in: Teubner-Taschenbuch der Mathematik Teil I I, Teubner
Verlag, Stuttgart-Leipzig, p.111-125, 1995

8.1 Begriffsbestimmung und Rückblick

Computeralgebra nutzt den Rechner in erster Linie nicht zum Berechnen von Zahlen, sondern zum Umgang mit mathematischen Formeln. In [2] wurde folgende Definition des Gebiets gegeben:
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 Strukuren 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.

Computeralgebra hat drei Facetten: Einmal den Nutzeraspekt, dann den der algorithmischen Umsetzung mathematischer Probleme, sowie den der Realisierung und Implementierung der für den Umgang mit mathematischen Symbolen geeigneten Datenstrukturen.

Vom Gesichtspunkt des Nutzers ist Computeralgebra besonders einfach zu beschreiben. Man stelle sich vor, man hätte alle Formeln aus Schulzeit, Studium und Beruf sofort zur Verfügung, würde diese anreichern mit dem algorithmischen Wissen von professionellen Mathematikern, Ingenieuren und Naturwissenschaftlern, und hätte dann noch jemanden, der den Umgang mit diesen Informationen intelligent erledigt, die Schreibarbeit abnimmt, die Formeln fehlerfrei ineinander einsetzt, Ableitungen bestimmt, Gleichungen löst, Graphiken zeichnet, Geometrie verdeutlicht, und benötigte Resultate beliebig genau numerisch ausrechnet. Diese Vorstellung deckt einen kleinen Ausschnitt dessen ab, was Computeralgebra heute ist und einen noch kleineren dessen, was sie morgen leisten wird.

Daß Computer mit mathematischen Objekten nicht nur numerisch sondern auch symbolisch umgehen können, sollte nicht verwundern. In gewisser Weise ist der symbolische Umgang einfacher als der numerische. Hat man zum Beispiel $5(\pi)^2/\pi$ auszurechnen, so wird man sich sowohl symbolisch, wie auch mit dem gesunden Menschenverstand, sehr leicht davon überzeugen, daß dies 5π ist. Betrachtet man die Aufgabe aber numerisch, so ist sie von nahezu hoffnungsloser Kompliziertheit: Man muß sich darüber klarwerden, daß es sich um einen Bruch handelt, bei dem sowohl im Nenner wie im Zähler undendliche Dezimalbrüche stehen, die ein Computer allenfalls approximativ auswerten kann, also muß man sich über die erforderliche Genauigkeit klarwerden. Mit dem Ergebnis, einer Approximation eines unendlichen Dezimalbruchs durch eine vielleicht 20-stellige, oder auch 100-stellige Zahl kann man zudem nicht einmal besonders viel anfangen da alle strukturellen Aus-

sagen verloren gegangen sind. Ein symbolisches Ergebnis hingegen hat den nicht zu unterschätzenden Vorteil, daß es *exakt* ist. Eine symbolische Berechnung ist nicht nur einfacher, sondern natürlicher und korrekter. Bei der Behandlung dieser und ähnlicher Aufgaben kommt es nur darauf an, die entsprechenden Manipulationsalgorithmen für symbolische Daten effizient und nutzerfreundlich auf dem Rechner bereitzustellen.

Daß Rechner auch zur Behandlung *symbolischer Daten* geeignet sind, hat wohl als erste Ada Augusta Countess of Lovelace bemerkt, die statt der romantischen Natur ihres Vaters den scharfen analytischen Verstand der Mutter geerbt hatte. Im Jahr 1842 schreibt sie über Babbage's Maschine (zitiert nach [10], p. 10):
Many persons ... imagine that the business of the engine is to give results in numerical notation, the nature of its processes must consequently be arithmetical and numerical rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or other general symbols; and in fact it might bring out its results in algebraical notation **were** *provisions made accordingly.*

Geht man noch weiter zurück, so kann man sogar den großen katalonischen Dichter und Philosophen Raimundus Lullus (1234-1316) als geistigen Wegbereiter der Gedanken, die der Computeralgbra zugrunde liegen, ansehen. Seine logischen Studien führten zum Versuch einer mechanischen Methode um aus Kombination allgemeiner Grundbegriffe zu Lösungen wissenschaftlicher Aufgaben zu führen. Er konstruierte deshalb eine Maschine, bei welcher mit Hilfe von Buchstaben, Zahlen und drehbaren Kreisen eine Ableitung "jeder Wahrheit" möglich sein sollte.

Erste Ansätze zur Realisierung allgemein brauchbarer Systeme sind allerdings erst in den 50-er Jahren dieses Jahrhunderts zu beobachten: 1953 wird in zwei Diplomarbeiten [16], [13] dargelegt, wie Digitalrechner formal differenzieren können. In den 60-er Jahren beginnt in den USA die systematische Entwicklungsarbeit zur Schaffung von Computeralgebrasytemen; 1970 werden die auf nahezu 2000 Seiten niedergelegten Formeln im unvollendeten Werke von Charles Eugène Delaunay [23], zu deren Korrektur er selbst mehr als 10 Jahre brauchte, mit dem Computeralgebrasytem MACSYMA in nur 20 Stunden überprüft. Heute würde diese Prüfung, bei der auf Seite 234 ein Fehler gefunden wurde, weniger als zwei Stunden dauern.

Bereits im Jahre 1981 wurde die Zahl der Computeralgebrapakete auf über 60 geschätzt, diese unterteilten sich in *general purpose* Systeme und Systeme für Spezialaufgaben, wie zum Beispiel Gruppentheorie, Zahlentheorie und andere algorithmisch besonders gut erschlossene mathematische Disziplinen.

Computeralgebra beginnt heute die mathematische Ausbildung an Hochschulen und Schulen zu durchdringen. Die immer wachsende Leistungsfähigkeit der Rechner erlaubt anspruchsvolle Systeme, mit nutzerfreundlichen Oberflächen, die früher Mainframes erforderten, auf Notebooks und Laptops einzusetzen. Daneben werden klassische Systeme wie REDUCE [11], und MACSYMA ständig verbessert. In welcher Weise Computeralgbra den Umgang unserer Kinder mit Mathematik prägen, und wie sie deren Verständnis von Wissenschaft und Technik beeinflussen wird, ist heute unabsehbar.

Auskunft über Computeralgebra ist heute leicht zu erhalten: Überblicke über vorhandene Systeme und Berichte über den aktuellen Stand von Forschung und Entwicklung im Bereich der Computeralgebra wurden in den letzten Jahren von verschiedenster Seite vorgelegt ([10], [2], [22], [19]). Wer

ständig aktualisierte Information über die gängigen Systeme wünscht, sollte von Zeit zu Zeit die von Paulo Ney de Souza veröffentlichte Liste konsultieren ([18], [17]). Darüber hinaus gibt es eine Vielzahl von Nutzergruppen, Special Interest Groups und Bulletin Boards, die sich mit dem Thema beschäftigen; Hinweise darauf findet man in [2].

8.2 Eine elementare interaktive Sitzung

Schaut man sich einfache Befehle in einem gängigen Computeralgebarsystem an, wie zum Beispiel in MAPLE [15], so stellt man zuerst fest, daß solche Systeme in der Arithmetik *exakt* rechnen, also bei ganzen und rationalen Zahlen nicht auf eine feste Stellenzahl beschränkt sind. Der Aufruf von

```
> 85235643658438743656743564346624 67/32879832643554354+23/45;
```

liefert zum Beispiel

$$\frac{1578437845526643404162889672269799}{6088857896954510}$$

Wer durch die Länge dieser Zahlen noch nicht beeindruckt ist, der versuche es einmal mit

```
> 10000! +2^10000;
```

um nach akzeptabler Zeit das korrekte 35-tausend-stellige Ergebnis zu erhalten. Die *Langzahlarithmetik* der Computeralgberasysteme enthält zahlentheoretische Funktionen, zum Beispiel zur Faktorisierung ganzer Zahlen

```
>ifactor(3553443884664);
```

$$(2)^3,(3)^2,(49353387287)$$

probabilistische Primzahltests

```
>isprime(49353387287); isprime(49353387289);
```

true

false

und vieles andere mehr. Ein schwierigerer Algorithmus liegt dem eingebauten *Gleichungslöser* zugrunde, der auf einfache und komplizierte, auf lineare und nichtlineare Gleichungssysteme angewandt

```
> solve({x+2*y=3,y+1/x=1}, {x,y});
```

bei Lösbarkeit in einer geeigneten Struktur sofort das Ergebnis liefert:

$$\{x=-1,y=2\},\{y=1/2,x=2\}$$

Da die Differentiation ein einfaches algorithmisches Verfahren ist, verwundert es nicht, daß auch Taylorreihen für solche Systeme kein Problem sind:

```
>A:=convert(taylor(sin(x^2+sqrt(Pi+x)),x=0,3),polynom);
```

$$\sin(\sqrt{\pi}) + \frac{\cos(\sqrt{\pi})x}{2\sqrt{\pi}} + \left(-\frac{\sin(\sqrt{\pi})}{8\pi} + \cos(\sqrt{\pi})\left(1 - \frac{1}{8\pi^{3/2}}\right)\right)x^2$$

Der *convert*-Befehl wurde um die Taylorreihe geschachtelt damit das Restglied verschwindet. Will man ein solches Ergebnis auswerten, so weist man der Variablen x einen Wert zu und ruft den vorher berechneten *Ausdruck A* noch einmal auf:

```
> x:=3: A;
```

$$\sin(\sqrt{\pi}) + \frac{3\cos(\sqrt{\pi})}{2\sqrt{\pi}} - \frac{9\sin(\sqrt{\pi})}{8\pi} + 9\cos(\sqrt{\pi})\left(1 - \frac{1}{8\pi^{3/2}}\right)$$

Dies führt natürlich, wie es dem Anspruch auf exaktes Rechnen auch entspricht, nur zu einer *symbolischen Auswertung*. Will man eine approximative Auswertung als Floatingpointzahl, so muß man das dem System mitteilen:

```
> evalf(A);
```

$$-1.302786911$$

Wünscht man statt der hier voreingestellten 10-stelligen Genauigkeit eine 123-stellige Genauigkeit, so muß man eine entsprechende Anweisung über die Stellenzahl voranstellen:

```
Digits:=123:evalf(A);
```

Danach wird nun alles 123-stellig ausgewertet.

Tabellenwerke gehören der Vergangenheit an. Will man zum Beispiel eine Tabelle der Werte des Wahrscheinlichkeitsintegrals haben

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_0^x \exp(-x^2/2)dx \ ,$$

wie sie etwa (mit anderer Schrittweite) in der beliebten Formelsammlung Bronstein-Semendjajew von 1991 zu finden ist, so kann man sich die folgende Tabelle, sogar in formatierter Form, durch ein Computeralgebraprogramm erzeugen:

x	$\Phi(x)$	x	$\Phi(x)$	x	$\Phi(x)$	x	$\Phi(x)$	x	$\Phi(x)$
0.00	0.000	0.20	0.079	0.40	0.155	0.60	0.225	0.80	0.288
0.01	0.003	0.21	0.083	0.41	0.159	0.61	0.229	0.81	0.291
0.02	0.007	0.22	0.087	0.42	0.162	0.62	0.232	0.82	0.293
0.03	0.011	0.23	0.090	0.43	0.166	0.63	0.235	0.83	0.296
0.04	0.015	0.24	0.094	0.44	0.170	0.64	0.238	0.84	0.299
0.05	0.019	0.25	0.098	0.45	0.173	0.65	0.242	0.85	0.302
0.06	0.023	0.26	0.102	0.46	0.177	0.66	0.245	0.86	0.305
0.07	0.027	0.27	0.106	0.47	0.180	0.67	0.248	0.87	0.307
0.08	0.031	0.28	0.110	0.48	0.184	0.68	0.251	0.88	0.310
0.09	0.035	0.29	0.114	0.49	0.187	0.69	0.254	0.89	0.313
0.10	0.039	0.30	0.117	0.50	0.191	0.70	0.258	0.90	0.315
0.11	0.043	0.31	0.121	0.51	0.194	0.71	0.261	0.91	0.318
0.12	0.047	0.32	0.125	0.52	0.198	0.72	0.264	0.92	0.321
0.13	0.051	0.33	0.129	0.53	0.201	0.73	0.267	0.93	0.323
0.14	0.055	0.34	0.133	0.54	0.205	0.74	0.270	0.94	0.326
0.15	0.059	0.35	0.136	0.55	0.208	0.75	0.273	0.95	0.328
0.16	0.063	0.36	0.140	0.56	0.212	0.76	0.276	0.96	0.331
0.17	0.067	0.37	0.144	0.57	0.215	0.77	0.279	0.97	0.333
0.18	0.071	0.38	0.148	0.58	0.219	0.78	0.282	0.98	0.336
0.19	0.075	0.39	0.151	0.59	0.222	0.79	0.285	0.99	0.338

Diese Tabelle wurde als ungeänderter Latex Code durch Übernahme der Ausgabe des Befehls `erftable(19,0.01,0.2,10)` erzeugt. Dabei wurden die nachfolgenden selbstgeschriebenen MAPLE Routinen verwandt

```
erftable:= proc(n,schritt,colbreite, einschub)
local i,j,a,x,y,a,b;
lprint(‘\\begin{small}\\begin{tabular}\
{|r|r||r|r||r|r||r|r||r|r|}\\hline‘);
lprint(‘\$x\$ & \$\\Phi (x)\$ &\
\$x\$ & \$\\Phi (x)\$ &\
\$x\$ & \$\\Phi (x)\$ &\
\$x\$ & \$\\Phi (x)\$ &\
\$x\$ & \$\\Phi (x)\$ \
\\\\ \\hline‘);
a:= evalf(1/(sqrt(2*Pi)));j:=0;
for i from 0 to n do
x:= i*schritt;
for k from 0 to 4 do
b:=filter(a*int(exp(-y^2/2),y=0..x+k*colbreite),3); if not k=4 then
lprint(filter(x+k*colbreite,2),‘&‘,b,‘&‘) else
lprint(filter(x+k*colbreite,2),‘&‘,b)
fi
od;
j:=j+1;lprint(‘\\\\‘);
if j= einschub then
lprint(‘\&\&\&\&\&\&\&\&\& \\\\‘); j:= 0 fi
od;
lprint(‘\\hline \\end{tabular} \\end{small}‘) end:
```

```
filter:= proc(x,n)
local i, b, c, r;
c:= trunc(x);
b:= cat(c,`.`);
for i from 1 to n do
r:=trunc(x*10^i);
a:=r-10*c;c:=r; b:=cat(b,a) od
end:
```

Etwa 75% des Programmcodes wurde von den Befehlen zur Erzeugung der Latex-Tabellenanweisungen und dem Ausgabefilter `filter` zur Erzeugung einer normierten Länge verwandt, die Berechnung der Tabellenwerte selbst nahm kaum Programmierarbeit in Anspruch. Jeder geschickte Student kann durch Modifikation dieses Programms (wobei er dann für die Tabellengestaltung flexible Makros verwenden sollte) fast jedes gewünschte Tabellenwerk selbst drucken; schöner formatiert als die wichtigsten kommerziellen Tabellenwerke ist sein Produkt dann allemal. Aber noch etwas fällt bei der Betrachtung dieses Beispiels auf: Computeralgebrasysteme haben eine sehr flexible Hochsprache, die auch dem Laien sehr schnell die Programmierung komplexer mathematischer Sachverhalte erlaubt, und die neben dem Zugriff auf mathematische Libraries auch den Zugriff auf Funktionen zur Manipulation von Zeichenketten erlaubt. Computeralgebrasysteme können mathematisch anspruchsvollere Aufgaben, wie das *Integrationsproblem*

$$\int \frac{x}{(x^3-1)}dx$$

lösen:

```
> int( x/(x^3-1), x );
```

$$\frac{\ln(x-1)}{3} - \frac{\ln(x^2+x+1)}{6} + \frac{\sqrt{3}\arctan(\frac{(2\,x+1)\sqrt{3}}{3})}{3}$$

oder

```
>int( exp(-x^2)*ln(x), x=0..infinity );
```

$$-\frac{\sqrt{\pi}\gamma}{4} - \frac{\sqrt{\pi}\ln(2)}{2}$$

Als Verallgemeinerung der Algorithmen, welche der Integration zugrunde liegen, können auch *gewöhnliche Differentialgleichungen*, zum Beispiel

$$c^2\frac{d^2}{dx^2}y(x) = y(x)\left(2\left\{\frac{d}{dx}y(x)\right\}^2 + 2\,y(x)\frac{d^2}{dx^2}y(x)\right)$$

gelöst werden

```
>dgl:=c^2*diff(y(x),x,x)=y(x)*diff(y(x)^2,x,x):

>dsolve(dgl,y(x));
```

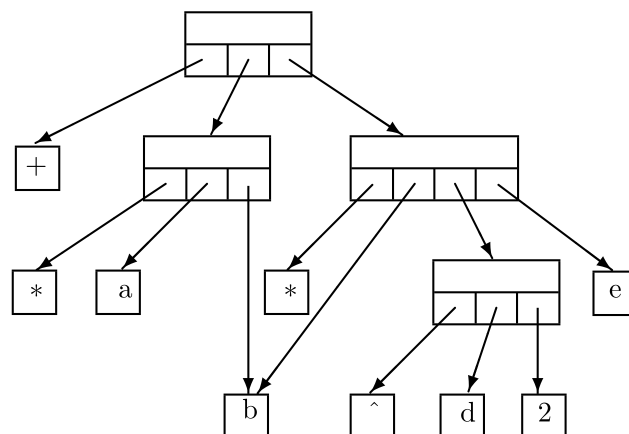
$$x = \sqrt{-1}\left(\frac{y(x)\sqrt{c^2-2\,y(x)^2}}{2} + c^2\sqrt{2}\arcsin(\frac{y(x)\sqrt{2}}{c})1/4\right)_C1^{-1} - _C2\ .$$

Moderne Systeme bieten neben dem Zugriff auf gute Algorithmen komfortable Interfaces für die verschiedensten Aufgaben: für Graphik, zur Herstellung von Filmen, zur interaktiven Fehlersuche und so weiter (siehe [6], [20], [21], [12]).

8.3 Datenstruktur und Evaluierung

Computeralgebrasysteme erlauben den Umgang mit den üblichen Daten der Mathematik und Informatik, wie *Listen, Tabellen, Mengen, Arrays, Polynome* usw. Diese Daten werden mit Hilfe von *Bäumen* dargestellt. Eine gute Kenntnis dieser Struktur ist die Voraussetzung für das effiziente Arbeiten mit jedem System.

In den meisten Systemen hat zum Beispiel der algebraische Ausdruck `a *



b + b * d ^ 2 * e;` intern die Form:

Dabei stehen die unterteilten Kästen für die inneren Knoten des Baumes. Der obere Teil deutet an, daß sich darin noch weitere Einträge, wie Typeklarationen und systeminterne Informationen, verbergen, die unteren Teile repräsentieren die Söhne des Ausdrucks. Die Kästen mit einem Eintrag sind die *Blätter* des Baumes. Daß zwei Knoten auf dasselbe Blatt zeigen können, was ja in der Natur nicht vorkommt, liegt daran, daß Computeralgebrasyteme an vielen Stellen das Prinzip der *eindeutigen Datenhaltung* realisieren: Wenn dieselben Daten an mehreren Stellen vorkommen, so sind sie physikalisch nur einmal im Speicher vorhanden. Diese Strategie spart Speicherplatz erhöht aber mitunter den Rechenaufwand.

Die Blätter und Teilbäume, die von einem durch einen Knoten repräsentierten Ausdruck ausgehen, heißen bei Computeralgebra-Systemen *Operanden*. Auf die einzelnen Blätter, wie auch auf Teilbäume, kann man mit einer geeignten Funktion (`op` in MAPLE und MuPAD) zugreifen. Zur Charakterisierung der

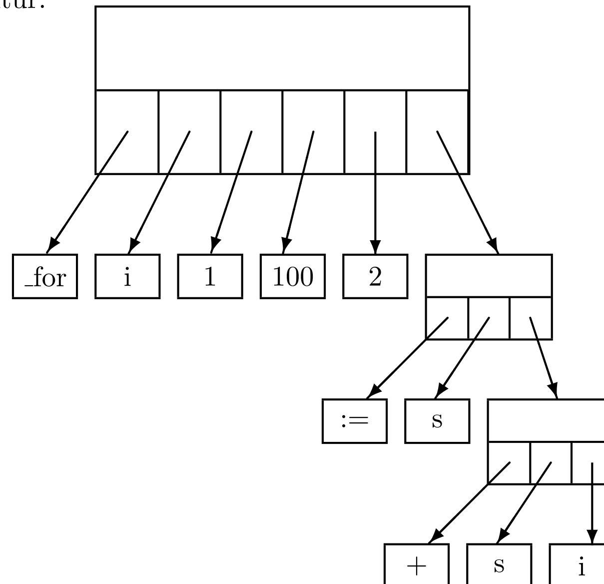
Operanden muß Infomationen über den Pfad mitgeben, der zum jeweiligen Blatt oder Teilbaum führt.

Mit diesem direkten Zugriff auf die Operanden eines Ausdrucks ist ein mächtiges Instrument zur bequemen Programmierung gegeben, dies insbesondere deshalb, weil jedes Computeralgebrasystem Funktionen zur Ersetzung beliebiger Operanden in einem Ausdruck zur Verfügung stellt.

Manche Computeralgebrasysteme (z.B. REDUCE oder MuPAD) fassen die einzelnen Konstrukte der *Hochsprache* des Systems wie

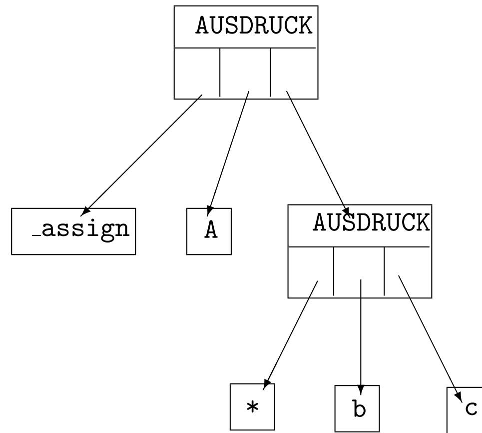
```
for i from 1 to 100 step 2 do s:= s + i end_for;
```

ebenfalls als algebraische Ausdrücke auf und verwenden dafür dieselbe Datenstruktur:

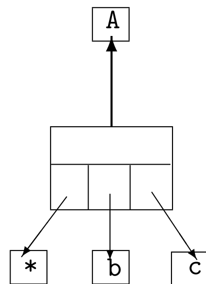


Dies erlaubt dem Nutzer *Zugriff* und *Substitution* auf die einzelnen Bausteine seiner Programme; er kann dadurch Programme verändern.

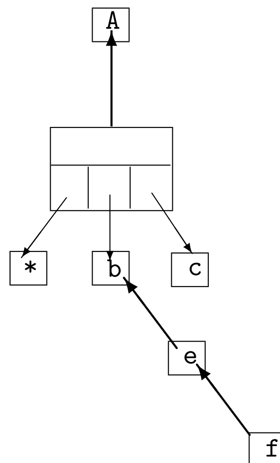
Zum Verständnis der Evaluierungsmechanismen von Computeralgebrasystemen, betrachte man die Zuweisung `A := b*c`. *Zuweisungen* sind Ausdrücke, die auf oberster Ebene durch eine Assignmentfunktion, sagen wir `_assign`, gebildet werden. Die Baumstruktur ist folgende:



Wenn man zum einfacheren Verständnis einen solchen `_assign`-Knoten durch einen dickgezeichneten Pfeil in umgekehrter Richtung symbolisiert, dann erhält man:



Eine mehrfache Zuweisung der Art `A := b*c; b := e; e := f;` führt dann zu:



Beim Aufrufen von `A` setzt das System, den dicken Pfeilen folgend, die Größen ein. Die dabei *hintereinander* durchlaufene Anzahl von dicken Pfeilen heißt *Substitutionstiefe*. Bei dem Beispiel ist die Substitutionstiefe 3. Beim interaktiven Arbeiten werden im allgemeinen alle Ersetzungen gemäß der dicken Pfeile ausgeführt (*vollständige Evaluation*), allerdings nur bis zu einer maximalen Tiefe, welche durch den Wert einer Variablen, z. B. `LEVEL`, festgelegt ist. Um nun zu verhindern, daß man durch rekursive Auswertungen in Endlos-Schleifen gerät, gibt es noch eine zweite Variable, z. B. `MAXLEVEL`, deren Wert im allgemeinen derselbe wie von `LEVEL` ist. Erreicht die Substitutionstiefe bei einer Auswertung den Wert dieser Variablen, so wird angenommen, daß eine *rekursive Zuweisung* vorliegt und eine Fehlermeldung wird ausgegeben. Manche Systeme erlauben eine Veränderung dieser fundamentalen Größen.

8.4 Wichtige Algorithmen

Schon bei der elementaren Arithmetik müssen wichtige Algorithmen implementiert werden, um symbolische Mathematik am Computer zu betreiben. Um die Gleichheit rationaler Zahlen zu erkennen, muß ein Computeralgebrasystem bei der Auswertung rationaler Zahlen die Operation des *größten gemeinsamen Teilers* `ggt` kennen. Um beim Vergleich

```
> bool(2356/1835 + 3453/1101= 8111/1835);
```

die richtige Antwort

$$true$$

zu bekommen, muß bei der Auswertung von `2356/1835 + 3453/1101` dieser Bruch, der auf einen Hauptnenner gebracht die Form

$$(2356 * 1101 + 3453 * 1835)/(1835 * 1101) = 8930211/2020335$$

hat, zu `8111/1835` gekürzt werden. Dafür muß der `ggt` von Zähler und Nenner berechnet werden. Dies ist eine einfache Aufgabe, die mit dem aus der Schule bekannten Euklidischen Algorithmus erledigt wird. Aber auch bei dieser simplen Frage ist ein Punkt erreicht, wo man durch Verbesserung bekannter Algorithmen mitunter Laufzeitverbesserungen um einen Faktor 10-100 erreichen kann. Berechnet man zum Beispiel bei der Vereinfachung von

$$\frac{a}{b} + \frac{c}{d} = \frac{p}{q}$$

die Werte von p, q zuerst durch $p = ad + bc,\ q = bd$ und kürzt dann durch den $\texttt{ggt}(p, q)$, so gelangt man zum selben Ergebnis wie bei der Berechnung durch

$$p = a\frac{d}{\texttt{ggt}(b, d)} + c\frac{b}{\texttt{ggt}(b, d)},\ q = bd/\texttt{ggt}(b, d)$$

und anschließender Kürzung durch den `ggt` [3], p. 62. Bei der Umstellung des Kerns von MuPAD von der ersten auf die zweite Methode wurde bei der

Berechnung der harmonischen Reihe von 1 bis 10 000 eine Laufzeitverbesserung um einen Faktor 100 erzielt.

Algorithmen, wie derjenige, welcher die Vereinfachung rationaler Zahlen durchführt, sind Teil des *Vereinfachers* eines Computeralgebrasystems. Die Qualität des Vereinfachers spielt eine große Rolle für die Qualität eines Computeralgebrasytems. Bei allgmeinen algebraischen Ausdrücken ist, anders als bei den rationalen Zahlen, nicht immer klar, was die einfachste Form ist. Zum Beispiel kann man sich bei den folgenden Formen desselben rationalen Ausdruckes durchaus darüber streiten, was die einfachere Form ist

$$\frac{x^{10} - y^{10}}{x - y}$$

$$x^9 + yx^8 + y^2x^7 + y^3x^6 + y^4x^5 + y^5x^4 + y^6x^3 + y^7x^2 + y^8x + y^9 \ .$$

Normalisierung, *Vereinfachung* und *Evaluierung* sind wegen ihrer Wichtigkeit für das Laufzeitverhalten im *Kern* des Systems implementiert. Da dieser, trotz seiner Wichtigkeit für den Nutzer, meist verborgen ist, werden im folgenden Algorithmen betrachtet, die dem Nutzer leichter zugänglich sind, zum Beispiel der Algorithmus eines *Differenzierers*. Im Allgemeinen hält man die Differentiation einer Funktion für ein Problem der Analysis, was aber seit Newton und Leibniz nur noch bedingt stimmt, da deren *Differentialkalkül* auf ein algebraisch-algorithmisches Rezept zur Berechnung von Differentialquotienten hinausläuft. Man sieht dies sofort ein, wenn man den Differenzierer in der Hochsprache eines Computeralgebrasystems schreibt. In MuPAD sähe ein Differenzierer, der auf Eingabe von `der(ausdruck,x)` die Ableitung von `ausdruck` nach `x` berechnet, folgendermaßen aus:

```
der := proc() local d, n; begin d:=type(args(1));   \#1\# case d of
DOM_INT do   \#2\# of DOM_RAT do of DOM_FLOAT do of DOM_COMPLEX do
of DOM_IDENT do if args(1)=args(2) then 1 else 0 end_if;    \#3\#
break of "_mult" do _plus(der(op(args(1), n), args(2))*
subsop(args(1), n=1) $ n=1..nops(args(1))); \#4\# break of "_plus"
do _plus(der(op(args(1), n), args(2)) $ n=1..nops(args(1))); \#5\#
break of "_power" do op(args(1), 1);  \#6\# op(args(1), 2);
args(1)*ln(last(2))*der(last(1), args(2))+last(1)*der(last(2),
args(2))*last(2)^(last(1)-1); break otherwise
_par(text2expr(d))(op(args(1)))* der(op(args(1), 1), args(2)) \#7\#
end_case; eval(last(1));  \#8\# end_proc:
```

In Zeile `#1#` wird der Datentyp des ersten der Prozedur `der` übergebenen Arguments `args(1)` festgehalten, danach wird dieser Typ mit den Grundtypen *ganze Zahl*, *rationale Zahl* bis *Bezeichner* verglichen. Dann wird, sofern dieses Argument gleich der Differentiationsvariablen ist, in `#3#` eine `1` ausgegeben, andernfalls eine `0`. Zeile `#4#` tritt in Aktion falls das erste Argument ein Produkt ist; es wird dann das Egebnis der Anwendung der Produktformel zurückgegeben. Zum Verständnis dieser Zeile muß man wissen, daß der Operator `_plus`, angewandt auf eine Folge, deren Summe zurückgibt, und daß `$` der Sequenzoperator ist, also `i$i=1..100` die Folge der Zahlen von 1 bis 100 erzeugt. Dies sind handliche Funktionen, da sie offensichtlich einen direkten

Zugriff auf die Bausteine der Datentypen zulassen. Wie praktisch solche Operationen sind, sieht man daran, daß zum Beispiel `_plus(i^3$i=1..10^3)` ganz offensichtlich ein Programm zur Summation der Kubikzahlen von 1 bis 1000 darstellt. Ganz analog werden dann in Zeilen `#5#` und `#6#` die Summen- und die Potenzregel abgehandelt. Der im Programm verwendete Funktionsaufruf `last(n)` liefert den n-ten vorangegangenen Ausdruck zurück; diese Ausdrücke werden in der sogenannten *Historytabelle* gespeichert. Es bleibt nur noch übrig Terme der Art $f(g(x))$ zu differenzieren; dies wird in Zeile `#7#` bewerkstelligt, wo die äußere Ableitung `_par(f)` mal innere Ableitung `der(g(x),args(2))` nach der Differentiationsvariablen, welche als zweites Argument vorkam, gebildet wird. Die dabei auftretende Funktion `text2expr` wandelte die durch die Typabfrage gegebene textuelle Information dabei wieder in ein algebraisches Datum um. Macht man nun durch Zuweisung dem System noch alle notwendigen äußseren Ableitungen bekannt

```
_par(sin) := cos:
_par(sinh):=cosh:
_par(cosh):=sinh:
usw.
```

so hat man in der Tat eine vollständige Implementation der Differentiation:

```
>>der(sin(x)*sin(x^(1/2)), x);

- 1/2   / 1/2 \
/ 1/2 \ x   sin(x) cos \ x /
cos(x) sin \ x / + -------------------------
2
```

Wirkliche Implementation des Differenzierers realisieren diese wichtige Funktion allerdings aus Effizienzgründen im Kern des Systems.

Andere wichtige Algorithmen sind:

Faktorisierung ganzer Zahlen: Dieser Algorithmus hat unter anderem Anwendungen im Public Key Verfahren der Kryptographie. Dabei handelt es sich um ein Verfahren, bei dem die Kenntnis des Verschlüsselungsverfahrens keine Entschlüsselung erlaubt. Entschlüsseln kann nur, wer die Zerlegung einer großen Zahl in ihre zwei Primfaktoren kennt, verschlüsseln kann aber jeder der die Zahl selbst kennt. Die Grenzen der Faktorisierung liegen gegenwärtig bei 120-stelligen Zahlen (sofern diese nicht von besonderer Struktur sind). Der einfachste Faktorisierungsalgorithmus für eine Zahl N besteht im Ausprobieren der Primzahlen $\leq N^{1/2}$. Dies führt, bezogen auf die Stellenzahl n, zu einem Algorithmus exponentieller Laufzeit mit dem sehr schlechten Exponenten $n/2$. Bessere Algorithmen verringern diesen Exponenten deutlich. Neuere Faktorisierungsverfahren sind einmal der des *quadratischen Siebs*, und der des *Zahlkörpersiebs*, welcher von besserer Komplexität ist, die sich aber erst bei sehr großer Stellenzahl auswirkt. Daneben gibt es Algorithmen, die mit zufällig gewählten Punkten elliptischer Kurven arbeiten [2], p. 29.

Polynomfaktorisierung: Auch die Faktorisierung von Polynomen hat vielfältige Anwendungen; zum Beispiel in der Kryptographie, aber auch überall da, wo man die Partialbruchzerlegung von rationalen Funktionen benötigt (für die Integration benötigt man diese allerdings nicht). Die Grundaufgabe ist die Faktorisierung bezüglich einer Variablen in endlichen Körpern. Dies ist mit dem klassischen Algorithmus von Berlekamp möglich. Unter den Verbesserungen und Modifikationen dieses Algorithmus haben sich Zufallsalgorithmen bewährt. Die Faktorisierung über Z oder Q wird durch Lösung des Problems in Restklassenkörpern und anschließendem Lifting durchgeführt (siehe [3], [8]).

Euklidischer Algorithmus: Verbesserungen des von der Schule her bekannten euklidischen Algorithmus spielen immer noch eine wichtige Rolle weil dieser Algorithmus zu den Grundoperationen in vielen Bereichen (ganze Zahlen, Polynome, Kettenbrüche) gehört. Eine interessante Variante ist der heuristische `ggt`, der die Faktorisierung von Polynomen in einer Variablen sehr beschleunigt [8].

Gröbnerbasen: Der euklidische Algorithmus spielt in Z und bei Polynomen einer Variablen eine besondere Rolle, da dort jedes Ideal Hauptideal ist, also von einem Element erzeugt wird, welches man als Basis des Ideals ansehen kann. Dies ist bei Polynomen mehrerer Variablen nicht mehr so, deshalb muß ein neuer Basisbegriff gefunden werden, der eine effektive Restklassenarithmetik erlaubt. Diese Rolle wird von den Gröbnerbasen übernommen. Diese sind Idealbasen, welche zu eindeutigen Normalformen führen, und die algorithmisch konstruiert werden. Die Konstruktion dieser Basen (Buchberger Algorithmus), die man auch braucht um Polynomgleichungen zu lösen, gehört zu den Standardalgorithmen.

Integration: Für den Praktiker spielt die Anwendung des Integrationsalgorithmus eine besondere Rolle. Man wird geneigt sein, rationale Funktionen durch Partialbruchzerlegung zu integrieren, dies setzt aber eine aufwendige Polynomfaktorisierung voraus und führt auch nur da zum Ziel, wo diese algorithmisch möglich ist. Nach Algorithmen von Hermite oder Horowitz braucht man aber nur eine quadratfreie Zerlegung und erhält für jedes Integral einer rationalen Funktion als Ergebnis eine Summe aus rationaler Funktion und Integral über einer rationalen Funktion von logarithmischer Art. Solche Integrale logarithmischer Art lassen sich nach Algorithmen von Rothstein und Trager lösen (siehe [3] und [8] wo man auch die Integrationsalgorithmen für kompliziertere Funktionenklassen findet). In [3] findet man zusätzlich eine kurze Zusammenfassung über Lösungsalgorithmen für gewöhnliche Differentialgleichungen.

Neben dieser willkürlichen Auswahl gibt es weitere wichtige Algorithmen, deren Bandbreite die strukturelle Reichhaltigkeit der Mathematik widerspiegelt. Effiziente Spezialsysteme für Zahlentheorie und Zahlkörper (PARI, THALES, KANT, SIEMATH), Gruppentheorie (GAP, CAYLAY), Kommutative Algebra (COCOA, MACCAULAY), Liealgebren (LIE), Algorithmenforschung und Entwurf (Aldes/SAC2, MAS) runden das Angebot an den

Nutzer von Computeralgebra ab [2] und geben Zugriff auf die algorithmische Durchdringung des jeweiligen Gebiets.

8.5 Programmierung und Effizienz

Bei erstmaliger Nutzung von Computeralgebra entsteht häufig der Eindruck, daß durch die Leistungsfähigkeit moderner Computer alle Effizienzerwägungen für den Mathematikanwender hinfort überflüssig seien: das Gegenteil ist der Fall! Gerade die anspruchsvolle Nutzung von Computeralgebra erfordert eine kritische Einstellung zur Effizienz algorithmischer Verfahren da anspruchsvolle Beispiele aus Bereichen formal einfach zu formulierender Mathematik häufig zu einem gigantischen *Intermediate Data Swell* führen (man siehe etwa [7], wo ein noch mit strukturellen Methoden handhabbares Beispiel angegeben ist, welches zu einem algebraischen Datum von mehreren Gigabyte führt).

Zur Demonstration der Notwendigkeit von Effizienzerwägungen seien die Fibonacci Zahlen betrachtet. Diese sind durch die Rekursion

$$f(n+2) := f(n) + f(n+1), \; f(1) = 1, \; f(0) = 0$$

definiert. Das folgende MuPAD Programm, welches man auf gleiche Weise in fast jedem anderen System schreiben kann, ist an Einfachheit und Transparenz kaum zu überbieten:

```
 fib := proc(n)
begin
if n<2 then n else fib(n-1)+fib(n-2) end_if; end_proc:
```

Der algorithmisch nicht geschulte Anwender wird, bevor er das Programm ausprobiert hat, darin eine besonders gelungene Lösung sehen, weil hier die mathematische Struktur der Rekursion deutlich abgebildet ist. Aber schon bei der Berechnung von `f(10)` stellt man fest, daß sich dieses Programm 176-mal selbst aufruft, und bei `f(18)` sind es schon 8360 Aufrufe. Offensichtlich hat das Programm exponentielle Laufzeit, was dazu führt, daß an die Berechnung von `f(10000)` gar nicht zu denken ist. Abhilfe schafft da schon die folgende Variation, bei der die *Option remember* verwandt wird:

```
fib_remember := proc(n)
option remember;
begin
if n<2 then n else fib_remember(n-1)
+fib_remember(n-2) end_if;
end_proc:
```

Bei dieser Option werden alle einmal berechneten Werte von Funktionen, die mit dieser Option ausgestattet sind, in einer Hashtabelle gespeichert. Beim Aufruf von `f(10000)` wird zuerst der Aufrufbaum (der Tiefe 1000)

angelegt, dann werden die Werte jeweils durch zweimaligen Zugriff auf die Hashtabelle eingesetzt. Dieses Verfahren ist schon bedeutend schneller, da zum Beispiel die Berechnung von `f(18)` nur 19 Aufrufe von `f()` benötigt, was schon bei diesem kleinen Argument zu einer Laufzeitverbesserung von etwa einem Faktor 100 führt. Die Komplexität dieses Algorithmus ist offensichtlich linear. Nachteil dieser Methode ist allerdings, daß durch den tiefen Aufrufbaum ein großer Stack benötigt wird was beim Aufruf von `f(1000)` auf kleineren Computern zum Absturz führt. Eine deutliche Verbesserung kann nun mit folgendem Programm erzielt werden, welches zudem noch kürzer (aber weniger transparent) ist:

```
fib_fast := proc(n) local i ;
begin 0; 1;
for i from 1 to n-1 do eval(last(1)+last(2)) end_for; end_proc:
```

Hierbei wird auf die *Historytabelle* zugegriffen: die jeweils beiden letzten Einträge werden solange addiert, wie der übergebene Parameter `n` in `fib_fast(n)` dies erfordert. Wieder ist die Komplexität linear, aber das speicherintensive Anlegen eines Aufrufbaums enfällt. Außerdem werden statt der nutzerdefinierten Funktion `f()`, welche wie jede Nutzerfunktion interpretiert wird, die Systemfuntionen `eval()` und `last()` verwandt, die als Binärcodobjekte vorliegen. Dies gibt wieder einen Laufzeitvorteil von etwa einem Faktor 20-30. Aber auch dieses Programm kann vom algorithmischen her noch deutlich übertroffen werden, da man durch jeweils einen Schritt die Berechnung des verdoppelten Arguments vornehmen kann; die Komplexität ist dann logarithmisch.

Wie das Beispiel zeigte, sind gute Laufzeiten nicht nur eine Frage der Komplexität der Algorithmen, sondern sie hängen empfindlich von den verwendeten Programmkonstrukten und der Art und Weise, wie diese im System implementiert sind, ab. Grundsätzliches läßt sich dazu nicht sagen, da dieselben Datenstrukturen in unterschiedlichen Systemen unterschiedlich implementiert sind. Als einheitliche Regel kann man allerdings festhalten, daß unnötige Funktionsaufrufe von nutzerdefinierten Prozeduren kostspielig sind, ebenso wie unnötige Evaluationen. Die Kostspieligkeit von Evaluationen führt dazu, daß man in vielen Systemen durch Zugriff auf die Historytabelle effizienter programmieren kann. Der Grund dafür ist darin zu sehen, daß diese Einträge nicht noch einmal evaluiert werden. Leider sind bei den meisten Computeralgebrasystemen die Evaluierungsmechanismen für den Nutzer keineswegs transparent. Auch die Art und Weise wie Parameter, die einer Prozedur übergeben werden, ausgewertet werden, unterscheiden sich bei verschiedenen Systemen.

Zu den wichtigen Interna eines Systems zählen: *Speicherverwaltung, eindeutige Datenrepresentation, Evaluierungsstrategien, Baumstruktur* und *Datenstruktur*, um nur einige zu nennen. Bei all diesen Implementierungsfragen gibt es keine optimalen Lösungen. Bei der Speicherverwaltung kann man zum Beispiel von einem, durch Erfahrung festgelegten, Durchschnittswert

der Größe eines algebraischen Datums ausgehen, und bevorzugt Speichersegmente dieser Größe freigeben, oder man kann, wegen der Bandbreite mathematischer Strukturen, eine solche Durchschnittsannahme ablehnen. Im ersten Fall wird eine Vielzahl von durchschnittlichen Problemen mit großer Geschwindigkeit abgearbeitet werden, im zweiten Fall wird die ständig notwendige Auflösung der Speicherfragmentierung zu Laufzeiteinbußen führen, dafür werden aber nichtstandard-Probleme effizienter erledigt. Bei der Datenhaltung kann man das Prinzip der eindeutigen Datenhaltung konsequent durchführen, oder man kann Kompromißstrategien verfolgen. Im ersten Fall wird sparsam mit dem Speicherplatz umgegangen, was aber Laufzeit kosten kann, da ständig überprüft werden muß, welche Daten physikalisch neu anzulegen sind und von welchen man nur logische Kopien braucht, im zweiten Fall wird das Laufzeitverhalten auf Kosten der Speicherökonomie verbessert. Bei der Baumstruktur kann man grundsätzlich auf n-äre Bäume setzen, oder möglichst (balancierte) binäre Bäume verwenden. Solche Entscheidungen haben, selbst wenn sie zu gleicher Komplexität führen, mitunter entscheidenden Einfluß auf die Geschwindigkeit, mit der einzelne Problemklassen behandelt werden.

8.6 Ausblick

Wichtige Neuerungen bzw. Änderungen bestehender Computeralgebrasysteme werden sein:

- *Domains*: Einführung von Kategorien und nutzerdefinierte Datentypen mit objektorientierten Konstrukten.
- *Module*: Möglichkeiten der Kompilierung zu Binärcodeobjekten und der dynamischen Linkung derselben.
- *Parallelisierung*.

Domains: Computeralgebrasysteme stellen Basisdatentypen zur Verfügung. Um der Bandbreite mathematischer Strukturen gerecht zu werden, können Computeralgebrasyteme in Zukunft nicht ohne die Möglichkeit zur Schaffung nutzerdefinierter Datentypen auskommen. Solche Datentypen müssen objektorientierte Programmierung in der Hochsprache erlauben, d.h. bei der Definition neuer Datentypen muß es dem Nutzer möglich sein, in der Definition zu vereinbaren, wie bestehende Systemfunktionen auf diese Daten reagieren. Um ein Beispiel zu nennen: bei der Definition eines speziellen Rings muß man auch definieren können, wie die bestehenden Operationen `+,*` auf die Elemente dieses Rings wirken, d.h. bestehende Funktionen müßen überladbar sein. Abstrakt gesehen sind Domains Zusammenfassungen der Operationen auf einem Datentypus und Domainelemente sind Daten mit Verweis auf die zugehörige Domain. Neben Domains werden alle zukünftigen Computeralgebrasysteme die Möglichkeit eröffnen, Kategorien

zu definieren. Kategorien, wie Ringe, Körper usw. sind dabei Zusammenfassungen algebraischer Strukturen mit gemeinsamen Eigenschaft zu einer Klasse. Die Einführung von Domains und Kategorien sind die Voraussetzung für die Entwicklung *generischer Algorithmen*, also solche die unabhängig von der Realisierung der jeweiligen algebraischen Struktur auf Kategorien wirken. Zum Beispiel sollte ein einmal geschriebener Euklidischer Algorithmus auf jeden neu definierten Euklidischen Ring anwendbar sein. In AXIOM und MuPAD sind diese Möglichkeiten bereits weitgehend im Kern implementiert, in anderen Systemen gibt es Implementationen auf Libraryebene.
Module: Computeralgebrasysteme sind interaktive interpretierende Systeme deren typische Nutzung aus dem sich wiederholenden Zyklus: Eingabe, Interpretation, Evaluierung und Ausgabe ergibt. Die Eingabesprache ist die Programmiersprache des Systems; im allgemeinen eine hochstrukturierte Sprache. Die dem Nutzer zur Verfügung stehenden Funktionen sind entweder System-Funktionen oder Library-Funktionen. Die System-Funktionen, die in Binärcode vorliegen, sind schnell und ein fester Bestandteil des Systems. Die Library-Funktionen werden in der Programmiersprache des Systems geschrieben.

Bisher steht der Nutzer oder der Entwickler bei der Schaffung neuer Funktionsbibliotheken vor der Alternative diese als System-Funktionen oder als Library-Funktionen zu implementieren. Neben den schon genannten Vorteilen haben die System-Funktionen den Nachteil der umständlichen Programmierung direkt im Kern, und einer damit verbundenen unerwünschten Aufblähung des Kerns. Library-Funktionen sind, da sie interpretiert werden, relativ langsam, haben aber den Vorteil, daß sie zur Laufzeit aus dem Speicher des Systems entfernt werden können, daß sie vorhandenen Funktionsbibliotheken dynamisch ergänzen, und daß sie einfach zu programmieren sind.

Zur Schaffung von Funktionen, welche die Vorteile beider Funktionsklassen in sich vereinigen werden zukünftig Modulfunktionen eingeführt. Diese neuen Funktionen bestehen aus Binärcode, zum Beispiel kompiliertem C-Code, der direkt vom System ausgeführt wird, also nicht interpretiert wird. Trotzdem sind Module als eigenständige Objekte kernunabhängig. Da sie zur Laufzeit lad- und ausladbar sind, bilden sie eine speicherfreundliche Ergänzung des Systems. Die Vorteile sind: Geschwindigkeit, Speicherdynamik und Erweiterbarkeit. Wegen der Kernunabhängigkeit sind sie relativ einfach zu programmieren und portabel, weiterhin erlauben sie die kernunabhängige Integration bestehender Software in das System.
Parallelisierung: Wegen der Komplexität wirklicher mathematischer Probleme wird in Zukunft bei Computeralgebrasystemen jede mögliche Effizienzsteigerung genutzt werden, insbesondere die Möglichkeit zur Parallelverarbeitung.

Ansätze in dieser Hinsicht sind vorhanden. Zum Beispiel das von Kaltofen et al. [4] entwickelte System DSC (System for Distributed Symbolic Computation), das auf Maple basierende und auf C/Linda aufsetzende System *Sugarbush* [1], sowie PAC, SAC-2, PARSAC-2 [14], PACLIB [9], und die Netzwerkver-

sionen von SAC-2 [2]. Daneben gibt es MuPAD [6], welches als paralleles general-purpose System entworfen wurde und auf die Architektur eines Netzwerks bestehend aus Shared Memory Maschinen zielt.

DSC ist ein Software-System für die verteilte Bearbeitung von großen Computeralgebra-Problemstellungen, welches auf den Internet Standard-Protokollen UDP und TCP basiert und auf heterogenen Unix-Netzwerken einsetzbar ist. DSC ist kein Computeralgebra-System im eigentlichen Sinne, es stellt die Basis für verteilte Applikationen in C und Lisp dar, dabei kann auch Source-Code verschickt werden, der dann von einem Remote-Server kompiliert wird.

Sugarbush hat zum Ziel das bestehende System mit geringem Aufwand zu einem parallelen System zu erweitern. Dabei werden mehrere Maple-Prozesse gestartet, zu denen jeweils ein *parallel communication transceiver (PCT)* gehört, der die jeweiligen Operationen auf entsprechende Linda-Konstrukte abbildet. Das System eignet sich hauptsächlich für *grobkörnige Parallelität*, da der Kommunikationsaufwand sehr hoch ist.

PACLIB ist ein listenorientiertes paralleles C-Library Paket, bestehend aus einer Sammlung von Algorithmen für Aufgaben von der Langzahlarithmetik bis zum Rechnen in endlichen Körpern. PACLIB erlaubt die Nutzung von feinkörniger Parallelität.

References

[1] B.W. Char: *Progress report on a system for general-purpose parallel symbolic algebraic computation*, Proceedings of the ACM-SIGSAM 1990, International Symposium on Symbolic und Algebraic Computation, T.T. Sasaki ed., ACM Press/Addison-Wesley, 1990

[2] *Computeralgebra in Deutschland - Bestandsaufnahme, Möglichkeiten, Perspektiven*, Herausgegeben von der Fachgruppe Computeralgebra der GI, DMV, GAMM, Passau und Heidelberg, 1993

[3] J. H. Davenport, Y. Siret und E. Tournier: *Computer Algebra: Systems and Algorithms for algebraic computation*, Academic Press, London - San Diego - New York - Boston - Sidney -Toronto - Tokyo, 1988

[4] A. Diaz, M. Hitz, E. Kaltofen, A. Lobo und T. Valente: *Process Scheduling in DSC and the Large Sparse Linear Systems Challenge*, Design and Implementation of Symbolic Computation Systems - DISCO 93, A. Miola ed., Lecture Notes in Computer Science, Springer Verlag, New York - Berlin - Heidelberg - London - Paris - Tokyo, 1993

[5] *Dictionary of Computing, Third Edition*, Oxford University Press, Oxford New-York-Tokyo, 1990

[6] B.Fuchssteiner, W. Wiwianka, K. Gottheil, A. Kemper, O.Kluge, K. Morisse, H. Naundorf, G. Oevel und T. Schulze: *MuPAD Multi Processing Algebra Data Tool: Benutzerhandbuch*, Birkhäuser Verlag, p.414, 1993

[7] B. Fuchssteiner: *Nichtlineare Dynamische Systeme: Eine Fallstudie für die Anwendung von Computeralgebramethoden*, in: Geometry and Analysis: Trends in Teaching and Research, B. Fuchssteiner und W. A. J. Luxemburg, eds., Bibliographisches Institut Mannheim, p.217-239, 1992

[8] K. O. Geddes, S.R. Czapor und G. Labahn: *Algorithms for Computeralgebra*, Kluwer Academic Press, Boston - Dordrecht - London, 1992

[9] H. Hong, A. Neubacher und W. Schreiner: *The Design of the SACLIB/PACLIB Kernels*, Design and Implementation of Symbolic Computation Systems - DISCO 93, Lecture Notes in Computer Science Vol. 722, Seiten 288–302, Springer Verlag, New York - Berlin - Heidelberg - London - Paris - Tokyo, 1993

[10] A. C. Hearn, Ann Boyle und B.F. Caviness: *Future Directions for Research in Symbolic Computation*, Siam Reports on Issues in the Mathematical Sciences, Philadelphia, 1990

[11] A.C. Hearn: *REDUCE User's Manual*, The Rand Corporation, Santa Monica CA, 1991

[12] R.D. Jenks und R.S Suter: *AXIOM, the Scientific Computing System*, Springer Verlag, New York - Berlin - Heidelberg - London - Paris - Tokyo, 1992

[13] H. G. Kahrimanian: *Analytical Differentiation by a Digital Computer*, MA Thesis, Temple University, 1953

[14] W. Kuechlin: *The S-Threads Environment for Parallel Symbolic Computation*, Computer Algebra and Parallelism, R. Zippel ed., Lecture Notes in Computer Science, Vol. 584, Springer Verlag, New York - Berlin - Heidelberg - London - Paris - Tokyo, 1990

[15] Maple V: *Library Reference Manual, Language Reference Manual, Tutorial*, Springer Verlag, New York - Berlin - Heidelberg - London - Paris - Tokyo, 1991 und 1992

[16] J. F. Nolan: *Analytical Differentiation on a Digital Computer*, SM Thesis, Mass. Inst. of Technology, 1953

[17] P. N. de Souza: *Computer Algebra Systems*, AMS Notices, 40/6, p.617-623, 1993

[18] P. N. de Souza: *Distributors of Computer Algebra Systems 11/93*, Internet: ca@math.berkeley.edu, (1993)

[19] F. Winkler, B. Kutzler und F. Lichtenberger: *Computer-Algebra Systems*, Risc-Linz series no 88-10.0, p.33 pages, 1988

[20] S. Wagon: *MATHEMATICA in Aktion*, Spektrum Akademischer Verlag, Heidelberg - Berlin - Oxford, 1993

[21] S. Wolfram: *Mathematica, Ein System für Mathematik auf dem Computer*, Addison-Wesley, Reading Mass. - New York - Amsterdam, 1992

[22] *MathPAD: Eine nutzerorientierte Information aus der MathPAD-Gruppe an der Universität-Gesamthochschule Paderborn*, Vol. 1, Heft 3, September 1991

[23] R. Pavelle, M. Rothstein und J. Fitch: *Computer algebra*, Scientific American, 245, p.102-113, December 1981