Computeralgebrasysteme

Programmierung und Analyse eines Computeralgebrasystems

Entstanden als Maturarbeit 2012
Gymnasium Münchenstein

Verfasser
Johannes Kapfhammer

Betreuungsperson
André Studer

Inhaltsverzeichnis

1 Abstract

Mathematik war nie eine reine Wissenschaft des Kopfes. Früher behalf an sich mit dem Kerbholz, dem Abakus und mit Rechenschiebern sowie mit Formelsammlungen und Tabellen. Die Taschenrechner, die vor einigen Jahrzehnten Einzug in die Schule gefunden haben, sind heute weitgehend von Computeralgebrasystemen abgelöst worden.

Diese Systeme sind für Schüler, Forschung und Industrie ein unerlässliches Werkzeug und bilden ein eigenes Marktsegment, dessen wichtigste Produkte Namen wie Axiom, Maple, Mathematica, Maxima, MuPAD oder Reduce tragen.

Man kommt nicht umhin, Computeralgebrasysteme als Hightech zu sehen, denn sie sind noch immer Gegenstand der Forschung. Sie sind aus der künstlichen Intelligenz hervorgegangen und werden teilweise immer noch dazugezählt.

In der Schule müssen sie für Ausbildungszwecke geeignet sein, weshalb ihre Anforderungen da weniger auf Leistung als auf einfache Bedienbarkeit ausgelegt sind. Diese Schulsysteme werden nicht als Computersoftware vermarktet, sondern als programmierbare Taschenrechner, d.h. sie stellen eine feste Verbindung eines CAS mit einer einfachen Hardware dar. Erstaunlicherweise gibt es hier nur zwei Hersteller, Texas Instruments und Casio, wobei ersterer in der Schweiz (und nicht nur da) eine Monopolstellung einnimmt.

In meiner Maturarbeit habe ich untersucht, wie Computeralgebrasysteme (CAS) im Allgemeinen aufgebaut sind und anhand dieser Erkenntnisse eines völlig neu von Grund auf programmiert. Der Kern in ihm, wie auch in vielen anderen Systemen, stellt ein "Interpreter" dar, der Code in einer dem CAS eigenen Programmiersprache ausführt. Es wird dabei ein regelbasierter Ansatz gewählt, bei dem die Umformungsregeln direkt angegeben werden können. Der Vorteil ist, dass mit vergleichweise wenig Aufwand vergleichsweise viel erreicht werden kann, der Nachteil ist die etwas langsamere Ausführung. Aus diesem Grund werden zeitkritische Routinen in das Hauptprogramm verlagert. Unter dem Link `http://www.gymmuenchenstein.ch/johannes/ma/` ist das funktionsfähige System zu finden.

Verglichen mit dem TI Voyage ist mein Halbjahresprojekt in ausgewählten Punkten besser, in dem Sinne, dass es Umformungen erledigen kann, bei denen der Voyage scheitert. Das grundlegende Design ist viel offener und erlaubt auf einfache Art und Weise, neue Regeln hinzuzufügen und andere zu modifizieren. Ein wichtiger Punkt der Computeralgebrasysteme ist ihr breiter Funktionsumfang, bei dem das von mir entwickelte CAS nicht mit dem Voyage und schon gar nicht mit einem seiner Software-Konkurrenten mithalten kann.

Es hat sich herausgestellt, dass der Ansatz des TI Voyages erhebliche Schwächen hat. Wenn mir ein vergleichbares Programm gelingt, dann müsste es in einem professionellen Entwurf möglich sein, ihn zu übertreffen. Es bleibt zu hoffen, dass zukünftige Rechner-Modelle für die Schule ein wesentlich verbessertes Leistungsvermögen zeigen.

2 Vorwort

Der Voyage hat mich schon immer fasziniert und es machte mir Spass, ihn mit Eingaben zu ärgern und seine Grenzen auszuloten. Immer wieder bin ich auf Stellen gestossen, an denen der Voyage ungenaue oder falsche Ergebnisse lieferte.

Weil mich das Gebiet der Programmierung sehr interessiert, dachte ich, ich könnte mich einmal an einen Gegenentwurf zum Voyage wagen und ein eigenes Computeralgebrasystem programmieren, das viele der Schwächen des Voyage nicht besitzt. Dieses kleine Programm steht im Zentrum meiner Maturarbeit.

3 Einleitung

Ein Computeralgebrasystem (CAS) muss, wenn es gut ist, die verschiedensten mathematischen Aufgaben erkennen und lösen können. Dazu ist es notwendig, dass das System mathematische Ausdrücke erkennt und so vorverarbeitet, dass sie von der inneren Logik vereinfacht werden können.

Mit welchen Ansätzen vereinfachen Computeralgebrasysteme mathematische Ausdrücke? Das ist die Leitfrage meiner Maturarbeit. Sie erfordert Recherche über den Aufbau verschiedenenr CAS-Typen.

Um mein Verständnis zu prüfen habe ich mich entschieden, ein eigenes CAS zu entwickeln, damit ich sehen kann, was alles zu einem CAS dazugehört, bis es funktioniert.

Gegen Ende möchte ich eine Bilanz ziehen und sehen, wie weit ich gekommen bin ob mein Programm Aufgaben, an denen der Voyage gescheitert ist, tatsächlich besser lösen kann.

Danksagung

Mein Dank geht an Herrn André Studer, der bereit war, diese Arbeit zu betreuen und mir dabei viele Freiheiten gelassen hat, meinen Eltern, die diese Arbeit auf verschiedene Weise unterstützt haben und der Schweizerischen Mathematik- und Informatikolympiade, die mein Interesse an der Mathematik und am Programmieren stark gefördert haben.

4 Schwächen anderer Computeralgebrasysteme

Um etwas über Computeralgebrasysteme zu erfahren, untersuchte ich die Funktionsweise von anderen Computeralgebrasysteme. Interessant sind dabei nicht die Eingaben, die richtig bearbeitet werden, sondern solche, zu denen ein unvollständiges oder sogar falsches Resultat angezeigt wird.

Alle Beispiele zum TI Voyage stammen aus einer ersten Version dieser Dokumentation, der Abgabe zur Maturarbeit. Die Beispiele zu Mathematica wurden nachträglich eingefügt.

4.1 Definitionsbereich

Die Gleichung $x = \frac{x}{x} - 1$ hat keine Lösung. Ist $x \neq 0$, dann ist $\frac{x}{x} = 1$ und somit $x = \frac{x}{x} - 1 = 0$. Allerdings widerspricht $x = 0$ der Annahme, denn bei $x = 0$ ist der Bruch $\frac{x}{x}$ undefiniert. Man sagt, 0 ist nicht im Definitionsbereich der Gleichung.

Gibt man die Gleichung beim Voyage ein, erhält man jedoch $x = 0$.

$$\text{solve}(x = \frac{x}{x} - 1, x) \quad \rightsquigarrow \quad x = 0$$

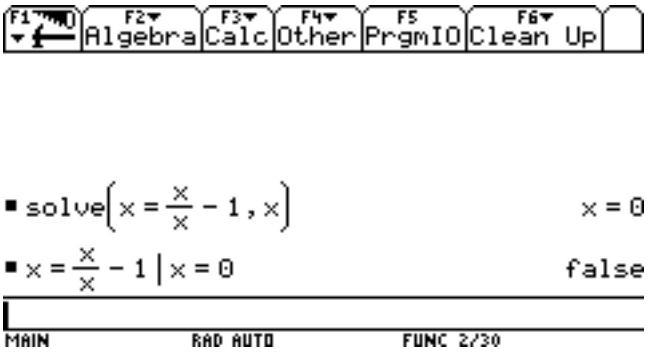


Abbildung 4.1: Der TI Voyage ignoriert den Definitionsbereich.

(Hinweis zur Notation: Links von dem "$\rightsquigarrow$" steht die Eingabe ins CAS, rechts davon die Ausgabe vom CAS. Das gleiche Beispiel ist auch im Bild 4.1 zu sehen.)

Erstaunlicherweise erhält man von Mathematica genau das gleiche, falsche, Ergebnis: $x = 0$.

```
In[1]:= Solve[x == x/x - 1, x]
Out[1]= {{x -> 0}}
```

Im Gegensatz zu Mathematica merkt der Voyage sogar, dass etwas nicht stimmt und zeigt klein an "Note: Domain of result may be larger" (bzw. in der deutschen Version: "HINW: Bereichsergeb. kann großer sein"). Besonders hilfreich ist das jedoch nicht, denn

der man muss erst einmal darauf kommen, dass sich der Hinweis auf die Umformung von $\frac{x}{x}$ nach 1 bezieht und nicht auf die Gleichung, denn deren Definitionsbereich wurde durch die Umformung eingeschränkt. Dennoch wird das Ergebnis $x = 0$ nicht in die Gleichung eingesetzt um zu überprüfen, ob es überhaupt stimmen kann, dies muss von Hand erledigt werden (siehe Abbildung 4.1).

Beide CAS formen zuerst $\frac{x}{x}$ nach 1 um. Die Gleichung lautet dann $x = 1 - 1$, was zu $x = 0$ führt. Das Problem ist nicht, dass $\frac{x}{x}$ nach 1 umgeformt wird, das ist durchaus zweckmässig und es wird von den Nutzern eines CAS auch erwartet, dass gekürzt wird.

Das Problem ist viel eher, dass sich das CAS nicht merkt, dass bei $x = 0$ der Ausdruck undefiniert ist.

These 1: Ein gutes *Computeralgebrasystem* merkt sich zu jeder Variable ihren Definitionsbereich.

4.2 Verzögerte Auswertung

Eine ungerade Zahl zu irgendeinem Exponenten modulo zwei ergibt immer eins, die zweite Rechnung in Abbildung 4.2 ist somit korrekt. Und die erste? Auf den ersten Blick ist sie unsinnig oder schlicht falsch, jedenfalls wenn man den Punkt am Ende nicht berücksichtigt.

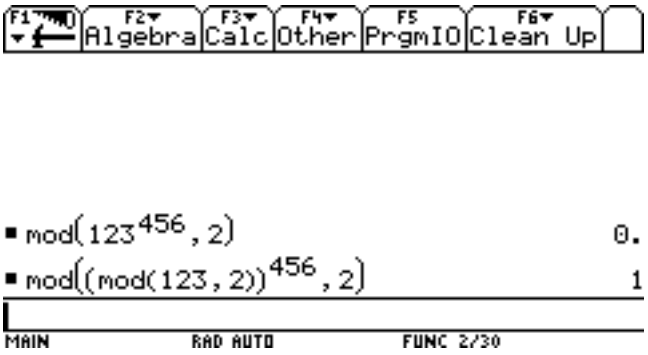


Abbildung 4.2: Modulo und grosse Zahlen.

Intuitiv ist klar, dass der *TI Voyage* zuerst 123^{456} auszurechnen versucht, dann aber, da die Zahl zu gross ist, von einer präzisen Ganzzahldarstellung auf eine gerundete *Fliesskommazahl*[1] umstellt. Dabei gehen niederwertige Ziffern verloren, insbesondere die letzte, und der Funktion mod bleibt nur noch übrig, die letzte Ziffer als 0 zu interpretieren.

Das ist schade, denn ein Mensch würde zuerst $123 \bmod 2 = 1$ ausrechnen und dann $1^{456} = 1$ und $1 \bmod 2 = 1$. Also so, wie die zweite Eingabe im Bild 4.2.

Dadurch, dass der *TI Voyage* strikt von innen nach aussen rechnet, also zuerst das erste Argument vereinfacht, dann das zweite und dann die Funktion ausführt, geht wertvolle Information verloren.

Ein kurzer Test zeigt, dass Mathematica gleich vorgeht wie der Voyage. Die Eingabe `Mod[123456789^12346789, 3]` benötigt ewig in der Auswertung. Wendet man den gleichen Trick an wie beim Voyage und schreibt `Mod[Mod[123456789,3]^12346789, 3]`, wird das Ergebnis ohne messbare Verzögerung ausgegeben. Was man eigentlich will,

[1] Eine Fliesskommazahl speichert Zahlen in der Form $a \cdot 10^b$, wobei a nur eine feste Anzahl Kommastellen besitzt. Manchmal wird auch eine andere Basis als 10 verwendet, näheres steht im Glossar.

ist das modulare Potenzieren, wofür ein spezieller Algorithmus existiert. Mathematica bietet dafür sogar eine Funktion `PowerMod[a, b, m]` an. Wer das weiss, kann `PowerMod[123456789, 12346789, 3]` eingeben. Trotzdem ist es eigentlich eine Aufgabe des CAS, und nicht des Nutzers, den passenden Algorithmus herauszusuchen.

Das Problem tritt nicht nur bei Modulo und Potenzen auf, sondern unter vielen Umständen. Eine Quadratzahl ist zum Beispiel nie eine Primzahl. Die kleinste Quadratzahl, 1, ist keine Primzahl und fällt damit weg. Jede grössere hat mindestens zwei gleiche Teiler ungleich 1. Also ist zum Beispiel $\left(2^{61}-1\right)^2$ keine Primzahl.

$$isPrime\left(\left(2^{61}-1\right)^2\right) \quad \rightsquigarrow \; false$$

Die Lösung liess drei Sekunden auf sich warten. Ein Mensch hingegen hätte es schneller sagen können. Wählt mein einen etwas grösseren Exponenten, sieht man, dass auch Mathematica zuerst das Argument ausrechnen möchte und erst danach probiert zu faktorisieren, was unter Umständen sehr lange dauern kann.

Bei der Eingabe werden Informationen geliefert, die bei weiteren Vereinfachungen gegebenenfalls verloren gingen. Es ist also ein Fehler, immer von innen nach aussen zu vereinfachen.

These 2: Ein gutes *Computeralgebrasystem* bietet Funktionen die Möglichkeit, an den unvereinfachten Ausdruck zu gelangen.

4.3 Erweiterbarkeit

Manchmal macht der Voyage nicht was er tun sollte. Meistens kann man, sobald man den Fehler ausfindig gemacht hat, die Eingabe entsprechend ändern, so auch bei Abbildungen 4.1 und 4.2. Bei 4.3 ist dies nicht möglich. Solche Probleme können nur gelöst werden, wenn man die inneren Vorgänge des CAS kennt und beeinflussen kann, und das ist beim Voyage nicht möglich.

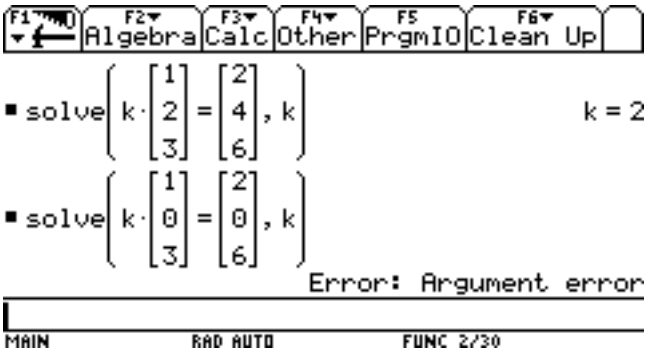


Abbildung 4.3: Unerklärlicher Fehler beim Rechnen mit Matrizen.

Ein anderes Beispiel sind logische Operatoren. Der *TI Voyage* kennt zwar das logische Und, Oder und das Exklusive Oder, nicht aber die Implikation oder das Verneinte Und. Es ist zwar möglich, diese Operatoren als Funktionen hinzuzufügen, damit sie in Und/Oder ausgedrückt werden. Es ist aber nicht möglich, einen Term aus Und/Oder so zu vereinfachen, dass ein Teilausdruck in eine Implikation geändert wird.

Im Vergleich dazu liegt der Erfolg von Mathematica unter anderem darin, dass viele Erkenntnisse in der Forschung Mathematica beigebracht und damit überprüft werden(4),

wodurch Mathematica immer auf dem aktuellen Stand der Forschung ist. Bei einem Schulrechner ist dies weniger wichtig, aber dennoch wünschenswert, sei es aus reinen Interesse an der Funktionsweise, zum Zweck der Fehlerbehebung oder für etwas Neues.

These 3: Ein gutes *Computeralgebrasystem* verbirgt die inneren Vorgänge nicht und erlaubt, diese zu beeinflussen und zu erweitern.

5 Computeralgebrasysteme

5.1 Definition

Vom Namen her ist ein Computeralgebrasystem (CAS) ein Computerprogramm, das Algebra betreiben, d.h. mit Symbolen rechnen kann.

So schreibt Wikipedia(8):

> The core functionality of a CAS is manipulation of mathematical expressions in symbolic form.

Wenn ich ein Programm schriebe, das Vielfache von a addieren kann – also beispielsweise von $2a + 3a + a + 5a$ auf $11a$ kommt –, das aber schon alles ist; zählt das schon als CAS?

Es ist sehr nahe dran. Viel näher als ein handelsüblicher Taschenrechner wie der *TI-30*, denn es hat einen wichtigen Schritt hinter sich: Es rechnet mit Variablen/Symbolen. Das Rechnen mit Symbolen ist vielleicht die einzige klare Abgrenzung, die sich machen lässt.

Im ersten Satz, bei der Definition von CAS, habe ich das "S" unterschlagen. Vergleicht man heute zwei Computeralgebrasysteme, so vergleicht man (a) ihre Mächtigkeit und (b) ihre Geschwindigkeit. Unter System muss man sich eine Sammlung von *Algorithmen* vorstellen, die eine Vielzahl von Aufgabentypen (und unendlich viele konkrete Eingaben) manipulieren kann.

Zu gerne hätte ich statt "manipulieren" "vereinfachen" geschrieben, aber je nach Kontext ist genau das Gegenteil der Fall. In der Praxis wird der Ausdruck oft in die so genannte Standardform gebracht, die aber je nach Situation kompliziert sein kann. Gemeinsam ist, dass sie die Eingabe umformen, oder eben manipulieren. Nicht umsonst benutzt Richard Fateman den Ausdruck "algebraic manipulation system" anstelle von "computer algebra system"(3).

Diese Sammlung von *Algorithmen* ist es auch, weshalb im Zitat oben von Kernfunktionalität die Rede ist. Wikipedia(8) zählt im folgenden fünfzehn Bereiche auf, die ein CAS eventuell unterstützen kann, aber nicht zwingend muss, und weitere elf, die manche CAS teilweise können, obwohl sie nicht mehr viel mit Computeralgebra zu tun haben. Das unterstreicht noch einmal den System-Charakter, beschreibt aber zu wenig genau, was ein CAS eigentlich tut.

Definition: Ein CAS kann Terme mit einer beliebigen Anzahl von freien Variablen symbolisch manipulieren und in eine zweckmässige äquivalente Form bringen.

Das äquivalent schliesst mit ein, dass z.B. $\frac{1}{3} \cdot a$ nicht in $1.33 \cdot a$ umgeformt werden darf. *numerische* Programme gehören nicht mehr zur Computeralgebra. Das heisst natürlich nicht, dass *numerische* Annäherungen verboten wären, ein CAS muss jedoch die Option bieten, darauf verzichten zu können.

5.2 Geschichtliche Ausgangslage

Als Charles Babbage den ersten Computer entwarf, seine "Analytical Engine" – die allerdings nie gebaut wurde –, schrieb Ada Lovelace wenige Jahre darauf das erste Computerprogramm. Sie schwärmte von den Möglichkeiten, die es eröffnen würde und schrieb (Zitiert nach (6)):

> [Babbage's Analytical Engine] can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraic notation, were provisions made accordingly. (Ada Augusta, Countess of Lovelace, 1844)

Der Wunsch nach einem Computeralgebrasystem ist noch älter als der erste Computer.

Mit der *Principia mathematica* (1910-1913) von Bertrand Russell und Alfred North Whitehead wurde versucht, die Mathematik vollständig und widerspruchsfrei in ein formales System zu giessen(5). Wäre das nicht die optimale Grundlage für Berechnungen am Computer? Kurz nachdem Heisenberg die Hoffnung der Physiker auf eine deterministische Welt zerstörte, zeigte Kurt Gödel:

> Jedes hinreichend mächtige[1] formale System ist entweder widersprüchlich oder unvollständig.

Und damit nicht genug, Alan Turing bewies, dass es in der Mathematik Probleme gibt, die nicht algorithmisch lösbar sind (6).

Turing begründete mit seiner Theorie die Informatik, die sich mit so genannten Turingmaschinen befasst. Auf dieser Grundlage konnte sich erst die Computeralgebra und damit auch Computeralgebrasysteme entwickeln.

Viele heutige Computeralgebrasysteme sind wie die *Principia mathematica* (5) aufgebaut, sie bestehen aus Axiomen und Regeln. Man könnte behaupten, ein CAS wäre von formalen Systemen der Mathematik, zum Beispiel der Zahlentheorie (man denke an die fünf Axiome von Peano), nicht zu unterscheiden.

Es gibt hier allerdings einen kleinen Unterschied: Während die Mathematik beschreibend ist, nutzt das CAS sein Wissen um aktiv zu handeln. Die Regeln der Computeralgebrasysteme somit aktiv sind, diejenigen der Mathematik passiv.

Für eine Umformung muss ein CAS von den vielen Möglichkeiten auf intelligente Weise die richtigen auswählen und das, so könnte man meinen, kann nur der Mensch. Die ersten CAS entstanden zu der Zeit, als man anfing, die künstliche Intelligenz (AI) zu untersuchen und nicht unbegründet waren Computeralgebrasysteme damals ein Teilgebiet der AI und selbst heute sehen das manche noch so.

[1] Ein mathematisches System ist mächtig genug, wenn sich darin Aussagen über natürliche Zahlen nachbilden lassen.

5.3 Computeralgebrasysteme im Vergleich

Von den vielen Computeralgebrasystemen, die es gibt werde ich später nur noch Derive (das ich mit "Voyage" meine) und Mathematica eingehen. Diese beiden CAS stehen stellvertretend für die die zweite Generationen von CAS, wie sie Gräbe (4) definiert hat:

> 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. (Gräbe, S.23)

Zu dieser Zeit musste man noch mit Lochkarten hantieren, weswegen es verständlich ist, dass die CAS damals noch kaum den Begriff verdient hatten.

> 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. (Gräbe, S.24)

Genau genommen ordnet Gräbe Derive in die zweite Generation ein. Das ist zwar korrekt, aber im Vergleich zu Mathematica (welches auch zur zweiten Generation gehört) sind die Eigenschaften wie das "Zwei-Ebenen-Modell" und insbesondere das "regelbasierte Simplifikationssystem" viel weniger stark ausgeprägt. Man kann es so sehen, dass Derive noch halb zur Generation 1 gehört, während Mathematica schon auf Generation 3 schielt.

> 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. (Gräbe, S.30)

Gemeint sind damit hauptsächlich "OO-Prinzipien", also die Objektorientierte Programmierung, welche heute von fast allen modernen Programmiersprachen unterstützt wird. Gerade in Bezug auf die Mathematik sind solche Paradigmen heikel, wodurch eine Integration von Grund auf geplant gewesen sein muss.

Während Gräbe eher eine Chronik der Computeralgebrasysteme angibt, beschreibt Fateman den Entwicklungsprozess(3). Nach ihm gibt es zwei Typen von Ansätzen: Den Prototyp/Hacker-Ansatz und den mathematisch-hierarchischen Ansatz. Alle gängigen CAS fallen in den Prototyp/Hacker-Ansatz, sie zeichnen sich dadurch aus, dass sie ein bestehendes System haben, das Spezialfall um Spezialfall erweitert wird, bis es "mehr oder weniger allgemein" ist. Der zweite Ansatz (hauptsächlich von Axiom vertreten) versucht, die Mathematik aus mathematischer Sicht zu modellieren.

5.4 Mein Computeralgebrasystem: Reckna

Der praktische Teil meiner Maturarbeit, der mir nicht zuletzt auch die theoretischen Hintergründe geliefert hat, war das Entwickeln eines eigenen Computeralgebrasystems. Zwar wusste ich grob, was ein CAS können sollte. Ich hatte aber nur nur mit dem TI Voyage Erfahrung, dessen Design mir nicht zusagte, nicht mit anderen CAS.

Mein erster Versuch übertraf meine ursprünglichen Erwartungen und brachte ein lauffähiges CAS hervor. Diese Version war meine Maturarbeit.

In den Sommerferien wurden Teile, die mir noch nicht optimal vorkamen, noch einmal überarbeitet.

Ein paar Daten zu meinem Computeralgebrasystem:

ID	Wert
Name	Reckna
Bedeutung	**R**egelbasierter, **e**rweiterbarer **C**omputeralgebra-**K**alkulator, **n**utzbringend für **A**lgebraaufgaben
Version	0.2
Kernfunktion	Symbolisches Rechnen
Features	Eigene Sprache, Lokalisierbar, FastCGI-fähig
Schnittstelle	Web, GUI, Terminal
Programmierdauer	146 Tage (12.9.11–22.01.12 und 30.5.12–19.6.12)
Grösse	8651 Zeilen (davon 6690 effektiver Code)
Sprachen	97% C++, 2% Makefile und 1% Perl
Dateien	91 (33 C++-Header, 26 C++-Code, 2 Tools)
Bibliotheken	C++-Standardbibliothek, GMPlib (für beliebig grosse Zahlen)
Plattformen	Theoretisch überall, getestet unter Linux 32 Bit und 64 Bit

Eine Web-Demonstration ist unter folgendem Link zu finden:

`http://www.gymmuenchenstein.ch/johannes/ma/`

Zum gegenwärtigen Zeitpunkt (26. August) ist dort noch die alte Version zu finden, aber ich plane die Webseite demnächst zu aktualisieren.

Die Mini-Programme, die zu sehen werden sind, werden in einer Datei abgespeichert und vom Mutterprogramm eingelesen. Dieser Prozess gehört zur Bedienung und wird bei den entsprechenden Beispielen weggelassen.

Ab Abschnitt 6.2 ist implizit immer von Reckna die Rede. Jedoch nicht von den Zeilen in C++, sondern von der Idee dahinter. Implementierungsdetails sind zwar interessant, aber viel wichtiger finde ich das Design des Computeralgebrasystems.

Hier bin ich ähnlicher Meinung wie Fateman, nämlich, dass der Aufbau eines CAS nach mathematischen Überlegungen erfolgen sollte und nicht danach, was einfach oder effizient umzusetzen ist.

6 Umformungsregeln

6.1 Typographische Manipulation

Das zentrale Thema dieser Maturarbeit sind Regeln. Eine Regel besagt zum Beispiel, dass ein Ausdruck der Form $x + 1$ in y umgeformt werden kann.

Als Einführung habe ich das pg-System gewählt, weil sich damit losgelöst von Computeralgebrasystemen einige grundlegende Eigenschaften der Regeln erklären lassen. Es stammt von Hofstadter:

> "DEFINITION: "x`p-g`x`-`" ist ein Axiom, wenn x nur aus Bindestrichen besteht.
>
> Beachten Sie, daß "x" beide Male für die gleiche Kette von Bindestrichen stehen muss. Z. B. ist "`--p-g---`" ein Axiom. Der Ausdruck "`xp-gx-`" ist natürlich kein Axiom (weil "x" nicht zum pg-System gehört); es ist eher eine Form, in die alle Axiome gegossen werden, und heisst Axiomen-Schema.
>
> Das pg-System besitzt nur eine einzige Erzeugungsregel:
>
> REGEL: Angenommen x, y und z stehen alle für einzelne Ketten, die nur Bindestriche enthalten, und angenommen, dass man weiss, daß "x`p`y`g`z" ein SATZ ist. Dann ist "x`p`y`-g`z`-`" ein Satz.
>
> Nehmen wir z. B. x als "`--`", y als "`---`", und z als "`-`". Die Regel sagt uns: Wenn "`--p---g-`" sich als SATZ erweist, dann auch "`-p----g--`".
>
> [...] Eine äusserst nützliche Übung ist die, ein Entscheidungsverfahren für die SÄTZE des pg-Systems zu finden. Schwierig ist das nicht, wenn Sie eine Zeitlang damit gespielt haben, kommen sie wahrscheinlich darauf. Versuchen Sie es." (Hofstadter (5))

Die erste Feststellung ist, dass alle Sätze eine gemeinsame Form besitzen; drei Gruppen von Bindestrichen getrennt durch, als erstes, ein `p` und danach ein `g`. Diese Eigenschaft gilt per Definition für jedes Axiom. Die Erzeugungsregel gibt sie insofern weiter, als dass jeder Eingabesatz, der sie erfüllt, einen neuen liefert, der sie ebenfalls erfüllt. □

Die Lösung zum Problem ist, die Bindestrichlängen zu bestimmen. Man findet heraus, dass sich die erste und die zweite immer zur dritten addiert.

Genauer: Wenn man "`-`" als 1, "`--`" als 2, und n mal "`-`" als n interpretiert, ausserdem `p` als "+" und `g` als "=", dann wird die Definition zu $x + 1 = (x + 1)$ und die Erzeugungsregel zu $x + y = z \Rightarrow x + (y + 1) = (z + 1)$.

Der Beweis lautet nun wie folgt: Jedes Axiom genügt der Addition, da $x+1 = (x+1)$ für alle $x \in \mathbb{N}$ mit ihr übereinstimmt. Das Argument mit der Erzeugungsregel läuft analog zum oberen ab: Die Erzeugungsregel erzeugt aus einem Satz, bei dem die Additivität gilt, einen neuen, bei dem sie ebenfalls gilt und da sie für jedes Axiom gilt, gilt sie auch für alle abgeleiteten. □

Die Sache hat einen Haken. Hofstadter hat zwar gezeigt, dass alle Sätze im pg-System Additivität besitzen, aber nicht, dass sich alle Gleichungen der Form $a+b = c,\ a, b, c \in \mathbb{N}$ im pg-System darstellen lassen. Ich möchte dies nun nachholen.

Wenn man sich ein paar Sätze des pg-Systems aufschreibt, kommt man immer von kleineren zu grösseren. Um zu testen, ob eine Kette ein Satz ist, muss man alle Sätze mit einer kleineren Länge aufschreiben und überprüfen, ob der gesuchte dabei ist.

Einfacher ist es, wenn man sich die Sache rückwärts ansieht. Ein jeder Satz ist entweder ein Axiom, oder er geht dank Erzeugungsregel auf einen kürzeren zurück. Man kann also sagen, (a) "x`p-g`x`-`" ist ein Satz und (b) "x`p`y`-g`z`-`" ist ein Satz, wenn auch "x`p`y`g`z" einer ist.

Jede Kette lässt sich somit auf eine mit nur einem Bindestrich zwischen `p` und `g` reduzieren, für die man dann den Test auf ein Axiom ausführt.

Das Schöne an dieser Lösung ist die ausschliessliche Verwendung typographischer Manipulationen, d.h. reiner Textersetzung. Dafür ist keinerlei Intelligenz vonnöten, wodurch sie einfach automatisierbar ist.

Zudem lässt sich nun die Frage nach der Äquivalenz zwischen Addition und pg-System beantworten: Damit $a + b = c$ ein Satz ist, muss entweder (a) $b = 0 \wedge c = a$ sein, oder (b) $a + (b - 1) = (c - 1)$. Bei jeder Addition kommt man hierbei zu einem Ende, die Äquivalenz ist somit erwiesen. □

6.2 Anwendung: Ableitungen

Regeln sind nicht nur theoretisch interessant, mit ihnen lassen sich auf natürliche Art und Weise mathematische Probleme lösen. In diesem Abschnitt wird das am Beispiel von Ableitungen gezeigt.

6.2.1 Intuitives Vorgehen

Um abzuleiten wendet man die "Ableitungsregeln" an. Diese kann man in der Formelsammlung nachschlagen.

Eine davon ist die Summenregel. Diese besagt $u + v$ wird abgeleitet zu $u' + v'$ (wobei x' für die Ableitung von x steht). Die Beziehung ist einseitig. Wenn es ums Ableiten geht, wird $u' + v'$ bei $u' + v'$ belassen und nicht nach $(u + v)'$ zurück umgeformt. Diese "Einbahnstrasse" kann man mit einem Pfeil ausdrücken.

$$(u + v)' \to u' + v'$$

Ebenso lassen sich Produkt- und Quotientenregel notieren:

$$(u \cdot v)' \to u' \cdot v + u \cdot v'$$

$$\left(\frac{u}{v}\right)' \to \frac{u' \cdot v' - u \cdot v'}{v^2}$$

Bei der Schreibweise x' als Ableitung von x ergibt sich das Problem, dass man nicht weiss, wonach abgeleitet wird. Dies macht es insbesondere schwierig zu entscheiden, ob eine Variable nach 1 oder nach 0 abgeleitet werden soll. Deshalb ist es besser, die obigen Regeln wie folgt umzuschreiben:

$$\frac{\mathrm{d}}{\mathrm{d}x}(u+v) \to \frac{\mathrm{d}}{\mathrm{d}x}u + \frac{\mathrm{d}}{\mathrm{d}x}v$$
$$\frac{\mathrm{d}}{\mathrm{d}x}(u \cdot v) \to \left(\tfrac{\mathrm{d}}{\mathrm{d}x}u\right) \cdot v + u \cdot \frac{\mathrm{d}}{\mathrm{d}x}(v)$$
$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{u}{v} \to \frac{\left(\tfrac{\mathrm{d}}{\mathrm{d}x}u\right) \cdot v - u \cdot \left(\tfrac{\mathrm{d}}{\mathrm{d}x}v\right)}{v^2}$$

Nun ist es möglich auszudrücken, dass eine Variable nach sich selbst abgeleitet wird:

$$\frac{\mathrm{d}}{\mathrm{d}x}x \to 1$$

Die letzte verbleibende Regel ist die Ableitung einer Konstanten c. Will man sich nicht auf die Konvention verlassen, dass c konstant ist, muss man dies explizit ausdrücken:

$$\frac{\mathrm{d}}{\mathrm{d}x}c \to 0 \text{ genau dann wenn } c \text{ frei von } x$$

Hier wird davon Gebrauch gemacht, dass c genau dann konstant bezüglich x ist, wenn im Ausdruck c die Variable x nicht vorkommt.

6.2.2 Formulierung in Regeln

Da sich $\frac{\mathrm{d}}{\mathrm{d}x}f$ nicht so einfach eingeben lässt, wird die Ableitung hier als $\mathrm{deriv}(f, x)$ notiert. $\mathrm{deriv}(f, x)$ ergibt die Ableitung (engl. "derivative") von f nach x. So entspricht beispielsweise $\mathrm{deriv}(2x^2, x)$ der Ableitung von $2x^2$ nach x, also $4 \cdot x$.

Sie soll in Reckna wie folgt benutzbar sein (Erklärung folgt):

```
In[1]> deriv(1, x)
Out[1]> 0
In[2]> deriv(y, x)
Out[2]> 0
In[3]> deriv(x, x)
Out[3]> 1
In[4]> deriv(x+2, x)
Out[4]> 1
In[5]> deriv(2*x, x)
Out[5]> 2
In[6]> deriv(2*x^2-x, x)
Out[6]> 2·x - 1
```

Da dies das erste Mal ist, in dem die Ausgabe von Reckna zu sehen ist, hier eine Erklärung der Zeilen: Wenn ein `In[`x`]>` steht, soll man seinen Eingabeterm eingeben. Hier war das `deriv(1, x)`, gefolgt von einem Zeilenumbruch (Enter-Taste). Es erscheint, nach kurzer Berechnungszeit `Out[1]> 0` sowie `In[2]>`, woraufhin eine erneute Eingabe erwartet wird.

In Reckna kann man die Regel $\frac{\mathrm{d}}{\mathrm{d}x}(u+v) \to \frac{\mathrm{d}}{\mathrm{d}x}u + \frac{\mathrm{d}}{\mathrm{d}x}v$ mehr oder weniger so eingeben, wie sie steht. Als erstes muss $\frac{\mathrm{d}}{\mathrm{d}x}$ mit der Funktion deriv ersetzt werden: $\mathrm{deriv}(u+v,x) \to \mathrm{deriv}(u,x) + \mathrm{deriv}(v,x)$. Der Pfeil $\to$ wird mit "`=>`" notiert:

```
deriv(u_ + v_, x_) => deriv(u_, x_) + deriv(v_, x_)
```

Regeln haben die Syntax <***Muster***> `=>` <***Ersatz***>. Jeder Ausdruck, der auf <***Muster***> (engl. "*pattern*") passt (engl. "*to match*"), wird mit <***Ersatz***> (engl. "*replacement*") ersetzt. Innerhalb der Regeln können *Platzhalter* (engl. "*placeholder*") für einen beliebigen Ausdruck stehen. Platzhalter sind Variablen, die mit einem Unterstrich ("_") enden. So stehen "`pi`" oder "`deriv()`" ausschliesslich für `pi` und `deriv()`, während "`x_`" für jeden beliebigen Ausdruck steht.

Gegeben sei der Ausdruck $\mathrm{deriv}(2+(y+3),y)$. Dieser passt auf die obige Regel mit $u_- = 2$, $v_- = y+2$ und $x_- = y$. Der Ersatz lautet in diesem Fall $\mathrm{deriv}(2,y)+\mathrm{deriv}(y+3,y)$, wenn man die Platzhalter mit ihren Werten ersetzt. $\mathrm{deriv}(y+3,y)$ passt wieder auf die Regel mit $u_- = y$, $v_- = 3$ und $x_- = y$, der Term wird also mit $\mathrm{deriv}(y,y) + \mathrm{deriv}(3,y)$ ersetzt. Der vereinfachte Ausdruck lautet somit $\mathrm{deriv}(2,y)+\mathrm{deriv}(y,y)+\mathrm{deriv}(3,y)$. Um das vollständig zu vereinfachen sind noch weitere Regeln erforderlich.

Aber erst einmal zu den anderen Operationen, diese können in der Form niedergeschrieben werden, wie sie in der Formelsammlung stehen:

```
deriv(a_-b_, x_) => deriv(a_, x_) - deriv(b_, x_)
deriv(a_*b_, x_) => b_*deriv(a_, x_) + a_*deriv(b_, x_)
deriv(a_/b_, x_) => (b_*deriv(a_, x_) - a_*deriv(b_, x_))/b_^2
```

Es fehlen noch folgende Regeln aus dem letzten Abschnitt:

$$\frac{\mathrm{d}}{\mathrm{d}x}x \to 1$$

$$\frac{\mathrm{d}}{\mathrm{d}x}c \to 0 \text{ genau dann wenn } c \text{ frei von } x$$

Die erste kann wieder 1:1 hingeschrieben werden:

```
deriv(x_, x_) => 1
```

Man beachte, dass hier x_- zweimal auftritt. Dies hat genau den erwarteten Effekt, dass beide Ausdrücke gleich sein müssen. Das bedeutet, dass die Regel nur dann passt, wenn der Wert für das erste `x_` demjenigen des zweiten `x_` entspricht. $\mathrm{deriv}(x,x)$, $\mathrm{deriv}(a,a)$ und $\mathrm{deriv}(2,2)$ werden also auf 1 umgewandelt.

Die zweite Regel unterscheidet sich von den bisherigen dadurch, dass sie eine Bedingung enthält. Dies kann man in Reckna wie folgt ausdrücken:

```
deriv(c_, x_) => 1 when freeOf(c_, x_)
```

Das "`when freeOf(c_, x_)`" bedeutet, dass die Regel nur genommen wird, wenn der Platzhalter `c_` frei von der Variablen in `x_` ist. Beispielsweise ist 1 frei von x, genauso wie y, und $e^{i\cdot\pi}$ ist frei von x, aber x, $f(x)$ und $f(mod(4,x))$ sind nicht frei von x. Auf die genaue Funktionsweise von `freeOf()` und auf das `when` wird später in Abschnitt 8.3 eingegangen.

Mit diesem freeOf() kann nun auch die Potenzregel ausgedrückt werden:

```
deriv(a_^n_, x_) => n_*x_^(n_-1)*deriv(a_, x_) when freeOf(n_, x_)
```

Diese Menge von 7 Regeln ermöglicht das Ableiten einfacher Ausdrücke. Eine Menge von Regeln heisst *Regelsatz* (engl. "ruleset"). Genau genommen ist es keine Menge, da die Regeln eine nummeriert sind, Regel 1, Regel 2, usw. (also eine Reihenfolge haben). Regeln mit kleineren Nummern haben Priorität vor den anderen. Wie ein Regelsatz angewandt wird, ist in Abbildung 6.1 zu sehen.

Mit dem Regelsatz, der in diesem Abschnitt entwickelt wurde, können schon einfache Ausdrücke abgeleitet werden. In Abschnitt 8.4 werden die Regeln noch etwas verfeinert und ergänzt. Als kleinen Vorgeschmack zeige ich die Erweiterung auf die wichtigsten trigonometrische Funktionen:

```
deriv(sin(a_), x_) =>  cos(a_)*deriv(a_, x_)
deriv(cos(a_), x_) => -sin(a_)*deriv(a_, x_)
deriv(tan(a_), x_) => deriv(a_, x_)/cos(a_)^2
```

Diese überschaubare Anzahl an Regeln reicht aus, um schon komplexere Ausdrücke wie $\text{deriv}(\sin(x^2) * \cos(2x) + x * (x+1), x)$ nach $\cos(2\cdot x)\cdot\cos(x^2)\cdot 2\cdot x + \sin(x^2)\cdot -1\cdot \sin(2\cdot x)\cdot 2 + x + 1 + x$ umzuformen.

6.3 Anwendung: Zahlzeichen

Die grundlegende Eigenschaft natürlicher[1] Zahlen ist, dass sie je genau einen eindeutigen Nachfolger besitzen. Bis auf die Null haben sie auch alle einen Vorgänger.

Peano, ein italienischer Mathematiker, stellte 1889 die nach ihm benannten Peano-Axiome auf, welche die natürlichen Zahlen folgendermassen beschreiben (Quelle: (9)):

1. 0 ist eine natürliche Zahl.
2. Jede natürliche Zahl n hat eine natürliche Zahl n' als Nachfolger.
3. 0 ist kein Nachfolger einer natürlichen Zahl.
4. Natürliche Zahlen mit gleichem Nachfolger sind gleich.

Mit Hilfe dieser Axiome kann die Addition und die Multiplikation streng formal definiert werden (Quelle: (9)):

[1] In diesem Abschnitt ist es zweckmässig, auch die Null als natürliche Zahl anzusehen. d.h. mit "natürlich" ist $\in \mathbb{N}_0$ gemeint.

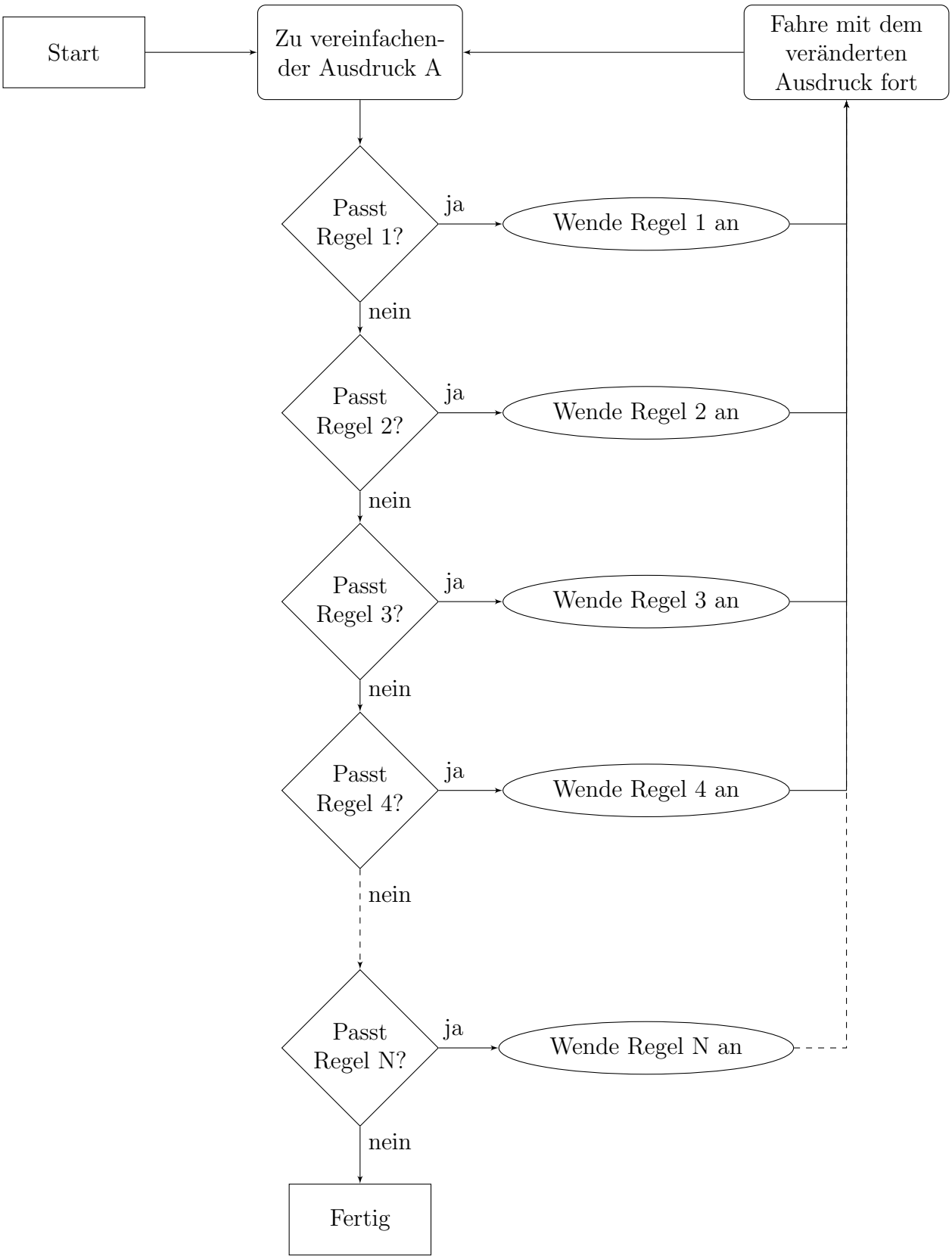


Abbildung 6.1: Anwendung eines Regelsatzes mit N Regeln an einem Ausdruck A.

$$n + 0 := n \tag{6.1}$$
$$n + m' := (n + m)' \tag{6.2}$$
$$n \cdot 0 := 0 \tag{6.3}$$
$$n \cdot m' := (n \cdot m) + n \tag{6.4}$$

Aus technischen Gründen sei im folgenden nicht n' der Nachfolger von n, sondern $\mathrm{s}(n)$ (s wie "Sukzessor").

Das reicht aus, jede natürliche Zahl zu repräsentieren. 1 wäre der Nachfolger von 0, also s(0), 2 entspräche dem Nachfolger von 1, s(s(0)), 3 ist s(s(s(0))) und 10 ist $\underbrace{\mathrm{s(s(s(s(s(s(s(s(s(s(}}_{10mal}\mathrm{0))))))))))$.

Es fällt die Ähnlichkeit zum pg-System auf, nur dass anstelle von Bindestrichen Funktionsaufrufe genutzt werden. Vielleicht ist diese Darstellung von Zahlen im Einersystem die grundlegendste überhaupt. Man denke an die ersten Zeugnisse von Zahlen, die als Anzahl Kerben in Knochen notiert waren.

Zahlen repräsentieren zu können mag vielleicht noch nicht allzu spektakulär sein, aber man kann mit ihnen auch rechnen. Und hier kommen die Regeln ins Spiel:

Die Formeln 6.1-6.2 sehen in der Syntax von Reckna folgendermassen aus:

```
a_ + 0     => a_
a_ + s(b_) => s(a_ + b_)
```

Und das genügt:

```
In[1]> 0 + 0
Out[1]> 0
In[2]> s(0) + s(0)
Out[2]> s(s(0))
In[3]> s(s(0)) + s(s(s(0)))
Out[3]> s(s(s(s(s(0)))))
```

Die Umformung für `In[3]` sieht so aus: $\mathrm{s(s(0)) + s(s(s(0)))} \Rightarrow \mathrm{s(s(s(0)) + s(s(0)))} \Rightarrow \mathrm{s(s(s(s(0)) + s(0)))} \Rightarrow \mathrm{s(s(s(s(0)) + 0)))} \Rightarrow \mathrm{s(s(s(s(s(0)))))}$

Wie erwartet funktioniert dies auch für die Multiplikation:

```
a_*0     => 0
a_*s(b_) => s((a_*b_) + a_)
```

Und auch das genügt:

```
In[4]> s(0) * 0
Out[4]> 0
In[5]> s(s(0)) * s(s(s(0)))
Out[5]> s(s(s(s(s(s(0))))))
In[3]> s(s(s(0)))*s(s(s(0)))+s(0)
Out[6]> s(s(s(s(s(s(s(s(s(s(0)))))))))
```

Es ist erstaunlich, dass sich so etwas fundamentales wie Addition und Multiplikation mit nur 4 Regeln ausdrücken lassen. Sonst wird nichts vorausgesetzt.

Die hier gezeigte Darstellung ist nicht optimal. Andere Ansätze (unser Dezimalsystem zum Beispiel), erlauben es, bei geringerem Speicherverbrauch schneller zu Rechnen.

6.4 Terminierung

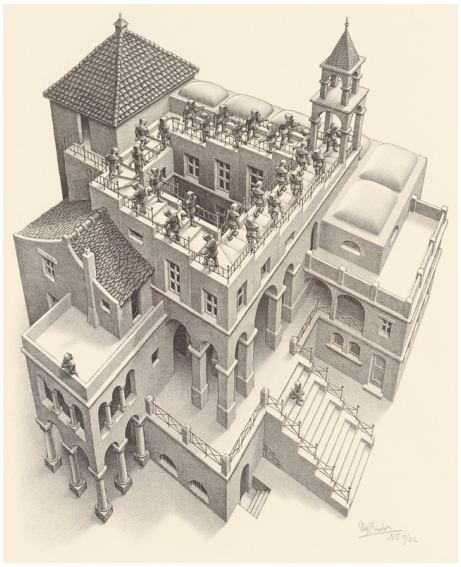


Abbildung 6.2: Treppauf, treppab, von M. C. Escher (Lithographie, 1960, ©1988 M.C.Escher)

Es scheint, als hätten wir mit Regeln einen Weg gefunden, mathematische Eigenschaften für den Computer verständlich zu beschreiben. Folgende Regel gibt die Kommutativität der Addition wieder:

1: `a_ + b_` ⇒ `b_ + a_`

Das führt allerdings zu einer Endlosschleife, da $x+1$ in $1+x$ und $1+x$ wieder zurück in $x+1$ umgewandelt wird: $x+1 \Rightarrow 1+x \Rightarrow x+1 \Rightarrow 1+x \Rightarrow x+1 \Rightarrow 1+x \Rightarrow \ldots$ Man sagt, der Regelsatz *terminiert* nicht.

So etwas will man möglichst vermeiden. In der Tat ist mir das selber mehrmals passiert, als ich die Regeln für mein *Computeralgebrasystem* entwarf und es lässt sich schwer vermeiden. Denn selten ist nur eine Regel schuld, oft sind es viele Regeln, die den Term nacheinander umformen, bis er wieder gleich wie am Anfang ist.

Von einem ähnlichen Typ ist folgende Regel:

1: `a_` ⇒ `a_/2 + a_/2`

Hier wird der Term immer länger und länger, ohne dass es zu einem Ende kommt. Das Prinzip ist mit Abbildung 6.2 vergleichbar, bei dem die Mönche immer höher gelangen. Kaum haben sie eine Umdrehung geschafft, befinden sie sich wieder in ihrer Ausgangslage um erneut eine Umdrehung zu laufen.

Rufen wir uns noch einmal die Ableitungsregeln zur Addition ins Gedächtnis. Handelt es sich auch hier um ein Aufsteigen (oder Absteigen) ohne Ende?

```
deriv(x_, x_) => 1
deriv(c_, x_) => 0 when freeOf(c_, x_)
deriv(a_+b_, x_) => deriv(a_, x_) + deriv(b_, x_)
deriv(a_-b_, x_) => deriv(a_, x_) - deriv(b_, x_)
deriv(a_*b_, x_) => b_*deriv(a_, x_) + a_*deriv(b_, x_)
deriv(a_/b_, x_) => (b_*deriv(a_, x_) - a_*deriv(b_, x_))/b_^2
deriv(a_^n_, x_) => n_*a_^(n_-1)*deriv(a_, x_) when freeOf(n_, x_)
```

Die Mönche, die abwärts steigen, sind im Glauben, dass sie irgendwann einmal am Boden ankommen werden. Wie kann man ihnen zeigen, dass sie niemals ankommen? Es reicht nicht aus, eine Stufe zu markieren und zu warten bis ein Mönch zweimal auf ihr steht, denn während er auf der anderen Seite war könnte sie durch Gotteshand nach unten versetzt worden sein. (Das gleiche gilt bei den Regeln, nur weil eine zweimal oder sogar hundertmal angewandt wurde heisst das noch nicht, dass man sich in einer Endlosschleife befindet.) Es reicht aber aus, eine Stufe zu markieren und die Höhe über dem Boden zu messen. Kommt der Mönch wieder bei ihr vorbei und ist immer noch in gleicher Höhe, kommt er seinem Ziel nie näher.

Wenn man formal beweisen will, dass in den Regeln nie eine Endlosschleife existieren kann, läuft es darauf hinaus, eine Bewertungsfunktion $\varphi\colon \text{Term} \to \mathbb{N}_0$ zu finden, die für den Term t die "Höhe über dem Boden misst", bzw. ihm eine Zahl für seine Komplexität zuweist. Wenn für jede Regel $a \Rightarrow b$ gilt $\varphi(a) > \varphi(b)$, dann gibt es keine Endlosschleife, der Satz an Regeln *terminiert* (weil es eine Mindestkomplexität, den Boden, geben muss).

Satz: Ein Regelsatz terminiert, wenn eine Abbildung φ von Ausdrücken nach $\mathbb{N}$ existiert, und die Werte bei jeder Regelanwendung streng monoton fallen.

Beweis: Angenommen, ausgehend von Term t_1 bilden die Regeln eine unendliche Folge von Terme $t_1, t_2, t_3, \ldots$. Dann müssen nach Voraussetzung auch $\varphi(t_1) > \varphi(t_2) > \varphi(t_3) > \cdots > 0$ eine unendliche Folge bilden. Zwischen $\varphi(t_1)$ und 0 existieren aber nur endlich viele natürliche Zahlen, Widerspruch. □

Man sieht, dass in den ersten beiden Zeilen die Funktion deriv() ganz verschwindet und in den restlichen immer der Operator aus dem deriv herausgeholt wird. Wenn man also die Summe der Anzahl Zeichen des ersten Arguments von deriv zählt, wird diese bei jeder Umformung strikt kleiner. deriv($\mathbf{x}+\mathbf{x}, x$) hat 3 Zeichen als erstes Argument, deriv($\mathbf{x}, x$) + deriv($\mathbf{x}, x$) hat zwei mal je 1 Zeichen, das macht 2 Zeichen insgesamt. deriv($\mathbf{x}, x$) hat 1 Zeichen, 1 hat 0 Zeichen als erstes Argument.

$$\varphi(x) = \text{Anzahl Zeichen aus } x\text{, die in einem ersten Argument von deriv() sind}$$

Ein kurzer Vergleich zeigt, dass $\varphi(x)$ bei jeder der sieben Regeln kleiner wird. □

Dieses Beweismuster ist, was Regeln betrifft, das geeignetste. Ein weiteres Beispiel dafür ist die Addition unserer Zahlzeichen:

```
a_ + 0      => a_
a_ + s(b_) => s(a_ + b_)
```

Hier ist genau das Gegenteil der Fall, der Ausdruck innerhalb der Funktion s() wird entweder grösser oder bleibt gleich. Man kann allerdings beobachten, dass das s() immer weiter nach links rutscht.

Sei die Eingabe eine Folge von Zeichen $c_1c_2c_3\ldots c_n$ (bei "`s(0)+s(s(0))`" wäre z.B. $c_1 =$ `s`, $c_2 =$ `(` und $c_{12} =$ `)`). Die Funktion φ ist nun folgendermassen definiert:

$$\varphi(c_1c_2c_3\ldots c_n) = \sum_{i=1}^{n} \begin{cases} i & c_i = \texttt{s} \\ 1 & c_i \neq \texttt{s} \end{cases}$$

Beim gezeigten Algorithmus nimmt diese Funktion in beiden Fällen ab, sie ist streng monoton fallend. Es ist also gezeigt, dass diese terminieren. □

Schwierig wird es, wenn verkürzende und verlängernde Regeln mitspielen. Zu den oberen zwei Regeln füge man folgende hinzu:

```
sum(0) => 0
sum(s(a_)) => s(a_) + sum(a_)
```

`sum(s(s(s(0))))` berechnet $0+1+2+3=6$, also s(s(s(s(s(0))))). Vom Gefühl her ist es offensichtlich, dass es immer terminiert, da die Argumente immer kleiner werden bis die Funktion sum gar nicht mehr vorkommt. Danach werden die Additionsregeln aufgerufen, die immer noch terminieren.

Oder kurz: sum für sich terminiert, die Addition für sich terminiert, also terminieren auch beide zusammen. Diese kurze Formulierung ist allerdings falsch.

```
a_ + s(b_) => s(a_ + b_)
s(a_ + b_) => a_ + s(b_)
```

Beide Regel terminieren, für sich alleine gesehen. Die zweite macht aber die erste wieder rückgängig, so dass beide Regeln abwechslungsweise angewandt werden ohne dass es zu einem Ende kommt.

Für ein Beweis müssen also immer alle Regeln berücksichtigt werden, was dazu führt, dass die Funktion $\varphi(t)$ sehr umständlich werden kann. Ausserdem kann es sehr schwierig werden, sie zu finden, sogar unlösbar schwierig. Einen tieferen Einblick über die Vorgehensweise bei Regeln bietet das Buch "Term Rewriting and All That" (2), an das auch die Beweistechnik angelehnt ist.

In der Praxis habe ich nicht bewiesen, dass der gesamte Regelsatz von Reckna korrekt ist, sondern einige representative Eingaben definiert und getestet, ob sie in eine Endlosschleife führen.

Dennoch sind die Überlegungen nicht nutzlos. Der Ansatz, ein Problem auf kleinere (weniger komplexe) Probleminstanzen zurückzuführen, ist für das Auffinden der Regeln wesentlich. Das Prinzip hat in der Informatik einen eigenen Namen: "Divide and Conquer".

6.5 Zusammenfassung

- Regeln geben vor, wie ein gewisser Ausdruck in einen anderen umgewandelt werden kann.
- In Reckna kann eine Regel zum Beispiel so aussehen: "`a_ + a_ => 2*a_`". "`a_+a_`" nennt man das *Muster*, "`2*a_`" den *Ersatz*, und "`a_`" den *Platzhalter*.
- Eine Regel *passt* auf einen Ausdruck A, wenn man die Platzhalter so wählen kann, dass das Muster identisch mit A ist.
- Passt eine Regel, so ersetzt man A durch A', wobei A' dem *Ersatz* der Regel entspricht, bei dem die Platzhalter durch ihre Werte ersetzt werden.
- Eine geordnete Menge von Regeln heisst *Regelsatz*.

- Regeln können mehrmals hintereinander angewendet werden.
- Ableitungen lassen sich sehr kurz mit Hilfe von Regeln beschreiben.
- Die Terminiertheit eines Regelsatzes lässt sich mit Hilfe einer Bewertungsfunktion φ nachweisen.
- Es ist manchmal schwierig herauszufinden, ob es Fälle gibt, an denen ein Regelsatz unendlich lange arbeitet.
- Das Grundprinzip beim Erstellen von Regeln ist, ein Problem in leichtere Teilprobleme zu zerlegen.

7 Syntax

Im letzten Kapitel wurde versucht, einen intuitiven Zugang zu den Regeln und Platzhaltern zu schaffen. Das Einführungsbeispiel handelte von typographischer Manipulation, später habe ich nicht weiter festgelegt, was ich mit "passen" meine.

So schön und einfach das textuelle Passen auf den ersten Blick auch sein mag, bei genauerem Hinsehen ist es nicht das gewollte.

Ausdruck	Muster	Textuell	Symbolisch
$\sin(\pi)$	`sin(x_)`	`x_` = "π"	`x_` $= \pi$
$1 - a^2$	`a_-b_`	`a_` = "1", `b_` = "a^2"	`a_` $= 1$, `b_` $= a^2$
$(1 - a)^2$	`a_-b_`	`a_` = "(1", `b_` = "a^2)"	—
sqrt$(1, 2, 3)$	`sqrt(x_, e_)`	`x_` = "$1, 2$"; `e_` = "3" `x_` = "1"; `e_` = "$2, 3$"	—
$2 \cdot x + x^2$	`a_·b_`	`a_` = "2", `b_` = "$x + x^2$"	—
$1 + 1$	`a_+␣␣␣b_`	—	`a_` $= 1$

(Das "␣" steht für ein Leerzeichen)

Das Passen nach rein textuellen Aspekten hat einige Fehler, zum Beispiel weiss es nichts über die Punkt-vor-Strich-Regel, weiss nichts von der Bedeutungen des Kommas und hat Probleme mit Leerzeichen. Hinzu kommt, dass die Werte der Platzhalter nicht immer eindeutig definiert sind. Einzelne Punkte können behoben werden, in dem man sie als Spezialfälle separat behandelt [1], aber das würde nicht nur einen riesigen Aufwand bedeuten, es würde den Ansatz für etwas missbrauchen, für das er nicht gedacht war. Wir wollen einen geeigneteren Ansatz suchen – und den gibt es.

7.1 Mathematische Ausdrücke in Reckna

Die mathematische Notation hat sich entwickelt bevor es Schreibmaschinen gab und ist, richtigerweise, nicht auf die Eingabe in den Rechner optimiert. Damit ein Ausdruck in ein CAS eingegeben werden kann, z.B. $\sqrt{5}$, muss festgelegt sein, wie man ihn in einer Sequenz von Zeichen darstellt [2].

Der Weg, der üblicherweise genommen wird, ist, jeder Funktion einen Namen zu geben. Für Brüche wird der Durch-Operator "/" genommen, für Potenzen der Operator "^".

[1] Tatsächlich können sogar alle Speziallfälle gelöst werden. Die Programmiersprache Markov basiert ausschliesslich auf Textersetzung und ihr wurde *Turing-Mächtigkeit* nachgewiesen, was bedeutet dass mit ihr alles erreicht werden kann.

[2] Neuerdings ist bei der Eingabe ein Trend zurück zur mathematischen Notation zu beobachten (z.B. von TI-Nspire, dem "Nachfolger" vom TI-Voyage und neueren Mathematica-Versionen). Beide beruhen aber darauf, dass sich diese in einen eindeutigen Einzeiler umformen lässt.

Mathematisch	Computergerecht
$a + b$	`a+b`
$a - b$	`a-b`
$a \cdot b$	`a*b`
$\frac{z}{n}$	`z/n`
b^2	`a^x`
$\sqrt{x}$	`sqrt(x)`
$\binom{n}{k}$	`binom(n, k)`
$\frac{\mathrm{d}}{\mathrm{d}x} f$	`deriv(f, x)`

Leerzeichen werden von den meisten Computerprogrammen ignoriert und dienen nur der Lesbarkeit.

Die generelle Umstellung fällt erfahrungsgemäss nicht schwer.

Bei Ausdrücken wie $2a$ fragt man sich, ob das nun `2*a` heissen muss oder ob auch `2a` erlaubt ist. Aus meiner Sicht ist `2*a` konsistenter und in Version 0.1 musste man das Mal-Zeichen immer explizit angeben.

In der neuen Version 0.2 kann das Zeichen immer weggelassen bzw. durch ein Leerzeichen ersetzt werden. Einzige Ausnahme ist, wenn hinter einer Variable eine öffnende Klammer steht, was immer als Funktionsaufruf interpretiert wird.

Eine Erwähnung wert sind ausserdem noch Kommentare. Alles, was in einer Zeile hinter einem "`#`" kommt, wird ignoriert.

7.2 Syntaxbaum

Es ist üblich, die Operatoren wie + zwischen zwei Ausdrücke zu setzen, also $1 + 1$. Es ist allerdings auch denkbar, sie wie einen Funktionsaufruf voranzustellen, wie in $+(1, 1)$. Bei dem Malzeichen geht das ebenfalls, so kann man $1 + 2 \cdot 3$ als $+(1, \cdot(2, 3))$ schreiben. Der Vorteil ist, dass es im Gegensatz zur Punkt-vor-Strich-Rechnung keine zusätzlichen Klammern benötigt. $(1+2)\cdot 3$ wird als $\cdot(+(1, 2), 3)$ geschrieben und benötigt gleich viele Klammern wie $+(1, \cdot(2, 3))$. Diese Schreibweise macht die Syntax ausserdem einheitlicher, weil das Plus zu einem Funktionsaufruf wird. Der Grund, weshalb ich sie eingeführt habe, ist der, dass sie sich leicht in einen (abstrakten) Syntaxbaum (siehe auch (7)) überführen lassen:

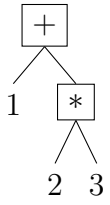


Abbildung 7.1: Syntaxbaum für $1 + 2 \cdot 3$

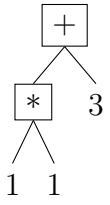


Abbildung 7.2: Syntaxbaum für $(1 + 2) \cdot 3$

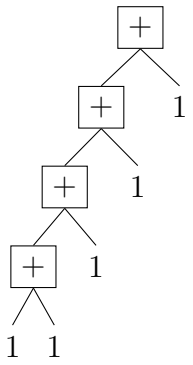


Abbildung 7.3: Baum für $1+1+1+1+1$

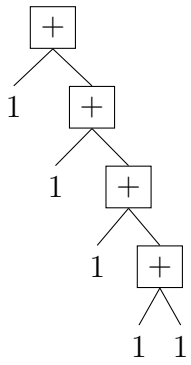


Abbildung 7.4: $1+(1+(1+(1+1)))$

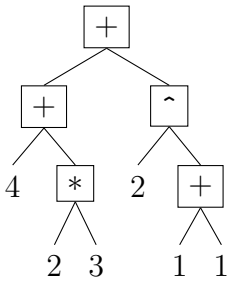


Abbildung 7.5: Baum für $4+2*3+(1+1)^2$

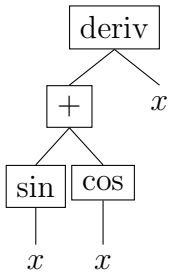


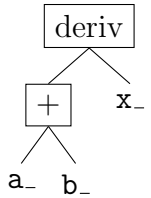
Abbildung 7.6: $\text{deriv}(\sin(x)+\cos(x), x)$

Definition: Zwei Ausdrücke gelten als *identisch*, wenn ihre Syntaxbäume identisch sind.

Das heisst, $(a+b)+c$ und $a\ +\ b\ +\ c$ sind identisch, $a+b+c$ und $a+(b+c)$ sind es nicht (weil die Addition linksassoziativ ist). $1+1$ und 2 sind ebenfalls nicht identisch, obwohl sie den gleichen Wert haben.

7.3 Regeln

Das Muster $\text{deriv}(\texttt{a\_}+\texttt{b\_}, \texttt{x\_})$ hat folgenden Syntaxbaum:



Legt man diesen auf Abbildung 7.6, stimmen die Bäume oben überein. Nur unten, wo die Platzhalter stehen, unterscheiden sie sich. Die Platzhalter nehmen die Werte ein, die auf ihrer Position stehen.

Es ist $\texttt{a\_} = \sin(x)+\cos(x)$, $\texttt{b\_} = x$ und $\texttt{x\_} = x$.

Deshalb passt die folgende Regel auf $\text{deriv}(\sin(x)+\cos(x), x)$:

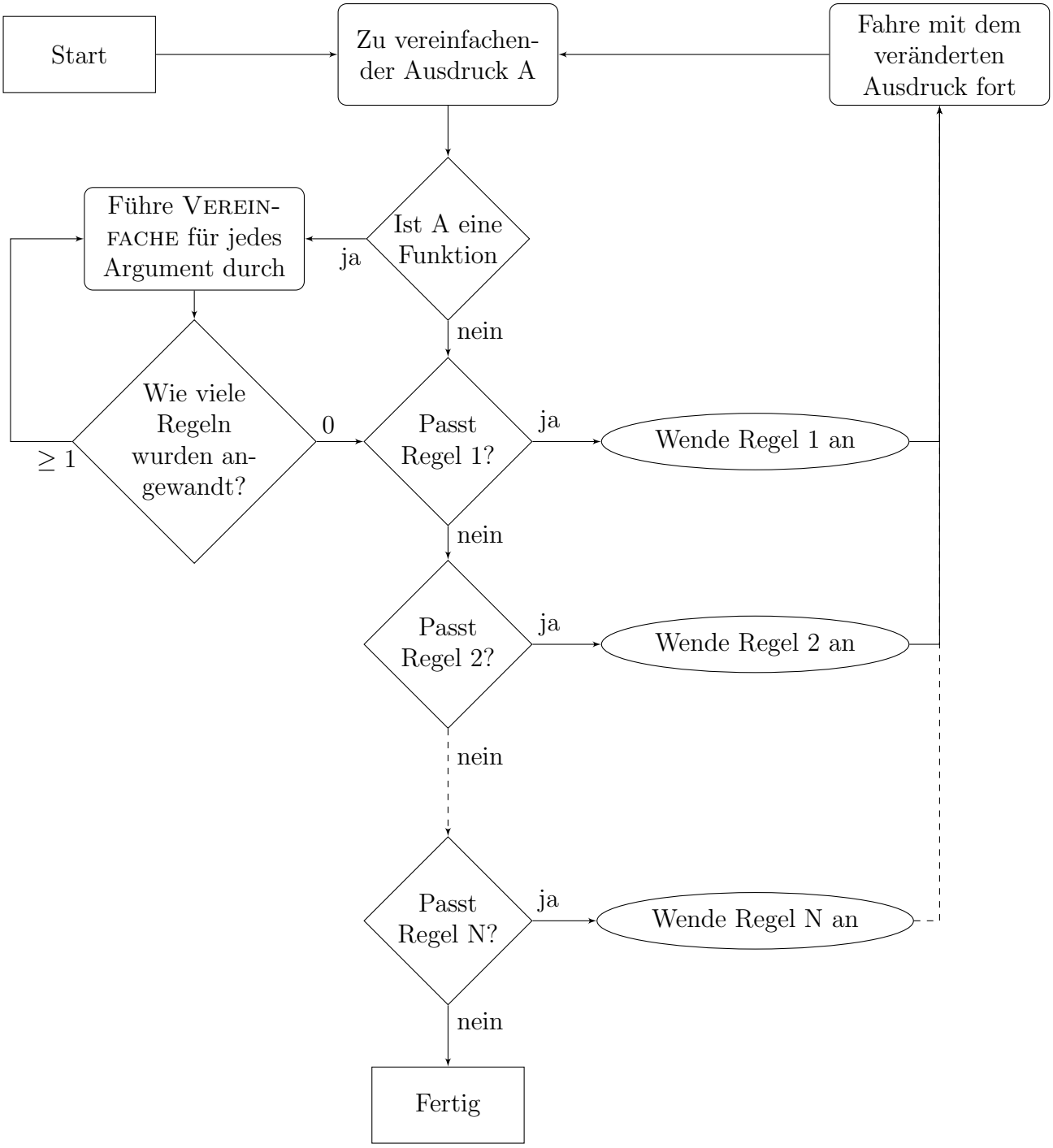


Abbildung 7.7: Rekursive Definition der Routine "Vereinfachen" bei einem Regelsatz von N Regeln an einem Ausdruck A.

```
deriv(a_+b_, x_) => deriv(a_, x_) + deriv(b_, x_)
```

Schreibt man den Syntaxbaum des Ersatzes auf und rückersetzt den Platzhalter mit dem Ausdruck, den er hatte, erhält man den umgewandelten Ausdruck.

7.4 Syntaktischer Zucker

In Programmiersprachen nennt man Syntaxelemente, die nicht zwingend notwendig sind, aber für einen schöneren Quelltext sorgen, syntaktischer Zucker (engl: "syntactic sugar").

Einer davon ist die Infix-Schreibweise von Funktionen. Man schreibt in der Mathematik zum Beispiel nicht $\mathrm{mod}(6, 4) = 2$, sondern $6 \bmod 4 = 2$. Von der Programmiersprache *Haskell* habe ich mir die Infix-Funktionen abgeschaut. Alle Funktionen, die mit "`" umschlossen sind, sind Infix-Funktionen. 6 mod 4 könnte man in Reckna zum Beispiel so schreiben: "`6`mod`4`".

Ein weiteres Syntaxelement ist der Punktoperator. Wer mit objektorientierten Programmiersprachen vertraut ist, bevorzugt vielleicht die Notation `a_.isEven()` vor `isEven(a_)`. Der Vorteil wird dann ersichtlich, wenn mehrere Operationen angewendet werden:

```
caesarOnChar(c_, key_) => c_.letterNumber().shiftMod(key_, 26).toLetter()
```

Ohne Punkt wäre das etwas unübersichtlicher und mit mehr verschachtelten Klammern:

```
caesarOnChar(c_, key_) => toLetter(shiftMod(letterNumber(c_), key_, 26))
```

Es muss gesagt sein, dass diese drei Arten, Funktionen aufzurufen, vollkommen äquivalent sind. Es besteht im Nachhinein keine Möglichkeit mehr, herauszufinden, wie sie eingegeben wurden.

Es gibt also drei Arten, 6 mod 4 zu schreiben:

1. `mod(6, 4)`
2. `6`mod`4`
3. `6.mod(4)`

Hier ist die zweite Variante am passendsten, im Zweifelsfall sollte aber die erste Variante bevorzugt werden.

8 Feinere Techniken

Der Syntaxbaum ist weit verbreitet, aber speziell bei mathematischen Ausdrücken hat er seine Schwierigkeiten. Zwei Repräsentationen von $1+1+1$, nämlich $(1+1)+1$ und $1+(1+1)$ stellen zwei unterschiedliche Bäume dar.

Yacas (1) sagt von sich, es sei ein einfach benutzbares, universelles Computeralgebrasystem. Es wurde von vielen Leuten geschrieben und besteht aus mehr als fünfmal so viel Code wie meines. Meine ersten Versuche mit Regeln machte ich mit ihm und ich habe die grundlegende Philosophie, möglichst alles auf der Ebene der Regeln zu implementieren, von ihm übernommen zu haben. Lange Zeit war ich überzeugt von Yacas, bis ich herausgefunden habe, dass es unfähig ist, einfache Additionen durchzuführen:

```
In> a+a+b
Out> 2*a+b
In> a+b+a
Out> a+b+a
```

Das unterstreicht, wie wichtig es ist, eine gute Repräsentation der Addition zu haben. Folgende Regel zum Beispiel läuft bei einem Baum von $(a+b)+a$ ins Leere:

```
a_ + a_ => 2*a_
```

Plus ist ein binärer Operator, d.h. er nimmt immer zwei Argumente entgegen. Allerdings ist er auch kommutativ, das heisst die Reihenfolge seiner Argumente spielen keine Rolle. Deshalb liegt es nahe, ihn als Funktion plus() anzusehen, die beliebig viele Argumente annehmen kann. $a+b+c$ ist aufgrund der Linksassoziativität $\text{plus}(\text{plus}(a,b),c)$, aber eigentlich ist das dasselbe wie $\text{plus}(a,b,c)$. Letztere Darstellung nennt man *flach* und hat sich als eine zweckmässige Form herausgestellt, die von vielen CAS verwendet wird.

Diese Technik nennt man, einen Ausdruck in Standardform ("canonical form") bringen (2). Die grundsätzliche Idee ist, dass zwei verschiedene Ausdrücke A_1 und A_2 vereinfacht werden nach A_1' bzw. A_2'. Im Idealfall sind A_1' und A_2' genau dann identisch, wenn A_1 und A_2 äquivalent sind.

8.1 Platzhaltertypen

Als ich mein CAS schrieb, habe ich lange Zeit gerätselt, wie ich die Addition, oder allgemein Funktionen beliebiger Anzahl Parameter, unterstützen kann. Ziel war, bei einem Ausdruck wie $\text{plus}(a,0,b,c,0)$ alle 0 zu entfernen.

Damit das möglich war, musste ich zwei neue Platzhaltertypen einführen.

8.1.1 Steh-Platzhalter (static placeholder)

In den letzten Kapitel gab es nur einen Platzhalter, der nur für für ein Argument an einem festen Platz stand. In meinem Programm hat er den Namen "static placeholder" und mein verzweifelter Versuch, diesen einzudeutschen brachte "Steh-Platzhalter" hervor.

8.1.2 Dehn-Platzhalter (greedy placeholder)

Ein weit verbreiteter Ansatz, den ich von Programmiersprachen kenne, ist, einen Parametertyp einzuführen, der mehrere Argumente fassen kann. Diesem Platzhalter habe ich den Namen "greedy placeholder" gegeben. Gierig deshalb, weil er die Werte von allen verbliebenen Argumenten annimmt.

Für den Dehn-Platzhalter gilt folgendes:

1. Variablen, die mit "___" enden, sind Dehn-Platzhalter.
   Beachte: "`x_`" und "`x___`" sind zwei *unterschiedliche* Namen.
2. Er schluckt alle nachfolgenden Argumente. Daher sollte er im Muster immer am Schluss der Parameterliste stehen.
3. Im Ersatz fügt der Dehn-Platzhalter alle seine Argumente in der gleichen Reihenfolge wieder ein[1].
4. Ein Dehn-Platzhalter kann beliebig viele Argumente aufnehmen, aber nicht keine.

Eine Anwendung ist zum Beispiel die folgende Regel:

```
deriv(plus(a_, b___)) => plus(deriv(a_), deriv(plus(b___)))
```

Ein paar Beispiele, was das konkret bedeutet:

Ausdruck vorher	`a_`	`b___`	Ausdruck nachher
$\text{deriv}(\text{plus}(1,1))$	1	1	$\text{plus}(\text{deriv}(1), \text{deriv}(\text{plus}(1)))$
$\text{deriv}(\text{plus}(x,y,z,2))$	x	$y,z,2$	$\text{plus}(\text{deriv}(x), \text{deriv}(\text{plus}(y,z,2)))$
$\text{deriv}(\text{plus}(y,z,2))$	y	$z,2$	$\text{plus}(\text{deriv}(y), \text{deriv}(\text{plus}(z,2)))$
$\text{deriv}(\text{plus}(z,2))$	z	2	$\text{plus}(\text{deriv}(z), \text{deriv}(\text{plus}(2)))$

Es hat sich herausgestellt, dass die Eigenschaft, dass der Dehn-Platzhalter immer mindestens ein Element aufnimmt, manchmal unpraktisch ist. Deswegen wurde in Version 0.2 von Reckna der leichte Dehn-Platzhalter eingeführt; er endet mit "..." und nimmt jede beliebige Anzahl Elemente, auch keine, auf.

8.1.3 Such-Platzhalter (mobile placeholder)

Als ich mir das Design überlegte, wollte ich $\text{plus}(a,0,b,c,0)$ umformen können. Die Lösung auf die ich gekommen bin, habe ich in keinem anderen CAS gefunden. Dabei ist sie einfach und mächtig zugleich: der Such-Platzhalter.

Bevor ich erkläre, wie er funktioniert, hier ein paar Beispiele:

[1] Im Fall, dass den Dehn-Platzhalter keine Funktion umschliesst wird aus den Argumenten eine Menge. Wie Mengen dargestellt werden, sehen wir später.

1: $\text{plus}(\text{plus}(\texttt{a\_\_\_}) \sim, \texttt{b\_\_\_}) \Rightarrow \text{plus}(\texttt{a\_\_\_}, \texttt{b\_\_\_})$
2: $\text{plus}(\texttt{a\_} \sim, \texttt{a\_} \sim, \texttt{b\_\_\_}) \Rightarrow \text{plus}(2 \cdot \texttt{a\_}, \texttt{b\_\_\_})$
3: $\text{plus}(0 \sim, \texttt{a\_\_\_}) \Rightarrow \text{plus}(\texttt{a\_\_\_})$

1. Ein Ausdruck, der mit "$\sim$" endet, ist ein Such-Platzhalter. Da er namenslos ist, wäre vielleicht Such-Modifikator oder Such-Attribut eine bessere Bezeichnung.
2. Während der Dehn-Platzhalter die Eindeutigkeit der Argumentzahl verletzt, verletzt der Such-Platzhalter die Eindeutigkeit des Ortes. Wenn der umschlossene Ausdruck am angegebenen Ort nicht passt, rückt er eins nach rechts und schaut, ob der dort passt. Im schlimmsten Fall probiert er alle Argumente aus.
3. Da der Such-Platzhalter keinen Namen hat, ist er nur im Muster erlaubt (nicht im Ersatz).

Ausdruck vorher	a_/a___	b___	Ausdruck nachher
$\text{plus}(\text{plus}(1,1),2)$	$1,1$	2	$\text{plus}(1,1,2)$
$\text{plus}(1,\text{plus}(2,3),4)$	$2,3$	$1,4$	$\text{plus}(2,3,1,4)$
$\text{plus}(1,\text{plus}(2,2),\text{plus}(3,3),4)$	$2,2$	$1,\text{plus}(3,3),4$	$\text{plus}(2,2,1,\text{plus}(3,3),4)$
$\text{plus}(a,a,1)$	a	1	$\text{plus}(2 \cdot a,1)$
$\text{plus}(1,a,2,a,3)$	a	$1,2,3$	$\text{plus}(2 \cdot a,1,2,3)$
$\text{plus}(1,\text{plus}(1,2),2,\text{plus}(1,2),3)$	$\text{plus}(1,2)$	$1,2,3$	$\text{plus}(2 \cdot \text{plus}(1,2),1,2,3)$
$\text{plus}(0,a,b)$	a,b	–	$\text{plus}(a,b)$
$\text{plus}(a,0,b)$	a,b	–	$\text{plus}(a,b)$
$\text{plus}(a,b,0)$	a,b	–	$\text{plus}(a,b)$

8.1.4 Funktionen-Platzhalter (function placeholder)

Lange Zeit waren diese drei Platzhalter für meine Zwecke ausreichend. Für einen Fall benötigte ich dann doch noch einen weiteren, den Funktionen-Platzhalter. Er entspricht dem Steh-Platzhalter, nur dass er für Funktionsnamen angewendet werden kann.

$\texttt{f\_}(getMeOut) \Rightarrow getMeOut$

Diese Regel vereinfacht $\text{plus}(\text{sin}(\text{sqrt}(getMeOut)))$ zu $\text{plus}(\text{sin}(getMeOut))$ zu $\text{plus}(getMeOut)$ zu $getMeOut$.

Dieser Platzhalter ist zum Beispiel für die Funktion freeOf() notwendig.

8.2 Anwendung: Grundrechenarten

Wir haben jetzt unser Werkzeug zusammen um die "Reckna Mathematics Library", kurz "RML" zu schreiben bzw. verstehen.

Als erstes wandeln wir die binären Operatoren in Funktionen um:

```
a_ + b_ => plus (a_,  b_)
a_ - b_ => plus (a_, -b_)
a_ * b_ => times(a_,  b_)
a_ / b_ => times(a_, b_^-1)
a_ ^ b_ => pow(a_, b_)
```

Dann wenden wir den Trick von oben an, dass verschachtelte Funktionen zusammengefügt werden:

```
# Flach machen
plus (plus (a...)~, z...) => plus (a..., z...)
times(times(a...)~, z...) => times(a..., z...)
pow(pow(a_, b_), c_)      => pow(a_, times(b_, c_))
```

Nach Abbildung 7.7, werden die Regeln der Reihe nach (die zuerst definierte zuerst) geprüft und sobald eine passt, wird diese angewandt. Ab jetzt können alle Regeln davon ausgehen, dass die Darstellung flach ist, d.h. dass es keine verschachtelten Plus- oder Malzeichen mehr hat.

```
# Neutrale Elemente
plus (0~, a...) => plus (a...)
times(1~, a...) => times(a...)

# Funktion ueberfluessig
plus (a_) => a_
times(a_) => a_
plus () => 0
times() => 1

# Spezialfaelle Potenzieren
pow(a_, 1) => a_
pow(a_, 0) => 1
pow(1, a_) => 1
pow(0, a_) => 0
```

Diese Zeilen sollen die Grundrechenarten in eine flache Darstellung bringen und einfache Bereinigungen durchführen. Um den Zwischenstand zu sehen ist es wohl am besten, diese zu testen.

```
In[1]> # Test der Addition
In[1]> a+(b+c)+d
Out[1]> plus(b, c, a, d)
In[2]> :T a+(b+c)+d
   0: [+, [+, a, [+, b, c]], d]
   1: [+, [+, a, [plus, b, c]], d]
   2: [+, [plus, a, [plus, b, c]], d]
   3: [+, [plus, b, c, a], d]
   4: [plus, [plus, b, c, a], d]
   5: [plus, b, c, a, d]
Out[2]> plus(b, c, a, d)
In[3]> plus(a, 0, b, 0, c, 0, 0)
Out[3]> plus(a, b, c)
```

Das “:T” in Eingabe 2 bedarf einiger Erklärung. T steht für “to trace” (verfolgen, nachverfolgen). Der Doppelpunkt bedeutet, dass das Programm direkt angesprochen wird und der darauf folgende Buchstabe nicht zum Term gehört. Gibt man “:h” ein, erhält man eine Liste der verfügbaren Direktiven[2]:

```
In[4]> :h
Optionen:
   :t  Zeige bei Umformung benutzte Regeln an
   :T  Zeige bei Umformung erhaltene Zwischenresultate an
   :f  Interne Repräsentation des Resultats
   :i  Interne Repräsentation der Eingabe
   :h  Diese Hilfe
```

Das “0:” zeigt das nullte Zwischenresultat, also die Eingabe, an. Man sieht alles im internen Format. Wie schon im Kapitel 7 gesehen, stellen Punkt-vor-Strich-Regeln ein Hindernis dar. Deshalb haben wir auch $a + b$ in $\mathrm{plus}(a, b)$ umgewandelt. Intern in der Darstellung des Syntaxbaumes haben wir ein + mit zwei Unterknoten a und b. Das kann als Funktionsaufruf $+(a, b)$ aufgefasst werden. Ändert man die runden Klammern zu eckigen erhält man `+[a, b]`. Das interne Format von Mathematica sieht genau so aus. Aus verschiedenen Gründen (darunter ästhetischen) habe ich mich entschieden, das “+” nach innen zu nehmen. $a + b$ ist also `[+, a, b]` und $\mathrm{plus}(a, \mathrm{plus}(b, c), d, \mathrm{f}())$ ist `[plus, a, [plus, b, c], d, [f]]`. Die Zeilen mit “1:”, “2:”, . . . , “5:” zeigen, wie der Ausdruck verändert wurde. Die letzte Zeile ist auch gleich das Resultat. Man sieht sehr schön, wie von innen heraus vereinfacht wurde.

Weiter mit der Multiplikation und dem Potenzieren:

```
In[5]> # Test der Multiplikation
In[5]> 1*2*(3*4)*1*5
Out[5]> times(3, 4, 2, 5)
In[6]> # Test des Potenzierens
In[6]> 1^2^3^4
Out[6]> 1
In[7]> 2^3^4
Out[7]> pow(2, pow(3, 4))
In[8]> # Nun beides
In[8]> (2^3)^4
Out[8]> pow(2, times(3, 4))
In[9]> ((2^3)^4)^2
Out[9]> pow(2, times(3, 4, 2))
In[10]> (2^3)^(4*2)
Out[10]> pow(2, times(4, 2, 3))
```

Auf dieser Basis lässt sich nun die Umformungsregeln der Potenzfunktionen umsetzen:

[2]Selbstverständlich ist Reckna nicht nur deutsprachig. Es lässt sich auf Deutsch und auf Englisch einstellen.

```
sqrt(x_) => x_^(1/2)
times(x_, x_) => pow(x_, 2)
times(x_~, x_~, z...) => times(pow(x_, 2), z...)
times(pow(x_, e_)~, x_) => times(pow(x_, e_+1))
times(pow(x_, e_)~, x_~, z...)=> times(pow(x_, e_+1), z...)
times(pow(x_, e_)~, pow(x_, f_))=> times(pow(x_, e_+f_))
times(pow(x_, e_)~, pow(x_, f_)~, z...) => times(pow(x_, e_+f_), z...)
```

8.3 Bedingungen

In Abschnitt 6.2 musste ich einmal die when-Anweisung benutzen. Das ist gehört Feature von den Regeln in Reckna, das noch nicht besprochen wurde.

Im gleichen Abschnitt habe ich auch gesagt, dass Regeln die Form `<`*Muster*`> => <`*Ersatz*`>` haben.

Das muss ich nun relativieren, denn sie können auch folgende alternative Form haben: `<`*Muster*`> => <`*Ersatz*`> when <`*Bedingung*`>`.

Die Bedingung hat folgenden Effekt: Es wird wie bisher geschaut, ob ein Ausdruck auf das Muster passt. Macht er das, werden die Platzhalter in der Bedingung so ersetzt, wie sie es im Ersatz täten. Dann wird die Bedingung vereinfacht, wie wenn sie ein normaler Ausdruck wäre. Ist die vereinfachte Version identisch mit "`true`", dann wird die Regel angewandt, ansonsten nicht.

Die when-Anweisung ermöglicht eine schöne Trennung zwischen Muster und Bedingung. Sie wird erst ausgewertet, wenn das gesamte Muster passt.

Zum Beispiel geht folgendes nicht:

```
hasEvenArgument(a_~, z...) => true when isEven(a_)
hasEvenArgument(z...)      => false
```

Hier ist "`a_` " *immer* das erste Argument, da `a_` auf alles passt und der Such-Platzhalter dadurch keine Wirkung zeigt.

Eine mögliche Lösung wäre, auf den Such-Platzhalter zu verzichten und Divide and Conquer anzuwenden:

```
hasEvenArgument(a_, z...) => true when isEven(a_)
hasEvenArgument(a_, z...) => hasEvenArgument(z...)
hasEvenArgument()         => false
```

In Fällen wie diesen, wo Such-Platzhalter auftreten, möchte man Bedingungen schon dann angeben, während das Muster mit einem Ausdruck verglichen wird. Es besteht die Möglichkeit, Inline-Bedingungen mit `[<`*Bedingung*`>]` anzugeben:

```
hasEvenArgument(a_[isEven(a_)]~, z...) => true
hasEvenArgument(z...)                  => false
```

8.4 Anwendung: Ableiten verbessert

Bisher hatte ich die freeOf-Funktion vorbehalten, das hole ich nun nach. Als erstes brauchen wir rudimentäre boolsche Algebra:

```
and() => true
and(true~, z...) => and(z...)
and(false~, z...) => false
```

Man beachte, wie einfach und selbsterklärend das ist. Mit den Ideen, die schon präsentiert wurden liesse sich das einfach auf einen Oder-Operator und sogar den Implikations-Operator ausweiten.

Dank dem Funktionenplatzhalter ist es nun möglich, freeOf zu implementieren.

```
# freeOf
freeOf(x_, x_) => false
freeOf(f_(), x_) => true
freeOf(f_(a_), x_) => freeOf(a_, x_)
freeOf(f_(a_, b...), x_) => freeOf(a_, x_) and freeOf(f_(b...), x_)
freeOf(c_, x_) => true
```

Rufen wir uns noch ein letztes mal die Ableitungen aus Abschnitt 6.2 ins Gedächtnis. Sie waren erstaunlich einfach:

```
deriv(a_+b_, x_) => deriv(a_, x_) + deriv(b_, x_)
deriv(a_-b_, x_) => deriv(a_, x_) - deriv(b_, x_)
```

Wegen unserer anderen Darstellung vom Plus müssen wir diese leicht anpassen. Dieser Teil bleibt gleich:

12: ***# Endpunkte***
13: $\text{deriv}(\texttt{c\_}, \texttt{x\_}) \Rightarrow 0 \ \texttt{when} \ \text{freeOf}(\texttt{c\_}, \texttt{x\_})$
14: $\text{deriv}(\texttt{x\_}, \texttt{x\_}) \Rightarrow 1$

Dieser bleibt ziemlich gleich:

15: ***# Summen- und Produktregel***
16: $\text{deriv}(\text{plus}(\texttt{a\_}, \texttt{b\_\_\_}), \texttt{x\_}) \Rightarrow \text{deriv}(\texttt{a\_}, \texttt{x\_}) + \text{deriv}(\text{plus}(\texttt{b\_\_\_}), \texttt{x\_})$
17: $\text{deriv}(\text{times}(\texttt{a\_}, \texttt{b\_\_\_}), \texttt{x\_}) \Rightarrow \texttt{a\_} \cdot \text{deriv}(\texttt{b\_\_\_}, \texttt{x\_}) + \text{times}(\texttt{b\_\_\_}) \cdot \text{deriv}(\texttt{a\_}, \texttt{x\_})$

Dieser sollte verständlich sein:

18: ***# Potenzregeln***
19: $\text{deriv}(\text{pow}(\texttt{a\_}, \texttt{n\_})), \texttt{x\_}) \Rightarrow \texttt{n\_} \cdot \texttt{x\_}^{\texttt{n\_}-1} \ \texttt{when} \ \text{freeOf}(\texttt{n\_}, \texttt{x\_})$
20: $\text{pow}(\text{deriv}(\texttt{n\_}, \texttt{a\_}), \texttt{x\_}) \Rightarrow \texttt{n\_}^{\texttt{x\_}} \cdot \text{deriv}(\texttt{a\_}, \texttt{x\_}) \ \texttt{when} \ \text{freeOf}(\texttt{n\_}, \texttt{x\_})$
21: $\text{deriv}(\ln(\texttt{a\_}), \texttt{x\_}) \Rightarrow 1/\texttt{a\_} \cdot \text{deriv}(\texttt{a\_}, \texttt{x\_})$

Über die Nutzung von "+" und "·" braucht man sich nicht zu kümmern, die Regeln der Grundrechenarten werden das schon für uns erledigen. Wichtig ist nur, dass im Muster vollständig die Funktionen benutzt werden.

Ab diesem Stadium ist es trivial, die Ableitungen um neue Funktionen zu erweitern. Exemplarisch sei dies an trigonometrischen Funktionen gezeigt.

22: ***# Trigonometrische Funktionen***
23: $\mathrm{deriv}(\sin(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow \cos(\mathtt{a\_}) \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
24: $\mathrm{deriv}(\cos(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow -\sin(\mathtt{a\_}) \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
25: $\mathrm{deriv}(\tan(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow \frac{1}{\cos(\mathtt{a\_})^2} \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
26: $\mathrm{deriv}(\cot(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow -\frac{1}{\sin(\mathtt{a\_})^2} \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
27: $\mathrm{deriv}(\arcsin(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow \frac{1}{\sqrt{1-x^2}} \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
28: $\mathrm{deriv}(\arctan(\mathtt{a\_}), \mathtt{x\_}) \Rightarrow \frac{1}{1+x^2} \cdot \mathrm{deriv}(\mathtt{a\_}, \mathtt{x\_})$
29: $\arccos(\mathtt{a\_}) \Rightarrow \frac{\pi}{2} - \arcsin \mathtt{a\_}$

In meiner Projektskizze gab ich als Ziel an, dass mein CAS einmal fähig sein sollte $\frac{\mathrm{d}}{\mathrm{d}x}\left(x^4 + x^3 + x^2 + x + \arccos(x)\right)$ umformen zu können. Das kann es schon fast:

```
In[1]> deriv(x^4+x^3+x^2+x+arccos(x), x)
Out[1]> 2·x^(2 + -1) + 1 + 3·x^(3 + -1) + 4·x^(4 + -1) + -1·(1 + -1·x^2)^(2^-1·-1)
```

Fehlt noch ein Weg, $2 + (-1)$ in 1 umzuformen. Diesen werden wir in Abschnitt 12.2 kennen lernen.

9 Definitionsbereich

Das wichtigste Feature von Version 0.2 ist der Definitionsbereich. In der Einleitung habe ich kritisiert, dass Mathematica die Gleichung $x/x-1=x$ nach $x=0$ auflöst. Nicht nur Mathematica macht das falsch, sondern alle mir bekannten CAS, eingeschlossen Axiom, welches den mathematisch-hierarchischen Ansatz verfolgt.

Mathematica:

```
In[1]:= Solve[x/x-1==x,x]
Out[1]= {{x -> 0}}
```

Axiom:

```
(1) -> solve(x/x-1 = x, x)
(1) ->
   (1)  [x= 0]
                              Type: List(Equation(Fraction(Polynomial(Integer))))
```

Reckna:

```
In[1]> solve(x/x-1 = x, x)
Out[1, 1]> {} if x /= 0
Out[1, 2]> undef if x = 0
```

9.1 Idee

Wenn ein Mensch einen Ausdruck $\frac{x}{x}$ vor sich hat, kürzt er ihn zu 1, schreibt dahinter gross $x \neq 0$. Man nennt das eine Fallunterscheidung.

Reckna macht genau das. Jeder Ausdruck besitzt seinen eigenen Definitionsbereich. Gilt $x \neq 0$, wird die Umformung $\frac{x}{x}$ einfach gemacht. Gibt es keine solche Einschränkung bezüglich x, wird der Ausdruck aufgesplittet, einmal in x, wobei $x \neq 0$ und einmal in `undef`, wobei $x = 0$.

Das sieht man oben im Beispiel wieder. Bei $x \neq 0$ führt das zur Lösung $x = 0$. solve macht dann eine Abfrage, ob $x = 0$ im Definitionsbereich liegt. Da das nicht so ist, ist die Lösung die leere Menge. Im andern Fall wird solve auf einen undefinierten Ausdruck aufgerufen, was undefiniert ist.

9.2 Umsetzung

Zuerst wird eine Syntax für das Aufsplitten benötigt. Diese sieht wie folgt aus:

```
a_/a_ => case a_/=0 => 1
           , a_ =0 => undef
```

Allgemein muss das `=>` von einem `case` gefolgt sein. Dann wird der Definitionsbereich angegeben und ein erneutes `=>` wird gefolgt von dem vereinfachten Ausdruck. Es können beliebig viele Zweige angegeben werden, diese werden mit Komma getrennt.

Der Definitionsbereich (engl. "domain") ist zuerst `r__domDefault`. Er kann durch eine Regel nach Belieben überschrieben werden. z.B.

```
r__domDefault => {}
```

Eine Bemerkung zum "{}": Das ist ein syntaktischer Zucker und steht für `list()`. So steht z.B. "`{1, 2, 3}`" für `list(1, 2, 3)`.

Der aktuelle Definitionsbereich kann mit `r__domGet()` abgefragt werden.

Bei einer Regel mit einem `case`-Ausdruck wird der Definitionsbereich verändert. Dafür wird `r__domUpdate(new_, domain_)` aufgerufen und der Rückgabewert wird als neuer Definitionsbereich genommen. Dies passiert einzeln für jeden Case-Zweig.

```
r__domUpdate(a_, {a_~, z...}) => {a_, z...}
r__domUpdate(a_, {z...})      => {a_, z...}
```

Am Schluss wird `r__domUpdate(domain_)` aufgerufen um den Definitionsbereich in einem schönen Format anzuzeigen. Das komplettiert unser Beispiel:

```
r__domBeautify({a_}) => a_
r__domBeautify(a_) => a_
```

Die Anwendung sieht fast aus wie erwartet:

```
In[1]> x/x
Out[1, 1]> 1 if x /= 0
Out[1, 2]> undef if x = 0
In[2]> x/x+x/x
Out[2, 1]> 2 if x /= 0
Out[2, 2]> undef if x = 0
Out[2, 3]> undef if {x = 0, x /= 0}
```

Es fehlt noch die Möglichkeit, widersprüchliche Definitionsbereiche zu ignorieren. Das passiert, wenn `r__domUpdate(domain_)` den Wert `false` zurückgibt. Folgende Zeilen schaffen somit Abhilfe:

```
r__domUpdate(a_=x_, {(a_/=x_)~, z...}) => false
r__domUpdate(a_/=x_, {(a_=x_)~, z...}) => false
```

Jetzt ist die Ausgabe wie gewünscht:

```
In[1]> x/x+x/x
Out[1, 1]> 2 if x /= 0
Out[1, 2]> undef if x = 0
```

9.3 Weitere Möglichkeiten

Für den `solve`-Befehl ist eine Funktion mayBe notwendig, der sagt, ob eine Variable einen bestimten Wert annehmen kann.

```
mayBe(x_, v_) => not((x_/=v_) `isIn` r__domGet())
a_ `isIn` {a_~, z...} => true
a_ `isIn` {z...}      => false
```

Jetzt funktioniert schon der wichtigste Teil:

```
In[1]> mayBe(x, x/x-1)
Out[1, 1]> false if x /= 0
Out[1, 2]> undef if x = 0
```

Der solve-Befehl wird in einem späteren Kapitel implementiert.

Das Potenzial ist mit den hier gezeigten Regeln noch lange nicht ausgeschöpft. Zum Beispiel stellt sich die Frage, ob a^b so belassen werden soll oder es einen Split mit $a = 0, b \leq 0$ geben soll. In meiner jetztigen Umsetzung tritt der Split erst auf, wenn sich eine nennenswerte Vereinfachung stattfindet, das verkompliziert allerdings die Regeln.

Ein Problem ist, dass jetzt $0 \cdot \ldots$ nicht mehr nach 0 umgeformt werden kann, da in den Punkten potentiell ein x/x vorkommen kann. Das habe ich so gelöst, dass es eine Funktion gibt, die einen Ausdruck nach Definitionsbereich untersucht, aber nur die dafür notwendigen Rechnungen durchführt.

Was noch fehlt ist, neue Regeln hinzufügen zu können. Auch wenn der Definitionsbereich $x = 0$ ist, bleibt ein x ein x und wird nicht durch 0 ersetzt.

10 Umformungsmodi

Neben den Such-Platzhaltern und dem Definitionsbereich gibt es ein drittes Alleinstellungsmerkmal in Reckna: Umformungsmodi.

10.1 Idee

Die Repräsentation vom Plus-Operator aus Abschnitt 8.2 hat den Nachteil, dass sie für den Anwender direkt sichtbar ist.

Wie in Kapitel 4 gesehen, ist es nicht immer optimal, von innen nach aussen zu vereinfachen.

Deswegen gibt es eine Erweiterung für die Regeln, die eine Dazugehörigkeit zu einem Umformungsmodus angibt. Die allgemeinste Syntax für Regeln lautet:
<*Modul*>| <*Muster*> =><*Ersatz*> when <*Bedingung*>
Der when-Teil und der <*Modul*>-Teil sind dabei optional.

Ein paar Beispiele:

<*Modul*>|	Langform	Bedeutung
(nichts angegeben)	main(0)	Modul main, Stelle 0
|	main(0)	Modul main, Stelle 0
main|	main(0)	Modul main, Stelle 0
main(0)|	main(0)	Modul main, Stelle 0
main(1)|	main(1)	Modul main, Stelle 1
main(-1)|	main(-1)	Modul main, Stelle −1
main(100)|	main(100)	Modul main, Stelle 100
main(-999)|	main(-999)	Modul main, Stelle −999
cute|	cute(0)	Modul cute, Stelle 0
cute(0)|	cute(0)	Modul cute, Stelle 0
cute(1)|	cute(1)	Modul cute, Stelle 1
cute(-1)|	cute(-1)	Modul cute, Stelle −1
down|	down(0)	Modul down, Stelle 0
down(0)|	down(0)	Modul down, Stelle 0
down(-1)|	down(-1)	Modul down, Stelle −1

down, main und cute sind dabei alle erlaubten Modulnamen, die Zahlen sind bei allen unbeschränkt.

Die Reihenfolge der Module ist in Abbildung 10.1 zu sehen. Das “passt” bedeutet in main und cute, das gleiche wie in Abbildung 7.7, dass immer zuerst die Argumente vereinfacht werden. Bei down ist das umgekehrt, es wird zuerst versucht, die ganze Funktion zu vereinfachen und erst dann die Argumente. Wenn das Paar down/main

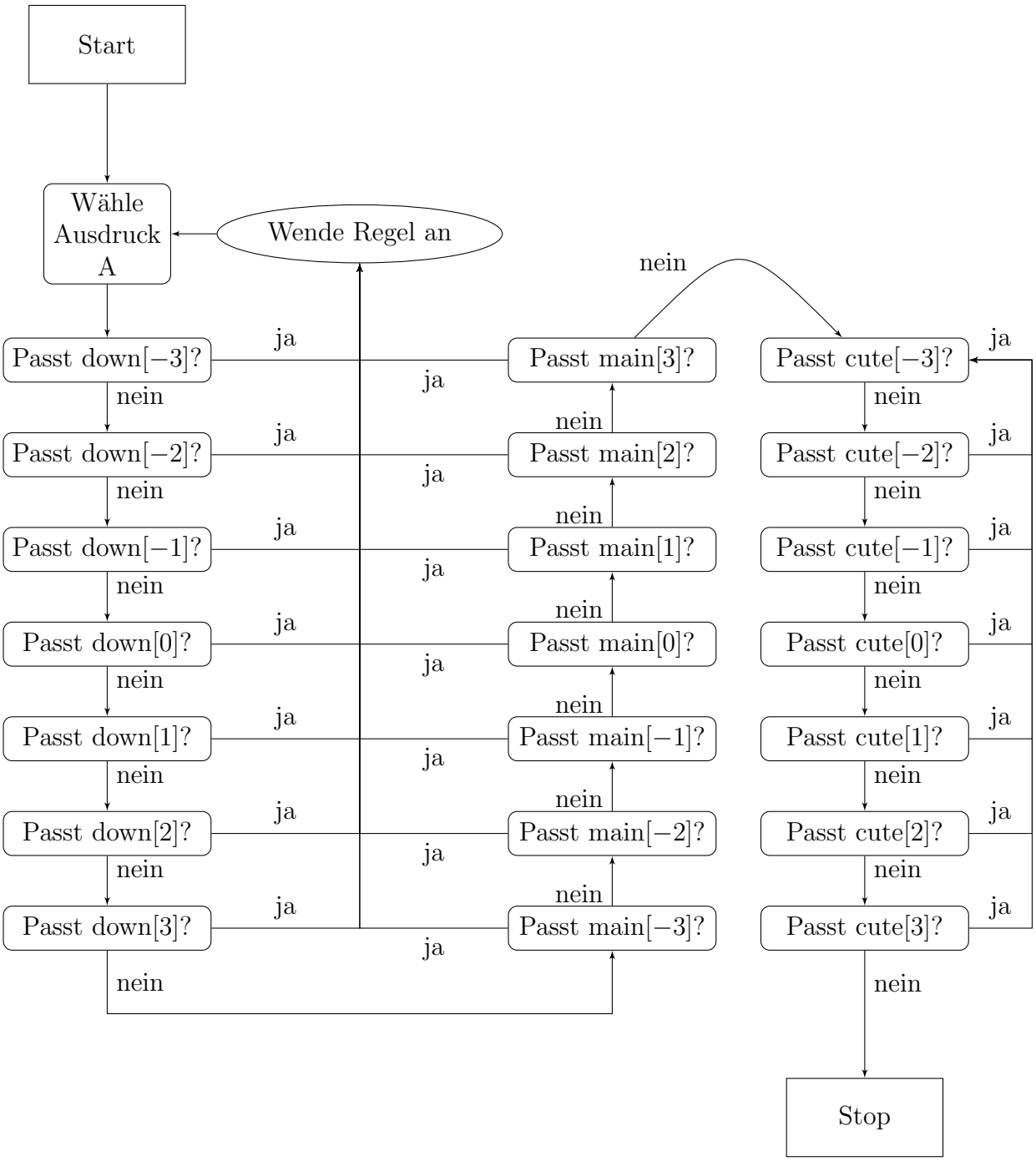


Abbildung 10.1: Die drei Regelmodule down, main und cute.

den Ausdruck fertig vereinfacht hat, hat cute die Aufgabe diesen für die Ausgabe zu verschönern. Es gibt keinen Weg mehr zurück in down oder main.

10.2 Anwendung: Grundrechenarten verschönern

Versuchen wir doch unser Ziel umzusetzen.

1: ***# Kosmetik: Flache Funktionen zurück in binäre Operationen***
2: `cute|` $\text{plus}(\texttt{a\_}, \texttt{b\_}) \Rightarrow \texttt{a\_} + \texttt{b\_}$
3: `cute|` $\text{plus}(\texttt{a\_}, \texttt{b\_}, \texttt{z\_\_\_}) \Rightarrow \text{plus}(\texttt{a\_} + \texttt{b\_}, \texttt{z\_\_\_})$
4: `cute|` $\text{times}(\texttt{a\_}, \texttt{b\_}) \Rightarrow \texttt{a\_} \cdot \texttt{b\_}$
5: `cute|` $\text{times}(\texttt{a\_}, \texttt{b\_}, \texttt{z\_\_\_}) \Rightarrow \text{times}(\texttt{a\_} \cdot \texttt{b\_}, \texttt{z\_\_\_})$
6: `cute|` $\text{pow}(\texttt{a\_}, \texttt{b\_}) \Rightarrow \texttt{a\_}^{\texttt{b\_}}$

Als ich das entwarf, war ich in Verlockung, folgendes zu schreiben:

`cute|` $\text{plus}(\texttt{a\_}, \texttt{z\_\_\_}) \Rightarrow \texttt{a\_} + \text{plus}(\texttt{z\_\_\_})$

Das verstösst allerdings gegen die Linksassoziativität der Addition. Korrekt wäre eine kleine Variation:

`cute|` $\text{plus}(\texttt{a\_}, \texttt{z\_\_\_}) \Rightarrow \text{plus}(\texttt{z\_\_\_}) + \texttt{a\_}$

Was leider den Nebeneffekt hat, die Reihenfolge der Parameter umzudrehen.

Test:

```
In[1]> a+0+b+0+c+0+0
Out[1]> a + b + c
In[2]> 1*2*3*2*1*2*3*2*1
Out[2]> 2·3·2·2·3·2
In[3]> 4^1^2^3^4^1^2^3^4
Out[3]> 4
In[4]> (4^3^2)^5
Out[4]> 4^(3^2·5)
```

Jeder Ausdruck wird zuerst in die Form mit plus(...) und ×(...) gebracht, dort vereinfacht und am Ende zurück in die Darstellung mit Operanden gebracht.

10.3 Anwendung: Mod-Befehl

Es gibt eine eingebaute Funktion `r__intMod(n, m)`, die n mod m ausrechnet (für eingebaute Funktionen siehe Abschnitt 12.2).

Schreiben wir also das folgende:

```
mod(n_, m_) => r__intMod(n_, m_)
```

Jetzt steht man immer noch vor dem Problem, dass bei 123^{456} mod 3 immer zuerst 123^{456} ausgerechnet wird.

Dies ist der Punkt, an dem diese "verzögerte Auswertung", die oben erwähnt wurde, sinnvoll ist. Den Fall, dass `n_` eine Potenz ist, können wir im down-Modus abfangen:

```
down| mod(pow(a_, b_), m_) => mod(mod(a_, m_)^b_, m_)
```

down wird vor main aufgerufen und ohne die Argumente vereinfacht zu haben, d.h. die zweite Regel wird vor der ersten angewandt wenn das Argument eine Potenzierung ist. Sie verpackt dann die Basis in eine Modulo-Operation. – Aber dann wird sie gleich wieder aufgerufen, was zu einer Endlosschleife führt.

Deshalb muss die Zeile direkt die eingebaute Funktion aufrufen:

```
down| mod(pow(a_, b_), m_) => r__intMod(r__intMod(a_, m_)^b_, m_)
```

Wenn `m_` gross ist, können die Zwischenresultate immer noch riesig werden. Aus diesem Grund gibt es eine weitere eingebaute Funktion `r__intPowM(b, e, m)`, die dafür den passenden Algorithmus liefert[1]:

```
down| mod(pow(a_, b_), m_) => r__intPowM(a_, b_, m_)
```

Es ist nun möglich, Modulo von grossen Potenzen zu berechnen:

```
In[1]> (123456789^987654321)‘mod‘314
Out[1]> 99
```

Die Eingabe `(123456789^987654321)‘mod‘314` ähnelt der mathematischen Notation.

In Mathematica ist das etwas umständlicher:

```
In[1]:= Mod[123456789^987654321, 314]
General::ovfl: Overflow occurred in computation.
Out[1]= Overflow[]
In[2]:= PowerMod[123456789, 987654321, 314]
Out[2]= 99
```

Die Eingabe `PowerMod[123456789, 987654321, 314]` ist nicht besonders intuitiv.

[1] Das Problem ist die “modular exponentiation”, wofür der Algorithmus “square and multiply” angepasst werden kann. Das habe allerdings nicht ich programmiert, sondern leite es an die GMPlib weiter.

11 Programmierung in Reckna

Dass sich Reckna dazu eignet um mathematische Umformungen zu betätigen haben wir im letzten Kapitel gesehen. Dabei wurde der gesamte Sprachumfang vorgestellt. Tatsächlich kann man sogar noch viel mehr damit anfangen, Sachen, an die man nicht sofort gedacht hätte.

11.1 Turing-Vollständigkeit

Programmiersprachen sind turingvollständig, d.h. sie können jeden algorithmisch berechenbaren Wert berechnen. Ein Beispiel ist *C++*, der Sprache in der Reckna geschrieben ist. Diese Spracheigenschaften reichen aus, um eine turingvollständige (turingmächtige) Sprache zu sein. Man kann zeigen, dass diese Sprache äquivalent zu einer Turingmaschine oder zu echten Programmiersprachen wie C++ oder Haskell ist.

Eine einfache Programmiersprache, die turingmächtig ist, ist das WHILE-Programm(10). Nachfolgend ist ein Programm aufgelistet, das das Quadrat einer Zahl ausrechnet.

Algorithmus 1 Quadrierung mit einem while-Programm

Require: $a \geq 0$
Ensure: $b = a^2$

```
1: i ← a
2: b ← 0
3: while i ≠ 0 do
4:     k ← a
5:     while k ≠ 0 do
6:         b ← b + 1
7:         k ← k − 1
8:     end while
9:     i ← i − 1
10: end while
```

Die Sprache hat folgende Syntaxelemente:

- `while` $x_i \neq 0$ `do` $AUSDRUCK$ `end`
- $x_i := x_j + c, c \in \mathbb{Z}$
- $AUSDRUCK$; $AUSDRUCK$

Diese Spracheigenschaften reichen aus, um eine turingvollständige Sprache zu sein. Man kann zeigen, dass diese Sprache äquivalent zu einer Turingmaschine oder zu echten Programmiersprachen wie C++ oder Haskell sind.

Gehört die Sprache von Reckna auch in dieser Kategorie?
Folgende Übersetzung:

1. Wir definieren eine Konvertierung $kon(n)$ Zahl⇔Alphabet. 1 ist "a", 2 ist "b", 3 ist "c", ... 26 ist "z". 27 ist "aa", 28 ist "ab". Das geht weiter bis "zz" und fährt dann mit "aaa" usw. fort. $Kon(n)$ sei $kon(n)$, nur dass Grossbuchstaben anstatt Kleinbuchstaben verwendet werden.

2. Wir weisen jeder Zeile eine Funktion `PROG`XX zu, wobei XX für Programmzeile z gleich $K(z)$ ist.

3. Jede Variable hat eine Nummer n und ihr Name in Reckna ist "$(kon(n))$_". 1 wäre z.B. `a_`.

4. Die Funktionen haben folgende Signatur: `Prog`XX`(a_, b_, c_, ...)`, wobei die Anzahl Argumente bei jeder Funktion immer gleich der Anzahl Variablen ist..

5. `while` $x\_i \neq 0$ wird zu: `Prog`XX`(a_, b_, ..., 0, ..., z_) => Prog`YY`(a_, b_, ...)` und `Prog`XX`(a_, b_, ..., z_) => Prog`XX*+1*`(a_, b_, ...)`.

6. $x\_i := x\_j + c$ wird zu `Prog`XX`(a_, b_, ..., x_, ...) => Prog`XX*+1*`(a_, b_, ..., x_ +` c`, ...)`.

Damit das etwas klarer wird, hier das Beispiel in Reckna:

Algorithmus 2 Quadrierung in Reckna im Stile eines while-Programms

```
1: ProgA(a_, b_, i_, k_) ⇒ ProgB(a_, b_, a_, k_)    # i := a
2: ProgB(a_, b_, i_, k_) ⇒ ProgC(a_, 0, i_, k_)     # b := 0
3: ProgC(a_, b_,  0, k_) ⇒ ProgK(a_, b_, 0, k_)     # if i = 0: goto ProgK
4: ProgC(a_, b_, i_, k_) ⇒ ProgD(a_, b_, i_, k_)    # else:
5: ProgD(a_, b_, i_, k_) ⇒ ProgE(a_, b_, i_, a_)    #   k := a
6: ProgE(a_, b_, i_,  0) ⇒ ProgI(a_, b_, i_, a_)    #   if k = 0: goto ProgI
7: ProgE(a_, b_, i_, k_) ⇒ ProgF(a_, b_, i_, a_)    #   else:
8: ProgF(a_, b_, i_, k_) ⇒ ProgG(a_, b_+1, i_, k_)  #     b := b + 1
9: ProgG(a_, b_, i_, k_) ⇒ ProgH(a_, b_, i_, k_-1)  #     k := k - 1
10: ProgH(a_, b_, i_, k_) ⇒ ProgE(a_, b_, i_, k_)   #   goto ProgE
11: ProgI(a_, b_, i_, k_) ⇒ ProgJ(a_, b_, i_+1, k_) #   i := i + 1
12: ProgJ(a_, b_, i_, k_) ⇒ ProgK(a_, b_, i_, k_)   # goto ProgC
```

Kombiniert man das mit der Repräsentation der natürlichen Zahlen aus Abschnitt 6.3, wird eine integrierte Unterstützung der Addition überflüssig. Die Regeln allein garantieren die Turingmächtigkeit.

11.2 Vererbung

Unsere Programmiersprache steht auf einer Stufe mit *C++*, *Haskell*, ja mit allen Programmiersprachen. Wie jede andere Programmiersprache auch.

Vererbung ist eine Ist-ein-Beziehung. Ein Quadrat ist ein Rechteck. In *Java* hiesse das:

```
class Rectangle {
    private double lenght;
    private double width;

    public Rectangle(double lenght, double width) { /* ... */ }

    public double getLenght() { return lenght; }
    public double getWidth () { return width; }
    public double setLenght(double value) { lenght = value; }
    public double setWidth (double value) { width  = value; }

    public double getArea() { return lenght * width; }
}

class Square extends Rectangle {
    public Square(int side)
    {
       super(side, side); // Aufruf von Basisklassenkonstruktor
    }
}
```

Was nützt uns das? Betrachten wir folgende Klasse:

```
class Kitchen {
    public Kitchen(Square tileSize, Rectangle kitchenSize) { /* ... */ }
    /* ... */
}

// Geht:
new Kitchen(new Square(1), new Rectangle(12, 10));

// Geht dank Vererbung:
new Kitchen(new Square(1), new Square(10));

// Geht nicht, wollten wir ja verbieten:
new Kitchen(new Rectangle(1, 2), new Rectangle(10, 20));

// Geht nicht, brauchen wir nicht zwingend:
new Kitchen(new Rectangle(1, 1), new Rectangle(10, 20));
```

Wenn wir irgendwo ein Rechteck wollen, erlauben wir implizit ein Quadrat und wenn wir ein Quadrat wollen, verbieten wir ein Rechteck.

Im Gegensatz zu dieser richtigen Anwendung von Vererbung hat unsere Klasse `Rectangle` einen Fehler:

```
Square s = new Square(3);
```

```
s.setLenght(4); // das geht, aber
s.setWidth(2);  // sollte nicht.
// s hat den Typ Quadrat, obwohl es jetzt ein 4x2-Rechteck ist
```

Das Beispiel wird oft gebraucht, um vor möglichen Fallstricken in der Vererbung zu warnen. Hier wäre das Problem, dass `Square` den Typ `Rectangle` einschränkt anstatt zu erweitern. Will man so eine Hierarchie, um sie wie in der Klasse `Kitchen` benutzen zu können, müsste man die Setter (`setLenght` und `getWidth`) entfernen. Möchte man diese trotzdem haben, bleibt einem nur noch übrig, eine Klasse `Rectangloid` zu schreiben, von der dann `Square` und `Rectangle` erben. Der Tenor ist: Ein Quadrat ist in der Programmierung kein Rechteck.

Ziel dieses Kapitel ist es, zu zeigen, dass Reckna Objektorientierung unterstützt. Was wäre hierfür passender als ein Paradoxon in Java, das in Reckna keines mehr ist?

`# new type: rectangle(l, w)`
`isRectangle(rectangle(l_, w_)) ⇒ true`
`getWidth(rectangle(l_, w_) ⇒ w_`
`getLenght(rectangle(l_, w_) ⇒ l_`
`setWidth(rectangle(l_, _), w_) ⇒ rectangle(l_, w_)`
`setLenght(rectangle(_, w_), l_) ⇒ rectangle(l_, w_)`

Jetzt wurde genügend Abstraktion geschaffen, die Funktion `getArea` generisch zu schreiben:

`getArea(r_) ⇒ getWidth(r_)*getLenght(r_)` **when** `isRectangle(r_)`

Weiter gehts mit dem Quadrat:

`# new type: square(s)` which is like a `rectangle(s, s)`
`isSquare(square(s_)) ⇒ true`
`isRectangle(square(s_)) ⇒ true`
`getWidth (square(s_) ⇒ s_`
`getLenght(square(s_) ⇒ s_`
`setWidth (square(s_), w_) ⇒ rectangle(s_, w_)`
`setLenght(square(s_), l_) ⇒ rectangle(s_, l_)`
`rectangle(s_, s_) ⇒ square(s_)`

Testen:

```
In[1]> s => rectangle(3, 3)
Out[1]> true
In[2]> s
Out[2]> square(3)
In[3]> {isSquare(s), isRectangle(s)}
Out[3]> {true, true}
In[4]> getArea(s)
Out[4]> 9
```

12 Mathematische Bibliothek

Vererbungen stellen nach Gräbe (4) einen wesentlichen Bestandteil der Computeralgebrasysteme dritter Generation dar. Ein Grund, weshalb ich die Adition und Subtraktion ganzer Zahlen so weit nach hinten verlegt habe, liegt darin, dass die Zahlen in eine Objekthierarchie eingebaut sind.

12.1 Listen

Jede Programmiersprache braucht etwas, mit dem sie Listen ausdrücken kann. In Reckna ist eine Liste eine reine Funktion, list. Den Ausdruck list$(1, 2, 3, 4, 5)$ könnte man sich als einen Funktionsaufruf vorstellen, aber in Reckna gibt es keine Funktionen. Es gibt nur Ausdrücke und Operationen auf Ausdrücke. Wenn es also keine Regel gibt, die list$(1, 2, 3, 4, 5)$ vereinfacht, bleibt der Ausdruck so, wie er ist.

```
In[1]> list(1, 2, 3, 4, 5)
Out[1]> {1, 2, 3, 4, 5}
In[2]> {3, 2, 1}
Out[2]> {3, 2, 1}
In[3]> :i {3, 2, 1}
Out[3]> [list, 3, 2, 1]
```

Weil Listen praktisch sind, gibt es eine Syntax für sie. {<...>} ist in Reckna eine Kurzform für list(<...>).

Im Voyage gibt es einen seq-Befehl.

$$seq(x^2, x, 1, 5) \quad \rightsquigarrow \quad \{1, 4, 9, 16, 25\}$$

Wir können das auch. Aber vorher müssen wir uns noch überlegen, wie Regeln den seq-Befehl aufrufen sollen. $seq(x^2, x, 1, 5)$ geht dann schief, wenn jemand eine Regel aufstellt, die jedes x in eine 5 umwandelt, also zum Beispiel `| x ⇒ 5;`.

Deshalb, so habe ich mich entschieden, steht jeder Regel eine garantiert einzigartige Variabel zu; der Paragraph $.

```
In[1]> $
Out[1]> $1
In[2]> $
Out[2]> $2
In[3]> $
Out[3]> $3
```

```
In[4]> a
Out[4]> a
In[5]> $+$
Out[5]> $4 + $4
```

Der Zähler startet bei 1 und erhöht sich bei jeder Nutzung. Damit die Garantie der Eindeutigkeit überhaupt funktioniert, lässt sich $2 nicht direkt eingeben.

Es wird noch eine zweite Hilfsfunktion benötigt und zwar die Funktion subst():

```
subst(e_, v_, r_) => r__subst(e_, v_, r_)
```

`r__subst(exp_, var_, repl_)` ist eine eingebaute Funktion, die jedes Vorkommen von `var_` innerhalb von `exp_` mit `repl_` ersetzt.

Wie könnte man seq implementieren? Wenn Start und Ziel den gleichen Wert haben, ist es trivial, dann entspricht es einem subst-Aufruf. Und sonst erledigt man es für Start (was ja trivial ist) und rekursiv für Start+1 bis Ziel und fügt die beiden Listen zusammen.

```
# seq
seq(a_, i_, t_, t_) => {subst(a_,i_,t_)}
seq(a_, i_, f_[.isConstInteger()], t_[.isConstInteger()]) =>
  subst(a_,i_,f_) `cons` seq(a_,i_,f_ + 1,t_)
```

Listen sind ohne Hilfsfunktionen kaum handhabbar. Dieses cons() ist ein Beispiel von so einer Hilfsfunktion. Eine weitere Funktion ist combine(), die zwei Listen zusammenfügt.

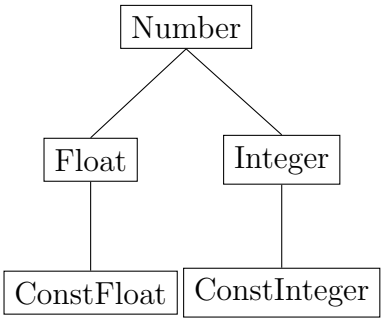
```
# Element vorne anfuegen
a_ `cons` {b...} => {a_, b...}

# Listen zusammenfuegen
{} `combine` {b...} => {b...}
{a_, a...} `combine` {b...} => a_ `cons` ({a...} `combine` {b...})
```

Als Erinnerung: `a`f`b` ist syntaktischer Zucker für `f(a,b)`.

12.2 Typsystem und Eingebaute Funktionen

Bis jetzt hatten wir nur mit reellen Zahlen zu tun. Das wird sich auch nicht ändern, aber zumindest prinzipiell will man sich das offen halten. Jede reelle Zahl ist vom Typ "Number". Ganzzahlen sind vom Typ "Integer", Kommazahlen sind vom Type "Float".



Kommazahlen sind in Reckna integriert, der Einfachheit halber lasse ich sie aber zukünftig weg. Damit man sie vernünftig unterstützen kann, muss man ein Konzept wie Mathematica haben, das zum Beispiel π je nach Bedarf auf

beliebig viele Kommastellen ausrechnen kann. Die ersten Stellen von Pi sind 3.141592653589793. Der Voyage besitzt nur eine bestimmte Genauigkeit und gibt ab einer gewissen Anzahl Stellen ein falsches Resultat:

$$\pi > 3.141592653589799 \quad \rightsquigarrow \; true$$

(Wären die letzte Ziffer eine 3 anstatt einer 9, würde das Resultat stimmen).

Wenn eine Ganzzahl ein Literal (0, 42 oder −1) ist, ist es vom Typ "ConstInteger". Die Unterscheidung ist mit den bisherigen Regeln der Kunst nicht durchführbar, deshalb gibt es die eingebaute Funktion `r__isConstInteger()`, die das für einen erledigt. Funktionen, die mit `r__` beginnen sind reserviert und der Zugriff auf sie sollte von normalen Funktionen gekapselt werden, da sie sofort aufgerufen werden ohne das Argument zu vereinfachen.

```
isConstInt(a_) => true when r__isConstInteger(a_)
```

Zur Addition gibt es eine eingebaute Funktion `r__intAdd(lhs, rhs)`, die allerdings nur funktioniert, wenn die Parameter Ganzzahlkonstanten sind.

```
# Konstanten
plus(i_[.isConstInt()]~, j_[.isConstInt()]~, z...)
  => plus(r__intAdd(i_, j_), z...)

times(i_[.isConstInt()]~, j_[.isConstInt()]~, z...)
  => times(r__intMul(i_, j_), z...)
```

Falls keine Ganzzahlkonstanten vorhanden sind, müssen alle Möglichkeiten für die beiden Platzhalter ausprobiert werden, was proportional zum Quadrat der Anzahl Parameter ist. Trotzdem ist es viel schneller als die Liste in linearer Zeit mit eigenen Regeln durchzugehen, zumal die Anzahl Argumente nur sehr selten gross ist.

12.3 Der `solve`-Befehl

Um Gleichungen per Formel lösen zu können, ist es sinnvoll, wenn $2(a+b)$ und $2a+2b$ die gleiche Darstellungsform haben. Dazu brauchen wir einen expand-Befehl:

```
# Ausmultiplizieren
expand(plus(times(plus(a_, b...)~, c...)~, z...))
 => expand(plus(times(a_, c...), times(plus(b...), c...), z...))
expand(times(plus(a_, b...)~, c...))
 => expand(plus(times(a_, c...), times(plus(b...), c...)))
expand(expanded_) => expanded_ # Keine weiteren Umformungen mehr
```

Es wird hier wieder das Prinzip aus Abbildung 6.1 zu Nutze gemacht, dass die letzte Regel nur dann ausgeführt wird, wenn die oberen nichts mehr finden. Dann liegt der Ausdruck nämlich in einer expandierten Form vor und wir sind fertig.

Reduzieren wir solve auf die Bestimmung der Nullstellen:

```
solve(a_ = b_, x_) => zeros(x_, expand(a_) + expand(-b_))
```

Es ist praktisch, wenn wir die Nullstellen wie beim Voyage als Liste ausgeben. Wenn eine Seite 0 ist und die andere faktorisiert vorliegt, können wir für beide Seiten die Nullstellen bestimmen:

```
zeros(x_, times(a_, b...)) => zeros(x_, a_) `combine` zeros(x_, times(b...))
```

Nachdem alles ausgeklammert wurde, hat man ein Polynom in x. Nun müssen die Koeffizienten bestimmt werden.

```
# Koeffizient extrahieren
coeffOf(x_, plus(x_~, z...)) => 1 + coeffOf(x_, plus(z...))
coeffOf(x_, plus(times(x_~, a...)~, z...))
  => times(a...) + coeffOf(x_, plus(z...))
coeffOf(x_, times(x_~, z...)) => times(z...)
coeffOf(x_, x_) => 1
coeffOf(x_, a_) => 0

noCoeff(x_, plus(a_, z...)) => noCoeff(x_, a_) + noCoeff(x_, plus(z...))
noCoeff(x_, a_) => a_ when a_`freeOf`x_
noCoeff(x_, a_) => 0
```

Jetzt können z.B. die Koeffizienten eines Polynom 2. Grades ausgelesen werden:

```
zeros(x_, e_) => _quadraticEq(x_,
                              coeffOf(x_^2, e_),
                              coeffOf(x_,   e_),
                              noCoeff(x_,   e_))
  when (x^2*coeffOf(x_^2, e_) + x_*coeffOf(x_,   e_) + noCoeff(x_, e_))
       `equals` e_
```

Die Bedingung stellt sicher, dass die Zerlegung auch wirklich vollständig war.

Die Methode kann leicht auf kompliziertere Gleichungen übertragen werden.

Jetzt kann mit der Lösungsformel angesetzt werden.

```
_midnight(x_, 0, b_, a_) => {-b_/(2*a_)}
_midnight(x_, delta_[.isNegative()], b_, a_) => {}
_midnight(x_, delta_, b_, a_)
  => {x_= (sqrt(delta_)-b_)/(2*a_), x_= (b_+sqrt(delta_))/(2*a_)*-1}
_quadraticEq(x_,  0,  0,  0) => IR
_quadraticEq(x_,  0,  0, c_) => {}
_quadraticEq(x_,  0, b_, c_) => -c_/b_
_quadraticEq(x_, a_, b_, c_) => _midnight(x_, b_^2 - 4*a_*c_, b_, a_)
```

Sollte der Ansatz unerfolgreich sein, möchte man gerne wieder den solve-Befehl zurückbekommen. Das ist mit einem kleinen Trick möglich:

```
# try
1| r__fail`try`e_ => e_
1| s_     `try`e_ => s_

# Solve
solve(a_ = b_, x_) => zeros(x_, expand(a_) + expand(-b_))
                      `try` _solve(a_ = b_, x_)
solve(true, x_) => IR
cute| _solve(e_, x_) => solve(e_, x_)

zeros(a_, b_) => r__fail
```

Auch hier wieder ein Beispiel für die Universalität der Umformungsmodi.

12.4 Ein Typ für Brüche

Während sich die Darstellung mit Potenzen für Variablen eignet, ist es mit jetztigen Mitteln unmöglich, z.B. $(-1) \cdot 2^{-2} \cdot 6^3$ zu vereinfachen. Zahlen rechnet man nicht mit Potenzgesetzen aus, sondern mit Brüchen. Deshalb gibt es in Reckna eine Klasse frac, die die Potenzen mit positivem Exponenten im Zähler und die mit negativem im Nenner multipliziert. Bei jeder Operation auf sie wird gekürzt.

Sie so detailliert vorzustellen wie die andereren Konzepte würde den Umfang dieser Arbeit noch weiter vergrössern. Mit ihr wird das letzte grosse Loch in der Bibliothek gestopft.

13 Auswertung

13.1 Ist Reckna ein gutes CAS?

In Kapitel 4 wurden drei Thesen formuliert, was ein gutes CAS leisten können sollte.

These 1: Ein gutes *Computeralgebrasystem* merkt sich zu jeder Variable ihren Definitionsbereich.

Genau genommen merkt sich Reckna zum Gesamtausdruck Definitionsbereichs-Einschränkungen einzelner Variablen.

These 2: Ein gutes *Computeralgebrasystem* bietet Funktionen die Möglichkeit, an den unvereinfachten Ausdruck zu gelangen.

Mit dem down-Modus wurde genau das erreicht.

These 3: Ein gutes *Computeralgebrasystem* verbirgt die inneren Vorgänge nicht und erlaubt, diese zu beeinflussen und erweitern.

Die Regeln sind nicht im eigentlichen Programm versteckt, sondern in der Interpretersprache Recklang geschrieben. Dank den Modi kann jede eingebaute Regel mit einer höher priorisierten überschrieben werden. Eingebaute Funktionen werden selten direkt aufgerufen und können bei Bedarf durch eigene ersetzt werden.

Das Werkzeug des Tracings erlaubt es, Schritt für Schritt zu verfolgen, wie ein Term umgeformt wurde.

Ist Reckna nun ein gutes CAS? Leider reicht es nicht aus, die drei Bedingungen zu erfüllen, um ein gutes CAS zu sein. Viel wichtiger sind die beiden Kriterien Geschwindigkeit und Mächtigkeit aus Abschnitt 5.1. Die Thesen sind Schritte darauf zu und was das betrifft, ist Reckna zumindest auf einem guten Weg.

13.2 Vergleich

13.2.1 Mathematica

Ein häufiger Kritikpunkt an Mathematica ist, dass seine Algorithmen geheim sind, obwohl sie eigentlich der Forschung dienlich wären (4). Das ist kontrovers, vor allem wenn man bedenkt, dass die Gelder für die Entwicklung hauptsächlich von Forschungseinrichtungen kommen. Für mich als Schüler würde eine Lizenz 128.00 € kosten (Quelle: Hersteller), obwohl ich mit den meisten Funktionen nicht viel anfangen kann.

Was die Funktionalität betrifft, lässt sich Reckna nicht mit Mathematica vergleichen, da liegt Mathematica, wie seine Mitstreiter Maple und MuPAD, uneinholbar weit vorne. Im Bereich der Programmierung bietet Reckna ähnliche Möglichkeiten wie Mathematica.

13.2.2 TI Voyage

Zu Beginn sah ich im Voyage ein unerreichbares Ziel. Denn der Voyage kostet pro Stück knapp doppelt so viel wie eine Mathematica-Lizenz (Quelle: eigene Erfahrung) und wird von bezahlten Programmieren entwickelt. Dass ein Schüler mit einem halben Jahr Aufwand an einen Punkt kommen konnte, in dem sein CAS in einigen Bereichen dem Voyage überlegen ist, stimmt bedenklich.

Der Aufbau von Derive ähnelt demjenigen anderer Computeralgebrasysteme aus den Sechzigerjahren (4). Bis vor kurzem konnte er nicht einmal mit Brüchen rechnen, der TI-89, der Vorgänger vom Voyage, tut sich heute noch schwer. Schon mit dem TI Voyage, aber viel stärker mit seinem Nachfolger TI-Nspire wird klar, dass der Fokus der Entwicklung hauptsächlich auf einer schicken grafischen Oberfläche liegt als in seiner Kernfunktionalität, dem symbolischen Umformen.

Durch die Erkenntnisse der letzten Jahre wird es immer einfacher, eine funktional gleichwertige Alternative zu entwickeln. Es wird sich zeigen, was Texas Instruments dem entgegenzusetzen hat.

13.3 Grenzen

Faktor Zeit. Es gibt noch vieles, was getan werden kann. Zum Beispiel das dynamische Hinzufügen von Regeln. Dies ist jedoch keine Einschränkung, sondern ein nächster Schritt, der getan werden muss. Das ist eine Balance, die jede Software finden muss, wie stark man auf die perfekte Lösung wartet (was beliebig lang dauern kann) oder wann man schon etwas Unfertiges veröffentlicht. Ohne Veröffentlichungen lebt kein Projekt.

Faktor Effizienz. Die meiste Arbeit wird von den Regeln getan. Obwohl hier Optimierungspotenzial vorhanden ist (durch Hashing kann die Zahl der Kandidaten reduziert werden[1], ein Rechner mit 4 Prozessoren kann 4 Regeln gleichzeitig testen), ist ein optimierter Algorithmus auf tiefster Ebene immer noch am schnellsten. Mathematica erledigt das meiste durch die eingebauten Funktionen, bei Reckna sind es zumindest auch die grundlegenden Operationen auf ganze Zahlen (Grundrechenarten, Kürzen, Modulo, Potenzieren). Ausserdem müsste der Code optimiert und in Maschinensprache kompiliert werden, damit er annähernd so schnell ausgeführt werden kann wie der Code im C++-Teil. Das ist nur durch Expertenwissen erreichbar, dem ich mich nicht gewachsen fühle.

Faktor Mächtigkeit. Damit Reckna mit dem TI Voyage konkurrieren kann, muss noch einiges implementiert werden. Das geschieht nicht einfach so und benötigt noch einiges an Aufwand. Für ein Ein-Mann-Projekt ist es schlicht unmöglich, sich darüber hinaus mit Mathematica zu messen.

[1] Diese Optimierung ist schon umgesetzt. Sie brachte einen enormen Geschwindigkeitszuwachs um einen Faktor im vierstelligen Bereich

13.4 Ausblick

Das Projekt Reckna steht für eine praktische Nutzung noch am Anfang. Für eine solche müsste noch einmal gleich viel Arbeit investiert werden, die aber hauptsächlich in die Umsetzung von Algorithmen gehen wird. Der Ansatz und die Architektur sind stabil genug, noch weitere Teilbereiche der Mathematik aufzunehmen ohne eine Umstrukturierung vornehmen zu müssen.

14 Schlusswort

Mit welchen Ansätzen vereinfachen Computeralgebrasysteme (CAS) mathematische Ausdrücke? Moderne CAS sind eigentlich Interpreter, die Code in einer eigenen Programmierumgebung, insbesondere einer eigenen Programmiersprache, ausführen, der die Eingabe auswertet. Für diesen Zweck haben sich Regeln als das Mittel der Wahl erwiesen. Das einzige heute noch verbreitete CAS, das nicht den Regelanatz verwendet, ist der TI Voyage. Aus der Architektur ergeben sich Vorteile wie Einfachheit und Erweiterbarkeit.

Regeln kann man mit dem Titelbild dieser Maturarbeit, dem Bild "Befreiung" von Escher (1955 Lithograph, ©1988 M. C. Escher) vergleichen. Schwierige Probleme, echte Vögel, werden auf einfachere Probleme zurückgeführt, die ihrseits auf einfachere zurückgehen bis die Lösung für den Computer einfach lösbar ist. Das Geflecht der Regeln muss möglichst dicht sein, da aber kein System vollständig sein kann, reisst es irgendwann auf. Mit zunehmendem Komplexheitsgrad können immer weniger Spezialfälle gelöst werden.

Reale CAS müssen aber auch Schnelligkeit vorweisen können und verlegen zeitkritische Routinen in den Kern. Wegen ihrer Pionierrolle und der Sorgen zur Geschwindigkeit haben diverse Entscheidungen zu einer Verkomplizierung der Sprache geführt.

Mein Ziel war, die Erkenntnisse dieser historisch gewachsenen Systeme zusammenzufügen in ein CAS, das diese von Grund auf im Design mit einbezogen hat. Es ist mir gelungen, meine Minimalanforderungen zu erfüllen und herausgekommen ist ein CAS, das flexibler ist als der Voyage und um einiges einfacher als gängige Computeralgebrasysteme, ihnen bezüglich Umfang aber nicht gewachsen ist.

Ein CAS ist ein System, das unter einer einheitlichen Oberfläche Zugriff auf verschiedene Algorithmen bietet. Viele neue CAS sind am Mangel an Funktionalität gescheitert, was eben auch ein nicht zu unterschätzender Nachteil an meinem Programm ist. Deshalb ist es für praktischen Einsatz nicht geeignet.

Als ich die Maturarbeit begonnen hatte, dachte ich, ein CAS sei ein Modell der Mathematik. In diesem Punkt lag ich (glücklicherweise) falsch. Beim Programmieren war ich erstaunt, wie schnell es ging, ein einfaches Computeralgebrasystem zu entwickeln. Als es darum ging, dieses zu verbessern, wurden meine Ideen immer komplexer und die Umsetzung schwerer. Trotzdem bin ich mit meinem Ergebnis zufrieden und es war für mich eine grosse Erfahrung aus Sicht der Informatik, einmal ein grösseres Programm geschrieben zu haben und eine mindestens ebenso grosse in Hinsicht auf die Mathematik.

Glossary

Algorithmus Ein Algorithmus ist eine Berechnungsvorschrift zur (endlichen) Lösung eines Problems. 10, 54

BigInt für (big integer). Bigints sind ganze Zahlen beliebiger Genauigkeit. Um sie zu nutzen wird oft die "GNU Multiple Precision Arithmetic Library" (GMPlib) genommen. 54

C ist eine maschinennahe Programmiersprache. 54

C++ ist eine Mehrparadigmen-Programmiersprache. 39, 41, 54

Computeralgebrasystem wird oft ales CAS abgekürzt. 8, 9, 20, 50, 54

Fliesskommazahl Eine Fliesskommazahl speichert zwei Zahlen a und b, die den Wert $a \cdot 10^b$ (oder auch einer anderen Basis) definieren. Da a und b begrenzt grosse Ganzzahlen sind, ist eine Fliesskommazahl eine gerundete Darstellung. 8, 54

Haskell ist eine funktionale Programmiersprache. 41, 54

Java ist eine objektorientierte Programmiersprache. 41, 54

Numerik ist das näherungsweise Berechnenmit Hilfe von *Fliesskommazahlen*. Im Gegensatz zum *BigInt* sind die Ergebnisse nie exakt. 11, 54

TI Voyage der TI Voyage ist ein Gerät mit Betriebssystem von Texas Instruments. Dieser Begriff wird hier synonym zum eingebauten *Computeralgebrasystem* (des Voyage 200 Operating Systems v3.10) benutzt symbol. 7–9, 54

TI-30 der TI-30 ist ein Taschenrechner, der nur mit Zahlen rechnen kann. 10, 54

Turing-Mächtigkeit Wenn eine Programmiersprache turing-mächtig ist, kann mit ihr jeder berechenbare Wert ausgerechnet werden. 23, 54

Literaturverzeichnis

[1] *Yacas*. – http://yacas.sourceforge.net/homepage.html

[2] Baader, Franz ; Nipkow, Tobias: *Term Rewriting and All That*. Cambridge : Cambridge University Press, 1999. – ISBN 0521779200

[3] Fateman, Richard J.: *On the Design and Construction of Algebraic Manipulation Systems*. 1990. – www.cs.berkeley.edu/~fateman/papers/asmerev94.ps

[4] H.-G. Gräbe, Universität L.: *Skript zum Kurs Einführung in das symbolische Rechnen*

[5] Hofstadter, Douglas R.: *Gödel, Escher, Bach. Ein Endloses Geflochtenes Band*. dtv, 2011. – ISBN 9783423300179

[6] Knuth, Donald E.: *The Art of Computer Programming (Volume 1)*. 1997. – ISBN ISBN 0-201-89683-4

[7] Wikipedia: *Abstrakter Syntaxbaum*. – http://en.wikipedia.org/wiki/Abstract_syntax_tree

[8] Wikipedia: *Computer algebra system*. – http://en.wikipedia.org/wiki/Computer_algebra_system

[9] Wikipedia: *Peano-Axiom*. – http://de.wikipedia.org/wiki/Peano-Axiom

[10] Wikipedia1: *WHILE-Programm*. – http://de.wikipedia.org/wiki/While-Programm