Größte gemeinsame Teiler in Polynomringen

und
Implementierung im $\mathcal{ISAC}$-Projekt

Stefan Karnel
skarnel@ist.tu-graz.ac.at

August 2002

Diplomarbeit in Technischer Mathematik

durchgeführt am

Institut für Mathematik (B)
der Technischen Universität Graz

Begutachter: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Clemens Heuberger
Betreuer: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Clemens Heuberger

Die Summe unserer Erkenntnis besteht aus dem,
was wir gelernt,
und aus dem,
was wir vergessen haben.
(Marie von Ebner-Eschenbach)

Danksagung

Diese Diplomarbeit ist meinen Eltern, meinen Freunden und meinen Professoren gewidmet, die mich während meines Studiums tatkräfig unterstützt haben. Besonders möchte ich mich bei Herrn Dr. Walther Neuper bedanken, der mich für diese Diplomarbeit motivieren konnte und Herrn Dr. Clemens Heuberger für seine großartige Hilfe bei den zahlreichen mathematischen Problemen dieser Arbeit.

Danke!

Ich versichere, diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubter Hilfsmittel bedient zu haben.

Graz/Klagenfurt, August 2002 Stefan Karnel

ZUSAMMENFASSUNG

Das Rechnen mit Brüchen ist ein grundlegender Teil von Computer-Algebra-Systemen (CAS). Dieser Diplomarbeit ist die Aufgabe gestellt, die wichtigsten Verfahren, die für das Bruchrechnen benötigt werden, vorzustellen und die geeigneten Verfahren im Rahmen des *ISAC*-Projektes zu implementieren.

Es wird gezeigt, dass die schwierigste Aufgabe die Berechnung des größten gemeinsamen Teilers ist. Dieser kann auf verschiedene Arten berechnet werden. Dabei wurden die modularen Zugänge als besonders geeignet befunden.

Die Anwendung der Verfahren zum Vereinfachen von Bruchtermen, Addieren von Bruchtermen, usw. stellt besondere Anforderungen aufgrund der schrittweise interaktiven Arbeitsweise in *ISAC*. Diese Anforderungen werden im Hinblick auf die erarbeiteten Verfahren spezifiziert.

INHALTSVERZEICHNIS

1. EINLEITUNG

Die in dieser Diplomarbeit behandelten Algorithmen sind theoretisch geklärt, in den gängigen Computer Algebra Systemen implementiert und somit allgemein und leicht zugänglich. Dass sie hier nochmals aufgearbeitet werden, bezieht seinen Sinn aus der besonderen Aufgabenstellung im $\mathcal{ISAC}$-Projekt. Dieses wird daher einleitend vorgestellt.

1.1 Über das $\mathcal{ISAC}$-Projekt

Die folgende Darstellung wurde aus [GGK+02] übernommen.

1.1.1 Ausgangslage

Die Ausgangslage für $\mathcal{ISAC}$ ist gekennzeichnet durch zwei aktuelle Entwicklungen: (1) durch ein Innovationsfenster in der Entwicklung von speziellen Konzepten und Werkzeugen der Computermathematik und der Softwaretechnologie, und (2) durch den zunehmenden Einsatz von Softwarewerkzeugen in der Lehre von Mathematik.

Zum Punkt (2) ist zu bemerken, dass die grundlegend nützlichen Softwareprodukte nicht für den pädagogischen, sondern für den industriellen und/oder wissenschaflichen Einsatz entwickelt wurden (während als 'Lernsoftware' entwickelte Produkte partikuläre und kurzfristige Randerscheinungen blieben). Kreative Pädagogen fanden dann meist Jahr(zehnt)e nach der Einführung der Softwareprodukte (zum Beispiel die Algebra Systeme wurden in den 70-iger Jahren entwickelt) auch Einsatzmöglichkeiten in der Lehre. Die zunehmende Intensität und Breite dieses Einsatzes zeigt nicht nur positive Erfahrungen, sondern auch Vorbehalte und Bedenken. Dies begründet das Anliegen von $\mathcal{ISAC}$, eine nächste Generation von Mathematiksoftware von vornherein unter pädagogischen Gesichtspunkten zu entwickeln.

Die Entwicklung kann sich auf die obgenannten (1) Innovationsfenster in zwei Wissenschaftsdisziplinen stützen: auf (a) die Entwicklung deduktiver Systeme (zum Beispiel [Pau94]) in der Computermathematik, und (b) Spezifikationswerkzeuge in der Diziplin der 'Formal Methods' der Softwaretechnologie.

(a) Die 'Theorem Prover' werden eingesetzt, um hochkomplexe Elektronik fehlerfrei zu entwerfen und gewünschte Eigenschaften von Softwarekomponenten zu verifizieren; nebenbei implementieren einige dieser Prover bereits eine große Portion von Mathematikwissen[1]. Für Lernzwecke erweist sich ihre schrittweise, interaktive Arbeitsweise und die vom interpretierenden System separierte Wissensbasis als günstig.

(b) Spezifikationswerkzeuge stellen die Entwicklung komplexer und sicherheitskritischer Software- und Hardware-Systeme auf eine formal-logische Basis, indem sie das betreffende System mithilfe (zumeist sehr elementarer) mathematischer Konzepte beschreiben. Diese Art der Beschreibung ist sehr hilfreich anzuwenden auf alle Probleme der angewandten Mathematik.

1.1.2 Zielsetzung des Gesamtprojektes

Ziel des *ISAC*-Projektes ist es, auf Basis neuester Entwicklungen in Computermathematik und Softwaretechnologie ein Mathematik-(Lern-)System mit grundlegend neuer Funktionalität zu entwickeln, die nach pädagogischen Erfordernissen ausgerichtet ist: ein solches System ([Neu02]) ...

... ist transparent punkto Wissensbasis
Das Wissen eines Mathematik-Systems soll für den Benutzer lesbar sein, in derselben Form, wie es vom System interpretiert wird (und *nicht* als *zusätzliche* Hilfe-Information).

Relativ alt ist der Vorschlag, Mathematik-Wissen entlang von drei Achsen zu strukturieren [Buc82], Theorien, Problemen und Methoden:
(1) Theorien beschreiben die deduktive Struktur der Mathematik (Axiome, Definitionen, Sätze und Beweise). *ISAC* baut auf die Wissensbasis des Theoremprovers Isabelle in HOL auf. Diese Wissensbasis umfasst bereits die grundlegenden Bereiche der Mittelschul-Mathematik; eine große Anzahl von Mathematikern erweitert sie in rasantem Tempo.
(2) Probleme erfassen den Anwendungsaspekt der Mathematik: Ein 'Problem' besteht darin, aus gegebenen Objekten neue Objekte zu konstruieren,

[1] Zum Beispiel `http://isabelle.in.tum.de/library/HOL/index.html`

die gewünschten Eigenschaften entsprechen. Die Objekte werden mithilfe der Sprache (Funktionskonstanten und Pädikate) beschrieben, die in den Theorien definiert ist. Probleme sollen vom Benutzer interaktiv spezifiziert werden können, und ebenso soll es möglich sein, dass das System automatisch ein vage spezifiziertes Problem zutreffend verfeinern kann.
(3) Methoden beschreiben die Algorithmen, die die jeweiligen Probleme lösen. Zur Beschreibung der Methoden verwendet *ISAC* eine Erweiterung der Sprache von Isabelle/HOL. Die Semantik dieser Sprache umfasst den Aufbau eines 'Beweisbaumes' für die ausgeführte Rechnung; die Konstruktion dieses Baumes erfolgt unsichtbar für den Programmierer der Methode – eine Vereinfachung für den Programmierer und eine Sicherheitsmaßnahme (direkte Manipulation des Baumes würde logische Fehler ermöglichen).

. . . 'weiß' was 'richtig und falsch' ist

Algebra Systeme wurden vor ca. einem Vierteljahrhundert für die Hand von Experten entwickelt, die wissen, was die Systeme tun. Algebra Systeme überlassen daher die Verantwortung dem Benutzer, die gewünschte Korrektheit zu überprüfen. Lernende sind mit dieser Verantwortung überfordert; je näher dem Anfängerstadium der Lernende und je komplexer das verwendete System, desto größer die Überforderung.

ISAC's Konzeption sieht daher einen formal-logischen Rahmen vor, innerhalb dessen Fehler des Studenten erkannt werden. Zum einen verwendet *ISAC* das **Typsystem** von Isabelle/HOL: jedes Objekt der Sprache hat einen 'Typ', und jede Operation und jedes Theorem berücksichtigt diesen Typ. Zum Anderen schafft die **Spezifikation** einer Aufgabenstellung den konkreten Rahmen, um Fehler zu erkennen. Die Spezifikation umfasst die Angabe einer geeigneten Theorie, eines Problems und einer Methode (also einen 3-dimensionalen, eindeutigen Verweis in die Wissensbasis). Hinter den Rechnungen auf dem Front-end baut *ISAC* einen **Beweisbaum** auf, der bestimmten logischen Strukturregeln entsprechen muss.

Eine (von einem Autor vorbereitete und vor dem Studenten normalerweise versteckte) **Formalisierung** stellt zusammen mit der Spefikation das Wissen dar, mithilfe dessen *ISAC* eine Aufgabe automatisch lösen kann.

. . . unterstützt schrittweises, formales Begründen

Unter der gesamten Flut von Mathematik-(Lern-)Software gibt es weltweit nur *ein einziges* Produkt[2] [Bee84], das schrittweises Bearbeiten von Re-

[2] www.matxpert.com

chengängen unterstützt (aber nicht die Spezifikationsphase, und damit auch nicht Teilen in Subprobleme und Problemlösungen der angewandten Mathematik — siehe oben).

Auf ISAC's elektronischem Arbeitsblatt kann der Benutzer Formeln und Taktiken eingeben. ISAC's Mathematik-Maschine antwortet auf die Eingabe stereotyp mit der resultierenden Formel und der zunächst anwendbaren Taktik. Die Antwort des Mathematik-Maschine wird jedoch durch ISAC's Dialogkomponente in flexibler Weise abgewandelt dem Benutzer präsentiert: Die Dialogkomponente kann die resultierende Formel als 'Lückentext' zum Ausfüllen vorgeben, oder sie kann eine Liste von Taktiken anbieten, aus der der Benutzer auszuwählen hat, etc.

. . . ist transparent punkto Funktionsweise

Ein Mathematik-(Lern-)System soll nicht nur transparent punkto Wissensbasis sein, sondern auch punkto Funktionsweise. Der Blick in die transparenten Mechanismen des Systems soll vor allem das schrittweise, formale Begründen unterstützen.

Diese Anforderung birgt Schwierigkeiten für die Implementierung von ISAC's Wissensbasis, denn Algebra Systeme bevorzugen generelle Methoden vor speziellen in ihren Implementationen, und kümmern sich dabei natürlich nicht um das Anliegen von Lernsystemen, nämlich sich möglichst auf elementare Methoden zu beschränken.

Das Re-engineering für Lernzwecke muss also die Standardmethoden, die Algebra Systeme zum Funktionieren bringen, elementarisieren – wenn nötig auch auf Umwegen.

. . . ist erweiterbar und anpassbar

Ein Mathematik-(Lern-)System darf kein perfektes, abgeschlossenes Produkt sein, weder für Schüler noch für Lehrer!
Folgende Komponenten sollen erweiterbar und anpassbar sein:
Die Wissensbasis ist transparent, wie erwähnt, mehr oder weniger lesbar für den Benutzer und getrennt vom Wissensinterpreter. Das sind auch die Voraussetzungen für beliebige Erweiterbarkeit.
Die Beispielsammlung ist im Allgemeinen vorbereitet, um im Tutoring vollständige Benutzerführung zu gewährleisten. Jede vorhandene Beispielsammlung wird in ISAC übertragen werden können, ohne deren Formatierung zu ändern.
Zusätzliche Erklärungen zu allen Elementen der Wissensbasis sind

möglich. Vom grundsätzlichen Design her sind die Elemente der Wissensbasis untereinander automatisch dicht verlinkt. Zusätzlich dazu sind Erklärungen beliebigen Typs zu allen Elementen der Wissensbasis möglich.
Der Dialog soll an verschiedene Lernstrategien, Fertigkeits- und Wissensstufen, sowie an unterschiedliche Formen der Lernorganisation vom verantwortlichen Pädagogen angepaßt werden können. Als Voraussetzung dafür verwaltet $\mathcal{ISAC}$ ein Usermodel.

Diese fünf Punkte wurden unter rein pädagogischen Gesichtspunkten formuliert. Man könnte sich jedoch vorstellen, dass einige der Punkte Auswirkungen auf die künftige Konzeption von Mathematiksoftware im Allgemeinen haben: Wäre es einem Ingenieur nicht viel leichter, die Verantwortung für die Verwendung einer softwaregenerierten Zahl zu übernehmen, wenn die betreffende Software transparent wäre und die Software das Zustandekommen dieser Zahlen nachvollziehen ließe?

Technische Aufgabenstellung

Die Aufgabe, die obgenannten Ziele technisch zu realisieren, impliziert die Anwendung von Konzepten und Techniken aus verschiedenen Disziplinen. Die entsprechenden Konzepte und Techniken sind im Grobdesign [Neu01c] von $\mathcal{ISAC}$ spezifiziert, ihre prinzipielle Eignung ist geklärt. In einigen Punkten werden die derzeitigen Grenzen von Technologie und Wissenschaft berührt [Neu01a]. Folgende Wissenschaftsdiziplinen sind in die Lösung der nachfolgend genannten Aufgaben involviert:

1. **Computermathematik**
   (a) **Die Definition der formalen Sprachen** zur Beschreibung des mathematischen Wissens sowie der konkreten Aufgaben der (angewandten) Mathematik obliegen der formalen Logik. $\mathcal{ISAC}$ verwendet Isabelle/HOL [Pau97] mit spezifischen Ergänzungen als Objektsprache, und nur für unvermeidlichen 'glue code' SML [MTHM97], die Metasprache von Isabelle und $\mathcal{ISAC}$.
   (b) **Die Struktur der Wissensbasis** entlang der beschriebenen Achsen Theorien, Probleme und Methoden ist neu, erscheint aber unproblematisch; die Struktur innerhalb der Achsen ist nicht

so klar. Theorien übernimmt *ISAC* ohne Änderung von Isabelle [Pau89]; die Strukturierung der Probleme ist im Wesentlichen neu zu entwickeln (Anfänge in [GL02]); eine Strukturierung der Methoden-Achse liegt derzeit jenseits des Erforschten.

(c) **Die meta-mathematischen Grundlagen** für die Mechanisierung des deduktiven Aspekts scheinen geklärt (in den Theorem-Provern), nicht jedoch für den anwendungsorientierten Aspekt: es fehlt ein 'calculus of solving'; erste Ansätze dafür könnten in [Far01] vorliegen.

(d) **Die Inhalte der Wissensbasis** werden nach etablierten Methoden und Konzepten der 'Symbolic Computation' implementiert. Die Erfordernisse der Interaktivität verlangen jedoch mehrmals eine neuartige Verwendung der Methoden (siehe [Kar02]).

2. **Softwaretechnologie**

(a) **Die Interpretation der Sprachen** wird durch die Erfordernisse der Interaktivität von *ISAC* vor neuartige Aufgaben gestellt; insbesonders gilt dies für *ISAC*'s Sprache zur Beschreibung der Methoden [Neu01c].

(b) **Die Softwarearchitektur** eines interaktiven, verteilten, www-basierten Mehrbenutzer-Systems wie *ISAC* (Tutoring- sowie Authoring-System) zu gestalten, ist eine Herausforderung, der *ISAC* durch Verwendung der Dinopolis-Middelware [Sch02] begegnet.

(c) **Das Dialogdesign** für den 'Arbeitsplatz des Mathematik-Studenten / das Mathematik-Autors' steht vor der Aufgabe, komplexe Zusammenhänge (die aus *ISAC*'s rigoros formal-logischem Rahmen resultieren) klar dazustellen und einfach handhabbar zu machen. Erste Schritte sind in [Kre02] beschrieben.

(d) **Die Wissens-Präsentation** übersichtlich und intuitiv zu gestalten, ist angesichts der hoch-komplexen und dicht-vernetzten Wissensbasis eine fordernde Aufgabe. Zu ihrer Bewältigung stehen relativ wenige Standardwerkzeuge zur Verfügung. Erste Überlegungen sind in [Gri02] zu finden.

3. **Lernpsychologie und Didaktik** werden eingebunden, sobald *ISAC* soweit gediehen ist, dass die Wissensbasis für praktischen Einsatz in der Lehre hinreicht und das Front-end eine erste Erfahrung für das vermutlich sehr neue 'touch und feel' erlaubt.

(a) Um **das Usermodel** für *ISAC* zu erstellen, wird auf Standardtechniken zurückgegriffen. Die neuartige Funktionalität und hochgradige Flexibilität von *ISAC* wird jedoch spezielle Anforderungen stellen.

(b) **Die Dialogstrategien** bauen auf das Usermodel und ebenso auf die Flexibilität von *ISAC*. Eine Herausforderung wird sein, eine einfache Sprache zur Beschreibung der vielen Möglichkeiten zu finden.

1.2 Organisatorische Realisierung

ISAC hat seinen Ursprung in einer Projektinitiative des 'Institute for Softwaretechnology of the United Nations University', UNU/IIST in Macao während der Jahre 1992/93 [3] mit dem Ziel einer Neukonzeption von Mathematiklernsoftware. Dieses Projekt wurde nicht zu Ende geführt, doch die Kontakte waren hergestellt - insbesonders zwischen den vorgesehenen österreichischen Teilnehmern, dem Institut für Softwaretechnologie der TU Graz und dem 'Research Institute for Symbolic Computation' der Universität Linz.

Mittlerweise hatten sich österreichische Pädagogen zum Thema 'Computereinsatz im Unterricht' organisiert, und zwar im 'Austrian Center for Didactics of Computer Algebra'[4] (ACDCA) für die Allgemeinbildenden Höheren Schulen, und die 'Arbeitsgemeinschaft Moderner Mathematikunterricht'[5] (AMMU) für die Berufsbildenden Höheren Schulen in Österreich. Beide Institutionen wurden schnell als Sammelpunkte für aufgelaufene Erfahrungen wirksam (etwa [Neu01b]), das ACDCA machte die österreichische Vorreiterrolle auf diesem Gebiet durch Organisation von Konferenzen[6] international bekannt, und trat durch wissenschaftliche Veröffentlichungen [HK98, KW00] hervor.

Auf diesen beiden Voraussetzungen, den wissenschaftlichen Kontakten und den praktischen Lehrerfahrungen aufbauend, wurde im *ISAC*-Projekt Mitte der 90-er Jahre begonnen, die grundlegenden Konzepte zu entwerfen. 1999/2000 entstand am Institut für Softwaretechnologie der TU Graz ein Prototyp[7], der die Realisierbarkeit der Konzepte belegte.

[3] `http://www.iist.unu.edu/home/Unuiist/newrh/I/3/2/page_7.html`

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

[5] `http://www.ammu.at`

[6] Teaching Mathematics with Derive, Krems 1992; Derive in Education, Krems 1993; ACDCA 5th Summer Academy, Gösing 1999; VISIT-ME, Wien 2002

[7] `http://www.ist.tugraz.at/projects/isac/`

Anschließend begann die Vergrößerung des Entwicklerteams und die Ausarbeitung der Detailkonzepte. Das $\mathcal{ISAC}$-Projekt steht derzeit am Ende der Entwurfsphase und soll auf breiterem organisatorischen Rahmen in die Implementierungsphase starten. Die Hoffnung ist, mit einem Aufwand von ca. 20 Mannjahren im akademischen Bereich ein hochqualitatives 'Open Source' Produkt zu erstellen, das jedem zum Erlernen von Mathematik zur Verfügung steht.

1.3 Positionierung der Diplomarbeit in $\mathcal{ISAC}$

Die vorliegende Diplomarbeit ist ein Teilprojekt von $\mathcal{ISAC}$. Zuerst wird dieses Projekt als Teil der gesamten technischen Aufgabenstellung positioniert, wie sie in 1.1.2 vorgestellt wurde. Dann wird die Zielsetzung der Teilaufgabe hinsichtlich der angestrebten Funktionalität, wie sie in 1.1.2 für $\mathcal{ISAC}$ insgesamt festgelegt ist, erklärt.

1.3.1 Position in der Gesamtaufgabe

Die Aufgabenstellung der vorliegenden Diplomarbeit betrifft ausschließlich Konzepte und Techniken aus der Computermathematik, insbesonders die Punkte (1a), (1b) und (1d).

ad (1a):
Die für das Bruchrechnen notwendigen Terme erster Ordnung werden mithilfe der in Isabelle/HOL vorgegebenen Termstruktur dargestellt. Für die Operationen unter den in der Diplomarbeit erarbeiteten Algorithmen werden die Terme jedoch in eine geeignetere Darstellung transformiert (siehe Kapitel 4). Die Algorithmen selbst werden in SML beschrieben; dadurch sind sie für den Benutzer nicht 'transparent' - um die gewünschte Transparenz zu erreichen, muss ein Umweg gewählt werden ('reverse rewriting').

ad (1b):
Die für das Bruchrechnen notwendigen Definitionen (reelle Zahlen und die zugehörigen Operationen) und Theoreme sind in der aktuellen Version von Isabelle/HOL bereits implementiert. Die wenigen, technisch begründeten Ergänzungen werden in einer einzigen Theorien-Datei, *Rationals.thy* zusammengefasst. Die erarbeiteten Algorithmen und Datenstrukturen stehen in *Rationals.ML*.

ad (1d):
Die Grundbausteine der Methoden sind vor allem Mengen von Umschreibe-Regeln ('rewrite rules'). Der zentrale Algorithmus dieser Arbeit, der Euklidische Algorithmus, wird nicht als Regelmenge dargestellt. Daher ergibt sich die Aufgabe, die Funktionalität von Regelmengen herzustellen, wie zum Beispiel in `norm_rational` (siehe Kapitel 4).

1.3.2 Bezüge zur Zielsetzung

Die vorliegende Diplomarbeit soll in mehrfacher Weise dazu beitragen, die in (1.1.2) überblicksmäßig angegebene Zielsetzung zur Funktionalität von *ISAC* zu erreichen. Die Anforderungen detaillieren sich folgendermaßen unter den genannten fünf Teilzielen:

Transparenz punkto Wissensbasis:
Ideal wäre es, wenn man das Wissen zum Bruchrechnen in einer Form ablegt, die sowohl der Benutzer lesen als auch das System mechanisch interpretieren kann. Letzteres ist zur Zeit aus technisch-mathematischen Gründen nicht möglich: Wie bereits erwähnt, wird der für die vorliegende Arbeit zentrale Euklidische Algorithmus (in entsprechender Verallgemeinerung) in SML implementiert. Er ist daher für den Benutzer nicht transparent - der Umweg zur Transparenz, 'reverse rewriting' wird unten nochmals angesprochen.

Entscheidend ist, dass alle weiteren Operationen auf Brüchen, wie Kürzen, Addieren und Vereinfachen, mithilfe von 'rewrite rules' beschrieben sind. Die wesentlichen dieser 'rewrite rules', also der Theoreme für das Bruchrechnen, sind in Isabelle/HOL bereits bewiesen und können mithilfe eines HTML-Browsers eingesehen werden.

Formal-logischer Rahmen ('richtig und falsch'):
Beim Rechnen mit Brüchen muss man immer darauf achten, dass der Nenner nicht 0 ist, da sonst der Wert des Bruches nicht definiert wäre. Beim 'Kürzen' nimmt man daher an, dass der Term, der gekürzt wird, nicht 0 sein darf. Tritt also das Kürzen in einer Berechnung auf, bei der man ein Ergebnis erhält, das den gekürzten Term 0 werden lässt, dann ist das Ergebnis nicht zulässig. Die erarbeiteten Algorithmen müssen also diese Annahmen an *ISAC* weitergeben, das sie auf der entsprechenden logischen Ebene berücksichtigt.

Unterstützung schrittweisen, formalen Begründens:
Alle Algorithmen, die mit Rewriting implementierbar sind, können Schritt[8] für Schritt den Rechenweg von der Angabe bis zum Ergebnis darstellen und damit die Richtigkeit begründen. Bei Algorithmen, die (noch) nicht mit Rewriting realisierbar sind (zum Beispiel die Algorithmen aus Kapitel 2 und 3), wird eine Art Backtracking mit versteckter Zusatzinformation verwendet um auf die nötigen Zwischenschritte zu kommen.

Die Implementierung dieses 'reverse rewriting' liegt außerhalb des Rahmens der Diplomarbeit. Diese stellt jedoch die notwendigen Funktionen bereit; die Verwendbarkeit dieser Funktionen für das 'reverse rewriting' wird spezifiziert.

Transparenz punkto Funktionsweise:
ISAC's Programmiersprache ist (noch) zu wenig mächtig, um den Euklidischen bzw. modularen Algorithmus zur Berechnung des größten gemeinsamen Teiler zu formulieren (und damit der Interaktion und der Steuerung durch den Benutzer zugänglich zu machen). Aber, wie bereits erwähnt, beim Bruchrechnen kann ein großer Teil der Algorithmen durch Rewriting realisiert werden, diese sind natürlich transparent.

Erweiterbarkeit und Anpassbarkeit:
Die implementierten Algorithmen gehören zum Kern der Wissensbasis und müssen in jedem Fall mitgeliefert werden. Andere Algorithmen und andere Autoren bauen auf diesen Algorithmen auf.

1.4 Voraussetzungen, Begriffe und Notation

1.4.1 Voraussetzungen

Diese Arbeit setzt mathematisches Grundverständnis sowie grundlegende Programmierkenntnisse voraus. Es wird aber in jedem Abschnitt Literatur genannt, mit deren Hilfe man sich die Grundlagen erarbeiten kann.

[8] Schritt = Anwendung einer Umschreiberegel ('rewrite rule')

1.4.2 Begriffe und Notation

Es gibt vier grundsätzliche Operationen, die für das Bruchrechnen notwendig sind:

- Erweitern und Kürzen
- Addieren von Bruchtermen
- Multiplizieren von Bruchtermen
- Mehrfach Brüche in einfache Brüche umwandeln

In dieser Arbeit werden diese gebräuchlichen Bezeichnungen in durchgängiger Weise verwendet und im Folgenden genauer erklärt.

Erweitern und Kürzen von Bruchtermen: Beim Erweitern bzw. Kürzen wird im Zähler und Nenner des Bruches mit einem gleichen Term ($\neq 0$) multipliziert.

$$\frac{a}{b} = \frac{a \cdot c}{b \cdot c}$$

Das Ziel beim Kürzen ist es, den Bruch auf seine "einfachste Form" zu bringen. Um das zu erreichen wird der größte gemeinsame Teiler (ggT) von Zähler und Nenner berechnet und dann sein Inverses mit Zähler und Nenner multipliziert. Der Bruch ist dann nicht mehr kürzbar, denn würde es noch einen Term geben, durch den Zähler und Nenner teilbar sind, dann wäre der zuerst berechnete Teiler nicht der größte, was aber im Widerspruch zur Definition des größten gemeinsamen Teilers steht.

$$\frac{a}{b} = \frac{a \cdot \frac{1}{\mathrm{ggT}(a,b)}}{b \cdot \frac{1}{\mathrm{ggT}(a,b)}}$$

Für diese Diplomarbeit stellt sich die Frage: "*Wie berechnet man den größten gemeinsamen Teiler?*"

Addieren von Bruchtermen: Beim Addieren von zwei Brüchen muss man die Nenner der beiden Bruchterme in Betracht ziehen. Sind beide Nenner gleich, dann spricht man von gleichnamigen Bruchtermen und man kann die Zähler einfach addieren.

$$\frac{a}{b} + \frac{c}{b} = \frac{a + c}{b}$$

Sind die beiden Nenner verschieden, dann spricht man von ungleichnamigen Bruchtermen. Hier muss man vor dem Addieren die Bruchterme auf gleichen Nenner bringen. Dies geschieht indem man sie auf den Hauptnenner (=kleinstes gemeinsames Vielfaches (kgV) der beiden Nenner) erweitert.

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot \frac{\mathrm{kgV}(b,d)}{b}}{b \cdot \frac{\mathrm{kgV}(b,d)}{b}} + \frac{c \cdot \frac{\mathrm{kgV}(b,d)}{d}}{d \cdot \frac{\mathrm{kgV}(b,d)}{d}} = \frac{a \cdot \frac{\mathrm{kgV}(b,d)}{b} + c \cdot \frac{\mathrm{kgV}(b,d)}{d}}{\mathrm{kgV}(b,d)}$$

Auch hier stellt sich die Frage: "*Wie berechnet man das kleinste gemeinsame Vielfache?*"

Multiplizieren von Bruchtermen: Bruchterme werden multipliziert indem man das Produkt der Zähler durch das Produkt der Nenner dividiert.

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

Doppelbruchterme in einfache Bruchterme umwandeln: Bruchterme, deren Zähler oder Nenner (mindestens) einen Bruchterm enthalten, werden Doppelbruchterme genannt. Ein Doppelbruchterm wird aufgelöst, indem man das Produkt der Außenglieder in den Zähler und das Produkt der Innenglieder in den Nenner schreibt.

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a \cdot d}{b \cdot c}$$

Wie man leicht sieht, sind alle Operationen bis auf kgV und ggT mit einfachen Rechenregeln beschreibbar. Diese einfachen Regeln können als Rewriting-Regeln implementiert werden. Es bleibt nur mehr die Frage nach den Algorithmen für ggT und kgV.

Da der folgende Zusammenhang gilt

$$a \cdot b = \mathrm{kgV}(a,b) \cdot \mathrm{ggT}(a,b)$$

kann das kgV auf den ggT zurückgeführt werden. Es ist daher nur ein Algorithmus zur Berechnung des ggTs notwendig, um das Bruchrechnen in $\mathcal{ISAC}$ implementieren zu können.

Noch ein Begriff ist wichtig für die praktische Anwendung der vorgestellten Algorithmen in $\mathcal{ISAC}$: der Begriff der *Normalform* [BN98]. Wenn in einem Bruchterm alle Brüche so addiert und multipliziert werden, dass nur mehr ein

einziger Bruchstrich auftritt, und der resultierende Bruch vollständig gekürzt ist, dann ist dieser Bruch eine Normalform.

Die praktische Bedeutung dieser Normalform liegt darin, dass beliebige Brüche dadurch auf Äquivalenz geprüft werden können, dass man sie in ihre Normalformen trasformiert, und dann feststellt, ob die Normalformen ident sind (siehe Kapitel 4). [9]

1.5 Struktur der Kapitel

Die Kapitel der vorliegenden Diplomarbeit sind folgendermaßen strukturiert:

Das 2. Kapitel enthält Allgemeines über univariate Polynome und deren ggT-Berechnung, angefangen von der Berechnung über Restfolgen bis hin zur modularen Methode.

Das 3. Kapitel deckt das analoge Gebiet (wie in Kapitel 2) für multivariate Polynome ab. Dabei wird auf die Problematik einer Verallgemeinerung des euklidschen Algorithmus für multivariate Polynome eingegangen und als Lösung der modulare Zugang erklärt.

Das 4. Kapitel beschreibt die Implementierung und Einbettung in *ISAC* mit den wichtigsten Algorithmen im originalen SML-Code.

Im Anhang sind wichtige Definition und Sätze zum mathematischen Hintergrund zu finden. Weiters werden auch einige allgemeine Befehle zum Pseudocode, der in den Kapiteln 2 und 3 verwendet wird, erklärt.

[9] Terme mit transzendenten Funktionen haben keine Normalformen; man kann bei solchen Termen daher im Prinzip Äquivalenz *nicht* feststellen.

2. POLYNOME IN EINER VARIABLE

2.1 Definition und Darstellung

Es sei R ein Ring und x eine Unbestimmte über R. Eine (formal) unendliche Summe

$$P(x) = a_0 + a_1x + a_2x^2 + \dots$$

mit $a_i \in R$ und $a_i = 0$ für alle bis auf endlich viele i wird Polynom über R genannt. Die Menge der Polynome über R wird mit $R[x]$ bezeichnet. Bis auf das Polynom $0(x) = 0 + 0x + 0x^2 + \dots$ gibt es zu jedem Polynom ein eindeutig bestimmtes $j \geq 0$, so dass $a_j \neq 0$ aber $a_k = 0$ für alle $k > j$ gilt. Dieses j nennt man Grad von $P(x)$, und es wird mit $\deg(P)$ bezeichnet.

Es seien die Polynome $P_1(x) = a_1 + a_2x + a_3x^2 + \dots$ und $P_2(x) = b_1 + b_2x + b_3x^2 + \dots$ Elemente aus $R[x]$. Diese Polynome sind gleich, wenn ihre Koeffizienten gleich sind, d.h., für alle $i = 1, 2, 3, \dots$ gilt $a_i = b_i$.

Wenn R ein Ring ist, in dem zu je zwei Elementen ihr größter gemeinsamer Teiler existiert, also R zum Beispiel ein ZPE-Ring[1] ist, dann wird der größte gemeinsame Teiler der Koeffizienten eines Polynoms $P(x) \in R[x]$ der Inhalt (Content - $\text{cont}(P)$) von $P(x)$ genannt. Dieser ist, so wie jeder größte gemeinsame Teiler, nur bis auf Assoziierte eindeutig bestimmt, d.h., es gibt eine Einheit u, so dass für zwei ggTs d_1 und d_2 die Gleichung $d_1 = u \cdot d_2$ gilt. Ist der Inhalt von P eine Einheit aus R, dann nennt man das Polynom $P(x)$ primitiv.

Das Polynom $\text{pp}(P) = \frac{P(x)}{\text{cont}(P)}$ gehört für $P(x) \neq 0$ zu $R[x]$ und ist primitiv. Es wird der primitive Teil von $P(x)$ genannt.

Aus der Algebra ist bekannt, dass, wenn $(R[x], +, \cdot)$ ein Ring ist und R ein Einselement besitzt, sich dann die Eigenschaften der Kommutativität bzw. der Nullteilerfreiheit auf $R[x]$ übertragen. [Fri96, Kapitel 15]

[1] siehe Abschnitt A.2

$(R[x], +, \cdot)$ wird Polynomring genannt. Im Weiteren wird immer ein kommutativer Ring mit Einselement vorausgesetzt.

Man kann Polynome in zwei verschiedenen Listendarstellungen angeben:

- Dichte Darstellung: $P(x)$ wird als Liste $(n, a_0, \ldots, a_n)$ dargestellt.
- Dünne Darstellung: $P(x)$ wird als Liste $((i_1, a_1), \ldots, (i_k, a_k), (n, a_n))$ dargestellt, wobei $a_j \neq 0$ für $j = i_1, \ldots, i_k, n$ mit $0 \leq i_1 < \cdots < i_k < n$ gilt und $a_j = 0$ sonst.

2.2 Operationen mit univariaten Polynomen

2.2.1 Addition

Die Summe von $P_1(x)$ und $P_2(x)$ ist definiert als

$$P(x) = c_0 + c_1 x + c_2 x^2 + \ldots \ ,$$

wobei für alle $i = 0, 1, 2, \ldots$ gilt $c_i = a_i + b_i$. Der Algorithmus für die Addition und Subtraktion ist eigentlich offensichtlich: Die Koeffizienten von gleichen Potenzen müssen addiert bzw. subtrahiert werden.

Input: $P_1(x), P_2(x) \in K[x]$.

Output: $P(x) \in K[x]$ mit $P(x) = P_1(x) + P_2(x)$
(Die Polynome sind in dichter Listendarstellung gegeben.)

Listing 2.1: addpoly

```
addpoly(p1,p2)
{
    m = first(p1);
    n = first(p2);
    p1 = red(p1);
    p2 = red(p2);
    o = max(m,n);
    while length(p1)>0 and length(p2)>0 do
        {
            p  = first(p1)+first(p2);
```

```
        p1 = red(p1);
        p2 = red(p2);
    }
if length(p1)>0 then p = conc(p,p1)
else p=conc(p,p2);
return conc(o,p);
}
```

2.2.2 Multiplikation

Die klassische Methode, um zwei univariate Polynome $P_1 = \sum_{i=0}^{m} p_i \cdot x^i$ und $P_2 = \sum_{j=0}^{n} q_j \cdot x^j$ zu multiplizieren, ist durch die Formel

$$P_1 \cdot P_2 = \sum_{l=0}^{m+n} (\sum_{i+j=l} p_i \cdot q_j) x^l$$

gegeben.

2.2.3 Division

Es sei R ein Ring und

$$P_1 = \sum_{i=0}^{n} a_i x^i \in R[x]$$

ein Polynom, dessen Koeffizient a_n eine Einheit in R ist. Dann gibt es zu jedem $P_2 \in R[x]$ eindeutig bestimmte Polynome $Q(x), R(x) \in R[x]$ mit

$$P_2(x) = Q(x)P_1(x) + R(x)$$

mit $\deg(R) < \deg(Q)$. [Bos99, S.30]

Für den einfachen Fall, das der Polynomring über einen Körper K betrachtet wird, ist folgender Satz hilfreich:

Satz 2.2.1. *Es sei K ein Körper, dann ist $K[x]$ ein euklidischer Ring, wobei der Grad die euklidische Norm ist. [Pet99, Satz 6.3]*

Daher kann der intuitive Divisions-Algorithmus angewendet werden.

Der folgende Algorithmus beschreibt die Division von Polynomen in einem Körper $K[x]$:

Input: $P_1(x), P_2(x) \in K[x]$, $P_2(x) \neq 0$, $\deg(P_1) \geq \deg(P_2)$.

Output: $Q(x), R(x) \in K[x]$ mit $P_1(x) = P_2(x)Q(x) + R(x)$ und $\deg(R) < \deg(P_2)$ oder $R(x) = 0$
(Die Polynome sind in dichter Listendarstellung gegeben.)

Listing 2.2: pdivk

```
pdivk(p1,p2)
{
  m=first(p1);
  n=first(p2);
  mn=m−n;
  p1‘=inv(red(p1));
  p2‘=inv(red(p2));
  lc2=first(p2‘);
  Q={};
  while(m >= n)
  {
    c=first(p1‘)/lc2;
    q=comp(c,q);
    p1‘=red(p1‘−c∗p2‘);
                          // multiplication and subtraction
    m=m−1;                // by coordinates
  }
  q=comp(mn,inv(q));
  while (first(p1‘)=0)
  {
    p1‘=red(p1‘);
  }
  r=comp(deg(p1‘),inv(p1‘));
  return {q,r}
}
```

Dieser Algorithmus stimmt nur, wenn R ein Körper ist, da die Division im Allgemeinen nicht zur Verfügung steht. Für Polynome aus $\mathbb{Z}[x]$ stellt das z.B. ein Problem dar. In [Pet99, Satz 6.4] wird eine Lösung angegeben:

Satz 2.2.2. *Es seien I Integritätsbereich und $P_1(x), P_2(x) \in I[x]$ mit $\deg(P_1) = m \geq \deg(P_2) = n$ und $P_2(x) \neq 0$, dann existieren eindeutige Polynome $Q(x)$, $R(x) \in I[x]$ mit*

$$(\mathrm{lc}(P_2))^{m-n+1} \cdot P_1(x) = Q(x)P_2(x) + R(x)$$

und $R(x) = 0$ oder $\deg(R) < \deg(P_2)$.

Die Polynome $Q(x)$ und $R(x)$ werden mit Pseudoquotient ($\mathrm{pquot}(P)$) und Pseudorest ($\mathrm{prest}(P)$) bezeichnet.

Daraus ergibt sich folgender Algorithmus zur Pseudodivision von Polynomen in $I[x]$:

Input: $P_1(x), P_2(x) \in I[x]$, $P_2(x) \neq 0$, $\deg(P_1) \geq \deg(P_2)$.

Output: $Q(x), R(x) \in I[x]$ mit $(\mathrm{lc}(P_2))^{m-n+1} \cdot P_1(x) = Q(x)P_2(x) + R(x)$ und $\deg(R) < \deg(P_2)$ oder $R(x) = 0$
(Die Polynome sind in dichter Listendarstellung gegeben.)

Listing 2.3: pdivi

```
void pdivi(p1,p2)
{
  m=first(p1);
  n=first(p2);
  mn=m-n;
  p1'=inv(red(p1));
  p2'=inv(red(p2));
  lc2=first(p2');
  q={};
  while (m>=n)
  {
    c=first(p1')*lc2^mn;
    q=comp(c,q);
    p1'=red(lc2*p1'-first(p1')*p2'); // multiplication und subtraction
    m=m-1;                           // by coordinates
  }
  q=comp(mn,inv(q));
  while (first(p1')=0)
  {
    p1'=red(p1');
  }
```

```
  r=comp(deg(p1‘),inv(p1‘));
  return {q,r}
}
```

Ein Problem, das bei dieser Version der Pseudodivision sehr schnell auftritt, ist, dass die Zahlen der Koeffizienten sehr groß werden können.

2.3 GGT von univariaten Polynomen

2.3.1 Berechnung mit polynomialen Restfolgen

Wenn K ein Körper ist, dann ist $K[x]$ ein euklidischer Bereich, und der $\mathrm{ggT}(P_1, P_2)$ für $P_1, P_2 \in K[x]$ kann mit Hilfe des euklidischen Algorithmus berechnet werden.

Der Anwendungsfall, dass Polynome P_1, P_2 über einem Integritätsbereich I gegeben sind, zum Beipiel $\mathbb{Z}$, und man den ggT bestimmen soll, tritt sehr häufig auf. [Win96, S. 82]

Daher wird in diesem Abschnitt speziell auf diesen Fall eingegangen. Es werden I einen ZPE-Ring und K den Quotientenkörper von I bezeichnen.

Ein Schlüsselergebnis lieferte Gauss:

Satz 2.3.1. *Es seien* $P_1(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ *und* $P_2(x) = b_0 + b_1x + b_2x^2 + \cdots + b_mx^m$ *aus* $I[x]$ *und primitive Polynome, dann ist* $Q(x) = P_1(x) \cdot P_2(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{m+n}x^{m+n}$ *wieder ein primitives Polynom. [Fri96, Kapitel 20]*

Daraus folgt, dass ggTs und Faktorisierung von Polynomen über I und K grundsätzlich gleich sind. D.h.:

1. Wenn $P_1, P_2 \in I[x]$ primitiv sind und Q ein ggT von P_1 und P_2 in $I[x]$ ist, dann ist Q auch ein ggT von P_1 und P_2 in $K[x]$.
2. Wenn $P \in I[x]$ in $I[x]$ primitiv und irreduzibel ist, dann ist P auch in $K[x]$ irreduzibel.

Nach dieser Folgerung kann die Berechnung von ggTs in $I[x]$ auf die Berechnung von ggTs in $K[x]$ zurückgeführt werden. Betrachtet man bei dieser Reduktion die Komplexität, dann sieht man, dass diese Methode nicht sehr effizient ist, weil die Arithmetik im Quotientenkörper viel aufwendiger als im darunterliegenden Integritätsbereich ist.

Man versucht also Methoden zu finden, die direkt mit dem ZPE-Ring $I[x]$ arbeiten.

Bis auf Multiplikation mit Einheiten kann man jedes Polynom $P(x)$ eindeutig in

$$P(x) = \mathrm{cont}(P) \cdot \mathrm{pp}(P) \tag{2.1}$$

zerlegen, wobei $\text{cont}(P) \in I$ und $\text{pp}(P)$ ein primitives Polynom in $I[x]$ ist.

Zwei von Null verschiedene Polynome $P_1(x), P_2(x) \in I[x]$ heißen ähnlich, wenn $\alpha, \beta \in I[x]\backslash\{0\}$ existieren, so dass $\alpha \cdot P_1(x) = \beta \cdot P_2(x)$. In diesem Fall schreibt man $P_1(x) \simeq P_2(x)$. Wie leicht zu sehen ist, gilt $P_1(x) \simeq P_2(x)$ genau dann, wenn $\text{pp}(P_1) = \text{pp}(P_2)$ gilt. Die Ähnlichkeit ist eine Äquivalenzrelation, die den Grad wahrt.

Man definiert sich nun die polynomiale Restfolgen(PRS):

Definition 2.3.1. *Es seien k eine natürliche Zahl größer 1 und $P_1, P_2, \ldots, P_{k+1}$ Polynome in $I[x]$. $P_1, P_2, \ldots, P_{k+1}$ ist eine polynomiale Restfolge, wenn gilt:*

- $\deg(P_1) \geq \deg(P_2)$,
- $P_i \neq 0$ *für* $1 \leq i \leq k$ *und* $P_{k+1} = 0$,
- $P_i \simeq \text{prest}(P_{i-2}, P_{i-1})$ *für* $3 \leq i \leq k+1$.

In [Win96, Lemma 4.1.2] wird der folgenden Satz, der Aufschluss über den Zusammenhang zwischen PRS und ggT geben soll, bewiesen.

Satz 2.3.2. *Sei $P_1, P_2, P_1', P_2' \in I[x]$, $\deg(P_1) \geq \deg(P_2)$, und $R \simeq \text{prest}(P_1, P_2)$.*

(a) Wenn $P_1 \simeq P_1'$ und $P_2 \simeq P_2'$ dann ist $\text{prest}(P_1, P_2) \simeq \text{prest}(P_1', P_2')$.

(b) $\text{ggT}(P_1, P_2) \simeq \text{ggT}(P_2, R)$.

Daraus ergibt sich der Algorithmus zur Berechnung des ggTs über polynomiale Restfolgen:

Input: $P_1(x), P_2(x) \in I[x]\backslash\{0\}$

Output: $\text{ggT}(P_1, P_2)$

Listing 2.4: GGT-PRS

```
GGT_PRS(P1,P2)
{
  if (deg(P1)>=deg(P2))
  {
```

```
    P1‘=pp(P1);
    P2‘=pp(P2);
  }
  else
  {
    P1‘=pp(P2);
    P2‘=pp(P1);
  }
  d=GGT(cont(P1),cont(P2))

  /*****************************************
  compute P3‘,P4‘,...,Pk‘,Pk+1‘=0 such that
  P1‘,P2‘,...,Pk‘,0 is a prs
  *****************************************/

  return d*pp(Pk‘);
}
```

Wenn also $P_1, P_2, \ldots, P_{k+1}$ eine polynomiale Restfolge ist, dann gilt:

$$\mathrm{ggT}(P_1, P_2) \simeq \mathrm{ggT}(P_2, P_3) \simeq \cdots \simeq \mathrm{ggT}(P_{k-1}, P_k) \simeq P_k.$$

Sind P_1 und P_2 primitiv, dann ist nach dem Gauss'schen Lemma der ggT auch primitiv, d.h. $\mathrm{ggT}(P_1, P_2) = \mathrm{pp}(P_k)$. Daraus folgt, dass man den ggT von Polynomen über den ZPE-Ring I mit diesem Algorithmus GGT-PRS berechnen kann.

Der GGT-PRS Algorithmus ist kein spezieller Algorithmus, sondern eine Gruppe von Algorithmen. Sie unterscheiden sich durch die Bestimmung der polynomialen Restfolge.

Verallgemeinerter Euklidischer Algorithmus

Der einfachste Algorithmus ist der sogenannte verallgemeinerte Euklidische Algorithmus. Man setzt einfach

$$P_i = \mathrm{prest}(P_{i-2}, P_{i-1}) \quad \forall i: \ 3 \leq i \leq k+1, \tag{2.2}$$

um die PRS zu berechnen.

Der große Nachteil dieser Wahl ist, dass die Koeffizienten enorm wachsen, wie im folgenden Beispiel zu sehen ist.

Beispiel:
Wir betrachten Polynome über $\mathbb{Z}$. Als Startpolynome nehmen wir:

$$P_1 = x^8 + x^6 - 3 \cdot x^4 - 3 \cdot x^3 + 8 \cdot x^2 + 2 \cdot x - 5,$$
$$P_2 = 3 \cdot x^6 + 5 \cdot x^4 - 4 \cdot x^2 - 9 \cdot x + 21.$$

Der erweiterte Euklidische Algorithmus erzeugt folgende polynomiale Restfolge:

$$P_3 = -15 \cdot x^4 + 3 \cdot x^2 - 9,$$
$$P_4 = 15795 \cdot x^2 + 30375 \cdot x - 59535,$$
$$P_5 = 1254542875143750 \cdot x - 1654608338437500,$$
$$P_6 = 12593338795500743100931141992187500.$$

Der ggT von P_1 und P_2 ist der primitive Teil von P_6, also 1. Man sieht, dass das Startpolynom und das Ergebnispolynom ganz "einfache" Koeffizienten besitzen, aber die Koeffizienten der Zwischenergebnisse werden sehr groß.

Für den Fall von Koeffizienten aus $\mathbb{Z}$ und univariaten Polynomen steigt nach [Knu81] die Länge der Koeffizienten schlimmstenfalls exponentiell. Für multivariate Polynome ist der Effekt noch viel größer.

Eine andere Möglichkeit ist, die Koeffizienten in jedem Schritt so weit wie möglich zu kürzen, d.h. immer den Inhalt des Zwischenergebnisses zu eliminieren.

Durch

$$P_i = \mathrm{pp}(\mathrm{prest}(P_{i-2}, P_{i-1}))$$

wird die Restfolge erzeugt, die man primitive polynomiale Restfolge nennt.

Das "Kürzen" ist aber sehr aufwendig, denn für jedes Zwischenergebnis muss der Inhalt bestimmt werden. Das bedeutet, dass sehr viele ggT-Berechnungen im Koeffizienten-Ring durchzuführen sind.

Daher ist das Ziel, die Koeffizienten so klein wie möglich zu halten, ohne zu viel Berechnungen durchführen zu müssen. Man setzt also

$$\beta_i P_i = \mathrm{prest}(P_{i-2}, P_{i-1}),$$

wobei β_i, das ein Faktor von

$$\mathrm{cont}(\mathrm{prest}(P_{i-2}, P_{i-1}))$$

ist, erst bestimmt werden soll.

Der bekannteste Algorithmus dieser Form ist der Subresultanten-PRS-Algorithmus von Collins (siehe [BT71]).

Subresultanten-PRS-Algorithmus

Es seien

$$P_1(x) = \sum_{i=0}^{m} a_i x^i \quad \text{und} \quad P_2(x) = \sum_{i=0}^{n} b_i x^i$$

zwei Polynome in $I[x]$ mit Grad m und n, wobei $m \geq n$ gilt.

Definition 2.3.2. *Es sei $M(P_1, P_2)$ die Sylvester-Matrix von P_1 und P_2, d.h.*

$$M(P_1,P_2) = \begin{pmatrix}
a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 & 0 & \dots & \dots & \dots & 0 \\
0 & a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 & 0 & \dots & \dots & 0 \\
 & & & & & \vdots & & & & & & \\
0 & \dots & \dots & \dots & 0 & a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 \\
- & - & - & - & - & - & - & - & - & - & - & - \\
b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0 & 0 & \dots & \dots & \dots & 0 \\
0 & b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0 & 0 & \dots & \dots & 0 \\
 & & & & & \vdots & & & & & & \\
0 & \dots & \dots & \dots & 0 & b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0
\end{pmatrix}.$$

Die Zeilen von $M(P_1, P_2)$ bestehen aus den Koeffizienten der Polynome $x^{n-1}P_1(x)$, $\dots$, $xP_1(x)$, $P_1(x)$ und $x^{m-1}P_2(x), \dots$, $xP_2(x)$, $P_2(x)$, d.h. es gibt n Zeilen mit Koeffizienten aus P_1 und m Zeilen mit Koeffizienten aus P_2. Die Resultante von P_1 und P_2 ist die Determinante von $M(P_1, P_2)$. Um die Subresultante zu bestimmen, entfernt man gewisse Zeilen und Spalten.

Unter $M(P_1, P_2)_{i,j}$ versteht man die Matrix, die entsteht, wenn man die letzten j Zeilen mit Koeffizienten aus P_1, die letzten j Zeilen mit Koeffizienten aus P_2 und die letzten $2j + 1$ Spalten außer der $(m + n - i - j)$-te Spalte entfernt.

Definition 2.3.3. *Die j-te Subresultante von P_1 und P_2 ist als*

$$S_j(P_1, P_2)(x) = \sum_{i=0}^{j} \det(M(P_1, P_2)_{i,j}) x^i$$

definiert und es gilt $\deg(S_j(P_1, P_2)(x)) \leq j$.

In [Win96, S. 87] wird durch folgenden Satz der Zusammenhang zwischen der Kette der Subresultanten der Polynome $P_1(x)$ und $P_2(x)$ und den Elementen der polynomialen Restfolge erklärt.

Es wird folgende Notation verwendet:

$$\begin{aligned} n_i &= \deg(P_i) \quad \forall i: \ 1 \leq i \leq k, \\ \delta_i &= n_i - n_{i+1} \quad \forall i: \ 1 \leq i \leq k-1. \end{aligned}$$

Satz 2.3.3. *Es seien $P_1, P_2 \in I[x]$ und $P_1, P_2, \ldots, P_k, P_{k+1} = 0$ eine polynomiale Restfolge in $I[x]$. Weiters seien $\alpha_i = \mathrm{lc}(P_{i-1})^{\delta_{i-2}+1}$ für $3 \leq i \leq k+1$ und $\beta_i \in I$ so, dass $\beta_i P_i = \mathrm{prest}(P_{i-2}, P_{i-1})$ für $3 \leq i \leq k+1$. Dann gilt für $3 \leq i \leq k$:*

$$\begin{aligned} &S_{n_{i-1}-1}(P_1, P_2) = \gamma_i P_i, \\ &S_j(P_1, P_2) = 0 \quad \forall j: \quad n_{i-1} - 1 > j > n_i, \\ &S_{n_i}(P_1, P_2) = \theta_i P_i, \\ &S_j(P_1, P_2) = 0 \quad \forall j: \quad n_k > j \geq 0, \end{aligned}$$

wobei

$$\gamma_i = (-1)^{\sigma_i} \cdot \mathrm{lc}(P_{i-1})^{1-\delta_{i-1}} \cdot \left(\prod_{l=3}^{i} \left(\frac{\beta_l}{\alpha_l} \right)^{n_{l-1}-n_{i-1}+1} \cdot \mathrm{lc}(P_{l-1})^{\delta_{l-2}+\delta_{l-1}} \right),$$

$$\theta_i = (-1)^{\tau_i} \cdot \mathrm{lc}(P_i)^{\delta_{i-1}-1} \cdot \left(\prod_{l=3}^{i} \left(\frac{\beta_l}{\alpha_l} \right)^{n_{l-1}-n_i} \cdot \mathrm{lc}(P_{l-1})^{\delta_{l-2}+\delta_{l-1}} \right),$$

$$\sigma_i = \sum_{l=3}^{i} (n_{l-2} - n_{i-1} + 1)(n_{l-1} - n_{i-1} + 1),$$

$$\tau_i = \sum_{l=3}^{i} (n_{l-2} - n_i)(n_{l-1} - n_i).$$

Für den Beweis sei auf die Artikel [Bro71, BT71] verwiesen.

Der Satz zeigt, dass die beiden Subresultanten $S_{n_{i-1}-1}(P_1, P_2)$ und $S_{n_i}(P_1, P_2)$ ähnlich zu P_i sind und alle Subresultanten dazwischen verschwinden. Die Folge der Subresultanten von P_1 und P_2 stimmt grundsätzlich mit einer polynomialen Restfolge, die mit P_1 und P_2 beginnt, überein. Diese spezielle polynomiale Restfolge eliminiert einen Großteil des Inhalts von den Zwischenresultaten und benötigt keine Berechnung des ggTs der Koeffizienten.

Wie in [Win96, S. 89] weiter gezeigt wird, gibt es eine sehr effiziente Art diese Folge von Subresultanten zu bestimmen.

Satz 2.3.4. *Es seien $P_1, P_2, \ldots, P_{k+1}$ gleich wie in obigen Satz. Um $\gamma_i = 1$ $\forall i: \ 3 \leq i \leq k+1$, muss man β_i folgend setzen:*

$$\begin{aligned}\beta_3 &= (-1)^{\delta_1+1},\\ \beta_i &= (-1)^{\delta_{i-2}+1} \cdot \mathrm{lc}(P_{i-2}) \cdot h_{i-2}^{\delta_{i-2}} \quad \textit{für } i = 4, \ldots, k+1,\end{aligned}$$

wobei

$$\begin{aligned}h_2 &= \mathrm{lc}(P_2)^{\delta_1},\\ h_i &= \mathrm{lc}(P_i)^{\delta_{i-1}} \cdot h_{i-1}^{1-\delta_{i-1}} \quad \textit{für } i = 3, \ldots, k.\end{aligned}$$

Mit den obigen Koeffizienten β_i erhält man die Subresultanten-polynomial-Restfolge (SR-PRS):

$$\begin{aligned}P_3 &= (-1)^{\delta_1+1} \cdot \mathrm{prest}(P_1, P_2),\\ P_i &= \frac{(-1)^{\delta_{i-2}+1}}{\mathrm{lc}(P_{i-2}) \cdot h_{i-2}^{\delta_{i-2}}} \cdot \mathrm{prest}(P_{i-2}, P_{i-1}) \quad \textit{für } i = 4, \ldots, k.\end{aligned}$$

Der folgende Algorithmus berechnet diese SR-PRS:

Input: $P_1(x), P_2(x) \in I[x]\backslash\{0\}$ und $\deg(P_1) \geq \deg(P_2)$

Output: $PRS = [f_1, f_2, \ldots, P_k]$

Listing 2.5: SR-PRS

```
PRS_SR(P1,P2) {
  PRS=[P2,P1];
  g=1; h=1; P‘=P2; i=3;
  while ((P‘!=0) && (deg(P_)>0))
  {
    d=deg(PRS[1])−deg(PRS[2]);
    P‘=prest(PRS[1],P[2]);
    if (P‘!=0)
    {
      Pi=P‘/(g∗hˆd);
      PRS=comp(Pi,PRS);
      g=lc(Pi);
      h=hˆ(1−d)∗gˆd;
      i=i+1;
    }
```

```
  }
  PRS=inv(PRS);
  return PRS;
}
```

Wendet man diesen Algorithmus auf univariate Polynome über den ganzen Zahlen an, so kann man (laut [Win96, S. 90]) eine obere Schranke für die Länge der Koeffizienten angeben.

Seien die Polynome $P_1(x), P_2(x) \in \mathbb{Z}[x]$ vom Grad m und n und die Koeffizienten seien durch d beschränkt, so sind die Längen der Koeffizienten durch

$$d^{m+n}(m+1)^{\frac{n}{2}}(n+1)^{\frac{m}{2}}$$

begrenzt.

2.3.2 Modulare Berechnung

Eine Alternative für die Berechnung des größten gemeinsamen Teilers zweier Polynome ist dieser modulare Zugang.

Als Motivation betrachten wir folgendes Beispiel:

$$P_1(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5,$$

$$P_2(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21.$$

Wenn $P_1(x)$ und $P_2(x)$ einen gemeinsamen Teiler $H(x)$ besitzen, dann gibt es $Q_1(x), Q_2(x)$, für die gilt:

$$P_1(x) = Q_1(x) \cdot H(x), \quad P_2(x) = Q_2(x) \cdot H(x). \tag{2.3}$$

Diese Relationen bleiben aufrecht, wenn man jeden Koeffizienten in (2.3) modulo 5 reduziert. Danach kann man den ggT von P_1 und P_2 sehr schnell berechnen, da jeder Koeffizient durch 5 beschränkt ist. In diesem Beispiel ist der ggT von P_1 und P_2 modulo 5 gleich 1. Vergleicht man die Grade auf beiden Seiten der Gleichungen in (2.3), so sieht man, dass auch der ggT der ursprünglichen Polynome 1 ist.

In diesem Abschnitt wird versucht eine Verallgemeinerung dieser Methode zu formen und daraus einen Algorithmus zur Berechnung des ggTs zu entwickeln.

In jeder modularen Näherung wird eine Grenze für die Anzahl der Module, die man verwendet, gebraucht (genauer gesagt, für das Produkt dieser Module). In unserem Fall sind die Größen der Koeffizienten von den Polynomen und deren ggT von Bedeutung.

In [Mig74, Mig83] wird die folgende Grenze gezeigt:

Satz 2.3.5. *(Landau-Mignotte Schranke) Seien* $P_1(x) = \sum_{i=0}^{m} a_i x^i$ *und* $P_2(x) = \sum_{i=0}^{m} b_i x^i$ *Polynome über* $\mathbb{Z}$ *mit* $a_m \neq 0$ *und* $b_n \neq 0$*, so dass* $P_2 | P_1$*. Dann gilt:*

$$\sum_{i=0}^{n} |b_i| \leq 2^n \left| \frac{b_n}{a_m} \right| \sqrt{\sum_{i=0}^{m} a_i^2}.$$

Da der ggT von P_1 und P_2 ein Teiler von P_1 und P_2 ist, und sein Grad durch das Minimum der Grade der Polynome beschränkt ist, und weiters der Leitkoeffizient des ggTs ein Teiler von a_m und b_n ist, und daher auch den ggT selbst teilt, gilt:

Folgerung 2.3.1. *Seien* $P_1(x) = \sum_{i=0}^{m} a_i x^i$ *und* $P_2(x) = \sum_{i=0}^{n} b_i x^i$ *Polynome über* $\mathbb{Z}$ *mit* $a_m \neq 0$ *und* $b_n \neq 0$*. Jeder Koeffizient vom* ggT *von* P_1 *und* P_2 *in* $\mathbb{Z}[x]$ *ist beschränkt durch:*

$$2^{\min(m,n)} \cdot \mathrm{ggT}(a_m, b_n) \cdot \min\left(\frac{1}{|a_m|} \sqrt{\sum_{i=0}^{m} a_i^2}, \frac{1}{|b_n|} \sqrt{\sum_{i=0}^{n} b_i^2} \right).$$

Der ggT von $P_1(x) \mod p$ und $P_2(x) \mod p$ muss kein modulares Abbild von einem ganzzahligen ggT sein, wie das folgende Beispiel zeigt:

Seien $P_1(x) = x - 3, P_2(x) = x + 2$. Der ggT über $\mathbb{Z}$ ist 1, aber modulo 5 sind P_1 und P_2 gleich und $\mathrm{ggT}(P_{1(5)}, P_{2(5)}) = x + 2$. Diese Situationen treten nach [Win96, S. 92] aber nicht sehr häufig auf.

Bei der Wahl von Primzahlen p verlangt man die Kommutativität des folgenden Diagramms, wobei $\Phi_p(f(x)) = f(x) \pmod p$ der Homomorphismus von $\mathbb{Z}[x]$ nach $\mathbb{Z}_p[x]$ ist.

$$\begin{array}{ccc} \mathbb{Z}[x] \times \mathbb{Z}[x] & \xrightarrow{\Phi_p} & \mathbb{Z}_p[x] \times \mathbb{Z}_p[x] \\ \mathrm{ggT\ in\ } \mathbb{Z}[x] \downarrow & & \downarrow \mathrm{ggT\ in\ } \mathbb{Z}_p[x] \\ \mathbb{Z}[x] & \xrightarrow{\Phi_p} & \mathbb{Z}_p[x] \end{array}$$

Satz 2.3.6. *Seien $P_1, P_2 \in \mathbb{Z}[x]$ und p eine Primzahl, die nicht die beiden Leitkoeffizienten der Polynome P_1 und P_2 teilt. Weiters seien $P_{1(p)}$ und $P_{2(p)}$ die Reduktionen von P_1 bzw. P_2 modulo p und $P_3 = \mathrm{ggT}(P_1, P_2)$ über $\mathbb{Z}$.*

1. $\deg(\mathrm{ggT}(P_{1(p)}, P_{2(p)})) \geq \deg(\mathrm{ggT}(P_1, P_2))$
2. *Wenn p die Resultante der Polynome $\frac{P_1}{P_3}$ und $\frac{P_2}{P_3}$ nicht teilt, dann ist der* $\mathrm{ggT}(P_{1(p)}, P_{2(p)}) = P_3 \mod p$.

Beweis. 1. Der $\mathrm{ggT}(P_1, P_2) \pmod p$ teilt $P_{1(p)}$ und $P_{2(p)}$, daher teilt er auch den $\mathrm{ggT}(P_{1(p)}, P_{2(p)})$. Daher gilt $\deg(\mathrm{ggT}(P_{1(p)}, P_{2(p)})) \geq \deg(\mathrm{ggT}(P_1, P_2) \mod p)$. Aber p teilt nicht den Leitkoeffizienten von $\mathrm{ggT}(P_1, P_2)$, deshalb ist $\deg(\mathrm{ggT}(P_1, P_2)) = \deg(\mathrm{ggT}(P_1, P_2) \mod p)$.

2. Es sei $P_{3(p)} = P_3 \mod p \neq 0$. Nach der Definition sind $\frac{P_1}{P_3}$ und $\frac{P_2}{P_3}$ relativ prim, daher gilt

$$\mathrm{ggT}(P_{1(p)}, P_{2(p)}) = P_{3(p)} \cdot \mathrm{ggT}\left(\frac{P_{1(p)}}{P_{3(p)}}, \frac{P_{2(p)}}{P_{3(p)}}\right).$$

Wenn der $\deg(\mathrm{ggT}(P_{1(p)}, P_{2(p)})) \neq \deg(P_{3(p)})$ ist, d.h. es gilt nicht $\mathrm{ggT}(P_{1(p)}, P_{2(p)}) \sim P_{3(p)}$, dann ist der ggT der rechten Seite nicht trivial. Daraus folgt $\mathrm{Res}\left(\frac{P_{1(p)}}{P_{3(p)}}, \frac{P_{2(p)}}{P_{3(p)}}\right) = 0$. Aus der Resultantendefinition (siehe Abschnitt 2.3.1) und der damit implizierten Anwendung des Entwicklungsatzes (siehe Abschnitt A.3.2) kann man schließen, dass p die Zahl $\mathrm{Res}\left(\frac{P_1}{P_3}, \frac{P_2}{P_3}\right)$ teilen muss.

□

Der ggT von Polynomen über $\mathbb{Z}_p$ ist nur bis auf die Multiplikation mit Konstanten ($\neq 0$) eindeutig bestimmt. Deshalb versteht man unter $\mathrm{ggT}(P_{1(p)}, P_{2(p)}) = P_3 \pmod p$, dass $P_3 \mod p$ ein ggT von $P_{1(p)}$ und $P_{2(p)}$ ist.

Aus Satz 2.3.6 weiß man, dass es endlich viele Primzahlen p gibt, die die Leitkoeffizienten von P_1 und P_2 nicht teilen, für die aber $\deg(\mathrm{ggT}(P_{1(p)}, P_{2(p)})) > \deg(\mathrm{ggT}(P_1, P_2))$ gilt. Wenn diese Grade übereinstimmen, dann nennt man p eine *glückliche* Primzahl.

Eine Möglichkeit den ggT von zwei ganzzahligen Polynomen P_1 und P_2 zu berechnen, ist die Landau-Mignotte Schranke M zu bestimmen, eine Primzahl p mit $p \geq 2M$, die nicht die Leitkoeffizienten von P_1 und P_2 teilt, zu

wählen, den ggT $P_{3(p)}$ von $P_{1(p)}$ und $P_{2(p)}$ zu berechnen, die Koeffizienten von $P_{3(p)}$ um 0 zu zentrieren (d.h. $\mathbb{Z}_p$ als $\{k|\frac{-p}{2} < k \leq \frac{p}{2}\}$ darzustellen), $P_{3(p)}$ als ganzzahliges Polynom zu interpretieren und zu überprüfen, ob das Polynom P_3 die Polynome P_1 und P_2 teilt. Wenn ja, dann hat man den ggT von $P_{1(p)}$ und $P_{2(p)}$ gefunden, wenn nein, war p eine *unglückliche* Primzahl und man wählt eine andere Primzahl. Weil nur endlich viele *unglückliche* Primzahlen existieren, terminiert dieser Algorithmus und liefert den ggT. Der Nachteil ist, dass p sehr groß werden kann und dann die Operationen mit den Koeffizienten sehr aufwendig werden.

Der folgende Algorithmus wählt verschiedene Primzahlen, berechnet den ggT modulo dieser Primzahlen und kombiniert diese modularen ggTs mittels einer Anwendung des Chinesischen Restsatzes. In $\mathbb{Z}_p[x]$ ist der ggT bis auf die Multiplikation mit Konstanten eindeutig definiert, deshalb ist man hier mit dem sogenannten Leitkoeffizienten-Problem konfrontiert. Der Grund für dieses Problem liegt darin, dass im Allgemeinen bei ganzzahligen Polynomen der ggT keinen Leitkoeffizienten gleich 1 besitzt, wobei in $\mathbb{Z}_p$ der Leitkoeffizient praktisch gewählt werden kann. Bevor man nun den Chinesischen Restsatz anwenden kann, muss man den $\text{ggT}(P_{1(p)}, P_{2(p)})$ normalisieren. Seien a_m, b_n die Leitkoeffizienten von P_1 und P_2. Der Leitkoeffizient des ggTs teilt den ggT der Leitkoeffizienten a_m und a_n. Daher normiert man für primitive Polynome den Leitkoeffizienten von $\text{ggT}(P_{1(p)}, P_{2(p)})$ auf $\text{ggT}(a_m, b_n)$ mod p und am Ende nimmt man den primitiven Teil des Ergebnisses.

Diese Überlegungen führen zu folgendem Algorithmus:

Input: $P_1, P_2 \in \mathbb{Z}[x]$ mit P_1 und P_2 sind primitiv

Output: $Q = \text{ggT}(P_1, P_2)$
(Ganzzahlen modulo m werden als $\{k|\frac{-m}{2} < k \leq \frac{m}{2}\}$ dargestellt)

Listing 2.6: GGT-MOD

```
GGT_MOD(P_1,P_2)
{
  d=gcd(lc(P_1),lc(P_2));
  M=2*d*LMS(P_1,P_2);     // LMS Landau-Mignotte bound for P_1 and P_2
                          // in fact any other bound for the size of
                          // the coefficients can be used
  while(true)
  {
    p=get_next_prime(d);      // a new prime not dividing d
```

```
    c_p=gcdp(P_1,P_2,p);        // calculates the gcd in Zp
                                // with leading coefficient lc(c_p)=1
    Q_p=(d mod p)*c_p;
    if (deg(Q_p)==0)
    {
      Q=1;
      return Q;
    }
    P=p;
    Q=Q_p;
    while(P<=M)
    {
      p=get_next_prime();       // a new prime not dividing d
      c_p=gcdp(P_1,P_2,p);      // calculates the gcd in Zp
                                // with leading coefficient lc(c_p)=1
      Q_p=(d mod p)*c_p;
      if (deg(Q_p)<deg(Q))
      {
        if (deg(Q_p)==0)
        {
          return 1;
        }
        P=p;
        Q=Q_p;
      }
      else
      {
        if (deg(Q_p)==deg(Q))
        {
          Q=apply_CRA2(Q,Q_p,P,p); // applies CRA_2 to the
                                  // coefficients of Q and Q_p
          P=P*p;
        }
      }
    }
    Q=pp(g);
    if (divides(Q,P_1) && divides(Q,P_2)) return Q;
                                // divides(P_1,P_2) tests if P_1|P_2
  }
}
```

Bemerkungen zu den verwendeten Funktionen:
`apply_CRA2(P1,P2,p1,p2)` wendet den Chinesischen Restalgorithmus (`CRA_2`, siehe Satz A.3.5 auf Seite 84) auf die Koeffizienten von `P1` und `P2` an.
`divides(P1,P2)` überprüft, ob `P1` ein Teiler von `P2` ist.
`gcd(c1,c2)` berechnet den ggT von zwei Konstanten `c1` und `c2` ($\in \mathbb{Z}$).
`gcdp(P1,P2,p)` berechnet den ggT von den Reduktionen von `P1` und `P2` modulo `p` im reduzierten Körper $\mathbb{Z}_p$.
`get_next_prime(d)` liefert eine Primzahl ($\in \mathbb{Z}$), die `d` nicht teilt.
`LMS(P1,P2)` berechnet die Landau-Mignotte-Schranke von zwei Polynomen `P1` und `P2` ($\in \mathbb{Z}[x]$).

Für die ggT-Berechnung in $\mathbb{Z}_p$ kann einfach der verallgemeinerte euklidische Algorithmus (siehe Seite 22) verwendet werden. Es muss aber in jedem Schritt darauf geachtet werden, dass die Ergebnisse der Rechenschritte immer modulo p reduziert werden. Da dann die Größe der Zahlen durch p beschränkt ist, treten hier die oben genannten Probleme nicht auf.

Normalerweise werden nicht so viele Primzahlen benötigt, wie die Landau-Mignotte-Schranke angibt, um die ganzzahligen Koeffizienten in GGT-MOD zu bestimmen. Wenn Q sich in einer Folge von Iterationen in der inneren WHILE-Schleife nicht ändert, dann wird im letzten Schritt überprüft, ob man schon fertig ist, wenn ja, dann ist das Ende erreicht und Q wird zurückgeliefert, sonst wählt man eine neue Primzahl und beginnt wieder von vorne.

Beispiel:
Gegeben seien die Polynome

$$\begin{aligned} P_1 &= 10 + 9x + 7x^2 + 8x^3 + 3x^4 + 3x^5, \\ P_2 &= 4 + 2x + 4x^2 + 7x^3 + 4x^4 + 3x^5. \end{aligned}$$

Der ggT der Leitkoeffizienten d ist 3. Die Schranke M für die Primzahlen beträgt ~ 2014. Als erste Primzahl wird 5 gewählt. Die reduzierten Polynome (modulo 5) sind

$$\begin{aligned} P_{1(5)} &= 0 - x + 2x^2 - 2x^3 - 2x^4 - 2x^5, \\ P_{2(5)} &= -1 + 2x - 1x^2 + 2x^3 - x^4 - 2x^5. \end{aligned}$$

Jetzt wird der ggT in $\mathbb{Z}_p$ berechnet, als Ergebnis erhält man $c_{(5)} = 2 - 2x + 2x^2 + x^3$. Da $d \pmod p = 3 \pmod 5 = -2$ ist, ergibt sich $Q_{(5)} = -4 + 4x - 4x^2 - 2x^3$. Also ist $P = 5$ und $Q = -4 + 4x - 4x^2 - 2x^3$.

Als Nächstes wählt man die nächst höhere Primzahl 7. Man erhält die reduzierten Polynome

$$\begin{aligned} P_{1(7)} &= 3 + 2x + 0x^2 + 1x^3 + 3x^4 + 3x^5, \\ P_{2(7)} &= -3 + 2x - 3x^2 + 0x^3 - 3x^4 + 3x^5. \end{aligned}$$

Die ggT-Berechnung in $\mathbb{Z}_p$ liefert $c_{(7)} = -3-1x-3x^2+2x^3$. Aus $d \pmod p = 3$ folgt $Q_{(7)} = -2 - 3x - 2x^2 - 1x^3$. Jetzt wird mit Hilfe des `apply_CRA2`-Algorithmus ein neuer Kandidat für den ggT berechnet:

$$Q = -9 + 4x - 9x^2 + 13x^3.$$

Dieser ist aber kein Teiler von P_1 und P_2.

Daher wird eine neue Primzahl gewählt: $p = 11$. Man erhält die reduzierten Polynome

$$\begin{aligned} P_{1(11)} &= -1 - 2x - 4x^2 - 3x^3 + 3x^4 + 3x^5, \\ P_{2(11)} &= 4 + 2x + 4x^2 - 4x^3 + 4x^4 + 3x^5. \end{aligned}$$

Es ergibt sich $c_{(11)} = 2 + 3x + 2x^2 + 1x^3$ und $Q_{(11)} = -5 - 2x - 5x^2 + 3x^3$. Durch `apply_CRA2` wird der neue Kandidat

$$Q = -61 + 101x - 61x^2 + 162x^3$$

bestimmt. Dieser ist wiederum kein Teiler. Daher wird wieder eine neue Primzahl gewählt, $p = 13$. Man erhält die reduzierten Polynome

$$\begin{aligned} P_{1(13)} &= -3 - 4x - 6x^2 - 5x^3 + 3x^4 + 3x^5, \\ P_{2(13)} &= 4 + 2x4x^2 - 6x^3 + 4x^4 + 3x^5. \end{aligned}$$

Es ergibt sich $c_{(13)} = 2 + 3x + 2x^2 + 1x^3$ und $Q_{(13)} = 6 - 4x + 6x^2 + 3x^3$. Durch `apply_CRA2` wird der neue Kandidat

$$Q = 2 + 3x - 2x^2 + 1x^3$$

bestimmt. Dieser teilt P_1 und P_2 und ist somit auch der ggT.

3. POLYNOME IN MEHREREN VARIABLEN

Die Darstellung von Polynomen in einem Computeralgebra-System beeinflusst wesentlich die Leistung des Systems. Erwartet man mit Polynomen in wenigen (ein oder zwei) Unbekannten zu arbeiten, dann spielt die Anordnung kaum eine Rolle; sonst hängt die bevorzugte Anordnung von der geplanten Anwendergruppe ab.

3.1 Definition und Darstellung

In Kapitel 2 wurde zu einem Ring R der Polynomring $R[x]$ betrachtet. Durch Iteration kann man den Polynomring in n Variablen $x_1, \ldots, x_n$ über R konstruieren:

$$R[x_1, \ldots, x_n] := (\ldots((R[x_1])[x_2])\ldots)[x_n].$$

Es ist auch möglich die meisten Definitionen aus dem Abschnitt 2 in direkter Weise auf den Fall mehrerer Variablen zu verallgemeinern. Ist $P(x_1, \ldots, x_n) \in R[x_1, \ldots, x_n]$ und $1 \leq j \leq n$, dann bezeichnet man mit $\deg_{x_j}(P)$ oder kürzer $\deg_j(P)$ den Grad von $P(x_1, \ldots, x_n)$ als Element von $R[x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n][x_j]$.

Es seien $\underline{x} = (x_1, \ldots, x_n)$ und $\underline{e} = (e_1, \ldots, e_n)^T \in \mathbb{Z}^n_{\geq 0}$, dann ist das spezielle Polynom $\underline{x}^{\underline{e}} = x_1^{e_1} \ldots x_n^{e_n}$ ein Element aus $R[\underline{x}]$ und es wird Monom genannt.

Lemma 3.1.1. *Es sei $P(\underline{x}) \in R[\underline{x}]$ kein Nullpolynom, dann gibt es paarweise verschiedene $\underline{e}_1, \ldots, \underline{e}_m \in \mathbb{Z}^n_{\geq 0}$ und von 0 verschiedene $a_1, \ldots, a_m \in R$, so dass*

$$P(\underline{x}) = \sum_{j=0}^{k} a_j \underline{x}^{\underline{e}_j}. \tag{3.1}$$

gilt [Pet99, Lemma 6.1].

Aus dem Lemma folgt eine dünne, distributive Darstellungsmöglichkeit für die multivariaten Polynome. Die Liste

$$(((e_{11},\ldots,e_{1n}),a_1),\ldots,((e_{m1},\ldots,e_{mn}),a_m))$$

ist nämlich $P(\underline{x})$ zugeordnet.

Die Zahl

$$\deg(P)=\max\{e_{j1}+\cdots+e_{jn}:j=1,\ldots,m\}$$

ist der Totalgrad von $P(x)$. Für den univariaten Fall stimmen der Totalgrad von $P(x)$ und der Grad bzgl. x von $P(x)$ überein. Die Anzahl der von Null verschiedenen Koeffizienten von $P(\underline{x})$ wird mit $n(P)$ bezeichnet.

Die Größen $\deg(P)$ und $n(P)$ sind eindeutig definiert. Dagegen ist die Reihenfolge der Monome in der Summe (vgl. Gleichung (3.1)) noch nicht festgelegt. Insbesondere, wenn $n\geq 2$ ist, dann können unterschiedliche Reihenfolgen für verschiedene Anwendungen benutzt werden.

Definition 3.1.1. *Eine Relation $\preceq$ auf einer Menge A heißt eine Wohlordnung, wenn für alle $a,b,c\in A$*

- $a\preceq b$ *und* $b\preceq a\Rightarrow a=b$,
- $a\preceq b$ *und* $b\preceq c\Rightarrow a\preceq c$

gilt und jede nicht leere Teilmenge von A ein kleinstes Element bezüglich $\preceq$ besitzt.

In der Theorie der multivariaten Polynome spielen folgende Wohlordnungen der Monome eine wichtige Rolle:

Es seien $\underline{e}^T=(e_1,\ldots,e_n)$ und $\underline{f}^T=(f_1,\ldots,f_n)$ aus $\mathbb{Z}^n_{\geq 0}$ die Exponenten von zwei Potenzprodukten pp_1 und pp_2.

- *Lexikographische Ordnung:* $pp_1\preceq pp_2$, wenn $\underline{e}=\underline{f}$ gilt oder es ein $1\leq j\leq n$ mit $e_i=f_i$ für alle $1\leq i<j$ und $e_j<f_j$ gibt.
  z.B. $u^2\cdot v^3\cdot x^2\cdot y^4\cdot z^8\preceq u^2\cdot v^3\cdot x^3\cdot y^4\cdot z^8\preceq u^3\cdot v\cdot x\cdot y\cdot z$.
- *Antilexikographische Ordnung:* $pp_1\preceq pp_2$, wenn $\underline{e}=\underline{f}$ gilt oder es ein $1\leq j\leq n$ mit $e_i=f_i$ für alle $j+1\leq i<n$ und $e_j<f_j$ gibt.
  z.B. $u^4\cdot v^2\cdot x^2\cdot y^2\cdot z\preceq u^3\cdot v^3\cdot x^2\cdot y^2\cdot z\preceq u^4\cdot v^3\cdot x^2\cdot y^2\cdot z^2$.

- *Totalgrad und lexikographische Ordnung:* $pp_1 \preceq pp_2$, wenn entweder $\sum_{i=1}^{n} e_i < \sum_{i=1}^{n} f_i$ ist oder im Falle von $\sum_{i=1}^{n} e_i = \sum_{i=1}^{n} f_i$ im Sinne der lexikographischen Ordnung $\underline{e} \preceq \underline{f}$ gilt.

  z.B. $x^5 \cdot y \cdot z \preceq x^2 \cdot y^3 \cdot z^3 \preceq x^5 \cdot y \cdot z^2$.

Es sei $\preceq$ eine Wohlordnung auf $\mathbb{Z}^n_{\geq 0}$ und $\underline{e}_1 \preceq \cdots \preceq \underline{e}_m$. Weiters bezeichnet $\mathrm{lc}(P)$ den Leitkoeffizienten des Polynoms $P(\underline{x})$, das heißt den Koeffizienten des Monoms mit dem größten Exponenten nach der Ordnung $\preceq$. Im univariaten Fall wird immer die natürliche Ordnung verwendet. In den anderen Fällen muss die zugehörige Ordnung angegeben werden.

3.2 Operationen mit multivariaten Polynomen

3.2.1 Addition

Bei der Addition werden ähnlich wie im univariaten Fall einfach die Koeffizienten mit gleichen Potenzprodukten addiert bzw. subtrahiert. Am einfachsten geschieht dies, indem man die Monome nach der gewählten Termordnung sortiert und dann die Potenzprodukte vergleicht.

Der folgende Algorithmus beschreibt eine solche Addition. Hier wird zum Ordnen der Terme die lexikographische Termordnung verwendet. Die Vergleichsoperatoren beziehen sich auch auf diese Termordnung.

Input: $P_1,\ P_2 (\in \mathbb{Z}[x_1, \ldots, x_n]$.

Output: $P \in \mathbb{Z}[x_1, \ldots, x_n]$ mit $P = P_1 + P_2$
(Die Polynome sind in obiger Listendarstellung gegeben.)

Listing 3.1: addpoly

```
addpoly(p1,p2)
  {
   p1‘ = sort_lex(p1); // sorted in lexicographic term order
   p2‘ = sort_lex(p2);
```

```
                    // compares the exponents by lex. term order
  if (first(p1') > first(p2))
     return comp(first(p1'),addpoly(red(p1'),p2'));
  else
     if (first(p1') = first(p2))
        return comp(addmonom(first(p1'),first(p2')),
                    addpoly(red(p1'),red(p2')));
     else
        return comp(first(p2'),addpoly(p1',red(p2')));
 }

// adds two monoms if the exponents are equal
addmonom((c1,e1),(c2,e2))
 {
  if (e1 = e2)
  {
    return (c1+c2,e1);
  }
  else
    raiseException("monoms must have the same exponents to add them");
 }
```

3.2.2 Multiplikation

Beim Multipizieren muss man die Monome des einen Polynoms mit allen Monomen des anderen Polynoms multiplizieren, d.h. man multipliziert die Koeffizienten und addiert die entsprechenden Exponenten. Danach sollten wieder die gleichen Potenzprodukte zusammengefasst werden.

Für die Multiplikation mit einem Skalar kann folgender Algorithmus verwendet werden.

Input: $P_1 \in \mathbb{Z}[x_1, \ldots, x_n]$ und $s \in \mathbb{Z}$.

Output: $P \in \mathbb{Z}[x_1, \ldots, x_n]$ mit $P = P_1 * s$
(Die Polynome sind in obiger Listendarstellung gegeben.)

Listing 3.2: smulpoly

```
smulpoly(p1,s)
```

```
    {
     if (s = 0) return {}
     else
        return comp(smulmonom(first(p1),s),smulpoly(red(p1),s);
    }

// multiplies a monom with a skalar
smulmonom((c,e),s)
    {
     return (c*s,e);
    }
```

Zum Multiplizieren von zwei multivariaten Polynomen dient der folgende Algorithmus.

Input: $P_1,\ P_2 \in \mathbb{Z}[x_1, \ldots, x_n]$.

Output: $P \in \mathbb{Z}[x_1, \ldots, x_n]$ mit $P = P_1 * P_2$
(Die Polynome sind in obiger Listendarstellung gegeben.)

Listing 3.3: mulpoly

```
mulpoly(p1,p2)
    {
        p1‘=p1;
        while(p1‘!=0)
            {
                p2‘=p2;
                while(p2‘!=0)
                    {
                        p=comp(mulmonom(first(p1‘),first(p2‘)),p);
                        p2‘=ref(p2‘);
                    }
                p1‘=red(p1‘);
            }
        return (simplify(p));
    }
```

Die Funktion mulmonom$(m1, m2)$ multipliziert zwei Monome $m1$ und $m2$, indem sie die Koeffizienten der beiden Monome multipliziert und deren Exponenten addiert.

Die Funktion simplify(p) addiert alle Koeffizienten von den Monomen mit gleichen Exponenten.

3.2.3 Division

Das Ziel dieses Abschnittes ist es, das multivariate Polynom P_2 durch das multivariate Polynom P_1 dividieren zu können. Dabei geht man immer von einer bestimmten Termordnung aus. Meist ist es die lexikographische Termordnung (siehe Abschnitt 3.1). Man ordnet die Monome in beiden Polynomen nach dieser Termordnung. Dann betrachtet man die führenden Monome (LM). Ist das LM von P_1 nicht durch das LM von P_2 teilbar, dann ist keine Division möglich. Andernfalls bestimmt man den Faktor q, der zum LM vom P_2 multipliziert werden muss, um 'gleich groß'[1] zu sein wie das LM von P_1. Jetzt wird das Polynom P_2 mit dem Faktor q multipliziert und das Ergebnis von P_1 abgezogen. Dieser Prozess wird so lange wiederholt, bis es keine Möglichkeit mehr zu dividieren gibt.

Diese Überlegungen führen zu folgendem Algorithmus:

Input: $P_1,\ P_2 \in \mathbb{Z}[x_1, \ldots, x_n]$.

Output: $Q,\ R \in \mathbb{Z}[x_1, \ldots, x_n]$ mit $P_1/P_2 = Q + R/P_2$
(Die Polynome sind in obiger Listendarstellung gegeben.)

Listing 3.4: divpoly

```
divpoly(p1,p2)
    {
     quot={};

     p1‘ = sort_lex(p1);
                                  // sorted in lexicographic term order
     p2‘ = sort_lex(p2);

     if (p2‘ = 0) raiseException("division by zero");

     if (first(p1‘)<first(p2‘)) return (0,p1‘);
                                  // no division possible
```

[1] Da es sich beim zugrundeliegenden Ring um keinen Körper handeln muss, kann es vorkommen, dass es keinen Faktor gibt, der die führenden Koeffizienten gleich werden lässt, dann wird der nächste kleinere Faktor bzw. nächst kleinere Koeffizient von q verwendet.

```
    else
    {
       while (first(p1‘)>=first(p2‘))
       {
           m = divmonom(first(p1‘),first(p2‘));
           if m = 0 then return (q,p1‘)
           else
               {
                   q = comp(m,q);
                   p1‘ = p1‘ − m ∗ p2‘;
               }
       }
       return (q,p1‘);
    }
   }

// divides two monoms if monom1 is greater than monom2
divmonom((c1,e1),(c2,e2))
    {
     c=c1 div c2;   // integer division
     e=e1−e2;       // by components

     return(c,e);
    }
```

Dieser Algorithmus arbeitet optimal, wenn P_1 durch P_2 teilbar ist.

3.3 GGT von multivariaten Polynomen

3.3.1 Verallgemeinerung des euklidschen Algorithmus

Die Anwendung des euklidschen Algorithmus auf multivariate Polynome funktioniert folgendermaßen:

Man muss die gleichen Definitionen, die für den univariaten Fall in Abschnitt 2.3 existieren, für den multivariaten Fall erklären.

Grundsätzlich betrachtet man die multivariaten Polynome als univariate Polynome mit multivariaten Koeffizienten.

$$P \in R[x_1, \ldots, x_n, y] \;\Rightarrow\; P \in R[x_1, ..., x_n][y].$$

Der Inhalt (Content) ist der größte gemeinsame Teiler der multivariaten Koeffizienten. Der primitive Teil ist wieder das Polynom dividiert durch dessen Inhalt.

Der restliche Algorithmus ändert sich im Prinzip nicht. Und siehe da, der Algorithmus scheint zu funktionieren.

Doch folgendes Beispiel scheint besonders interessant zu sein:

$$\mathrm{ggT}(x^4 + x \cdot y + x^3 \cdot y + y^2, x + 5 \cdot x^2 + y + 5 \cdot x \cdot y + x^2 \cdot y^3 + x \cdot y^4)$$

Der Algorithmus liefert 1, d.h. es wird kein größter gemeinsamer Teiler gefunden. Bei der Konstruktion des Beispiels wurde aber absichtlich ein Teiler eingebaut.

$$x^4 + x \cdot y + x^3 \cdot y + y^2 = (x + y) \cdot (x^3 + y)$$

$$x + 5 \cdot x^2 + y + 5 \cdot x \cdot y + x^2 \cdot y^3 + x \cdot y^4 = (x + y) \cdot (x \cdot y^3 + 5 \cdot x + 1)$$

Das Ergebnis sollte also $x + y$ sein. Was ist passiert?

Das Problem liegt ganz alleine in der Definition des euklidschen Algorithmus. Dort heißt es, dass ein euklidscher Ring zugrunde gelegt wird. Doch der Ring der multivariaten Polynome ist kein euklidscher Ring!

Daher ist eine Verallgemeinerung des euklidschen Ringes auf die multivariaten Polynome nur mit erheblichem Mehraufwand möglich. Außerdem tritt hier die gleiche Problematik des univariaten Falles in größerem Ausmaß auf. Daher wird zur Lösung dieses Problems auf den modularen Algorithmus verwiesen.

3.3.2 Modularer Ansatz

Beim modularen Ansatz für multivariate Polynome geht man vom gleichen Ansatz wie im univariaten Fall (vgl. Abschnitt 2.3.2) aus. Die Eingabepolynome sind Elemente aus $\mathbb{Z}[x_1, \ldots, x_n][y]$ und die Koeffizienten sind aus $\mathbb{Z}[x_1, \ldots, x_n]$ und die Hauptvariable ist y. Bei dieser Methode rechnet man modulo irreduziblen Polynomen $p(x) \in \mathbb{Z}[x_1, \ldots, x_n]$. Genauer gesagt, verwendet man lineare Polynome der Form $p(x) = x_n - r$, wobei $r \in \mathbb{Z}$ ist. Die Reduktion eines Polynoms $Q(x_1, \ldots, x_n, y)$ modulo $p(x_n)$ ist einfach die Auswertung von $Q(x_1, \ldots, x_n, y)$ an der Stelle $x_n = r$.

Für ein Polynom $P_1 \in \mathbb{Z}[x_1, \ldots, x_n][y]$ und $r \in \mathbb{Z}$ schreibt man P_{1,x_n-r} für P_1 mod $x_n - r$. Außerdem kann der Beweis des Satzes 2.3.6 auf den multivariaten Fall verallgemeinert werden, sodass man folgenden Satz erhält:

Satz 3.3.1. *Seien $P_1, P_2 \in \mathbb{Z}[x_1, \ldots, x_n][y]$ und $r \in \mathbb{Z}$, so dass $x_n - r$ nicht die beiden Leitkoeffizienten (bzgl. y) der Polynome P_1 und P_2 teilt. Weiters sei $P = \mathrm{ggT}(P_1, P_2)$. Dann gilt:*

1. $\deg_y(\mathrm{ggT}(P_{1,x_n-r}, P_{2,x_n-r})) \geq \deg_y(P)$
2. *Wenn $x_n - r$ kein Teiler der Resultanten von $\frac{P_1}{P_3}$ und $\frac{P_2}{P_3}$ ist, dann ist der Grad von $\mathrm{ggT}(P_{1,x_n-r}, P_{2,x_n-r})$ gleich dem Grad von P.*

Daraus kann der folgende Algorithmus entwickelt werden.

Input: $P_1, P_2 \in \mathbb{Z}[x_1, \ldots, x_n, y, z]$.

Output: P mit $P = \mathrm{ggT}(P_1, P_2)$
(Die Polynome sind in obiger Listendarstellung gegeben.)

Listing 3.5: gcdpoly

```
gcdpoly(p1,p2)
{
  if (areUnivariat(p1,p2)) return(gcdpoly_univariat(p1,p2));
  else
  {
    r = 0;
    currentDegree = infinity;
    InterpolationList = {};
```

```
cont = gcdpoly(content(p1,z),content(p2,z));
p1‘ = p1/cont;
p2‘ = p2/cont;
delta = gcdpoly(lc(p1‘,z),lc(p2‘,z));

while(true)
{
  r=r+1;
  if (subs(lc(p1‘,z),y,r) == 0) && (subs(lc(p2‘,z),y,r) == 0)
  {
    print("both leading coefficients vanish");
  }
  else
  {
    gcd_r = gcdpoly(subs(p1‘,y,r),subs(p2‘,y,r));
    gamma_r = subs(delta,y,r)/lc(gcd_r,z);
    gcd_r = gamma_r * gcd_r;
    d = deg(gcd_r,z);
    if (d<currentDegree)
    {
      currentDegree = d;
      InterpolationList = {(r,gcd_r)};
    }
    else
    {
      if (d == currentDegree)
      {
        InterpolationList = comp(InterpolationList,(r,gcd_r));
      }
    }
  }
  if (length(InterpolationList) > min(deg(p1‘,y),deg(p2‘,y)))
  {
    print("enough interpolation points collected");
    candidate = NewtonInterpolation(InterpolationList,y);
    if (isIntPoly(candidate))
    {
      print("candidate found");
      candidate = pp(candidate,z);
      if (isDivisor(candidate,p1‘) && isDivisor(candidate,p2‘))
      {
        print("candidate is gcd");
        return (cont*candidate);
```

```
          }
          else print("wrong candidate");
        }
        else print("candidate is not a integer polynomial");
      }
      InterpolationList = first(InterpolationList);
      print("drop earlier results");
    }
  }
}
```

Erklärungen zu den verwendeten Funktionen:
`areUnivariat(p1,p2)`: Diese Funktion überprüft, ob `p1` und `p2` univariate Polynome sind
`deg(p,x)`: Diese Funktion berechnet den Grad bzgl. der Hauptvariable `x`.
`gcdpoly_univariat(p1,p2)`: Diese Funktion berechnet den ggT von zwei univariaten Polynomen (siehe Abschnitt 2.3)
`isDivisor(p1,p2)`: Diese Funktion bestimmt, ob das Polynom `p1` das Polynom `p2` teilt.
`isIntPoly(p)`: Diese Funktion überprüft, ob `p` ein ganzzahliges Polynom ist
`lc(p,x)`: Diese Funktion bestimmt den führenden Koeffizienten des Polynoms `p` bzgl. der Hauptvariable `x`. (Achtung: Ergebnis ist Polynom!)
`length(list)`: Diese Funktion bestimmt die Anzahl der Elemente in der Liste `list`.
`min(x,y)`: Die Funktion bestimmt das Minimum von `x` und `y`.
`NewtonInterpolation(list,var)`: Diese Funktion berechnet aus einer Liste von Stützstellen `list` mit Hilfe der Newton Interpolation ein interpolierendes Polynom in der Variable `var`, wobei die Stützstellen wiederum Polynome sein können.
`pp(p,x)`: Diese Funktion berechnet den primitiven Teil des Polynoms `p` bezüglich der Hauptvariable `x`.
`subs(p,x,c)`: Diese Funktion wertet das Polynom `p` über der Hauptvariable `x` in `c` aus.

Zur Korrektheit des Algorithmus:.
(mit tatkräftiger Unterstützung von Clemens Heuberger)

Berechnung der Teil-ggTs:

Es sei $R := \mathbb{Z}[x_1, \ldots, x_n]$.

Gegeben seien zwei Polynome $P_1, P_2 \in R[y][z]$. Wie im univariaten Fall kann man sich hier ohne Beschränkung der Allgemeinheit auf die primitiven Teile

von P_1 und P_2 beschränken, da gilt:

$$\mathrm{ggT}(P_1, P_2) = \mathrm{ggT}(\mathrm{cont}_z(P_1), \mathrm{cont}_z(P_2)) \cdot \mathrm{ggT}(\mathrm{pp}_z(P_1), \mathrm{pp}_z(P_2)).$$

wobei cont_z eine Abbildung von $R[y][z]$ in $R[y]$ ist.

Es gilt also o.B.d.A. $\mathrm{cont}_z(P_1) = \mathrm{cont}_z(P_2) = 1$.

Angenommen $c = \mathrm{ggT}(P_1, P_2)$, dann teilt der Leitkoeffizient bzgl. z die Leitkoeffizienten der Polynome P_1 und P_2 und damit auch den ggT der beiden. D.h. wenn $\gamma = \mathrm{ggT}(\mathrm{lc}_z(P_1), \mathrm{lc}_z(P_2))$ ist, dann folgt, dass es ein $d \in R[y]$ gibt, für das $\mathrm{lc}_z(c) \cdot d = \gamma$ gilt.

Neues Ziel:
Berechne ein Polynom f mit $f := c \cdot d$.
Da c primitiv ist (weil P_1 und P_2 primitiv sind), gilt

$$c = \frac{f}{\mathrm{cont}_z(f)}.$$

c teilt P_1 nach der Voraussetzung, dass $c = \mathrm{ggT}(P_1, P_2)$ ist. Wenn man zu c und zu P_1 den gleichen Faktor multipliziert, dann ändert sich bezüglich der Teilbarkeit nichts, es gilt $c \cdot d \mid P_1 \cdot d$. Nach Definition von f gilt sofort $f \mid P_1 \cdot d$. Analoges gilt für die Teilbarkeit von P_2.

Wenn man jetzt beide Seiten reduziert, also $f_{y-r} \mid d_{y-r} \cdot P_{1(y-r)}$ und $f_{y-r} \mid d_{y-r} \cdot P_{2(y-r)}$, bleibt die Teilbarkeit erhalten. Da f_{y-r} beide reduzierten Polynome teilt, muss es auch den ggT der beiden reduzierten Polynome teilen. Dieser ggT ist aus $R[z]$ und hat daher eine Unbekannte weniger. Betrachtet man diese ggT-Berechnung rekursiv, dann kann man voraussetzen, dass dieser ggT bestimmt werden kann.

Falls $y - r \nmid d$ und $y - r \nmid \mathrm{Res}_x(\frac{P_1}{c}, \frac{P_2}{c})$, dann gilt nach Satz 3.3.1

$$\deg_z(\mathrm{ggT}(P_{1(y-r)}, P_{2(y-r)})) \;=\; \deg_z(c_{y-r}) = \deg_z(f_{y-r}). \tag{3.2}$$

Daraus folgt, dass q in der Gleichung $f_{y-r} \cdot q = d_{y-r} \cdot \mathrm{ggT}(P_{1(y-r)}, P_{2(y-r)})$ aus $R[z]$ sein muss.

Da $f_{y-r} \in R[z]$ und $\mathrm{ggT}(P_{1(y-r)}, P_{2(y-r)}) \in R[z]$ und $d \in R[y]$ ist, folgt, dass $d_{y-r} \in R$ sein muss. Daher muss auch $q \in R$ sein.

Daraus ist leicht die Gleichung für die Leitkoeffizienten ableitbar:

$$\mathrm{lc}_z(f_{y-r}) \cdot q \;=\; d_{y-r} \cdot \mathrm{lc}_z(\mathrm{ggT}(P_{1(y-r)}, P_{2(y-r)})). \tag{3.3}$$

Aus der Gradbedingung (3.2) und $y - r \nmid d$ und $y - r \nmid \mathrm{Res}_z(\frac{P_1}{c}, \frac{P_2}{c})$ folgt, dass $\mathrm{lc}_z(f_{y-r}) = \mathrm{lc}_z(f)_{y-r}$ ist.

Damit ist klar, dass der Leifkoeffizient $\mathrm{lc}_z(f) = \mathrm{lc}_z(c) \cdot d = \gamma$ ist.

Die Gleichung (3.3) kann auf diese Form gebracht werden:

$$\frac{d_{y-r}}{q} = \frac{\mathrm{lc}_z(f)_{y-r}}{\mathrm{lc}_z(\mathrm{ggT}(a_{y-r}, b_{y-r})}.$$

Wie man daraus erkennen kann, ergibt sich:

$$f_{y-r} = \frac{d_{y-r}}{q} \cdot \mathrm{ggT}(a_{y-r}, b_{y-r})$$

bzw.

$$f_{y-r} = \gamma_{y-r} \cdot \mathrm{pp}_x(\mathrm{ggT}(a_{y-r}, b_{y-r})),$$

wobei γ_{y-r} und f_{y-r} im Algorithmus `gamma_r` und `gcd_r` entsprechen.

Newton Interpolation Bei der Newton-Interpolation werden die einzelnen Stützstellen, also die reduzierten Polynome `gcd_r`, zu einem ggT-Kandidaten kombiniert. (siehe [Sch93])

Abbruchkriterium Das Analogon zur Landau-Mignotte-Schranke im univariaten Fall ist hier viel leichter zu bestimmen. Da der ggT vom Grad in der Hauptvariable nicht größer als das Minimum der Grade der beiden Polynome werden kann, werden für die Newton-Interpolation höchstens $\min(\deg(P_1), \deg(P_2))$ reduzierte ggT-Polynome (`gcd_r`) gebraucht, um einen Kandidaten (`candidate`) für den ggT zu erhalten.

□

Beispiel:

Gegeben seien die zwei Polynome

$$\begin{aligned} P_1 &= x^3 + x^2y + xy^2 + y^3 + x^2z + y^2z + xz^2 + yz^2 + z^3, \\ P_2 &= (x^4 + x^3y + xy^3 + y^4 + x^3z + y^3z + xz^3 + yz^3 + z^4. \end{aligned}$$

Jetzt werden die reduzierten Polynome bestimmt und dann deren ggT:

$\mathtt{r} = 1$:
$P_{1(z-1)} = 1 + x + x^2 + x^3 + y + x^2y + y^2 + xy^2 + y^3$
$P_{2(z-1)} = 1 + x + x^3 + x^4 + y + x^3y + y^3 + xy^3 + y^4$
`gcd_r`$= \mathrm{ggT}(P_{1(z-1)}, P_{2(z-1)}) = 1 + x + y$

$\mathtt{r} = 2$:
$P_{1(z-2)} = 8 + 4x + 2x^2 + x^3 + 4y + x^2y + 2y^2 + xy^2 + y^3$
$P_{2(z-2)} = 16 + 8x + 2x^3 + x^4 + 8y + x^3y + 2y^3 + xy^3 + y^4$
$\mathtt{gcd\_r}$= ggT$(P_{1(z-2)}, P_{2(z-2)}) = 2 + x + y$

$\mathtt{r} = 3$:
$P_{1(z-3)} = 27 + 9x + 3x^2 + x^3 + 9y + x^2y + 3y^2 + xy^2 + y^3$
$P_{2(z-3)} = 81 + 27x + 3x^3 + x^4 + 27y + x^3y + 3y^3 + xy^3 + y^4$
$\mathtt{gcd\_r}$= ggT$(P_{1(z-3)}, P_{2(z-3)}) = 3 + x + y$

$\mathtt{r} = 4$:
$P_{1(z-4)} = 64 + 16x + 4x^2 + x^3 + 16y + x^2y + 4y^2 + xy^2 + y^3$
$P_{2(z-4)} = 256 + 64x + 4x^3 + x^4 + 64y + x^3y + 4y^3 + xy^3 + y^4$
$\mathtt{gcd\_r}$= $ggT(P_{1(z-4)}, P_{2(z-4)}) = 4 + x + y$

Jetzt ist die Anzahl der Stützstellen größer als das Minimum der maximalen Grade von z in P_1 und P_2. Daher wird mit Newton-Interpolation ein Kandidat für den ggT erzeugt.

$$\mathtt{candidate} = x + y + z.$$

Da dieser Kandidat ein Teiler von P_1 und P_2 ist, ist der ggT gefunden.

3.3.3 Weitere alternative GGT-Algorithmen

In [SS92] werden drei weitere Algorithmen zur Berechnung des größten gemeinsamen Teilers vorgestellt.

Gröbner-Basis-Methode

Der erste Algorithmus berechnet den ggT durch Gröbner Basen. Er basiert auf folgendem Satz [SS92, Theorem 1]:

Satz 3.3.2. *Seien* $\Gamma = \{P_1, P_2, \ldots, P_s\}$ *eine Gröbner Basis vom Ideal* (P_1, P_2) *in* $K[y, \ldots, z][x]$ *und* $d_i = \deg(P_i)$ *für* $i = 1, \ldots, s$*, wobei* d_k *der kleinste Wert aus der Menge* $\{d_1, d_2, \ldots, d_s\}$ *ist, dann existiert ein Polynom* $C \in K[y, \ldots, z]$*, so dass* $P_k = C \cdot ggT(P_1, P_2)$*.*

Der resultierende Algorithmus wird die *Gröbner-Basis-Methode* genannt:

1. Berechne die Gröbner Basis $\Gamma = \{P_1, P_2, \ldots, P_s\}$ vom Ideal (P_1, P_2) in $K[y, \ldots, z][x]$.

2. Sei P_k das Element mit minimalen Grad (bzgl. x) von Γ und $\deg(P_k) = 0$, dann ist das Ergebnis 1 sonst $\mathrm{pp}(P_k)$.

Dieser Algorithmus findet aber keine praktische Anwendung, da die Berechnung der Gröbner Basen eine sehr komplexe und damit für diese Zwecke ineffiziente Aufgabe darstellt.

Die beiden nächsten Algorithmen nehmen auf die Terminologie der abgeschnittenen Potenzreihen Bezug. (siehe Anhang A.3.1)

Subresultanten-Methode mit Potenzreihen-Koeffizienten

Bei dieser Methode geht man von zwei Polynomen in der folgenden Darstellung aus:

$$\begin{aligned} P_1 &= a_m x^m + a_{m-1}x^{m-1} + \cdots + a_0, \quad a_m \neq 0, \\ P_2 &= b_n x^n + b_{n-1}x^{n-1} + \cdots + b_0, \qquad b_n \neq 0, \end{aligned}$$

wobei man annimmt, dass $m \geq n$ ist.

Weiters wird das Polynom $S^{(j)}$ als die Subresultante j-ter Ordnung von P_1 und P_2 (siehe Abschnitt 2.3.1) definiert und ihr Grad $\deg(S^{(j)})$ ist üblicherweise j.

Im Folgenden werden Determinanten $D_i^{(j)}$ verwendet. Diese entsprechen den Determinanten $\det(M(P_1, P_2)_{i,j})$ aus Abschnitt 2.3.1.

Der grundlegende Satz für diese Methode ist:

Satz 3.3.3. *Es seien* $d = \deg(\mathrm{ggT}(P_1, P_2))$ *und* $g = \mathrm{ggT}(\mathrm{lc}(P_1), \mathrm{lc}(P_2))$. *Dann gilt:*

$$\begin{aligned} &g | D_i^j, \qquad i = j-1, \ldots, 0, \\ &\frac{D_d^{(d)}}{g} | D_i^{(d)}, \qquad i = d, \ldots, 0. \end{aligned}$$

Angenommen, man kennt den Wert $d = \deg(\mathrm{ggT}(P_1, P_2))$, dann konstruiert man sich einfach die Subresultante $S^{(d)}$, indem man die Determinanten $D_i^{(d)}$ für $i = d, d-1, \ldots, 0$ berechnet. Man erhält dann den $\mathrm{ggT}(P_1, P_2)$ als den primitiven Teil von $S^{(d)}$,

$$\mathrm{ggT}(P_1, P_2) = \mathrm{pp}(S^{(d)}).$$

Wie man oben sieht, teilt g den Leitkoeffizienten von $S^{(d)}$ und $\tilde{P} = \frac{S^{(d)}}{\frac{\mathrm{lc}(S^{(d)})}{g}}$ ist ein Vielfaches von G, wobei $G = \mathrm{pp}(\tilde{P})$ ist.

Wenn man $\tilde{P}$ folgend darstellt

$$\tilde{P} = g_d x^d + g_{d-1} x^{d-1} + \cdots + g_0, \tag{3.4}$$

dann sieht man, dass

$$g_d = g \quad \text{und} \quad g_i = \frac{D_i^{(d)}}{\frac{D_d^{(d)}}{g}}, \text{ für } i = d-1, \ldots, 0. \tag{3.5}$$

(3.4) und (3.5) zeigen, dass man g_d und die g_is braucht und nicht die Determinaten $D_d^{(d)}$ und $D_i^{(d)}$ selbst. Die Größen der g_is sind gewöhnlich viel kleiner als die der Determinaten. Für die Berechnung der g_is aus der Gleichung (3.5) werden nicht alle Terme der Determinaten $D_d^{(d)}$ und $D_i^{(d)}$ benötigt. Man braucht nur einige Terme mit kleineren Exponenten. Daher lässt man bei der Berechnung der Determinante einfach die unwichtigen Terme weg, was die Berechnung sehr effizient macht. Man eliminiert also die unwichtigen Terme systematisch, indem man die Koeffizienten von P_1 und P_2 als abgeschnittene Potenzreihen betrachtet.

Es seien $E, e_y, \ldots, e_z$ definiert wie in Lemma A.3.4. Lemma A.3.2 zeigt, dass für die Berechnung von $\tilde{P}$ die Determinaten $D_i^{(d)}$ nur mit einer Signifikanz $(E, e_y, \ldots, e_z)$ zu berechnen sind. Zu beachten ist auch, dass wenn P_1 und P_2 primitiv sind, dann gilt:

$$\begin{aligned} ran_u(P_i) &= (0, some), \quad i = 1, 2 \\ ran_u(G) &= (0, some), \end{aligned}$$

für jede Variable $u \in y, \ldots, z$.

Daraus ergibt sich der folgende Algorithmus:

Input: $P_1,\ P_2 \in \mathbb{Z}[y, \ldots, z][x]$.

Output: $G \in \mathbb{Z}[y, \ldots, z][x]$ mit $G = \mathrm{ggT}(P_1, P_2)$

Listing 3.6: ASR_GCD

```
ASR_GCD(p1,p2)
```

```
{
 Schaetze d=deg(ggT(p1,p2)) durch eine modulare Methode ab;
 // d stellt also eine obere Schranke dar
 if (d==0) return 1;
 else
 {
    g=ggT(lc(p1),lc(p2));
    E=min{E_i−E_i'+E"|i=1,2};
    e_u=min{e_ui−e_ui'+e_u"|i=1,2 und u ein Element aus {y,...,z}}
    // wobei die E_i usw. wie in Abschnitt A.3.1 definiert sind
 }
 while(true)
 {
    Konstruiere die Determinanten D[i][d], i=d,d−1,...,0 der
    Subresultante d−ter Ordnung S[d] und berechne die
    Determinanten D[i][d] bis zu (E,ey ,..., ez) signifikante
    Terme fuer (y ,..., z);

    Berechne P~ nach der Potenzreihendivision bis auf
    (E,ey ,..., ez) signifikante Terme;

    G=pp(P~);

    if (divides(G,p1) && divides(G,p2)) return G;
    else d=d−1;
 }
}
```

PRS-Methode mit Potenzreihen-Koeffizienten

Wie in Abschnitt 3.3.3, wo man mit Polynomen geringen Grades bei der Division rechnet, indem die Polynome als Potenzreihen betrachten und die Terme höheren Grades einfach abgeschnitten werden, kann man diese Idee auch für den PRS-Algorithmus verwenden.

Angenommen man berechnet die PRS $(P_1, P_2, \ldots, P_k \neq 0, P_{k+1} = 0)$, indem man die Terme höheren Grades in den Koeffizienten systematisch weglässt und $\tilde{P}_k = \frac{gP_k}{\mathrm{lc}(P_k)}$ auf $(E, e_y, \ldots, e_z)$ signifikante Terme berechnet, dann folgt aus Lemma A.3.2, dass $\tilde{P} = \gamma G$, wobei $G = \mathrm{ggT}(P_1, P_2)$ und $\gamma = \frac{\mathrm{ggT}(\mathrm{lc}(P_1),\mathrm{lc}(P_2))}{\mathrm{lc}(G)}$.

Die PRS kann mit den konventionellen Formeln (siehe Abschnitt 2.3.1) berechnet werden, wobei man aber eben nur Terme bis zu einem bestimmten Grad betrachtet.

Das einzige Problem in der obigen Methode ist die Behandlung des Genauigkeitsrückganges. Man beachte, dass, wenn die Genauigkeit in den P_is gleich bleibt, aber die Genauigkeit der $\mathrm{lc}(P_i)$s sich veringert, dann auch die Genauigkeit der PRS sinkt.

Man verwendet die folgende Regel:
Ungeachtet der Tatsache, ob sich die Genauigkeit verringert oder nicht, berechnet man die PRS, indem man nur die signifikanten Terme betrachtet.

Lemma 3.3.1. *Seien P_1 und P_2 in $K[y,\ldots,z][x]$ und ($\tilde{P}_1$,$\tilde{P}_2$,...,$\tilde{P}_k \neq 0$, $\tilde{P}_{k+1} = 0$) eine PRS, so dass $P_i \equiv \tilde{P}_i$ bzgl. $(E, e_y, \ldots, e_z)$ signifikante Terme für $i = 1, 2$, und $\tilde{P}$ nach obiger Beschreibung generiert wird. Angenommen $\tilde{P}_k$ ist $(E', e'_y, \ldots, e'_z)$ signifikant, also $E - E' = e_y - e'_y = \cdots = e_z - e'_z \geq 0$, und G sei der $\mathrm{ggT}(P_1, P_2)$ in $K[x, y, \ldots, z]$, dann*

$$G|\tilde{P}_k \quad \textit{bis auf } (E', e'_y, \ldots, e'_z).$$

Seien $(E, e_y, \ldots, e_z)$ wie in Lemma A.3.2 definiert und die PRS wie im letzten Lemma berechnet. Weil $(E, e_y, \ldots, e_z)$ oft keine scharfe obere Schranke ist, kann man G korrekt aus den $\tilde{P}_k$s berechnen.

Ausgenommen im folgenden, sehr unglücklichen Fall kann man die Sinnhaftigkeit von $\tilde{P}_k$ mit der Trial-Division überprüfen:
Man berechnet einfach $\tilde{P} \equiv \frac{g\tilde{P}_k}{\mathrm{lc}(\tilde{P}_k)}$ auf alle möglichen Terme und überprüft, ob $\mathrm{pp}(\tilde{P})$ die beiden Polynome P_1 und P_2 teilt.

Der unglückliche Fall, dass $\deg(\tilde{P}) < \deg(G)$ ist, tritt dann auf, wenn die höhergradigen Terme in $\tilde{P}_k$ auf Grund des Genauigkeitsverlustes verschwinden. Man beachte, dass dieser Fall nicht auftritt, wenn die PRS normal ist, d.h. $\deg(\tilde{P}_{i+1}) = \deg(\tilde{P}_i) - 1$ für alle $i = 2, 3, \ldots$.

Man kann einfach den unglücklichen Fall vermeiden, indem man eine untere Schranke für die signifikanten Terme angibt.

Satz 3.3.4. *Es seien $G = \mathrm{ggT}(P_1, P_2)$, $g = \mathrm{ggT}(\mathrm{lc}(P_1), \mathrm{lc}(P_2))$ und*

$$E_{\min} = \min(\textit{Termgrade der Terme von } g).$$

Weiters sei $(P_1, P_2, \ldots, P_k \neq 0, P_{k+1} = 0)$ die PRS, wie sie in obigem Lemma definiert wurde. Es seien $\mathrm{lc}(\tilde{P}_k)$ und $\tilde{P} = \frac{gP_k}{\mathrm{lc}(P_k)}$ abgeschnitten nach $(\tilde{E}, \tilde{e_y}, \ldots, \tilde{e_z})$ signifikanten Termen. Wenn $\tilde{E} \geq E_{\min}$ und $\mathrm{pp}(\tilde{P})$ sowohl P_1 als auch P_2 teilt, dann ist $G = \mathrm{pp}(\tilde{P})$

Aus diesen Überlegungen ergibt sich der folgende Algorithmus:

Input: $P_1,\ P_2 \in \mathbb{Z}[y,\ldots,z][x]$.

Output: $G \in \mathbb{Z}[y,\ldots,z][x]$ mit $G = \mathrm{ggT}(P_1, P_2)$

Listing 3.7: APRS_GCD

```
APRS_GCD(p1,p2)
    {
     g=ggT(lc(p1),lc(p2));
     Ed=E=min{E_i−E_i‘+E“|i=1,2};
     edu=e_u=min{e_ui−e_ui‘+e_u“|i=1,2 und u ein Element aus {y,...,z}};
        // wobei die E_i usw. wie in Abschnitt A.3.1 definiert sind
     Emin=min(Grade der Terme aus g);
     while(true)
     {
        for (int i=1;i<2;i++) Ps[i]=P[i];
            // Abschneiden zu hoeher gradiger Terme (Es,esy,....esx)
        Berechnen der PRS (P3,...,Pk,Pk+1) mit allen ömglichen Termen;
        lcpk=lc(Pk);
            // Abschneiden zu hoeher gradiger Terme!
        if (deg(Pk)=0) return 1;
        else P=g∗Pk/lcpk;
            // Abschneiden zu hoeher gradiger Terme!
        G=pp(P);
        if (divides(G,p1) && divides(G,p2))
        {
            if ((PRS ist normal)||E>=Emin) return(G);
        }
        Ed=E=2∗Ed−Ed;
        edu=eu=edu+Ed−Es fuer alle u aus {y,...,z};
     }
    }
```

4. IMPLEMENTIERUNG IM *ISAC*-PROJEKT

Dieses Kapitel ist der Implementierung und Anwendung der in Kapitel 2 und 3 dargestellten Funktionen gewidmet. Nach kurzen Bemerkungen zur verwendeten Programmiersprache werden die gewählten Datenstrukturen beschrieben.

Der Abschnitt 4.3, GGT-Berechnung, stellt die Implementierung der in den vorhergehenden Kapiteln erarbeiteten Funktionen dar. Abschnitt 4.4. beschreibt die praktische Anwendung der Funktionen für das Bruchrechnen.

Der letzte Abschnitt, 'reverse rewriting', spezifiziert die Funktionen, die *ISAC* schrittweise begründendes Bruchrechnen ermöglicht, ohne sie auszuführen, und nennt offene Fragen hierzu.

Im Folgenden wird der Aufruf der implementierten SML-Funktionen auf dem SML-Toplevel (mit `ML>` als Prompt) zeichengetreu und von den zurückgegebenen Werten nur die relevanten gezeigt (jeweils ohne Prompt).

4.1 Programmiersprache

Das *ISAC*-Projekt basiert auf zwei Programmiersprachen. Einerseits wird Java verwendet, für das Benutzerinterface und die Netzwerkkommunikation. Andererseits wird SML (Standard ML of New Jersey v110.9.1, October 19, 1998) verwendet, für das mathematische Basiswissen und die Beweisführung durch Isabelle. Für diese Diplomarbeit ist jedoch nur SML von Bedeutung.

Was ist SML? SML hat seine Wurzeln als Meta-Sprache zur Definition von Beweistaktiken in interaktiven Beweisern (theorem-prover). Über Jahre entwickelte sich die Sprache dann zu einer ausgewachsenen Programmiersprache, mit exzellenten Features sowohl für kleine als auch für größere Programme.

Folgende Eigenschaften machen SML zu einer interessanten Sprache:

- SML ist eine funktionale Sprache.
- Das wichtigste Element in SML ist der Ausdruck (Expression), der ausgewertet und weiter in der Berechnung verwendet wird.
- SML ist side-effect free, d.h. Ausdrücke werden abgearbeitet, das Ergebnis wird aber nicht dauerhaft gespeichert, außer man verwendet sogenannte Referenzvariable. Mithilfe dieser Variablen kann auch imperativ programmiert werden. Die vorliegende Arbeit verwendet diese Sprachelemente, um die Algorithmen 1-zu-1 zur Literatur zu implementieren.
- SML unterstützt Funktionen höherer Ordnung. Das sind kurz gesagt Funktionen, die Funktionen als Argument akzeptieren und verwenden.
- SML unterstützt Type-Polymorphismus, d.h. eine Funktion kann für mehrere Argumenttypen definiert werden.
- SML unterstützt auch Pattern Matching.
- SML ist eine streng typisierte Sprache, d.h. der Typ aller Werte wird bei der Compilierung ermittelt und geprüft. So können viele Fehler schon vorab erkannt werden.
- SML erlaubt es, abstrakte Datentypen elegant und einfach zu implementieren.

4.2 Datenstrukturen

ISAC übernimmt die Termstruktur von Isabelle. Diese ist zum Rechnen mit Algorithmen nur beschränkt geeignet. Daher werden eigene "interne" Datenstrukturen für Polynome angelegt. (siehe auch Abschnitte 2.1 bzw. 3.1)

In diesem Abschnitt werden diese "internen" Datenstukturen vorgestellt.

Univariate Polynome werden intern als Listen der ganzzahligen Koeffizienten dargestellt. Die Koeffizienten werden so geordnet, dass bei der niedrigsten Potenz also x^0 begonnen wird und die Liste bei der höchsten Potenz x^n endet. Die fehlenden Potenzprodukte werden durch 0 ersetzt.

z.B.

$$5 \cdot x^5 + 4 \cdot x^3 + 2 \cdot x^2 + x + 19 \Rightarrow [19, 1, 2, 4, 0, 5]$$

In SML sieht die Definition so aus:

```
type uv_poly = int list;
```

Multivariate Polynome werden auch als Listen implementiert, aber sie haben etwas komplexere Elemente. Wie in Abschnitt 3.1 bestehen die einzelnen Elemente - Monome - aus einem Tupel, wobei der erste Teil der Koeffizient und der zweite eine Liste der Exponenten ist.

z.B.
$5 \cdot x^5 \cdot y^3 + 4 \cdot x^3 \cdot z^2 + 2 \cdot x^2 \cdot y \cdot z^3 - z - 19$
$\Rightarrow [(5, [5, 3, 0]), (4, [3, 0, 2]), (2, [2, 1, 3]), (-1, [0, 0, 1]), (-19, [0, 0, 0])]$

Die Reihenfolge ist nicht festgelegt. Die Definition in SML erfolgt folgendermaßen:

```
type mv_monom = (int * int list);
        (* (coefficient,list of exponents) *)
type mv_poly = mv_monom list;
```

Die polynomiale Normalform wird in $\mathcal{ISAC}$ folgendermaßen definiert:

- die Koeffizienten und Exponenten sind ganze Zahlen
- die Polynome sind vollständig ausmultipliziert,
  d.h. $a_0 + a_1 \cdot x + \cdots + a_n \cdot x^n$ für univariate Polynome
  bzw. $a_0 + a_1 \cdot x_1^{e11} \cdot x_2^{e12} \ldots x_m^{e1m} + \cdots + a_n \cdot x_1^{en1} \cdot x_2^{en2} \ldots x_m^{enm}$
  für multivariate Polynome
- zwischen den Monomen steht immer ein +, d.h. $a - b \;\Rightarrow\; a + (-1) \cdot b$
- die Monome sind in lexikographischer Termordnung sortiert

Beispiele:

$2 \cdot x + (-50) \cdot x^3$
$1 + (-10) \cdot x + 25 \cdot x^2$
$(-1) \cdot x \cdot y^2 + 7 \cdot x^3$

Die binomiale Normalform ist gleich definiert wie die obige, nur ist hier neben dem + auch der binären Operator − erlaubt.

Beispiele:

$$2 \cdot x - 50 \cdot x^3$$
$$1 - 10 \cdot x + 25 \cdot x^2$$

4.2.1 Umwandlung von Isabelle-Termen in die interne Datenstruktur

Die Eingaben an den Kernel erfolgen prinzipiell in Strings. Dabei werden numerische Werte als Zahlen, und "Unbekannte" als Buchstaben dargestellt. Die Zeichen für Addition und Multiplikation sind die üblichen, nämlich `+` und `*`. Zum Potenzieren muss `^^^` verwendet werden[1]. Die Subtraktion wird für die polynomiale Normalform als Addition umgeschrieben[2].

$$a - b = a + (-1) * b$$

Hier wird das obige Beispiel als String eingegeben und der Variablen `t` zugewiesen:

```
ML>  val t="5 * x^^^5 * y^^^3 + 4 * x^^^3 * z^^^2 +
           2 * x^^^2 * y * z^^^3 + (-1) * z + (-19)";
val t =
"5 * x^^^5 * y^^^3 + 4 * x^^^3 * z^^^2 + 2 * x^^^2 * y * z^^^3
 + (-1) * z + (-19)": string
```

Der Befehl `parse` überprüft, ob der String das richtige Format hat, und liefert dann einen sogenannten `cterm`, der in der Variable `t1` gespeichert wird. Der Typ `option` bedeutet, dass `None` zurück geliefert wird, wenn der Vorgang auf einen Fehler stößt.

```
ML> val t1=parse thy t;
val t1 =
  Some
  "5 * x ^^^ 5 * y ^^^ 3 + 4 * x ^^^ 3 * z ^^^ 2 +
   2 * x ^^^ 2 * y * z ^^^ 3 + -1 * z + -19" : cterm option
```

[1] Eine vorübergehende Darstellung in der ISAC-Version, unter der diese Diplomarbeit implementiert wurde.

[2] Die Berücksichtigung von Binomen in der Form $(a - b)^2 = a^2 - 2ab - b^2$, wie sie im elementaren Algebraunterricht gebräuchlich sind, wird mit der binomialen Normalform behandelt.

`cterm` ist ein Datentyp, der als String dargestellt wird und nur schwer bearbeitet werden kann. Deshalb wird der `cterm` in den Isabelle-Typ `term` umgewandelt und in der Variable `t2` gespeichert. Dieser stellt das Polynom in einer Baumstruktur dar; diese ist sowohl für das 'matching' und 'rewriting' geeignet, als auch für die Umwandlung in die hier benötigte interne Struktur.

```
ML>  val t2=(term_of o the) t1;
val t2 =
Const ("op +","[RealDef.real, RealDef.real] => RealDef.real") $
(Const ("op +","[RealDef.real, RealDef.real] => RealDef.real") $
 (Const ("op +","[RealDef.real, RealDef.real] => RealDef.real") $
  (Const (#,#) $ (# $ # $ (# $ #)) $ (Const # $ (# $ #) $
                                          (# $ # $ Free #)))
  $
  (Const ("op *","[RealDef.real, RealDef.real] => RealDef.real") $
   (Const # $ (# $ #) $ Free (#,#)) $
   (Const (#,#) $ Free (#,#) $ Free ("3","RealDef.real")))) $
 (Const ("op *","[RealDef.real, RealDef.real] => RealDef.real") $
  Free ("-1","RealDef.real") $ Free ("z","RealDef.real"))) $
Free ("-19","RealDef.real") : term
```

`Free(Name/Wert,Typ)` beschreibt eine Variable, bzw. Zahlen-Konstante
`Const(Name,Typ)` beschreibt eine Funktions-Konstante (algebraische Operation, Prädikat, etc.).

Da diese Darstellung nur schwer zu lesen ist, kann man sich den Term mit `atomty` ausgeben lassen.

```
ML>  atomty thy t2;
*** -------------
*** Const ( op +, [real, real] => real)
*** . Const ( op +, [real, real] => real)
*** . . Const ( op +, [real, real] => real)
*** . . . Const ( op +, [real, real] => real)
*** . . . . Const ( op *, [real, real] => real)
*** . . . . . Const ( op *, [real, real] => real)
*** . . . . . . Free ( 5, real)
*** . . . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . . . Free ( x, real)
*** . . . . . . . Free ( 5, real)
*** . . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . . Free ( y, real)
*** . . . . . . Free ( 3, real)
*** . . . . Const ( op *, [real, real] => real)
```

```
*** . . . . . Const ( op *, [real, real] => real)
*** . . . . . . Free ( 4, real)
*** . . . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . . . Free ( x, real)
*** . . . . . . . Free ( 3, real)
*** . . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . . Free ( z, real)
*** . . . . . . Free ( 2, real)
*** . . . Const ( op *, [real, real] => real)
*** . . . . Const ( op *, [real, real] => real)
*** . . . . . Const ( op *, [real, real] => real)
*** . . . . . . Free ( 2, real)
*** . . . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . . . Free ( x, real)
*** . . . . . . . Free ( 2, real)
*** . . . . . Free ( y, real)
*** . . . . Const ( RatArith.pow, [real, real] => real)
*** . . . . . Free ( z, real)
*** . . . . . Free ( 3, real)
*** . . Const ( op *, [real, real] => real)
*** . . . Free ( -1, real)
*** . . . Free ( z, real)
*** . Free ( -19, real)
val it = () : unit
```

Mit dem Befehl `term2poly` kann man den Isbelle-Term in die interne Struktur `mv_poly` umwandeln lassen.

```
ML> term2poly t2 ["x","y","z"];
val it = Some
[(5,[5,3,0]),(4,[3,0,2]),(2,[2,1,3]),(~1,[0,0,1]),(~19,[0,0,0])]
: mv_poly
```

Die Signatur lautet:

```
val term2poly = fn : term -> string list -> mv_poly option;
```

Das erste Argument der Funktion muss ein Polynom in polynomialer Normalform (siehe Seite 55) sein, wobei die Termordnung nicht relevant ist.
Das zweite Argument stellt die Liste der vorhandenen Variablennamen dar. Die Reihenfolge ist für die Berechnung nicht wichtig, es müssen aber mindestens alle Variablen des Terms angegeben werden!

Mit der Funktion `((map free2str) o vars) term` kann man die Variablen von einen Term automatisch bestimmen. Das manuelle Eingeben der Variablen bietet aber den Vorteil, dass man mit mehr verschiedenen Variablen rechnen kann, als im Polynom vorkommen. Es kann auch zu folgenden Problem kommen:

Man betrachte das folgende Beispiel:

$$\frac{x \cdot y + y}{x \cdot z + z}$$

Wenn man hier den Zähler und Nenner getrennt umwandelt, dann erhält man beide Male die gleiche Darstellung $[(1,[1,1]),(1,[0,1])]$, aber jeweils mit einer anderen Interpretation für die Exponenten. Daher muss man zuerst die Variablen für den gesamten Ausdruck bestimmen und mit ihnen die Umwandlungen durchführen!

Für die Umwandlung von Polynomen sind die beiden Darstellungen `x^^^1` und `x` ident.

Als Alternative wird hier auch die Funktion `binom2poly` vorgestellt.

Die Signatur lautet:

```
val binom2poly = fn : term -> string list -> mv_poly option;
```

Im Prinzip funktioniert diese Funktion gleich, wie `term2poly`. Der einzige Unterschied ist, dass diese Funktion Polynome in binomialer Normalform (siehe Abschnitt 4.2) akzeptiert.

4.3 ggT-Berechnung

Dieser Abschnitt stellt die Implementierung der in den Kapiteln 2 und 3 erarbeiteten Funktionen zur Berechnung des ggT dar.

Bei der ggT-Berechnung wird davon ausgegangen, dass alle Polynome multivariat sind und der univariate Fall nur einen Spezialfall darstellt.

4.3.1 Multivariater Fall

Der implementierte Algorithmus zur Berechnung des größten gemeinsamen Teilers von zwei multivariaten Polynomen entspricht dem Algorithmus von Abschnitt 3.3.2 auf Seite 42. Er kann zwei Polynome in der internen Darstellung `mv_poly` verarbeiten und liefert wieder ein Polynom in der internen Darstellung `mv_poly`.

Seine SML-Signatur lautet:

```
val mv_gcd = fn : mv_poly -> mv_poly -> mv_poly
```

Die Exponentenliste entspricht der Reihenfolge der reduzierten Variable in umgekehrter Reihenfolge, wobei der erste Eintrag der Hauptvariable entspricht.

Beispiel:

```
ML> val ggt1 = mv_gcd [(4,[2,2]),(8,[1,1]),(4,[0,0])]
                      [(2,[1,1]),(2,[0,0])];
val ggt1 = [(2,[1,1]),(2,[0,0])] : mv_poly
```

Im univariaten Fall werden in `mv_gcd` die Polynome vom Typ `mv_poly` in die interne Struktur `uv_poly` umgewandelt und dann dem entsprechenden Algorithmus zugeführt.

4.3.2 Univariater Fall

Für den Spezialfall der univariaten ggT-Berechnung ist auch der modulare Algorithmus implementiert, da es beim Berechnen über Restfolgen bei der Pseudodivision zu sehr großen Zwischenergebnissen kommen kann, die in SML als ganze Zahl nicht dargestellt werden können (Begrenzungen: -1073741824 bis 1073741823). Daher weicht man hier mit der Darstellung der Zahlen in $\mathbb{Z}_p$ aus. Der Algorithmus kann zwei Polynome in der internen Darstellung `uv_poly` verarbeiten und liefert wieder ein Polynom in der internen Darstellung `uv_poly`. Im Wesentlichen entspricht der Algorithmus dem Algorithmus 2.6 auf Seite 30.

Die Signatur lautet:

```
val uv_mod_gcd = fn : uv_poly -> uv_poly -> uv_poly
```

Das Beispiel demonstriert die Funktionsweise:

```
ML> val ggt1 = uv_mod_gcd [1,2,1] [1,1];
val ggt1 = [1,1] : uv_poly
```

4.4 Rechnen mit Bruchtermen

Das Rechnen mit Bruchtermen baut auf die oben beschriebenen Funktionen zur GGT-Berechnung auf. Die hier vorgestellten Funktionen zu den Grundrechenarten bilden die mathematische Grundlage für das im nachfolgenden Abschnitt beschriebene 'reverse rewriting' dar.

4.4.1 Kürzen von Bruchtermen

Für das Kürzen wurden zwei Funktionen implementiert:

cancel_ kürzt einen Bruchterm und liefert die Bedingung mit, unter der die Division durch 0 zu vermeiden ist (in der Folge als 'Division-0-Bedingung' bezeichnet).

factor_out_ggt hebt den ggT heraus und belässt ihn als Faktor neben Zähler und Nenner stehen.

Die Funktion cancel_ nimmt als Argument einen Isabelle-Term, der als äußersten Operator das Divisionszeichen (dh. die Funktion `HOL.cancel`) hat und deren beide Argumente Polynome in der auf Seite 55 definierten Normalform darstellen. Die beiden Polynome werden in die interne Struktur umgewandelt und der ggT der beiden berechnet. Dann wird der Zähler und Nenner durch den ggT dividiert. Der Bruch wird wieder als Isabelle-Term zusammengefügt und die Division-0-Bedingungen angehängt.

Die Signatur lautet:

```
val cancel_ = fn : theory -> (term -> term * term list) option
```

Das folgende einfache Beispiel demonstriert die Funktionalität. Dabei stellt

$$t = \frac{x^2 - 2 \cdot x \cdot y + y^2}{x^2 - y^2}$$

dar.

Erzeuge Isabelle-Term:

```
ML>  val t="(( x^^^2 + (-2) * x * y + y^^^2) /
             (x^^^2 + (-1) * y^^^2))";
val t = "(( x^^^2 + (-2) * x * y + y^^^2) /
          (x^^^2 + (-1) * y^^^2))" : string
ML> val t1=(term_of o the) (parse thy t);
val t1 =
     Const (#,#) $
     (# $ # $ (# $ #)) $
     (Const # $ (# $ #) $ (# $ # $ (# $ #))) : term
```

Wende die Funktion `cancel_` an:

```
ML> val t2 = cancel_ thy t1;
val t2 = Some (Const # $ (# $ #) $ (# $ # $ Free #),
               [Free # $ (# $ #)]) : (term * term list) option
```

`t3` enthält den gekürzten Bruch und `t4` die Division-0-Bedingung.

```
ML> val t3 = term2str (#1(the(t2)));
val t3 = "(x + (-1) * y) / (x + y)" : string
ML>  val t4 =  term2str (hd(#2(the(t2))));
val t4 = "not (x + (-1) * y = 0)" : string
```

Das ist in lesbarer Form

$$\frac{(x + (-1) \cdot y)}{(x + y)} \quad \wedge \quad (x + (-1) \cdot y \neq 0).$$

Die Funktion factor_out_gcd nimmt als Argument einen Isabelle-Term, der als äußersten Operator das Divisionszeichen (dh. die Funktion `HOL.cancel`) hat, dessen beide Argumente Polynome in der auf Seite 55 definierten Normalform darstellen. Die beiden Polynome werden in die interne Struktur umgewandelt und der ggT der beiden berechnet. Dann wird im Zähler und im Nenner im Gegensatz zum ersten Fall der ggT nur herausgehoben und nicht gekürzt. Anschließend wird der Bruch wieder als Isabelle-Term zusammengefügt.

Die Signatur lautet:

```
val factor_out_gcd =
         fn : theory -> term -> (term * term list) option
```

Das folgende Beispiel zeigt mit obigem `t1` die Funktionalität.

```
ML> val t2=factor_out_gcd thy t1;
val t2 =
        Const (#,#) $
        (# $ # $ (# $ #)) $
        (Const # $ (# $ #) $ (# $ # $ Free #)) : term
ML> val t3 = term2str (#1(the t2));
val t3 = "((-1) * y + x) * ((-1) * y + x) /
          ((y + x) * ((-1) * y + x))" : string
```

Das ist in lesbarer Form

$$\frac{(x + (-1) \cdot y) \cdot (x + (-1) \cdot y)}{(x + y) \cdot (x + (-1) \cdot y)}.$$

4.4.2 Addieren von Bruchtermen

Ziel dieses Abschnittes ist ein Algorithmus, der für eine beliebig lange Summe von Bruchtermen alle Terme auf einen Nenner bringt.

z.B. die folgenden Summanden

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

sollen so umgeformt werden, dass man sie auf einen Bruchstrich schreiben kann, also

$$\frac{1 \cdot (x-1)}{(x+1)(x-1)} + \frac{1 \cdot (x+1)}{(x+1)(x-1)} + \frac{1}{(x+1)(x-1)}.$$

Von einem derartigen Ausdruck kann man dann mit Rewriting-Schritten weiter rechnen.

Im Prinzip berechnet man das kleinste gemeinsame Vielfache von den Nennern der Bruchterme und "faktorisiert" quasi anschließend den Nenner.

Die benötigte Funktion ist also das kgV von Polynomen. Wie in Abschnitt 1.4.2 schon erwähnt wurde, kann man das kgV über den ggT ausrechnen.

$$\mathrm{kgV}(a,b) = \frac{a \cdot b}{\mathrm{ggT}(a,b)}$$

Berechnung des kgV
Der implementierte Algorithmus zur Berechnung des kleinsten gemeinsamen Vielfaches von zwei multivariaten Polynomen kann zwei Polynome in der internen Darstellung verarbeiten und liefert wieder ein Polynom in der internen Darstellung.

Die SML-Signatur der Funktion `mv_lcm` lautet:

```
val mv_lcm = fn : mv_poly -> mv_poly -> mv_poly
```

Das folgende Beispiel demonstriert die Funktionalität.

Zu berechnen ist:

$$\begin{aligned} &\mathrm{kgV}(4 \cdot x^2 - 8 \cdot x \cdot y + 4 \cdot y^2, x^2 - y^2) = \\ &= \mathrm{kgV}(4 \cdot (x-y)^2, (x-y) \cdot (x+y)) \\ &= 4 \cdot (x-y)^2 \cdot (x+y) \\ &= 4 \cdot x^3 - 4 \cdot x^2 \cdot y - 4 \cdot x \cdot y^2 + 4 \cdot y^3 \end{aligned}$$

```
ML> val kgv = mv_lcm [(4,[2,0]),(~8,[1,1]),(4,[0,2])]
                     [(1,[2,0]),(~1,[0,2])];
val kgv = [(4,[3,0]),(~4,[2,1]),(~4,[1,2]),(4,[0,3])] : mv_poly
```

Berechnung des gemeinsamen Nenners
Auch hier werden wieder zwei Funktionen zur Verfügung gestellt:

add_fractions_ ... liefert gleich das Ergebnis mit den Division-0-Bedingungen (für die direkte Berechnung)

common_nominators_ ... liefert die erweiterten Brüche mit gleichen Nennern (für die stufenweise Berechnung)

Die Funktion add_fractions_ wandelt die zu addierenden Bruchterme in eine Liste um. Anschließend berechnet sie den gemeinsamen Nenner von den ersten beiden gekürzten Termen der Liste und erzeugt einen neuen Bruchterm. Dann wird der gemeinsame Nenner dieses Bruchterms und des nächsten in der Liste berechnet. Dieser Prozess wird so lange fortgeführt, bis die Liste leer ist. Dann werden die Zähler mit den entsprechenden Faktoren multipliziert und anschließend addiert. Zuletzt wird der gesamte Bruch gekürzt. Damit erhält man eine eindeutige Darstellung, den sogenannten normierten Bruch.

Voraussetzung ist auch für diese Funktion, dass es sich um Bruchterme handelt, die aus Termen in Normalform bestehen. Enthält die Liste nur ein Element, dann wird einfach die Funktion `cancel_` aufgerufen.

Die Signatur lautet:

```
val add_fractions_ =
        fn : theory -> term -> (term * term list) option
```

Das folgende Beispiel demonstriert die Funktionalität:

```
ML> val t = "((7*x^^^2 + y)/(x + 3*y)) +
          ((-24*x^^^2*y + 5*x*y + 21*y^^^2)/(x^^^2 + -9*y^^^2)) +
          ((4*x^^^2 + -6*y)/(x + -3 * y))";
val t = "((7 * x^^^2 + y) / (x + 3 * y)) +
         ((-24 * x^^^2 * y + 5 * x * y + 21 * y^^^2) /
                                          (x^^^2 + -9 * y^^^2)) +
         ((4 * x^^^2 + -6 * y) / (x + -3 * y))" : string
```

Hier wird der entsprechende Isabelle-Term erzeugt.

```
ML> val t1=(term_of o the) (parse thy t);
val t1 =  Const (#,#) $
          (# $ # $ (# $ #)) $
          (Const # $ (# $ #) $ (# $ # $ (# $ #))) : term
```

Jetzt kann die Funktion aufgerufen werden.

```
ML> val t2=add_fractions_ thy t1;
val t2 = Some (Const # $ (# $ #) $ (# $ # $ Free #),
             [Free # $ (# $ #)]) : (term * term list) option
```

Das Ergebnis wird in eine lesbare Form gebracht. `t3` enthält, den komplett vereinfachten Term und `t4` die Division-0-Bedingung.

```
ML> val t3 = term2str (#1(the t2)));
val t3 = "11 * x ^^^ 2 / (x + 3 * y)" : string
ML> val t4 = term2str (hd(#2(the t2))));
val t4 = "not ((-1) * x + 3 * y = 0)" : string
```

Für die Funktion common_nominators_ gelten die gleichen Voraussetzungen wie für `add_fractions_`.

Diese Funktion berechnet aber zuerst das kgV von allen Nennern der Bruchterme in der Liste und multipliziert alle Zähler und Nenner mit den entsprechenden Faktoren, damit alle Nenner (ausmultipliziert gedacht) die gleichen Polynome repräsentieren. Dann wird der Nenner "quasi faktorisiert", d.h. es werden alle bekannten Faktoren herausgehoben und das Polynom als Produkt der Faktoren dargestellt.

Die Signatur lautet:

```
val common_nominators_ =
          fn : theory -> term -> (term * term list) option
```

Das folgende Beispiel verwendet die Summe der Bruchtermen `t1` von oben. Auf diese Summe wird dann die Funktion angewendet und anschließend in `t3` lesbar dargestellt.

```
ML> val t2 = common_nominators_ thy t1;
val t2 = Some (Const # $ (# $ #) $ (# $ # $ (# $ #)),[])
         : (term * term list) option
ML> val t3 = term2str (#1(the t2));
val t3 =
"(7 * x ^^^ 2 + y) * ((-1) * x + 3 * y) /
 ((x + 3 * y) * ((-1) * x + 3 * y)) +
 ((-24 * x ^^^ 2 * y + 5 * x * y + 21 * y ^^^ 2) * (-1) /
  ((x ^^^ 2 + -9 * y ^^^ 2) * (-1)) +
  (4 * x ^^^ 2 + -6 * y) * ((-1) * x + (-3) * y) /
  ((x + -3 * y) * ((-1) * x + (-3) * y)))" : string
```

4.5 Reverse Rewriting für schrittweise Interaktivität

Mit 'reverse rewriting' (RR)[3] wird ein im $\mathcal{ISAC}$-Projekt entwickeltes Verfahren bezeichnet, das schrittweise begründendes Vorgehen in Algorithmen ermöglicht, die nicht mit Rewriting arbeiten. Das in dieser Diplomarbeit entwickelte Bruchrechnen ist die erste Anwendung dieses Verfahrens. Die Idee des Verfahrens soll folgendes Beispiel erläutern.

Der folgende Bruch soll schrittweise gekürzt werden:

$$\frac{9-x^2}{9-6\cdot x+x^2} \quad = \quad \text{Regel: } 6=2\cdot 3 \tag{4.1}$$

$$= \frac{9-x^2}{9-2\cdot 3\cdot x+x^2} \quad = \quad \text{Regel: } 9=3^2 \tag{4.2}$$

$$= \frac{9-x^2}{3^2-2\cdot 3\cdot x+x^2} \quad = \quad \text{Regel: } a^2-2\cdot a\cdot b+b^2=(a-b)\cdot(a-b) \tag{4.3}$$

$$= \frac{9-x^2}{(3-x)\cdot(3-x)} \quad = \quad \text{Regel: } 9=3^2 \tag{4.4}$$

$$= \frac{3^2-x^2}{(3-x)\cdot(3-x)} \quad = \quad \text{Regel: } a^2-b^2=(a+b)\cdot(a-b) \tag{4.5}$$

$$= \frac{(3+x)\cdot(3-x)}{(3-x)\cdot(3-x)} \quad = \quad \text{Regel: if } c\neq 0 \text{ then } \frac{a\cdot c}{b\cdot c}=\frac{a}{b} \tag{4.6}$$

$$= \frac{(3+x)}{(3-x)} \quad \text{Division-0-Bedingung: } 3-x\neq 0 \tag{4.7}$$

Die Erzeugung dieser Schritte beruht auf folgender Idee: Die Funktion `factor_out_gcd` erzeugt, unsichtbar für den Benutzer, die Formel (4.6), die ebenso unsichtbar mittels einer *kanonischen Regelmenge* (dh. *konfluente* und *terminierenden* Regelmenge [BN98]) zurückverwandelt wird in die Formel (4.1). Dabei werden die angewendeten Umschreiberegeln protokolliert und in symmetrischer Form (linke und rechte Seite vom Gleichheitszeichen vertauscht) bereitgestellt – das ist genau die Liste der im Beispiel oben angewendeten Umschreiberegeln.

4.5.1 Der Interpreter und seine Datenstrukturen

Das Bruchrechnen folgt gewissen Algorithmen. Algorithmen werden in $\mathcal{ISAC}$ durch *Skripts* beschrieben. Der Interpreter der Skripts ist so konstruiert [Neu01c], dass er in folgender Weise mit dem Benutzer interagiert:

[3] Der deutsche Ausdruck müsste heißen: 'rückwärts Umschreiben'. Da 'Umschreiben' schon nicht für 'rewriting' gebräuchlich ist, wird hier auch auf das Eindeutschen des zusammengesetzten Ausdrucks verzichtet.

1. Ausgehend von der aktuellen Formel findet der Intepreter die nächste anwendbare Regel (Funktion `next_rule`),
2. übergibt der Interpreter diese Regel dem Benutzer als Vorschlag für den nächsten Schritt, und auch die Kontrolle darüber, welche Regel tatsächlich angewendet wird,
3. nimmt der Interpreter die vom Benutzer eingegebene Regel entgegen (dies kann auch eine andere sein als die vorgeschlagene!) und sucht sie im zugehörigen Skript auf (Funktion `locate_rule`); wenn dies gelungen ist,
4. kann der Interpreter mit Punkt 1 fortfahren.

Im Punkt 3 hat der Benutzer die Wahl, anstatt der Regel eine Formel einzugeben. Diese Formel ist als Ergebnis des nächsten Schrittes anzusehen. Der Interpreter hat also die Aufgabe, eine gültige Ableitung von der aktuellen Formel zur neu eingegebenen Formel zu erzeugen (Funktion `attach_rule`). Diese Ableitung braucht eine Regelmenge aus den RRS (siehe unten).

Nun wird, wie erwähnt, Bruchrechnen, insbesonders das Kürzen, nicht in ISAC-Skripts beschrieben, sondern in SML-Funktionen dem Euklidischen Algorithmus (bzw. entsprechenden Verallgemeinerungen) folgend. Daher weist der Datentyp für die Skripts eine Alternative für das RR auf:

```
datatype scr = (*script*)
    Script of ... normal scripts
  | Rfuns of (*reverse rewrite functions*)
             {init_state : term -> rrlsstate,
              normal_form: term -> (term * term list) option,
              locate_rule: rule list list -> term -> rule
                           -> (rule * (term * term list)) list,
              next_rule  : rule list list -> term -> rule option,
              attach_form: rule list list -> term -> term
                           -> (rule * (term * term list)) list};
and rls =      (*ruleset*)
    Rls of ... normal rulesets
  | Rrls of  (*reverse rewrite set*)
            {preconds: term list,
             scr     : scr};
```

Anstelle des Skripts in den üblichen Algorithmen tritt also beim RR ein Satz von Funktionen, den *reverse rewrite functions (RRF)*, die weiter unten beschrieben werden.

Der Datentyp `scr` enthält die statischen Informationen, die für alle Anwendungen gleichermaßen gelten; die dynamischen Daten für die Interaktion bezüglich eines speziellen Bruchterms werden im *Interpreterstatus* gehalten:

```
datatype istate =(*interpreter state*)
  ScrState of scrstate (*for script interpreter*)
| RrlsState of   (*for reverse rewriting*)
   (term *       (*the current formula*)
    rule list    (*reverse rule sets*)
          list * (*may be serveral, eg. in norm_rational*)
    (rule *      (*Thm (+ Thm generated from Calc) resulting in*)
     (term *     (*... rewrite with ...*)
      term list))(*... assumptions*)
        list);   (*derivation from given term to normalform
                    in reverse order with sym_thm*)
```

Für die gegenständliche Beschreibung ist nur das zweite Element von Bedeutung, die Liste der `rule list`'s, die *reverse rule sets (RRS)*. Diese Liste taucht als Argument in einigen RRF (rewrite functions – siehe unten) auf. Der Datentyp `rule` ist folgendermaßen definiert:

```
datatype rule =
  Thm of (string * thm)
| Calc of
         string *
         (string -> term -> theory -> (string * term) option);
```

Eine Regel (`rule`) besteht entweder aus einem Theorem `Thm` oder einer Rechenanweisung `Calc`. Das Theorem wird mit seinem Namen als `string` und der zugehörigen Formel `thm` beschrieben. Eine Rechenanweisung wird mit dem Namen des Operators als `string` und der zugehörigen Rechenfunktion für die Evaluation der numerischen Argumente beschrieben.

4.5.2 Funktionen für schrittweise Interaktion

Der Datentyp Skript (`scr`) enthält Funktionen, die der Interpreter verwendet, um die auf Seite 66 beschriebene schrittweise Interaktion mit dem Benutzer zu realisieren. Diese Funktionen werden 'reverse rewrite functions' (RRS) genannt und sind folgendermaßen festgelegt:

Die Initialisierung des Interpreterstatus erfolgt, bevor die Interaktion bezüglich eines bestimmten Bruchterms beginnt. Dies geschieht durch die Funktion `init_state`, die insbesonders die in der Folge notwendigen RRS bereitstellt:

```
val init_state = fn : term -> istate
```

Diese Funktion initialisiert also den Status des interaktiven Interpreters. Der Status wird in folgender Datenstruktur gespeichert:

```
type rrlsstate =      (*Status für das 'reverse rewriting'*)
     (term *          (*der aktuelle Term*)
      term *          (*der resultierende Term*)
      rule list       (*die 'reverse rule list'*)
      list *          (*es kann mehrere Listen geben,
                        z.B. bei 'norm_rational'*)
      (rule *         (*Thm (+ Thm generiert von Calc) liefert ...*)
       (term *        (*Resultatsterm mit seinen*)
        term list))   (*Einschränkungen*)
      list);          (*die Abstammung von einem gegebenen Term
                        von der Normalform in umgekehrter Reihen-
                        folge mit sym_thm kann von hier extrahiert
                        werden*)
```

Übergabeparameter:
`term` : Term, der bearbeitet werden soll

Rückgabewert:
`istate` : Interpreterstatus

Die Normalform-Funktion ist jene Funktion, die das gewünschte Ergebnis in *einem* Schritt berechnet, zum Beispiel einen Bruchterm kürzt oder Summen in einem Bruchterm auf gleichen Nenner bringt und addiert. Der Rückgabewert umfasst die resultierende Formel (`term`) und eventuell entstandene Annahmen (`term list`, beim Bruchrechnen insbesonders Division-0-Bedingungen). Wenn die Normalform-Funktion den aktuellen Term nicht umschreiben kann, sie also nicht anwendbar ist, liefert sie nichts zurück (`None`, Typ `option`).

```
val normal_form = fn : term -> (term * term list) option
```

Übergabeparameter:
`term` : Term, der in Normalform gebracht werden soll

Rückgabewert:
`term` : Term in Normalform
`term list` : eventuelle Division-0-Bedingungen

Eine Regel im RRS lokalisieren ist jene Aufgabe, die notwendig ist, um dem Benutzer die Freiheit in der Wahl der Regel zu überlassen, die als nächste auf die aktuelle Formel angewendet werden soll. Die Regel kann also auch so gewählt werden, dass sie zwar auf die aktuelle Formel anwendbar ist, aber nichts mit der aktuellen Normalform-Funktion zu tun hat. Wenn die Mathematik-Maschine zum Beispiel gerade mit dem Kürzen befasst ist, und der Benutzer eine Regel zum Addieren eingegeben hat, dann wird sie der Interpreter beim Kürzen 'nicht lokalisieren', sondern versuchen, diese Regel in einem anderen Skript zu 'lokalisieren' (und bezüglich des Kürzens eine *leere* Liste zurückliefern).

Wenn die Funktion, `locate_rule`, eine Regel lokalisieren konnte, liefert sie eine Ableitung von der aktuellen Formel bis zu jener Formel zurück, auf die die Regel letztendlich angewendet wird. Beim Bruchrechnen wird die Liste immer *ein* Element enthalten, bestehend aus der Regel, die auf die aktuelle Formel anwendbar ist, und die resultierende Formel mit der Liste der entstandenen Annahmen.

```
val locate_rule = fn : rule list -> term -> rule
                       -> (rule * (term * term list)) list.
```

Diese Funktion überprüft also, ob eine Regel R eine 'Kürzungsregel' ist. Wenn sie eine ist, dann liefert die Funktion die Liste der Regeln (mit den zugehörigen resultierenden Termen), die angewendet werden müssen, bevor man die Regel anwendet.

Übergabeparameter:
`rule list` : die 'reverse rule'-Liste
`term` : Term, auf den die Regel angewendet werden soll
`rule` : Regel, die angewendet werden soll

Rückgabewert:
Eine Liste von Tripel:
`rule` : Eine Rewiting-Regel die ...
`term` : ... den Term liefert ...
`term list` : ... mit den Division-0-Bedingungen.
Die Liste ist leer, wenn die Regel nichts mit Kürzen zu tun hat.

Die nächste Regel bestimmen braucht als Argument *ein* RRS (`rule list`, wie die Funktion `locate_rule` oben auch) und die aktuelle Formel. Es ist möglich, dass keine Regel im RRS gefunden wird, die auf die aktuelle Formel anwendbar ist; daher ist der Rückgabewert vom Typ `option`. Normalerweise tritt dieser Fall nur auf, wenn die aktuelle Formel den Wert erreicht hat, den auch die Normalform-Funktion berechnet. Der Interpreter kann diesen Fall also von eventuellen Fehlerfällen unterscheiden.

```
val next_rule = fn : rule list -> term -> rule option
```

Diese Funktion liefert also die nächste Regel, die angewendet werden muss um zu kürzen.

Übergabeparameter:
`rule list` : die 'reverse rule'-Liste
`term` : ein Bruchterm, für den ...

Rückgabewert:
`rule option` : ... diese Regel vorgeschlagen wird um zu kürzen; wenn es keine Regel mehr gibt, dann ist der Term schon vollständig gekürzt.

Eine Ableitung für die eingegebene Formel wird von der letzten der RRF errechnet. `attach_form` nimmt drei Argumente: wieder ein RRS, die aktuelle Formel und die neu eingegebene Formel, für die eine Ableitung aus der aktuellen Formel berechnet werden soll. Die Ableitung besteht, wie bei `locate_rule`, aus einer Liste von Regeln gepaart mit resultierender Formel plus Annahmen. Wenn keine Ableitung erstellt werden kann, wird eine leere Liste zurückgegeben.

```
val attach_form = f : rule list -> term -> term
                      -> (rule * (term * term list)) list
```

Diese Funktion überprüft also einen eingegebenen Term TI, ob er zum aktuellen Kürzungsprozess passt, indem sie versucht ihn von gegebenen Term TG abzuleiten.

Übergabeparameter:
`term` : TG, der letzte Term vom aktuellen Kürzungsprozess (Benutzereingabe oder Mathematikmaschine)
`term` : TI, der nächste Term als Benutzereingabe

Rückgabewert:
Eine Liste von Tripel:
`rule` : Die Regel, die angewendet werden muss, um zum Term TI zu gelangen,
`term` : ... liefert diesen Term ...
`term list`: ... mit den Division-0-Bedingungen.
Es kann mehrere solche Regeln mit den zugehörigen Termen geben. Die Liste ist leer, wenn der eingegebene Term nicht zum Kürzungsprozess gehört.

4.5.3 Die Reverse Rulesets für das Bruchrechnen

ISAC realisiert die schrittweise begründende Interaktion mit dem Benutzer mit Hilfe von Regelmengen (konfluente und terminierende, kurz 'kanonische', Termrewritingsysteme [BN98]). Algorithmen, die im Allgemeinen nicht mit Rewriting funktionieren (können) wie das Kürzen von Brüchen, werden in ISAC in den oben beschriebenen 'reverse rule sets' (RRS) abgebildet, die wiederum schrittweise Interaktivität realisieren.

Da die RRS die Regeln von kanonischen Regelmengen umkehren (zum Beispiel wird aus $(a-b)^2 \rightarrow a^2 - 2ab + b^2$ die Regel $a^2 - 2ab + b^2 \rightarrow (a-b)^2$) lässt sich intuitiv erwarten, dass die Regelmengen in gewisser Weise 'anti-konfluent' werden und *nicht* terminieren. Diese Frage ist allgemein noch nicht gelöst. Die gegenständliche Diplomarbeit hat die Funktionen bereitgestellt, um von experimenteller Seite an die Frage heranzugehen.

An dieser Stelle werden die 'reverse rule sets' spezifiziert (ihre Implementierung liegt, wie erwähnt, ausserhalb des Rahmens dieser Diplomarbeit). Da die 'reverse rule sets' dieselben Voraussetzungen haben und dieselben Ergebnisse wie die zugeörigen Normalform-Funktionen liefern, werden die Namen der Normalform-Funktionen übernommen, und auch die Abkürzung RRS. Folgende RRS sind also in ISAC zur Realisierung schrittweisen Bruchrechnens vorgesehen:

Die Grundoperationen für Bruchterme, Kürzen und Addieren erledigen die folgenden RRS `cancel` und `add_fractions`:

```
val cancel =
  Rrls
    {preconds=#,
     scr=Rfuns
           {attach_form=#,init_state=#,locate_rule=#,
            next_rule=#,normal_form=#}} : rls
```

Vorbedingungen: Der zu kürzende Term besitzt *einen* Bruchstrich als äussersten Operator; Zähler und Nenner sind Polynome in der Normalform laut Definition von Seite 55.
Ergebnis bei der Anwendung: die Taktik `Rewrite_Set cancel` liefert:
(a) `Error 'not-applicable'`, falls eine der Vorbedingungen nicht erfüllt ist, oder der Bruchterm bereits vollständig gekürzt ist,
(b) `Some (term, asms)` wobei `term` der vollständig gekürzte Bruch ist, und `asms` die Liste der erzeugten Division-0-Bedinungen.
Ein Beispiel findet sich auf Seite 61.

```
val add_fractions =
  Rrls
    {preconds=#,
     scr=Rfuns
           {attach_form=#,init_state=#,locate_rule=#,
            next_rule=#,normal_form=#}} : rls
```

Vorbedingungen: Der zu kürzende Term besitzt (mindestens) ein Plus als äussersten Operator; Zähler und Nenner der Summanden sind Polynome in der Normalform laut Definition von Seite 55, und alle Summanden sind vollständig gekürzte Brüche oder Subterme ohne Bruchstrich.
Ergebnis bei der Anwendung: die Taktik `Rewrite_Set cancel` liefert:
(a) `Error 'not-applicable'`, falls eine der Vorbedingungen nicht erfüllt ist,
(b) `Some (term, asms)`, wobei `term` ein Term mit *einem* Bruchstrich und `asms` die Liste der erzeugten Division-0-Bedinungen ist.
Ein Beispiel findet sich auf Seite 64.

Die Grundoperationen für Binome, Kürzen und Addieren, erledigen die folgenden RRS `cancel_binom` und `add_binoms`. Diese beiden RRS kommen der gewohnten Schreibweise entgegen, die insbesonders bei Binomen das $-$ anstelle des $+(-\dots)$ verwenden (siehe Seite 56). Die beiden RRS entsprechen daher völlig den oben definierten Funktionen `cancel` und `add_fractions`, mit dem Unterschied, dass die andere polynomiale Normalform in Argument und Resultat auftritt.

```
val cancel_binom =
  Rrls
    {preconds=#,
     scr=Rfuns
         {attach_form=#,init_state=#,locate_rule=#,next_rule=#,
            normal_form=#}} : rls

val add_binoms =
  Rrls
    {preconds=#,
     scr=Rfuns
         {attach_form=#,init_state=#,locate_rule=#,next_rule=#,
            normal_form=#}} : rls
```

Beispiele sind:

$$\frac{9-6x+x^2}{9-x^2} \quad \xrightarrow{Rewrite\_Set\ \ cancel\_binom} \quad Some\left(\frac{3-x}{3+x}\ ,\ [\,3-x\neq 0\,]\right)$$

$$\frac{1}{9-x^2}+\frac{2}{9+6x+x^2} \quad \xrightarrow{Rewrite\_Set\ \ add\_binoms} \quad Some\left(\frac{1\cdot(3+x)+2\cdot(3-x)}{(3+x)\cdot(3+x)\cdot(3-x)}\ ,\ [\,]\right)$$

Die Normalform für Bruchterme dient u.a. dazu, beliebige Terme auf Äquivalenz zu überprüfen. Das folgende RRS berechnet die Normalform:

```
val norm_rational =
  Rrls
    {preconds=#,
     scr=Rfuns
           {attach_form=#,init_state=#,locate_rule=#,
            next_rule=#,normal_form=#}} : rls
```

Vorbedingungen: Der zu normierende Term ist beliebig, mit der Einschränkung, dass *nur* natürlichzahlige Exponenten auftreten dürfen (dh. auch keine Wurzeln).
Ergebnis bei der Anwendung: die Taktik `Rewrite_Set norm_rational` liefert einen Bruch mit *einem* Bruchstrich, der auch vollständig gekürzt ist, und Zähler und Nenner in polynomialer Normalform laut Definition auf Seite 55 hat. Falls der Nenner 1 werden sollte, wird er angeschrieben.
Beispiele sind:

$$\frac{\frac{1}{x}-\frac{1}{x+1}}{\frac{x}{x+1}} \quad \xrightarrow{Rewrite\_Set\ \ norm\_rational} \quad Some\left(\frac{1}{x^2}\ ,\ [\,x+1\neq 0\,]\right)$$

$$\frac{5}{35\cdot x-25\cdot y}\cdot(49\cdot x^2-70\cdot x\cdot y+25\cdot y^2) \quad \xrightarrow{Rewrite\_Set\ \ norm\_rational} \quad Some\left(\frac{7\cdot x+(-5)\cdot y}{1}\ ,\ [\,7\cdot x+(-5)\cdot y\neq 0\,]\right)$$

4.5.4 Ein Beispiel

Der folgende Bruch soll gekürzt werden, wobei die Binom-Schreibweise verwendet wird:

$$\frac{9-x^2}{9-6\cdot x+x^2} \quad \xrightarrow{Rewrite\_Set\ \ cancel\_binom} \quad Some\left(\frac{3+x}{3-x}\ ,\ [3-x\neq 0]\right)$$

Zuerst wird die String-Repräsentation des Bruchterms durch einige Hilfsfunktionen zu einem Term `t` geparst:

```
ML> val t = (term_of o the o (parse thy))
            "(9 - x ^^^ 2) / (9 - 6 * x + x ^^^ 2)";
val t =
  Const (#,#) $
  (# $ # $ (# $ #)) $
  (Const # $ (# $ #) $ (# $ # $ Free #)) : term
```

Die Form, in der SML den Typ `term` zurückgibt, ist schlecht zu interpretieren (man sieht einige Konstanten, `Const`, und Variable, `Free`; die Datenelemente dazwischen sind in den `#` komprimiert). Eine Hilfsfunktion macht Terme als Strings wieder lesbar:

```
ML> term2str t;
 val it = "(9 - x ^^^ 2) / (9 - 6 * x + x ^^^ 2)" : string
```

Nun holt man die 'reverse rewrite functions' (RRF) aus dem 'reverse rewrite set' (RRS) `cancel_binom`:

```
ML> val Rrls {scr=Rfuns
              {init_state=ini,locate_rule=loc,
               next_rule=nex,normal_form=nor,...},...} = cancel_binom;
val ini = fn :
    term -> rrlsstate
val loc = fn :
    rule list list -> term -> rule -> (rule * (term * term list)) list
val nex = fn :
    rule list list -> term -> rule option
val nor = fn :
    term -> (term * term list) option
```

Mithilfe der Normalform-Funktion, hier mit `nor` bezeichnet, kann der Bruchterm in *einem* Schritt gekürzen werden:

```
ML> val Some (t',_) = nor t;
ML> term2str t';
val it = "(3 + x) / (3 - x)" : string
```

Der Zweck des RRS liegt allerdings in schrittweise begründender Interaktivität mit dem Benutzer. Für diese muss der Interpreter-Status mithilfe von `ini` initialisiert werden, von dem insbesonders die 'reverse rule sets' `revsets` von Interesse sind:

```
ML> val (_,_,revsets,_) = ini t;
val revsets =
   [[Thm ("sym_#power_3_2","9 = 3 ^^^ 2"),
     Thm
       ("sym_real_plus_minus_binom",
        "?a ^^^ 2 - ?b ^^^ 2 = (?a + ?b) * (?a - ?b)"),
```

```
    Thm ("sym_#mult_2_3","6 = 2 * 3"),
    Thm
      ("sym_real_minus_binom_times",
       "?a ^^^ 2 - 2 * ?a * ?b + ?b ^^^ 2 = (?a - ?b) * (?a - ?b)"),
       Thm ("real_mult_div_cancel2",
            "?k ~= 0 ==> ?m * ?k / (?n * ?k) = ?m / ?n")
  ]] : rule list list
```

Dieses RRS ist offenbar konfluent und terminierend (was im Allgemeinen, wie erwähnt, nicht erwartet werden kann). Nun fragt man sich, welche Regel vom RRS bei gegebener aktueller Formel `t` und gegebenem `revset` vorgeschlagen wird:

```
ML> val Some r = nex revsets t;
val r = Thm ("sym_#mult_2_3","6 = 2 * 3") : rule
```

Das RRS schlägt also vor, eine der Konstanten zu faktorisieren. Hier wird diesem Vorschlag gefolgt und die Regel auf die aktuelle Formel angewendet. Da man ebenso eine andere Regel wählen hätten können, nimmt man die *locate_rule*-Funktion `loc`, die die Regel im RRS zu 'lokalisieren' versucht:

```
ML> val [(r,(t,asm))] = loc revsets t r;
val r = Thm ("sym_#mult_2_3","6 = 2 * 3") : rule
val t =
   Const (#,#) $
   (# $ # $ (# $ #)) $
   (Const # $ (# $ #) $ (# $ # $ Free #)) : term
val asm = [] : term list
```

Diese Regel kann das RRS natürlich lokalisieren, und liefert genau *einen* Ableitungsschritt zurück, der aus der gewählten Regel und dem Resultat ihrer Anwendung besteht – die Hilfsfunktion zeigt das Resultat als String:

```
ML> term2str t;
val it = "(9 - x ^^^ 2) / (9 - 2 * 3 * x + x ^^^ 2)" : string
```

Der nächste Schritt verläuft ebenso, dem Vorschlag des RRS folgend, und den Vorschlag zum übernächsten einholend:

```
ML> val Some r = nex revsets t;
val r = Thm ("sym_#power_3_2","9 = 3 ^^^ 2") : rule
ML> val [(r,(t,asm))] = loc revsets t r;
val r = Thm ("sym_#power_3_2","9 = 3 ^^^ 2") : rule
val t =
   Const (#,#) $
   (# $ # $ (# $ #)) $
   (Const # $ (# $ #) $ (# $ # $ Free #)) : term
val asm = [] : term list
ML> term2str t;
```

```
ML> val it = "(9 - x ^^^ 2) / (3 ^^^ 2 - 2 * 3 * x + x ^^^ 2)" : string
ML> val Some r = nex revsets t;
val r = Thm ("sym_#power_3_2","9 = 3 ^^^ 2") : rule
```

Im folgendem Schritt folgt man *nicht* diesem Vorschlag und zerlegt nicht die Konstante `9 = 3 ^^^ 2` , sondern faktorisiert gleich den Nenner. Dazu nimmt man eine *andere* Regel, `sym_real_minus_binom_times`:

```
ML> val rrr = Thm ("sym_real_minus_binom_times",
                   sym_real_minus_binom_times);
ML> val [(r,(t,asm))] = loc revsets t rrr;
val r =
   Thm
     ("sym_real_minus_binom_times",
      "?a ^^^ 2 - 2 * ?a * ?b + ?b ^^^ 2 = (?a - ?b) * (?a - ?b)") : rule
val t =
   Const (#,#) $
   (# $ # $ (# $ #)) $
   (Const # $ (# $ #) $ (# $ # $ Free #)) : term
val asm = [] : term list
ML> term2str t;
val it = "(9 - x ^^^ 2) / ((3 - x) * (3 - x))" : string
```

Die *locate*-Funktion `loc` konnte also auch eine andere als die vorgeschlagene Regel erfolgreich lokalisieren, und die Ableitung mit dem gewünschten Ergebnis errechnen.

Wenn man dagegen eine Regel anwendet, die für das Kürzen nicht zweckmässig erscheint, zum Beispiel das Kommutativgesetz der Multiplikation, erhält man *keine* Ableitung, also die leere Liste:

```
ML> real_mult_commute;
val it = "?z * ?w = ?w * ?z" : thm
ML> val rrr = Thm ("real_mult_commute", real_mult_commute);
ML> val derivation = loc revsets t rrr;
val derivation = [] : (term * term list) list
```

Die leere Liste zeigt dem Interpreter an, dass der Benutzer das Kürzen offenbar unterbrochen hat und vielleicht nun etwas anderes vorhat — der Interpreter muss in den entsprechenden Skripts nach einer Regel zu diesem Vorhaben suchen.

Nun folgt man wieder den Vorschlägen des Systems ...

```
ML> val Some r = nex revsets t;
val r = Thm ("sym_#power_3_2","9 = 3 ^^^ 2") : rule
ML> val [(r,(t,asm))] = loc revsets t r;
ML> term2str t;
val it = "(3 ^^^ 2 - x ^^^ 2) / ((3 - x) * (3 - x))" : string
```

```
ML>  val Some r = nex revsets t;
val r =
   Thm ("sym_real_plus_minus_binom",
        "?a ^^^ 2 - ?b ^^^ 2 = (?a + ?b) * (?a - ?b)") : rule
ML> val [(r,(t,asm))] = loc revsets t r;
ML> term2str t;
val it = "(3 + x) * (3 - x) / ((3 - x) * (3 - x))" : string
ML>  val Some r = nex revsets t;
val r =
   Thm ("real_mult_div_cancel2",
        "?k ~= 0 ==> ?m * ?k / (?n * ?k) = ?m / ?n") : rule
ML> val [(r,(t,asm))] = loc revsets t r;
ML> term2str t;
val it = "(3 + x) / (3 - x)" : string
ML> term2str asm;
val it = "3 - x ~= 0" : string
```

... und kommt mit drei Schritten zum erwarteten Ergebnis `(3 + x) / (3 - x)` mit der notwendigen Division-0-Bedingung `3 - x ~= 0`.

5. ZUSAMMENFASSUNG

In dieser Diplomarbeit wird versucht, eine Lösungvariante für das Bruchrechnen im *ISAC*-System zu finden. Dabei wird überlegt, was alles notwendig ist, um überhaupt Bruchrechnen zu ermöglichen. Man beschränkt sich auf Brüche mit Polynomen ohne transzendente Funktionen, da das Zulassen von transzendenten Funktionen die Komplexität einer Diplomarbeit sprengen würde.

Im ersten Kapitel wird das *ISAC*-Projekt vorgestellt und seine Prinzipen dargestellt. Weiters wird die Diplomarbeit in diesem Projekt positioniert und deren Ziele abgesteckt.

In den zwei folgenden Kapiteln wird der mathematische Hintergrund für die Berechnung des größten gemeinsamen Teilers für univariate und multivariate Polynome, der als grundlegende Funktionalität benötigt wird, vorgestellt. Dabei werden verschiedene Algorithmen aufgeführt, inklusive den implementierten modularen Zugängen.

In letzten Abschnitt dieser Diplomarbeit werden die bei der Implementierung verwendeten Datentypen präsentiert und anhand von einigen Bespielen die Verwendung der Algorithmen von den vorangegangenen Kapiteln besprochen. Weiters wird die Einbettung der Datenstrukturen und Algorithmen im *ISAC*-Projekt gezeigt. Die Funktionen, die die Interaktivität des *ISAC*-Systems gewährleisten sollen, werden über ihre Signatur und ihre Funktionsweise spezifiziert.

Wie man dieser Arbeit entnehmen kann, braucht Computer-Algebra ein sehr tiefgreifendes Wissen, sowohl von Mathematik als auch von Didaktik. Da die Algorithmen immer allgemeiner werden, das heißt immer komplexere Aufgaben lösen können, geraten die einfachen Rechenwege, wie sie in Schulen unterrichtet werden, immer mehr in den Hintergrund. Daher stellt die Präsentation von Berechnungen mit CAS, so dass sie von jedermann verstanden werden können, eine besondere Herausforderung dar. Das *ISAC*-Projekt ist der erste Schritt in diese neue Richtung, indem es den ganzen Rechenvorgang durch kleine verständliche Rechenschritte darstellen kann. Für Schüler und Studenten sollte so ein Werkzeug geschaffen werden, mit dem sie experimentell und effektiv lernen können.

ANHANG

A. GRUNDLEGENDE DEFINITIONEN

A.1 Definitionen und Hinweise zum Pseudo-Code

Prinzipiell ist der Pseudo-Code an den ANSI-C Standard angelehnt. Es werden ein paar spezielle Befehle für Listen eingeführt:

first(list): Diese Funktion liefert das erstes Element der Liste *list*

$$\mathit{first}((a_1, \ldots, a_n)) = a_1$$

red(list): Diese Funktion liefert die Liste *list* ohne ihr erstes Element

$$\mathit{red}((a_1, \ldots, a_n)) = (a_2, \ldots, a_n)$$

inv(list): Diese Funktion liefert die Liste *list* in umgekehrter Reihenfolge

$$\mathit{inv}((a_1, \ldots, a_n)) = (a_n, \ldots, a_1)$$

comp(elem,list): Diese Funktion hängt ein Element *elem* an der Liste *list* an

$$\mathit{comp}(c, (a_1, \ldots, a_n)) = (c, a_1, \ldots, a_n)$$

conc(list1,list2): Diese Funktion hängt eine Liste *list2* an eine zweite Liste *list1* an

$$\mathit{conc}((a_1, \ldots, a_n), (b_1, \ldots, b_n)) = (a_1, \ldots, a_n, b_1, \ldots, b_n)$$

A.2 Definitionen aus der Mathematik

Integritätsbereich: Ein kommutativer, nullteilerfreier Ring $(I, +, \cdot)$ heißt Integritätsbereich. Gibt es ein Element $e \in I$ mit der Eigenschaft $ex = xe = x$ für alle $x \in I$, dann wird e als Einselement von I bezeichnet. Das Einselement ist immer eindeutig bestimmt.

Euklidischer Ring: Ein kommutativer Ring R heißt Euklidischer Ring, wenn eine Funktion $\varphi : R\backslash\{0\} \to \mathbb{N}_0$ existiert, so dass gilt:

1. $\forall a, b \in R$ mit $a \cdot b \neq 0$ gilt: $\varphi(a) \leq \varphi(ab)$
2. zu allen $a, b \in R$ mit $b \neq 0$ existieren $q, r \in R$ mit $a = qb + r$, so dass $r = 0$ oder $\varphi(r) < \varphi(b)$ gilt. (Division mit Rest)

Ein Euklidischer Ring, der Integritätsbereich ist, wird Euklidischer Bereich genannt.

Primes Element: Sei R ein kommutativer Ring mit Eins, und seien $a_1, \ldots, a_n \in R$. $a_1, \ldots, a_n$ heißen relativ prim, falls 1_R ein größter gemeinsamer Teiler von $a_1, \ldots, a_n$ ist.
Ein $p \in R$ heißt prim, falls gilt:

1. $p \neq 0$ und p ist keine Einheit.
2. $\forall a, b \in R : p|ab \Rightarrow (p|a \vee p|b)$

Irreduzibles Element Es sei R ein kommutativer Ring mit Eins. Ein $c \in R$ heißt irreduzibel, wenn gilt:

1. $c \neq 0$ und c ist keine Einheit.
2. $\forall a, b \in R : c = ab \Rightarrow (a$ oder b ist eine Einheit$)$

ZPE-Ring: Ein Integritätsbereich I heißt ZPE-Ring (Zerlegung in PrimElemente), falls gilt:

1. Jede Nichteinheit $a \neq 0$ ist als Produkt von irreduziblen Elementen darstellbar, d.h. $a = c_1 \cdot \ldots \cdot c_n$, wobei c_i für $i = 1, \ldots, n$ irreduzibel ist.
2. Diese Darstellung ist eindeutig bis auf die Reihenfolge der irreduziblen Elemente und die Assoziiertheit.

In einem ZPE-Ring ist jedes irreduzible Element prim.

Ein Integritätsbereich ist genau dann ein ZPE-Ring, wenn jede Nichteinheit $a \neq 0$ als Produkt von Primelementen darstellbar ist, d.h. $a = p_1 \cdot \ldots \cdot p_n$ und p_i ist ein Primelement für $i = 1, \ldots, n$. [Fri96]

Quotientenkörper: Sei R ein kommutativer Ring und sei S eine multiplikativ abgeschlossene Menge, dann nennt man $S^{-1}R$ den Ring der Brüche. Falls $S = R\backslash\{\text{Nullteiler}\} \neq \emptyset$ gilt, dann heißt $S^{-1}R$ der komplette Ring der Brüche von R. Falls R ein Integritätsbereich und $S = R\backslash\{0\}$ ist, dann heißt $S^{-1}R$ der Quotientenkörper.

Größter gemeinsamer Teiler (ggT): Sei R ein Ring und seien $a_1, \ldots, a_n \in R$. Ein $d \in R$ heißt ein größter gemeinsamer Teiler (ggT) von $a_1, \ldots, a_n$, falls gilt:

1. d teilt alle a_i
2. Falls ein $c \in R$ alle a_i teilt, dann gilt $c|d$.

Man schreibt $d = \mathrm{ggT}(a_1, \ldots, a_n)$, wenn d ein größter gemeinsamer Teiler von $a_1, \ldots, a_n$ ist.

A.3 Wichtige Sätze aus der Algebra

Satz A.3.1 (Euklidischer Algorithmus). *Es sei R ein euklidischer Ring. Für zwei Elemente $x, y \in R \setminus \{0\}$ betrachtet man die Folge $z_0, z_1, \cdots \in R$, die induktiv durch*

$$
\begin{aligned}
z_0 &= x \\
z_1 &= y \\
z_{i+1} &= \begin{cases} \textit{der Rest von } z_{i-1} \textit{ bei Division durch } z_i, \textit{ falls } z_i \neq 0, \\ 0 \textit{ sonst} \end{cases}
\end{aligned}
$$

bestimmt ist. Dann gibt es einen kleinsten Index $n \in \mathbb{N}$ mit $z_{n+1} = 0$. Für dieses n gilt $z_n = \mathrm{ggT}(x, y)$.
[Bos99]

Satz A.3.2 (Entwicklungssatz von Laplace). *Ist $n \geq 2$ und A eine quadratische $n \times n$ Matrix, so gilt für jedes $i \in \{1, \ldots, n\}$*

$$\det A = \sum_{j=1}^{n} (-1)^{i+j} \cdot a_{ij} \cdot A'_{ij}$$

(Entwicklung nach der i-ten-Zeile)

und für jedes $j \in \{1, \ldots, n\}$

$$\det A = \sum_{i=1}^{n} (-1)^{i+j} \cdot a_{ij} \cdot A'_{ij}$$

(Entwicklung nach der j-ten-Spalte).

Dabei bezeichnet A'_{ij} *die Matrix, die jeweils entsteht, wenn man die i-te Zeile und j-te Spalte in A streicht.*
[Fis95, Satz 3.3.3]

Satz A.3.3 (Chinesisches Rest Problem). *Gegeben sind in einem euklidschen Ring D:*
$r_1, \ldots, r_n \in D$ *(Reste)*
$m_1, \ldots, m_n \in D \setminus \{0\}$ *(Moduli)*
Gesucht ist ein $r \in D$*, so dass für alle* $1 \leq i \leq n$ *gilt:*

$$r = r_i \pmod{m}_i$$

Satz A.3.4 (Chinesischer Restsatz). *Seien* $A_1, \ldots, A_n$ *Ideale in einem Ring R, so dass* $R^2 + A_i = R$ *für alle i und* $A_i + A_j = R$ *für alle* $i \neq j$*. Wenn* $b_1, \ldots, b_n \in R$*, dann existiert ein* $b \in R$ *so, dass*

$$b \equiv b_i(\pmod{A})_i) \quad \textit{für } i = 1, \ldots, n.$$

Weiters ist b eindeutig bestimmt modulo dem Ideal $A_1 \cap \cdots \cap A_n$*.*
[Hun96]

Satz A.3.5 (Chinesischer Restalgorithmus). *Der Satz A.3.4 besagt, dass das Chinesische Rest Problem immer eine Lösung besitzt. Im speziellen Fall* $n = 2$ *gilt das also auch und der Algorithmus* `CRA_2` *löst dieses Problem.*

Input: r_1, r_2, m_1, m_2 *(stellen das Chinesische-Rest-Problem dar)*

Output: r *(Lösung des Chinesischen-Rest-Problems)*

Listing A.1: CRA_2

```
CRA_2(r1,r2,m1,m2)
    {
        c  = m1^(−1) mod m2;
        r1‘= r1 mod m1;
        d  = (r2−r1‘)c mod m2;
        r  = r1‘ + d ∗ m1;
        return(r);
    }
```

[Win96, S. 55]

A.3.1 Terminologie der abgeschnittenen Potenzreihen

Eine genauere Ausführung dieser Thematik ist in [SS92, Abschnitt 3] zu finden. Hier werden nur die wichtigsten Definitionen und Sätze angeführt.

Definition A.3.1 (Exponentenbereich und Termgradbereich). *Sei*

$$C = \sum_{i=1} c_i u^{e_i} \times \textit{(Monome die nicht u enthalten)}, \quad c_i \neq 0, \quad c_i \in K.$$

Man definiert den Exponentenbereich von der Variable u von C als $\mathrm{ran}_u(C) = (e_{\min}, e_{\max})$*, wobei* $e_{\min} = \min\{e_i | i = 1, 2, \dots\}$ *und* $e_{\max} = \max\{e_i | i = 1, 2, \dots\}$ *sind. Der Termgradbereich wird ähnlich definiert,* $\mathrm{tran}(C) = (E_{\min}, E_{\max})$*, wobei* $E_{\min}/E_{\max}$ *das Minimum/Maximum der Termgrade aller Terme von C bezeichnen.*

Lemma A.3.1. *Für Polynome, also endliche Potenzreihen, P_1 und P_2 gilt*

$$\begin{aligned} \mathrm{ran}_u(C_1 C_2) &= \mathrm{ran}_u(C_1) + \mathrm{ran}_u(C_2) \\ \mathrm{tran}(C_1 C_2) &= \mathrm{tran}(C_1) + \mathrm{tran}(C_2) \end{aligned}$$

Definition A.3.2 (Signifikante Terme). *C und $\tilde{C} \in K[y, \dots, z]$ seien so, dass aus C höher gradige Terme abgeschnitten werden müssen um $\tilde{C}$ zu erhalten. Sei* $\mathrm{tran}(C) = (E, E + \textit{einige})$*, besitzen C und $\tilde{C}$ die gleichen Terme, deren Termgrad aus dem Bereich $(0, E + \tilde{E})$ und deren Grade bzgl. u im Bereich $(0, E + \tilde{e}_u)$ für alle $u \in \{y, \dots, z\}$ liegen, und $\tilde{C}$ habe keine Terme außerhalb dieser Bereiche. Dann kann man sagen, dass $\tilde{C}$ ist $(\tilde{E}, \tilde{e}_y, \dots, \tilde{e}_z)$ signifikant w.r.t. C. Alle Terme von $\tilde{C}$ werden als die $(\tilde{E}, \tilde{e}_y, \dots, \tilde{e}_z)$ signifikanten Terme von C bezeichnet.*

Lemma A.3.2. *Seien C_1, C_2 und $D \in K[y, \dots, z]$ und $C_1 = C_2 \cdot D$. Weiters sei* $\mathrm{tran}(D) = (E_l, E_h)$ *und* $\mathrm{ran}_u(D) = (\textit{einige}, e_u)$ *für $u = y, \dots, z$. Dann braucht man, wenn man D durch die Potenzreihendivision von C_1 und C_2 berechnet, nur $(E_h - E_l, e_y - E_l, \dots, E_z - E_l)$ signifikante Terme von C_1 und C_2.*

Von hier an werden der maximale Termgrad der Koeffizienten mit $\mathrm{tdeg}(\mathrm{coef}(P))$ bezeichnet und der maximale Grad bzgl. u als $\deg_u(\mathrm{coef}(P))$.

Lemma A.3.3. *Für P_1 und $P_2 \in K[y, \dots, z][x]$ setzt man $G = \mathrm{ggT}(P_1, P_2)$. Es seien*

$$\begin{aligned} E &= \mathrm{tdeg}(\mathrm{coef}(G)), \\ E'' &= \mathrm{tdeg}(\mathrm{lc}(G)), \\ E_i &= \mathrm{tdeg}(\mathrm{coef}(P_i)), \\ E_i' &= \mathrm{tdeg}(\mathrm{lc}(P_i)) \quad \text{für } i = 1, 2. \end{aligned}$$

Weiters sei für alle Variablen $u \in \{y, \ldots, z\}$

$$\begin{aligned} e_u &= \deg_u(\mathrm{coef}(G)), \\ e''_u &= \deg_u(\mathrm{lc}(G)), \\ e_{ui} &= \deg_u(\mathrm{coef}(P_i)), \\ e'_{ui} &= \deg_u(\mathrm{lc}(P_i)) \quad \text{für } i = 1, 2. \end{aligned}$$

Dann gilt:

$$\begin{aligned} E &\leq \min\{E_i - E'_i + E'' | i = 1, 2\} \\ e_u &\leq \min\{e_{ui} - e'_{ui} + e''_u | i = 1, 2\} \end{aligned}$$

Lemma A.3.4. *Für* P_1 *und* $P_2 \in K[y, \ldots, z][x]$ *seien* $G = \mathrm{ggT}(P_1, P_2)$, $g = \mathrm{ggT}(\mathrm{lc}(P_1), \mathrm{lc}(P_2))$, $\gamma = \frac{g}{\mathrm{lc}(G)}$ *und* $\tilde{P} = \gamma G$. *Seien*

$$\begin{aligned} E &= \mathrm{tdeg}(\mathrm{coef}(\tilde{P})), \\ E'' &= \mathrm{tdeg}(g), \\ E_i &= \mathrm{tdeg}(\mathrm{coef}(P_i)), \\ E'_i &= \mathrm{tdeg}(\mathrm{lc}(P_i)) \quad \text{für } i = 1, 2. \end{aligned}$$

Weiters sei für alle Variablen $u \in \{y, \ldots, z\}$

$$\begin{aligned} e_u &= \deg_u(\mathrm{coef}(\tilde{P})), \\ e''_u &= \deg_u(g), \\ e_{ui} &= \deg_u(\mathrm{coef}(P_i)), \\ e'_{ui} &= \deg_u(\mathrm{lc}(P_i)) \quad \text{für } i = 1, 2. \end{aligned}$$

Dann gilt:

$$\begin{aligned} E &\leq \min\{E_i - E'_i + E'' | i = 1, 2\} \\ e_u &\leq \min\{e_{ui} - e'_{ui} + e''_u | i = 1, 2\} \end{aligned}$$

LITERATURVERZEICHNIS

[Bee84] Michael J. Beeson. Mathpert: Computer support for learning algebra, trig, and calculus. In A. Voronkov, editor, *Logic programming and automated reasoning: international conference LPAR '92*, pages 454–456. Springer-Verlag, 1984. Volume 624 of Lecture Notes in Computer Science.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Bos99] Siegfried Bosch. *Algebra.* Ausgabe 3., überarb. und erw. Aufl.. Springer, 1999.

[Bro71] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the ACM (JACM)*, 18(4):478–504, 1971.

[BT71] W. S. Brown and J. F. Traub. On euclid's algorithm and the theory of subresultants. *Journal of the ACM (JACM)*, 18(4):505–514, 1971.

[Buc82] Bruno Buchberger. Algorithmic mathematics: Problem types, data types, algorithm types. Lecture notes, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, 1982. 320 pages.

[Far01] William M. Farmer. A formal framework for managing mathematics. RISC, A-4232 Schloss Hagenberg, September 24-26 2001.

[Fis95] Gerd Fischer. *Lineare Algebra.* 10. Auflage. Vieweg, 1995.

[Fri96] Sophie Frisch. *Algebra.* ÖH-Skriptum zur Vorlesung, TU-Graz, Austria, 1996.

[GGK+02] Matthias Goldgruber, Andreas Griesmayer, Stefan Karnel, Alan Krempler, and Walther Neuper. Das $\mathcal{ISAC}$-Projekt. Status und Ausblick. Technical report, IICM, Institute f.Softwaretechnology, University of Technology, Graz, Austria, September 2002. im Druck.

[GL02] Matthias Goldgruber and Richard Lang. Eine explizite Hierarchie von Typen elementarer Gleichungen. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*, 2002. im Druck. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/visitme02-M023.pdf.

[Gri02] Andreas Griesmayer. Tools für die interaktive Spezifikation von Problemen. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*, 2002. im Druck. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/visitme02-p03.pdf.

[HK98] Helmut Heugl and Walter Klinger. Forschungsprojekt 'Der Mathemematikunterricht im Zeitalter der Informationstechnologie'. Technical report, ACDCA, Austrian Center for the Didactics of Computer Algebra, 1998.

[Hun96] Thomas W. Hungerford. *Algebra.* Springer, 1996.

[Kar02] Stefan Karnel. Computer Algebra für Brüche – angepasst an Ausbildungszwecke. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*, 2002. im Druck. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/visitme02-M01-paper.pdf.

[Knu81] D.E. Knuth. *The art of computerprogramming, vol. 2, seminumerical algorithms, 2nd edn.* Addison-Wesley, 1981.

[Kre02] Alan Krempler. Zum Design eines elektronischen Arbeitsblattes für Mathematik. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*, 2002. im Druck. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/visitme02-p01-paper.pdf.

[KW00] Walter Klinger and Walter Wegscheider. Elektronische Lernmedien im Mathematikunterricht (Einfluss auf das Lehren und Lernen, den Lehrplan und die Leistungsbeurteilung). Technical report, ACDCA, Austrian Center for the Didactics of Computer Algebra, 2000.

[Mig74] M. Mignotte. An inequality about factors of polynomials. *Math. Comput. 28*, pages 1153–1157, 1974.

[Mig83] M. Mignotte. Some useful bounds. *In: Buchberger, B., Collins, G.E., Loos, R. (eds.): Computer algebra, symbolic and algebraic computation 2nd edn.*, pages 259–263, 1983.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, London, 1997.

[Neu01a] Walther A. Neuper. A 'calculemus-approach' to high-school math ? Siena, Italy, June 21-22 2001. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/calculemus01.ps.gz.

[Neu01b] Walther A. Neuper. Künftige Softwaretools für den Mathematik-Unterricht. Arbeitspapier für die AMMU, Arbeitsgemeinschaft moderner Mathematik-Unterricht ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/ammu1.doc, 2001.

[Neu01c] Walther A. Neuper. *Reactive User-Guidance by an Autonomous Engine Doing High-School Math.* PhD thesis, IICM - Inst. f. Softwaretechnology, Technical University, A-8010 Graz, 2001. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/wn-diss.ps.gz.

[Neu02] Walther Neuper. Re-engineering von Algebra-Systemen zum Mathematik-Lernen. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*, 2002. im Druck. ftp://ftp.ist.tugraz.at/pub/projects/isac/publ/visitme02-intro.pdf.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *JAR*, 5(3):363–397, 1989.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Topias Nipkow.

[Pau97] Lawrence C. Paulson. *Isabelle's object-logics.* University of Cambridge, Computer Laboratory, July 1997.

[Pet99] Attila Pethő. *Algebraische Algorithmen.* Vieweg, 1999.

[Sch93] Hans Rudolf Schwarz. *Numerische Mathematik.* 3. überarb. und erw. Aufl. Teubner, 1993.

[Sch02] Klaus Schmaranz. *Dinopolis - a massively distributed middleware.* habilitation thesis, IICM, University of Technology, Graz, Austria, 2002.

[SS92] Tateaki Sasaki and Masayuki Suzuki. Three new algorithms for multivariate polynomial GCD. *J. Symbolic Comput.*, 13(4):395–411, 1992.

[Win96] Franz Winkler. *Polynomial algorithms in computer algebra.* Springer, 1996.