Summationsalgorithmen und ihre Implementierung in MuPAD

Masterarbeit von Stefan Scheel
Betreuer: Prof. Dr. W. Koepf

Universität Kassel

Februar 2008

Inhaltsverzeichnis

Einleitung

Die Arbeit beschäftigt sich mit einem Themenbereich aus der Computeralgebra. Die algorithmische Betrachtung von mathematischen Fragestellungen gab es schon immer. Aber die Entwicklung der Computer und ihre Geschwindigkeit in Verbindung mit der Größe ihres Speicherplatzes, haben der Mathematik ganz neue Türen geöffnet. Es wurden Computeralgebrasysteme[1] entwickelt, die zu immer wichtigeren Werkzeugen für Mathematiker und Naturwissenschaftler wurden. Diese Computerprogramme lösen Rechenaufgaben aus verschiedenen Bereichen der Mathematik und dies nicht nur wie ein Taschenrechner mit Zahlen, sondern auch mit symbolischen Ausdrücken (Variablen, Funktionen, Matrizen). Durch diese und die zunehmende Geschwindigkeit der Computer war plötzlich ein algorithmisches Lösen von komplizierten Fragestellungen möglich geworden.

In meiner Arbeit befasse ich mich mit dem Thema der algorithmischen Summation. Speziell stelle ich die Summation von hypergeometrischen Termen vor.

Auch im Bereich der algorithmischen Summation sorgte die Geschwindigkeit der Computer zur Weiterentwicklung. Der erste Algorithmus, der ohne ein Computeralgebrasystem wahrscheinlich nicht gefunden worden wäre, war der Algorithmus zur unbestimmten Summation ([Gos78]). Diese wurde 1978 von R. W. Gosper Jr. während der Entwicklung eines der ersten symbolischen Algebraprogramme Macsyma entdeckt.

Die ersten Schritte zur algorithmischen Summation gehen zurück bis zu Schwester Celine Fasenmyer, die 1945 in ihrer Dissertation schrieb, wie man eine Rekursionsgleichung für eine Summe hypergeometrischer Terme findet. Ihre Methode ist nicht sehr effektiv, aber sie führte später zur Automatisierung durch einen Computer. Sie wurde von Wilf und Zeilberger aufgearbeitet und implementiert und ist in [PWZ97] nachzulesen. Zeilberger selbst entwarf einen Algorithmus, der dieses Problem viel effizienter löst ([Zei91]). Dazu nutzte er eine angepasste Form des Gosper-Algorithmus von 1978 ([Zei91]).

In der Einleitung zu dem Buch [PWZ97] schrieb Donald Knuth: „Science is what we understand well enough to explain to a computer. Art is everything else we do. During the past several years an important part of mathematics has been transformed from an Art to a Science.“ Dieses Zitat charakterisiert die Entwicklung der algorithmischen Mathematik mit Hilfe von Computeralgebrasystemen. Für die algorithmische Summation von hypergeometrischen Termen heißt das, dass man nicht mehr brillante Erkenntnisse braucht, um eine Sum-

[1] Zum Beispiel: Mathematica, Maple, MuPAD und viele mehr.

me mit Binomialkoeffizienten oder ähnlichen Formeln auszuwerten, sondern wir können mechanischen Prozeduren folgen, die uns dann systematisch die Antwort erzeugen.

In dieser Arbeit beschäftige ich mich mit der Frage, ob eine gegebene Summe in eine „einfache Form“ zu bringen ist. Eine „einfache Form“ ist beispielsweise eine Funktion in der Variablen n, in der das Summationszeichen und die Laufvariable k nicht mehr vorkommen. Unser Ziel ist es eine „geschlossene Form“ für eine Summe zu finden, das heißt, wir suchen eine Linearkombination von hypergeometrischen Termen.

Die Summen, die wir betrachten, haben die Form

$$s_n = \sum_{k=m}^{n} a_k,$$

wobei k der Summationsindex ist und a_k ein hypergeometrischer Term bezüglich k. Ein Term a_k heißt hypergeometrisch, falls der Quotient

$$\frac{a_{k+1}}{a_k} \in \mathbb{Q}(k)$$

eine rationale Funktion ist, das heißt, der Quotient ist rational bezüglich k. Die Beschränkung auf hypergeometrische Terme hat den Vorteil, dass wir nur noch mit Polynomen rechnen müssen. Eine Linearkombination von hypergeometrischen Termen bezeichen wir als „geschlossene Form“.

In der Arbeit sprechen wir von unbestimmter und bestimmter Summation. Bei der bestimmten Summation summieren wir über ein festes Intervall aus $\mathbb{Z}$. Dieses Intervall kann durchaus auch $]-\infty,\ \infty[$ sein, das heißt die Summe läuft über alle $k \in \mathbb{Z}$.

Zuerst betrachten wir jedoch die unbestimmte Summation und stellen den Gosper-Algorithmus vor. Dieser findet eine diskrete Stammfunktion s_k mit

$$a_k = s_{k+1} - s_k,$$

wobei der gegebene Term a_k hypergeometrisch ist. Wir werden sehen, dass die Stammfunktion ein rationales Vielfaches von a_k ist. Ein Term a_k, für den der Algorithmus eine diskrete Stammfunktion findet, heißt gospersummierbar. Der Algorithmus ist abgeschlossen bezüglich der Multiplikation, aber nicht bezüglich der Addition, daher betrachten wir danach eine linearisierte Form des Gosper-Algorithmus. Wir können eine gegebene Linearkombination von hypergeometrischen Termen in Äquivalenzklassen einteilen. Dadurch erhalten wir eine Linearkombination von paarweise verschiedenen ähnlichen Termen, auf die wir dann den Gosper-Algorithmus anwenden können. Als Resultat erhalten wir dann eine Linearkombination aus vereinfachten und nicht vereinfachten Stammfunktionen. Falls die Linearkombination nur aus vereinfachten hypergeometrischen Stammfunktionen besteht, haben wir eine „geschlossene Form“ gefunden.

Als nächstes betrachten wir die bestimmte Summation und versuchen eine „geschlossene Form“ für die Summe

$$s_n = \sum_{k \in \mathbb{Z}} F(n,k)$$

zu finden, wobei das gegebene $F(n,k)$ hypergeometrisch bezüglich k und n ist. Eine „geschlossene Form“ kann nur (ohne Grenzwertbetrachtung) gefunden werden, wenn für jedes $n \in \mathbb{Z}$ (bzw. $n \in \mathbb{N}_0$) nur endlich viele k mit $F(n,k) \neq 0$ existieren, das heißt, der Summand hat einen endlichen Träger. Der Algorithmus von Doron Zeilberger ([Zei91]) nutzt eine angepasste Form des Gosper-Algorithmus, um eine Rekursion für die Summe s_n zu finden.

Da unser Ziel eine „geschlossene Form“ für eine gegebene Summe ist, müssen wir eine hypergeometrische Lösung für die Rekursionsgleichung finden, die wir durch den Zeilberger-Algorithmus erhalten haben. Aus einer Rekursion erster Ordnung erhalten wir den hypergeometrischen Quotienten s_{n+1}/s_n, aus diesem können wir dann die hypergeometrische Lösung s_n bestimmen. Ist die Ordnung aber höher, brauchen wir einen weiteren Algorithmus: den Petkovšek-Algorithmus. Dieser findet für eine homogene Rekursionsgleichung alle rationalen hypergeometrischen Lösungen, die existieren. Der Algorithmus gibt uns Lösungen der Form s_{n+1}/s_n zurück, diese können wir dann leicht in hypergeometrische Terme s_n umformen. Die Lösung ist dann eine Linearkombination von hypergeometrischen Termen.

Danach beschäftigen wir uns mit der Summation mit festen Summationsgrenzen für k. Falls ein Term a_k gospersummierbar ist, ist der Fall sehr einfach, denn wir erhalten durch den Gosper-Algorithmus eine diskrete Stammfunktion s_k. Indem wir die Grenzen für k in $s_{k+1} - s_k$ substituieren, erhalten wir eine „geschlossene Form“ für die bestimmte Summe.

Beim Zeilberger-Algorithmus ist die Summation über einem festen Summationsintervall nicht so trivial. Der Algorithmus liefert uns eine Darstellung

$$F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) = G(n,k+1) - G(n,k),$$

wobei $\sigma_j(n)$ $(j = 1, \ldots, J)$ rationale Funktionen sind, die abhängig von n, aber unabhänig von k sind. Auf der linken Seite haben wir eine Rekursion mit rationalen Koeffizienten in n für die Summe s_n erhalten. Das $G(n,k)$ auf der rechten Seite ist die diskrete Stammfunktion der linken Seite.

Da $F(n,k)$ einen endlichen Träger hat, gibt es natürliche Summationsgrenzen für die Summe s_n. Summieren wir über ein Intervall, das außerhalb dieser natürlichen Summationsgrenzen liegt, ist die rechte Seite auf Grund einer Teleskopsumme gleich Null. Wenn wir über ein Intervall summieren, das innerhalb der natürlichen Summationsgrenzen liegt, erhalten wir eine inhomogene Rekursion für die Summe s_n.

Da der Petkovšek-Algorithmus nur homogene Rekursionsgleichungen lösen kann, müssen wir die inhomogene Rekursionsgleichung in eine homogene umwandeln. Mit Hilfe des Petkovšek-Algorithmus können wir dann eine „geschlossene Form“ für s_n finden, falls sie existiert.

Zum Abschluss fassen wir alle Algorithmen zusammen. Dadurch erhalten wir einen Algorithmus, der für eine Summe, deren Summand aus einer Linearkombination von hypergeometrischen Termen besteht, eine geschlossene Form findet, falls sie existiert.

Die Algorithmen sind in der Arbeit als Pseudocode dargestellt. Einige Beispiele in der Arbeit werden mit Hilfe von implementierten Befehlen in MuPAD gelöst. Diese Prozeduren sind auf der beiliegenden CD in dem Package `summation.mu` zu finden. Die von mir vorgestellten und implementierten Algorithmen sind zum größten Teil noch nicht in MuPAD implementiert und bieten daher eine Verbesserung des in MuPAD verwendeten Summationsbefehls `sum`.

Danksagung

Ich danke Prof. Dr. Wolfram Koepf für die gute und freundliche Betreuung während der Implementierung und Erstellung dieser Arbeit. Auch möchte ich mich für die Unterstützung und Hilfestellung bei MuPAD-Gruppe von Sciface und bei Peter Horn, der mich bei der Implementierung betreut hat, bedanken.

Kapitel 1

Theoretische Grundlagen

1.1 Einführung

Wir werden zuerst einige Grundlagen vorstellen, die wir im Laufe der Arbeit verwenden werden. Aber zunächst betrachten wir das Problem der Integration als Motivation für das folgende Kapitel.

Mit der Stammfunktion F einer Funktion $f(x)$ und dem Hauptsatz der Differential- und Integralrechnung können wir jedes bestimmte Integral mit der Rechenregel

$$\int_a^b f(x)dx = F(b) - F(a)$$

lösen. Mit großen Datenbanken, in denen die Stammfunktionen vermerkt sind, wird das Problem der bestimmten Integration lösbar. Ein algorithmisches Verfahren ist allerdings sehr wünschenswert, da keine Datenbank vollständig ist und oft viele komplizierte Umformungen nötig sind, um die Einträge in der Datenbank zu finden. Für die Integration gibt es solche algorithmischen Verfahren, auf die wir hier aber nicht näher eingehen wollen. Es kann aber in [Ris70] nachgelesen werden.

Bei der Summation stellt sich ein ähnliches Problem. Wir werden im weiteren Verlauf der Arbeit sehen, wie wir eine diskrete Stammfunktion s_k für den gegebenen Summanden a_k finden können. Es ergibt sich dann die Teleskopsumme

$$\sum_{k=a}^{b} a_k = s_{b+1} - s_b + \ldots + s_{a+1} - s_a = s_{b+1} - s_a. \tag{1.1}$$

Wenn wir eine diskrete Stammfunktion s_k für a_k gefunden haben, ist die bestimmte Summation also trivial.

Im nächsten Kapitel werden wir den Gosper-Algorithmus kennen lernen und werden sehen, dass wir mit diesem in der Lage sind für eine große Klasse von Eingabetermen, nämlich für hypergeometrische Terme, eine hypergeometrische Stammfunktion zu finden, falls eine solche existiert.

Zunächst wollen wir aber definieren in welchen Körpern wir uns bewegen. Die vorgestellten Algorithmen dieser Arbeit benutzen Faktorisierungsalgorithmen. Diese können nur über $\mathbb{Q}$ garantiert werden. Daher setzen wir für die ganze

Arbeit stillschweigend voraus, dass wir alles über dem Körper $K = \mathbb{Q}$ oder $\mathbb{Q}$ adjungiert um eine feste Anzahl von Konstanten (beispielsweise: $\mathbb{Q}(\alpha_1, \alpha_2, \alpha_3)$) betrachten. Es sei $K(k)$ der Körper der rationalen Funktionen bezüglich der Variablen k über dem Körper K und $K[k]$ sei der Ring der Polynome in k über K.

Definition 1.1.1 *Ein Term a_k heißt hypergeometrisch über K, falls Polynome p_k und q_k aus $K[k]$ existieren, so dass a_k eine lineare homogene Rekursion erster Ordnung*

$$p_k a_{k+1} - q_k a_k = 0,$$

mit Polynomkoeffizienten in $K[k]$ für alle $k \in \mathbb{N}$ erfüllt. Weiter heißt das, dass a_{k+1}/a_k in $K(k)$ liegt.

Die Klasse der hypergeometrischen Terme umfasst trivialer Weise sowohl Polynome als auch rationale Funktionen in k. Aber zusätzlich gehören auch Produkte und Summen von Binomialkoeffizienten dazu, da

$$\frac{\binom{n+1}{k}}{\binom{n}{k}} = -\frac{k-n}{k+1} \in K(k) \tag{1.2}$$

gilt. Fakultäten gehören dazu, weil

$$\frac{(k+1)!}{k!} = k+1 \in K(k) \tag{1.3}$$

ist. Es gibt noch eine weitere Klasse, die den hypergeometrischen Termen angehört, und zwar die Klasse der Pochhammersymbole.

Definition 1.1.2 *Mit Pochhammersymbol bezeichnen wir*

$$(a)_k = \begin{cases} 1 & \text{falls } k = 0 \\ a \cdot (a+1) \cdots (a+k-1) & \text{falls } k \in \mathbb{N}. \end{cases} \tag{1.4}$$

Es ist zu erwähnen, dass es eine erweiterte Definition von Pochhammersymbolen für alle $k \in \mathbb{Z}$ (als steigende oder fallende Faktorielle) gibt. Aber wir wollen uns in der Arbeit nur auf den Fall $k \in \mathbb{N}$ beschränken. In Kapitel 5 werden wir sehen, dass die Pochhammersymbole eine wichtige Rolle bei der Ausgabe von hypergeometrischen Lösungen spielen.

Die Pochhammmersymbole gehören der Klasse der hypergeometrischen Terme an, weil

$$\frac{(a)_{k+1}}{(a)_k} = k + a \in K(k) \tag{1.5}$$

gilt.

Damit wir hypergeometrische Terme erkennen können, nehmen wir eine Vereinheitlichung vor. Dafür führen wir die Eulersche Gamma-Funktion ein.

Definition 1.1.3 *Die Gamma-Funktion ist durch*

$$\Gamma(z) = \int_0^\infty e^{-t} t^{z-1} dt.$$

definiert. Das Integral existiert für $z \in \mathbb{C}$ mit Re $z > 0$.

Mit der Gamma-Funktion lassen sich Fakultäten, Binomialkoeffizienten und Pochhammersymbole darstellen. Die Darstellungen lassen sich aus der folgenden Rekursion

$$\Gamma(z+1) = \int_0^\infty e^{-t} t^z dt - t^z e^{-t} \Big|_{t=0}^{t=\infty} = z\Gamma(z) \tag{1.6}$$

herleiten. Wir werden nun den `simpcomb` Algorithmus von W. Koepf aus seinem Buch [Koe97] vorstellen. Dieser Algorithmus ist in der Lage zu entscheiden, ob ein Term a_k hypergeometrisch ist, mit anderen Worten, ob a_{k+1}/a_k eine rationale Funktion in k ist. Der Algorithmus drückt die Funktion in Form von Gamma-Funktionen aus.

Algorithmus 1.1.4 (`simpcomb`) *Der Algorithmus entscheidet, ob ein Term a_{k+1}/a_k rational ist.*

1. *Input: a_{k+1}/a_k, wobei a_k ein Produkt aus Quotienten von rationalen Funktionen, Potenzen, Fakultäten, Gamma-Funktionen, Binomialkoeffizienten und Pochhammersymbolen, welche rational linear in ihren Argumenten sind, ist.*

2. *(`togamma`) Bilde a_{k+1}/a_k. Schreibe Fakultäten, Binomialkoeffizienten und Pochhammersymbole nach folgenden Regeln um:*

$$k! = \Gamma(k+1), \tag{1.7}$$

$$\binom{n}{k} = \frac{z\cdot(z-1)\cdots(z-k+1)}{k!} = \frac{\Gamma(z+1)}{k!\Gamma(z-k+1)} \quad \text{und} \tag{1.8}$$

$$(z)_k = z\cdot(z+1)\cdots(z+k-1) = \frac{\Gamma(z+k)}{\Gamma(z)}. \tag{1.9}$$

   Vermeide negative Argumente. Im Fall der Binomialkoeffizienten kann dies mit folgender Regel getan werden:

$$\binom{n}{k} \to \begin{cases} (-1)^k \frac{\Gamma(k-a)}{\Gamma(k+1)\Gamma(-n)} & \text{falls } n \in \mathbb{Z},\ n < 0 \\ 0 & \text{falls } n-k \in \mathbb{Z},\ n-k < 0 \\ \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} & \text{sonst.} \end{cases}$$

3. *(`simplifygamma`) Schreibe die erhaltenen Ausdrücke mit folgender Regel um:*

$$\Gamma(n+j) = n\cdot(n+1)\cdots(n+j-1)\cdot\Gamma(n).$$

   Wenn immer die Argumente n und $n+j$ von zwei Gamma-Funktionen eine ganzzahlige positive Differenz j haben, reduziere den resultierenden Bruch durch Kürzen der gemeinsamen Gammaterme.

4. *(*`simplifypower`*) Schreibe den resultierenden Ausdruck mit Hilfe der Regel*

$$b^{n+j} = b^j b^n$$

um. Wenn immer die Argumente n und $n+j$ von zwei Potenzen eine ganzzahlige positive Differenz j haben, reduziere den resultierenden Bruch durch Kürzen der Potenz mit dem Argument j.

5. *Der Ausdruck a_{k+1}/a_k ist genau dann rational, wenn der im letzten Schritt resultierende Bruch u_k/v_k rational ist, das heißt, $u_k,\ v_k \in K[k]$.*

6. *Output: $(u_k,\ v_k)$.*

Wir werden im weiteren Verlauf nicht mehr genauer auf die Gamma-Funktion oder den `simpcomb`-Algorithmus eingehen. Aber in allen von mir implementierten Befehlen in MuPAD wird dieser Algorithmus verwendet, um zu entscheiden, ob ein Term hypergeometrisch ist oder nicht. Dieser Algorithmus löst unsere spezielle Fragestellung viel effektiver als alle eingebauten Funktionen in MuPAD.

In der gesamten Arbeit ist es unser Ziel die Eingabeterme in eine möglichst „einfache Form“ zu bringen. Dazu definieren wir den Terminus „geschlossene Form“:

Definition 1.1.5 *Eine Funktion s_n liegt in geschlossener Form vor, wenn sie sich als Linearkombination mit einer festen Anzahl ($r \in \mathbb{N}$) von hypergeometrischen Termen darstellen lässt.*

Falls wir einen hypergeometrischen Term $s_{n+1}/s_n \in K(n)$ als Quotient einer hypergeometrischen Lösung erhalten, können wir diesen in einen hypergeometrischen Term s_n umschreiben. Dazu geben wir folgenden Algorithmus an:

Algorithmus 1.1.6 *Der Algorithmus wandelt den hypergeometrischen Quotienten $s_{n+1}/s_n \in K(n)$ in einen hypergeometrischen Term s_n um.*

1. *Eingabe: Ein hypergeometrischer Quotient s_{n+1}/s_n.*

2. *Berechne mit Hilfe von Algorithmus 1.1.4 die Polynome $u_n,\ v_n \in K[n]$, so dass*

$$\frac{s_{n+1}}{s_n} = \frac{u_n}{v_n} \quad \text{mit } \mathrm{ggT}(u_n, v_n) = 1$$

gilt. Falls Algorithmus 1.1.4 ausgibt, dass s_n nicht rational ist, brich ab und gib aus, dass kein hypergeometrischer Term s_n existiert.

3. *Führe für u_n und v_n eine rationale Faktorisierung durch. Falls keine linearen Faktoren entstehen, brich ab und gib aus, dass keine rationale Faktorisierung existiert. Falls die Faktorisierung erfolgreich war, haben u_n und v_n in die Form:*

$$u_n = A(n+\alpha_1)(n+\alpha_2)\cdots(n+\alpha_p) \text{ und } v_n = B(n+\beta_1)(n+\beta_2)\cdots(n+\beta_{q+1}).$$

4. *Berechne für alle* β_i $(i = 1, \ldots, q+1)$*, die eine ganze Zahl sind, das kleinste* $\beta_i = m$*. Falls* $m < 1$*, shifte alle* α_j $(j = 1, \ldots, p)$ *und* β_i *um* $-m+1$*. Dadurch erhalten wir die umgeschriebenen* α_j *und* β_i*.*

5. *Schreibe* α_j *und* β_j *in die Form:*

$$s_n = \frac{u_n}{v_n} = \frac{A^n}{B^n} \frac{(\alpha_1)_n (\alpha_2)_n \cdots (\alpha_p)_n}{(\beta_1)_n (\beta_2)_n \cdots (\beta_{q+1})_n}.$$

6. *Ausgabe: Der hypergeometrische Term* s_n*.*

Nachdem wir nun einige wichtige theoretische Grundlagen definiert haben widmen wir uns im nächsten Kapitel nun der algorithmischen Summation von hypergeometrischen Termen.

Kapitel 2

Gosper-Algorithmus

2.1 Einführung

Wir wollen uns zunächst mit der Frage beschäftigen, in welchen Fällen eine unbestimmte Summe ein hypergeometrischer Term ist. Dazu werden wir im Verlaufe des Abschnitts einen Algorithmus kennen lernen.

Im letzten Kapitel haben wir als Motivation das Problem der Integration angesehen, nun wollen wir unser Summationsproblem betrachten. Der Gosper-Algorithmus ist ein Verfahren, das für einen (unbestimmt zu summierenden) Term a_k eine diskrete Stammfunktion s_k der Form

$$s_{k+1} - s_k = a_k \tag{2.1}$$

findet, so dass s_k ein hypergeometrischer Term ist, das heißt, dass s_k der Eigenschaft

$$\frac{s_{k+1}}{s_k} \in K(k)$$

genügt. Da der Algorithmus einen hypergeometrischen Term findet, liegt das Resultat in einer geschlossenen Form vor. Ähnlich wie bei der Integration verhält es sich auch bei der Summation: Haben wir eine diskrete Stammfunktion s_k von a_k gefunden, so ist die bestimmte Summation trivial, denn wir erhalten eine Teleskopsumme

$$\sum_{k=n}^{m} a_k = (s_{n+1} - s_n) + (s_n - s_{n-1}) + \ldots + (s_{m+1} - s_m) = s_{n+1} - s_m,$$

die sich an den Grenzen auswerten lässt. Auf diesen Fall werden wir genauer in Kapitel 5 eingehen. Für den Moment und in diesem Kapitel betrachten wir immer die unbestimmte Summation mit dem Summationsindex k, wobei wir nur über ganze Zahlen summieren.

Aus der Forderung, dass s_k ein hypergeometrischer Term ist, folgt sofort, dass auch a_k ein hypergeometrischer Term sein muss. Dieses ergibt sich aus der Beobachtung:

$$\frac{a_{k+1}}{a_k} = \frac{s_{k+2} - s_{k+1}}{s_{k+1} - s_k} = \frac{s_{k+1}}{s_k} \frac{\frac{s_{k+2}}{s_{k+1}} - 1}{\frac{s_{k+1}}{s_k} - 1} = \frac{u_k}{v_k} \in K(k), \tag{2.2}$$

wobei u_k und v_k Polynome aus $K[k]$ mit $\mathrm{ggT}(u_k, v_k) = 1$ sind. Daher können wir uns in diesem Kapitel auf die unbestimmte Summation von hypergeometrischen Termen beschränken.

Definition 2.1.1 *Ein Term a_k heißt gospersummierbar, wenn er eine hypergeometrische diskrete Stammfunktion s_k hat.*

Zu beachten ist, dass s_k ein rationales Vielfaches von a_k ist, auf dies werden wir im weiteren Verlauf noch genauer eingehen. Weiter ist zu beachten, dass nicht für alle hypergeometrischen Terme eine diskrete hypergeometrische Stammfunktion existiert.

2.2 Unbestimmte Summation

Zunächst wollen wir uns einen Überblick über den Algorithmus von Gosper verschaffen, um ihn später im Detail zu erläutern und zu beweisen. Dieser Teil der Arbeit bezieht sich auf Kapitel 5 aus [Koe97].

Die Eingabe des Gosper-Algorithmus ist ein hypergeometrischer Term a_k, wobei a_{k+1}/a_k, wie oben gesehen, durch die beiden teilerfremden Polynome u_k und v_k repräsentiert wird. Diese Idee wird durch die Betrachtung des folgenden Beispiels deutlicher:

Beispiel 2.2.1
Wir haben $s_k = (2k)!/k!$ gegeben. Wie wir dieses Resultat erhalten haben, spielt zu diesem Zeitpunkt keine Rolle. Aus diesem s_k wollen wir nun a_k berechnen. Es ergibt sich dann

$$a_k = s_{k+1} - s_k = \frac{(2k+2)!}{(k+1)!} - \frac{(2k)!}{k!} = (4k+1) \cdot \frac{(2k)!}{k!},$$

ein Produkt aus einem polynomialen Teil $p_k = (4k+1)$ und einem faktoriellen Term $b_k = (2k)!/k!$. Da $b_{k+1}/b_k = q_{k+1}/r_{k+1}$ rational ist, können wir annehmen, dass q_k und r_k Polynome sind.

In dem Beispiel haben wir gesehen, wie ein solches a_k im Speziellen aussehen könnte. Als nächstes geben wir eine grobe allgemeine Zusammenfassung des Gosper-Algorithmus. Für die im Beispiel auftretenden q_k und r_k gilt die Eigenschaft

$$\mathrm{ggT}(q_k, r_{k+j}) = 1 \qquad \text{für alle } j \in \mathbb{N}_0. \tag{2.3}$$

Genügen q_k und r_k nicht dieser Eigenschaft, können sie durch ein algorithmisches Verfahren so umgeschrieben werden, dass sie die Eigenschaft (2.3) erfüllen. Dieses werden wir in Lemma 2.2.2 kennen lernen. Später werden wir sehen, welche Bedeutung der größte gemeinsame Teiler $\mathrm{ggT}(q_k, r_k)$ genau hat.

Für a_k erhalten wir folgende Relation

$$\frac{a_{k+1}}{a_k} = \frac{p_{k+1}}{p_k} \frac{q_{k+1}}{r_{k+1}}, \tag{2.4}$$

wobei p_k dem polynomialen und q_{k+1}/r_{k+1} dem faktoriellen Teil entspricht.

Als letztes definiert Gosper eine Funktion f_k, wie folgt:

$$f_k = \frac{s_{k+1}}{a_{k+1}} \frac{p_{k+1}}{r_{k+1}} \quad \text{oder} \quad s_k = \frac{r_k}{p_k} f_{k-1} a_k. \tag{2.5}$$

Mit $s_{k+1} - s_k = a_k$ sehen wir sehr schnell, dass diese Funktion rational ist:

$$f_k = \frac{s_{k+1}}{a_{k+1}} \frac{p_{k+1}}{r_{k+1}} = \frac{s_{k+1}}{s_{k+2} - s_{k+1}} \frac{p_{k+1}}{r_{k+1}} = \frac{1}{\frac{s_{k+2}}{s_{k+1}} - 1} \frac{p_{k+1}}{r_{k+1}}. \tag{2.6}$$

Gosper zeigt des Weiteren mit Hilfe von (2.3), dass f_k ein Polynom ist. Durch Einsetzen und Umformen ergibt sich:

$$a_k = s_{k+1} - s_k = \frac{r_{k+1}}{p_{k+1}} f_k a_{k+1} - \frac{r_k}{p_k} f_{k-1} a_k.$$

Wenn wir nun mit p_k/a_k multiplizieren und (2.4) verwenden, erhalten wir eine lineare inhomogene Rekursiongleichung für f_k:

$$p_k = \frac{a_{k+1}}{a_k} \frac{p_k}{p_{k+1}} r_{k+1} f_k - r_k f_{k-1} = q_{k+1} f_k - r_k f_{k-1}. \tag{2.7}$$

Gosper entwickelte einen Algorithmus, welcher den möglichen Grad der Polynomlösung f_k der Rekursion (2.7) a priori bestimmt, dieser Algorithmus wird in Satz 2.2.6 näher erläutert. Das führt dazu, dass wir f_k als ein allgemeines Polynom mit einem festen Grad gegeben haben. Von diesem können wir dann die Koeffizienten durch Lösen eines linearen Gleichungssystems finden und damit zum Abschluss auch eine Lösung für s_k aus (2.5) angeben.

Falls der Algorithmus kein Polynom f_k findet, ist bewiesen, dass keine hypergeometrische Stammfunktion s_k von a_k existiert.

Nach diesem Überblick werden wir im nächsten Abschnitt den Gosper-Algorithmus genauer betrachten.

2.2.1 Der Gosper-Algorithmus

Da wir jetzt einen allgemeinen Überblick des Algorithmus erhalten haben, werden wir zunächst alle Teile genauer analysieren und beweisen, bevor wir den Algorithmus zusammenfassen.

Lemma 2.2.2 *Gegeben sei $a_{k+1}/a_k \in K(k)$. Dann können p_k, q_k und r_k mit (2.4) so gewählt werden, dass*

$$\mathrm{ggT}(q_k, r_{k+j}) = 1 \qquad \textit{für alle } j \in \mathbb{N}_0 \tag{2.8}$$

gilt.

Der folgende Beweis liefert gleichzeitig einen Algorithmus, der p_k, q_k und r_k so bestimmt, dass sie Gleichung (2.8) genügen.

Beweis: Zunächst wählen wir $p_k = 1$, $q_k = u_{k-1}$ und $r_k = v_{k-1}$. Die Indexverschiebung von k ist notwendig, da wir in unserer Ausgangsgleichung (2.4) q_{k+1}

und r_{k+1} betrachten. Jetzt ist entweder die Eigenschaft (2.8) erfüllt und der Beweis ist erbracht oder Eigenschaft (2.8) ist nicht erfüllt. Dann existiert aber ein festes $j \in \mathbb{N}_0$ und eine $g_k \in K[k]$, so dass

$$\mathrm{ggT}(q_k, r_{k+j}) = g_k \neq 1 \tag{2.9}$$

ist. Sei J die Menge aller $j \in \mathbb{N}$, die die Eigenschaft (2.9) erfüllen. Diese Menge J heißt Dispersionsmenge. Wir werden in Satz 2.2.3 erläutern wie wir diese algorithmisch bestimmen können.

Wir können nun für ein $j \in J$ den gemeinsamen Teiler g_k herausdividieren. Dadurch erhalten wir die neuen Funktionen p'_k, q'_k und r'_k wie folgt:

$$p'_k = p_k g_k g_{k-1} \cdots g_{k-j+1}, \quad q'_k = \frac{q_k}{g_k} \quad \text{und } r'_k = \frac{r_k}{g_{k-j}}. \tag{2.10}$$

Denn es gilt:

$$\frac{p'_{k+1}}{p'_k}\frac{q'_{k+1}}{r'_{k+1}} = \frac{p_{k+1}g_{k+1}g_k \cdots g_{k-j+2}}{p_k g_k g_{k-1} \cdots g_{k-j+1}}\frac{q_{k+1}}{g_{k+1}}\frac{g_{k-j+1}}{r_{k+1}} = \frac{p_{k+1}}{p_k}\frac{q_{k+1}}{r_{k+1}}.$$

Mit (2.9) folgt nun

$$\mathrm{ggT}(q'_k, r'_{k+j}) = \mathrm{ggT}\left(\frac{q_k}{g_k}, \frac{r_{k+j}}{g_k}\right) = 1,$$

so dass für p'_k, q'_k und r'_k die Gleichung (2.9) nicht mehr gültig ist. Durch Anwendung der Ersetzungsregeln (2.10) für alle $j \in J$ erfüllen p_k, q_k und r_k die Eigenschaft (2.8). Falls die Menge J mehr als eine Zahl enthält, muss vor jedem Ersetzungsschritt die ggT-Eigenschaft überprüft werden, denn es ist möglich, dass eines der vorigen g_k bereits der Bedingung genügt. □

Nun befassen wir uns mit der im Beweis erwähnten Dispersionsmenge. Für zwei Polynome q_k und r_k ist

$$\mathrm{disp}(q_k, r_k) = \max\{j \in \mathbb{N}_0 \big| \mathrm{ggT}(q_k, r_{k+j}) \neq 1\}$$

die Dispersion von q_k und r_k. Um eine vollständige Notation zu erhalten, sei das Maximum der leeren Menge gleich $-\infty$. Wir beachten, dass die Notation nicht symmetrisch ist. Die Dispersionsmenge von q_k und r_k ist definiert durch

$$J = \{j \in \mathbb{N} \big| \mathrm{ggT}(q_k, r_{k+j}) \neq 1\}.$$

Die Dispersion von zwei Polynomen gibt Informationen über die Shift-Strukturen, ob gemeinsame Teiler existieren, falls das Polynom r_k in seiner Variablen k um j geshiftet wird.

Im nächsten Algorithmus werden wir die Dispersionsmenge J bestimmen. Es gibt zwei Möglichkeiten diese zu bestimmen: Eine ist die algorithmisch effiziente rationale Faktorisierung, die wir im nächsten Satz betrachten werden, eine andere besteht darin, sie mit Hilfe von Resultanten zu errechnen.

Die Menge J sind die nichtnegativen ganzzahligen Nullstellen des Polynoms in j, das durch die Resultante von q_k und r_{k+j} entsteht. Die Dispersionsmenge

mit Hilfe der Resultante zu berechnen, scheint ein kurzer und sehr eleganter Weg zu sein. Aber es stellt sich heraus, dass es ineffizienter ist als die Bestimmung der Dispersionsmenge durch die rationale Faktorisierung. Ein Nachteil ist, dass die Resultante das zu lösende Problem zu einem Problem mit zwei Variablen macht, obwohl die Dispersionsmenge nur die Information über eine Variable benötigt. Des Weiteren erzeugt die Resultante der zwei Polynome q_k und r_{k+j} mit dem Grad m und n bezüglich k ein Polynom mit dem Grad nm bezüglich j, dies ist genauer auf den Seiten $79-84$ in [PWZ97] erläutert.

Wir werden hier die Dispersionsmenge mit der effizienteren Methode der rationalen Faktorisierung berechnen. Eine wichtige Tatsache hierfür ist, dass die rationale Faktorisierung algorithmisch realisiert werden kann. Da die Berechnung einer Resultante ein Polynom (in j) mit viel höherem Grad erzeugt, ist die Berechnung der Nullstellen viel zeitaufwendiger als die Faktorisierung der Ausgangspolynome q_k und r_{k+j}. Weiter ist die Implementierung der meisten Computeralgebrasysteme für die rationale Faktorisierung von Polynomen genauso effizient wie die, die alle ganzzahligen Nullstellen von Polynomen findet.

Der Algorithmus zur Bestimmung der Dispersionsmenge mit Hilfe der rationalen Faktorisierung wurde zuerst von Koornwinder[1] implementiert und von Man und Wright in [MW94] beschrieben. Er ist ein effizienter Weg um die Dispersionsmenge zu erhalten und macht die Berechnung der Resultante überflüssig. Wir verwenden den Algorithmus aus [Koe97].

Algorithmus 2.2.3 (Dispersionsmenge) *Der folgende Algorithmus bestimmt die Dispersionsmenge*

$$J = \{j \in \mathbb{N}_0 \big| \mathrm{ggT}(q_k, r_{k+j}) \neq 1\}$$

zweier Polynome q_k, $r_k \in K[k]$ unter Voraussetzung, dass ein Faktorisierungsalgorithmus in $K[k]$ existiert. Dies ist insbesondere dann anwendbar, falls $q_k,\ r_k \in \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_p)[k]$.

1. *Eingabe: Zwei Polynome q_k und $r_k \in K[k]$.*
2. *Faktorisiere q_k und r_k über K.*
3. *Setze $J = \{\}$. Berechne für jedes Paar irreduzibler Faktoren s_k von q_k und t_k von r_k die Menge $D = \mathrm{disp}(s, t, k)$ in folgenden Schritten:*
   (a) *Falls die Grade $m = \deg(s_k, k)$ und $n = \deg(t_k, k)$ verschieden sind, gib $D = \{\}$ aus*[2].
   (b) *Berechne die Koeffizienten*[3] *$a = \mathrm{coeff}(s_k, k, n)$, $b = \mathrm{coeff}(s_k, k, n-1)$, $c = \mathrm{coeff}(t_k, k, n)$ und $d = \mathrm{coeff}(t_k, k, n-1)$.*
   (c) *Falls $j := \frac{bc-ad}{acn} \notin \mathbb{N}_{\geq 0}$, dann gib $D = \{\}$ aus.*
   (d) *Test, ob $cs_k - at_{k+j} = 0$. Ist dies der Fall, setze $D = \{j\}$, andernfalls $D = \{\}$ und gib D aus.*

[1] Der Algorithmus ist in Maple im Package `zeilb` implementiert und in [Koo93] nachzulesen.
[2] $\deg(a_k, k)$ steht für den Grad des Polynoms a_k bezüglich der Variablen k.
[3] $\mathrm{coeff}(a_k, k, n)$ steht für den Koeffizienten von k^n des Polynoms a_k.

$J = J \cup D.$

4. *Ausgabe:* J.

Beweis: Als erstes zeigen wir, dass die Unterroutine $\mathrm{disp}(q, r, k)$ für zwei über $K[k]$ irreduzible Polynome q_k und r_k die Dispersion bestimmt.

Wenn die Polynome nicht den gleichen Grad haben, können wir keine Dispersion finden. Daher nehmen wir zunächst an, dass q_k und r_k eine Dispersion $j \geq 0$ und den gleichen Grad haben. Daher müssen sie auch ein Vielfaches voneinander sein. Nach Definition gilt dann

$$\mathrm{ggT}(q_k, r_{k+j}) = g_k \neq 1,$$

wobei g_k ein rationaler Faktor von q_k und g_{k-j} ein rationaler Faktor von r_k ist. Da nach Voraussetzung q_k und r_k irreduzibel sind, muss g_k den gleichen Grad wie q_k und r_k haben.

Daher haben die zwei Polynome

$$q_k = ak^n + bk^{n-1} + \ldots \quad \text{und} \quad r_k = ck^n + dk^{n-1} + \ldots \tag{2.11}$$

die Dispersion $j \in \mathbb{N}_0$ genau dann, wenn

$$\frac{c}{a} q_k = r_{k+j} = c(k+j)^n + d(k+j)^{n-1} + \ldots = ck^n + (cnj + d)k^{n-1} \ldots, \tag{2.12}$$

wobei wir den binomischen Lehrsatz verwendet haben. Die entstandene Identität (2.12) kann nur gültig sein, wenn die Koeffizienten von k^{n-1} übereinstimmen. Es muss also nach (2.11) und (2.12) gelten $\frac{bc}{a} = cnj + d$ oder wie in Algorithmus 2.2.3 angegeben wurde

$$j = \frac{bc - ad}{acn}. \tag{2.13}$$

Damit muss j eine aus (2.13) resultierende, nichtnegative ganze Zahl sein. Mit dem gegebenen j können wir nun r_{k+j} bestimmen und überprüfen, ob die Gleichung (2.12) gilt.

Der Algorithmus berechnet die ganze Dispersionsmenge J von q_k und r_k, indem die Unterroutine disp für jedes irreduzible Paar von Faktoren die Dispersion bestimmt und sie in der Menge J zusammenfasst. □

Die Funktionen zur Bestimmung der Dispersion und der Dispersionsmenge sind nach den oben erläuterten Algorithmen in MuPAD implementiert. Im folgenden Beispiel vergleichen wir den Zeitaufwand des Algorithmus mit Resultante und ohne Resultante.

Beispiel 2.2.4
Wir geben zwei Polynome in faktorisierter Form vor, so dass wir die Dispersionsmenge abgelesen können. Wir werden den Zeitaufwand der beiden Prozeduren in MuPAD vergleichen. Der Algorithmus, der die Resultante verwendet, ist sehr leicht wie folgt zu implementieren. In dem Algorithmus nehmen wir mit `assume` an, dass die Parameter beider Eingabeterme ganze Zahlen sind und verwenden

den `solve` Befehl, der durch den Zusatz `Domain=Dom::Integer` nur ganzzahlige Lösungen sucht.

MuPAD

```
>> dispersionsset:=proc(q,r,k)
      begin
      assume(indets(q) ,Type::Integer);
       assume(indets(r) ,Type::Integer);
      select(solve(polylib::resultant(q, subs(r,k=k+j),k),j,
      Domain=Dom::Integer),testtype, Type::NonNegInt);
   end_proc:
```

Der von mir implementierte Befehl `dispersionsmenge(q,r,k)` gibt die Dispersionsmenge für zwei Polynome aus, wobei die Eingabeterme `q` und `r` die zu betrachtenden Polynome sind und `k` die Variable der Polynome.

Zunächst geben wir die Polynome q_k und r_k in MuPAD ein:

MuPAD

```
>> q:=expand(4*k^3*6*(k+3000)*(k+a)^2*(k+b));
>> r:=expand(subs(q,k=k-799)):
```

Output

```
144000*a*k^5 + 48*a*k^6 + 72000*b*k^5 + 24*b*k^6 + 72000*k^6
+ 24*k^7 + 72000*a^2*k^4 + 24*a^2*k^5 + 72000*a^2*b*k^3
+ 24*a^2*b*k^4 + 144000*a*b*k^4 + 48*a*b*k^5
```

Wobei unser r_k das Polynom q_k ist, in das wir $k = k-799$ substituiert haben. Dieses geben wir nicht aus, da der Ausdruck zu komplex ist und einen Großteil des Bildschirms füllt. Die Parameter in `dispersionset` und `dispersionsmenge` sind die selben.

MuPAD

```
>> t:=time(): dispersionsmenge(q,r,k), time = (time()-t)*unit::ms;
>> t:=time(): dispersionsset(q,r,k),   time = (time()-t)*unit::ms;
```

Output

```
{799, 3799}, time = 108*ms
{799, 3799}, time = 119799*ms
```

Hier sehen wir, dass die Berechnung mit dem Algorithmus ohne Verwendung der Resultante viel effizienter die beiden Shifts 799 und 3799 findet. Der Zeitaufwand mit `dispersionsmenge` ist 108 Millisekunden und der mit Hilfe der Resultanten 119799 ms. Wir können weiter zeigen, dass allein der Aufwand zur Berechnung der Resultante schon größer ist als der ganze Algorithmus zur Bestimmung der Dispersionsmenge durch rationale Faktorisierung.

MuPAD

```
>> t:=time(): polylib::resultant(q, subs(r,k=k+j),k):
   time = (time()-t)*unit::ms;
```

Output

```
time = 61139*ms
```

Daran sehen wir schon, dass die Bestimmung der Resultante sehr aufwendig ist. Wenn wir nun den Grad des Polynoms in j ermitteln, ergibt sich:

MuPAD

```
>> degree(polylib::resultant(q, subs(r,k=k+j),k));
```

Output

```
49
```

Damit sehen wir, dass durch das Bilden der Resultante ein Polynom in j vom Grad 49 entsteht. Wir beobachten auch, dass die Koeffizienten, die entstehen, sehr groß sind, daher ist die Berechnung der Nullstellen sehr aufwendig. In unserem Fall braucht der Algorithmus mit Hilfe der Resultante ungefähr 1000 mal länger als mit der rationalen Faktorisierung.

Um die Berechnung der Resultante zu beschleunigen, sollten die Polynome vorher faktorisiert werden. Da kommt natürlich die Frage auf, warum einige Autoren in ihren Büchern oder in implementierten Algorithmen überhaupt noch die Methode mit der Resultante verwenden. Der einzige Grund ist, dass die auf der Resultanten basierende Methode allgemeiner ist und sie über jedem Körper berechnet werden kann, indem die Determinate bestimmt werden kann. Auf der anderen Seite ist kein allgemeiner Faktorisierungsalgorithmus für Polynome über allen Körpern bekannt. Da wir aber $K = \mathbb{Q}$ vorgegeben haben, können wir ohne Bedenken den Algorithmus, der auf der rationalen Faktorisierung basiert, zur Bestimmung der Dispersionsmenge verwenden.

In (2.6) haben wir schon gesehen, dass f_k rational ist. Mit Hilfe des nächsten Lemmas können wir sogar zeigen, dass f_k ein Polynom ist.

Lemma 2.2.5 *Die durch (2.5), (2.7) und (2.8) beschriebene Funktion f_k ist ein Polynom.*

Beweis: Angenommen f_k ist kein Polynom. Damit ist f_k rational und hat die Form

$$f_k = \frac{c_k}{d_k},$$

wobei c_k und d_k Polynome sind und $\deg(d_k, k) > 0$ ist. Dies werden wir nun zu einem Widerspruch führen. Es gilt dann:

$$\mathrm{ggT}(c_k, d_k) = 1 = \mathrm{ggT}(c_{k-1}, d_{k-1}). \tag{2.14}$$

Indem wir (2.7) mit $d_k d_{k-1}$ multiplizieren und $d_k f_k$ durch c_k ersetzen, erhalten wir

$$d_k d_{k-1} p_k = d_k d_{k-1}(q_{k+1} f_k - r_k f_{k-1}) = c_k d_{k-1} q_{k+1} - c_{k-1} d_k r_k. \tag{2.15}$$

Nun sei $j \geq 0$ die Dispersion von d_k mit sich selbst. Sei j die größte ganze Zahl, die die Eigenschaft

$$\mathrm{ggT}(d_k, d_{k+j}) = g_k \neq 1 \tag{2.16}$$

erfüllt. Dieses j existiert, da $0 \in \text{disp}(d_k, d_k, k)$ ist. Da j maximal und d_{k+j} ein Vielfaches von g_k ist, erhalten wir

$$\text{ggT}(d_{k-1}, d_{k+j}) = 1 = \text{ggT}(d_{k-1}, g_k). \tag{2.17}$$

Wenn wir k in (2.16) um $-(j+1)$ verschieben, ergibt sich

$$\text{ggT}(d_{k-(j+1)}, d_{k-1}) = g_{k-(j+1)} \neq 1 \tag{2.18}$$

und verschieben wir auf der linken Seite von (2.17) nun k um j, gilt mit der Eigenschaft $d_{k-(j+1)}$ ein Vielfaches von $g_{k-(j+1)}$, dass

$$\text{ggT}(d_{k-(j+1)}, d_k) = 1 = \text{ggT}(g_{k-(j+1)}, d_k). \tag{2.19}$$

Als Nächstes betrachten wir unsere Hauptgleichung (2.15) genauer. Zu zeigen ist, dass die Terme dieser Gleichung Polynome sind. Um diese zu prüfen, teilen wir die Gleichung durch g_k und wir erhalten:

$$\frac{d_k d_{k-1} p_k}{g_k} = \frac{c_k d_{k-1} q_{k+1}}{g_k} - \frac{c_{k-1} d_k r_k}{g_k}. \tag{2.20}$$

Mit Gleichung (2.16) wissen wir, dass g_k ein Teiler von d_k ist, daraus können wir folgern, dass die linke Seite der Gleichung ein Polynom ist. Mit der gleichen Argumentation ist auch der ganz rechte Term ein Polynom, was zur Folge hat, dass auch $c_k d_{k-1} q_{k+1}$ durch g_k teilbar ist. Durch die Gleichung (2.17) folgt, dass d_{k-1} und g_k teilerfremd sind. Weiter können wir sagen, dass g_k ein Teiler von d_k ist und mit (2.14) folgt, dass auch c_k und g_k teilerfremd sein müssen. Somit muss q_{k+1} durch g_k teilbar sein und damit ist auch g_{k-1} ein Teiler von q_k.

Als nächstes teilen wir (2.15) durch $g_{k-(j+1)}$ und erhalten

$$\frac{d_k d_{k-1} p_k}{g_{k-(j+1)}} = \frac{c_k d_{k-1} q_{k+1}}{g_{k-(j+1)}} - \frac{c_{k-1} d_k r_k}{g_{k-(j+1)}}. \tag{2.21}$$

Da nach (2.18) $g_{k-(j+1)}$ ein Teiler von d_{k-1} ist, folgt, dass die linke Seite von (2.21) ein Polynom ist. Das Gleiche gilt für den mittleren Teil der Gleichung. Daraus folgt: $c_{k-1} d_k r_k$ ist durch $g_{k-(j+1)}$ teilbar. Aus (2.19) erhalten wir, dass d_k und $g_{k-(j+1)}$ teilerfremd sind. Des Weiteren gilt nach (2.14), dass c_{k-1} und $g_{k-(j+1)}$ teilerfremd sind, da sich aus (2.18) ergibt, dass $g_{k-(j+1)}$ ein Teiler von d_{k-1} ist. Damit ergibt sich, dass r_k durch $g_{k-(j+1)}$ teilbar ist und g_{k-1} ein Teiler von r_{k+j} ist.

Dadurch haben wir gezeigt, dass q_k und r_{k+j} durch g_{k-1} teilbar sind. Dies widerspricht der Aussage (2.8) aus Lemma 2.2.2. Damit ist gezeigt, dass f_k ein Polynom ist. □

Der letzte Schritt zur Vervollständigung des Gosper-Algorithmus ist, dass wir eine Gradschranke für das Polynom f_k suchen. Kennen wir eine Gradschranke a priori, so sind wir in der Lage ein allgemeines Polynom in die Gleichung (2.7) einzusetzen und wir können f_k durch einen Koeffizientenvergleich berechnen. Wenn wir keine Lösung für unser Gleichungssystem finden, können wir daraus schließen, dass kein solches f_k existiert.

Anstatt den Satz speziell für die Gleichung (2.7) zu betrachten, werden wir ihn allgemeiner zeigen und aufschreiben[4].

Satz 2.2.6 (Gradschranke) *Für jede Polynomlösung f_k der Rekursion*

$$A_k f_{k+1} + B_k f_k = C_k$$

mit den Polynomen $A_k,\ B_k,\ C_k \in K[k]$ erhalten wir die Gradschranke durch folgenden Algorithmus:

1. *Ist $\deg(A_k - B_k, k) \leq \deg(A_k + B_k, k)$ dann gilt*

$$\deg(f_k, k) = \deg(C_k, k) - \deg(A_k + B_k, k).$$

2. *Ist hingegen $n = \deg(A_k - B_k, k) > \deg(A_k + B_k, k)$, so sei a der Koeffizient von k^n des Polynoms $A_k - B_k$ ($a \neq 0$) und b der Koeffizient von k^{n-1} von $A_k + B_k$.*

   (a) *Falls $-2b/a \notin \mathbb{N}_{\geq 0}$ dann gilt*

$$\deg(f_k, k) = \deg(C_k, k) - n + 1.$$

   (b) *Ist aber $-2b/a \in \mathbb{N}_{\geq 0}$, dann gilt*

$$\deg(f_k, k) \in \max\{-2b/a, \deg(C_k, k) - n + 1\}.$$

Der Satz ist allgemein formuliert, in unserem Fall sieht die Rekursion aus (2.7) wie folgt aus:

$$p_k = q_{k+1} f_k - r_k f_{k-1}. \tag{2.22}$$

Um den Satz nun auf unser spezielles Problem anzuwenden, müssen wir nur $C_k = p_k$, $A_k = q_{k+1}$ und $B_k = -r_k$ setzen. Besonders bei der Implementierung ist es wichtig, dass die Indexverschiebungen und Vorzeichenwechsel beachtet werden. Wir kommen nun zum Beweis des Satzes.

Beweis: Wir schreiben unsere Rekursionsgleichung um und erhalten

$$C_k = (A_k + B_k)\frac{f_{k+1} + f_k}{2} + (A_k - B_k)\frac{f_{k+1} - f_k}{2}. \tag{2.23}$$

Für jedes Polynom $f_k \neq 0$ gilt dann die Gradbeziehung

$$\deg(f_{k+1} - f_k, k) = \deg(f_{k+1} + f_k, k) - 1, \tag{2.24}$$

da beim Polynom $f_{k+1} - f_k$ die höchste Potenz wegfällt, wobei sie bei $f_{k+1} + f_k$ stehen bleibt. Setzen wir nun noch $\deg(0) = -1$, dann gilt (2.24) allgemein für alle $f_k \in K[k] \setminus \{0\}$.

Gilt also nun $\deg(A_k - B_k, k) \leq \deg(A_k + B_k, k)$, dann ist der zweite Summand der Gleichung (2.23) definitiv vom Grad kleiner als der erste und daraus folgt die erste Behauptung.

[4] Zu finden in [Koe06].

Wenn nun $\deg(A_k - B_k, k) > \deg(A_k + B_k, k)$, ist die Situation schwieriger aufzulösen. Sei m der Grad von f_k und $f_k = ck^m + \ldots$. Dann folgt für die höchsten Koeffizienten von (2.23):

$$C_k = (b + a\frac{m}{2})ck^{m+n-1} + \ldots.$$

Daraus resultiert dann Behauptung (2). □

Es ist zu erwähnen, dass es sich beim Algorithmus zur Bestimmung der Gradschranke aus Satz 2.2.6 nicht in allen Fällen um den echten Grad handelt, denn in Fall (2*b*) gibt es zwei Möglichkeiten, in den anderen Fällen wird der Grad des resultierenden Polynoms aber exakt bestimmt.

Wenn wir nun eine Polynomlösung für das Polynom f_k suchen, müssen wir nur den Lösungsansatz in (2.22) einsetzen und durch einen Koeffizientenvergleich erhalten wir unser Resultat. Ist die Gradschranke keine nichtnegative ganze Zahl oder besitzt das lineare Gleichungssystem keine Lösung, ist erwiesen, dass die Rekursionsgleichung (2.22) keine Polynomlösung hat. In Bezug auf das Summationsproblem heißt das, dass keine diskrete Stammfunktion s_k von a_k existiert, welche ein hypergeometrischer Term ist.

Wir fassen nun den Gosper-Algorithmus zusammen, dieser besteht aus den kennen gelernten Algorithmen.

Algorithmus 2.2.7 (Gosper-Algorithmus) *Gegeben ist ein hypergeometrischer Term a_k. Der folgende Algorithmus bestimmt, ob eine hypergeometrische diskrete Stammfunktion s_k existiert und gibt sie dann aus.*

1. *Eingabe: Ein (rationaler) hypergeometrischer Term $a_k \neq 0$.*
2. *Berechne den Quotienten von a_{k+1}/a_k. Wir suchen $u_k, v_k \in K[k]$, so dass*
$$\frac{a_{k+1}}{a_k} = \frac{u_k}{v_k}$$
gilt.
3. *Berechne p_k, q_k und $r_k \in K[k]$ mit der Eigenschaft (2.8) aus Lemma 2.2.2.*
4. *Bestimme die Gradschranke M für das Polynom $f_k \in K[k]$ mit dem Algorithmus aus Satz 2.2.6. Falls $M < 0$, brich ab. Wir wissen dann, dass keine hypergeometrische diskrete Stammfunktion für den Eingabeterm existiert.*
5. *Setze das allgemeine Polynom*
$$f_k = b_0 + b_1 k + b_2 k^2 + \ldots + b_M k^M$$
in die Rekursionsgleichung
$$p_k = q_{k+1} f_k - r_k f_{k-1}$$
für f_k ein. Wir bestimmen die Koeffizienten, indem wir das resultierende lineare Gleichungssystem nach den Unbekannten b_l ($l = 0, \ldots, M$) auflösen.

6. *Falls das lineare Gleichungssystem keine Lösung besitzt, brich ab, denn dann existiert keine hypergeometrische diskrete Stammfunktion für den Eingabeterm.*

7. *Ausgabe:* $s_k := \frac{r_k}{p_k} f_{k-1} a_k$.

Der Befehl `gosper(a,k)` ist die von mir in MuPAD implementierte Variante des oben vorgestellten Algorithmus, wobei `a` dem Eingabeterm a_k und `k` dem Summationsindex entspricht. Die Funktion berechnet die hypergeometrische diskrete Stammfunktion s_k, falls sie existiert.

Wenn der Gosper-Algorithmus eine hypergeometrische Stammfunktion s_k für a_k findet, können wir nach (2.5) sagen, dass s_k ein rationales Vielfaches von a_k ist:

$$s_k = R_k a_k \qquad \text{mit} \quad R_k = \frac{r_k}{p_k} f_{k-1}.$$

Wir nennen R_k rationales Zertifikat des Gosper-Algorithmus. Wenn wir nämlich R_k gegeben haben, ist es sehr leicht zu prüfen, ob $s_k = R_k a_k$ eine hypergeometrische Stammfunktion von a_k ist. Mit einfacher rationaler Arithmetik können wir dann folgende Aussage überprüfen

$$s_{k+1} - s_k = R_{k+1} a_{k+1} - R_k a_k = a_k.$$

Diese Gleichung ist äquivalent zu

$$\frac{R_k + 1}{R_{k+1}} = \frac{a_{k+1}}{a_k}.$$

Daran sehen wir, ohne zu wissen, wo das Resultat R_k herkommt, ob $s_k = R_k a_k$ eine Stammfunktion ist. Wir haben auch gesehen, dass sowohl für die Anwendung des Gosper-Algorithmus als auch die Prüfung des Resultats durch R_k nur rationale Arithmetik nötig ist.

Der in MuPAD implementierte Befehl `gospertest(a,s,k)` überprüft, ob s_k eine hypergeometrische diskrete Stammfunktion des eingegebene Term a_k ist. Bei dem Befehl ist `a` der hypergeometrische Eingabeterm a_k und `s` die zu überprüfende Stammfunktion. Der Befehl gibt `TRUE` aus, falls s_k eine hypergeometrische Stammfunktion zum eingegebenen a_k ist und sonst `FALSE`. Intern verwendet er aber nicht das rationale Zertifikat, sondern überprüft $s_{k+1} - s_k - a_k = 0$ aus Gleichung (2.1). Ist diese Gleichung gleich Null, ist s_k eine diskrete Stammfunktion zu a_k.

Zum Abschluss dieses Abschnittes werden wir uns nun mit einigen Beispielen beschäftigen, um die Anwendungen des Algorithmus zu verdeutlichen.

Beispiel 2.2.8 (Polynome)
Jedes Polynom a_k ist gospersummierbar, da für jedes Polynom a_k eine polynomiale Antidifferenz s_k existiert. Wenn wir nicht die gemeinsamen Teiler aus a_{k+1}/a_k herausdividieren, führt uns die Anwendung des Gosper-Algorithmus zu folgender Wahl $p_k = 1,\ q_k = a_k$ und $r_k = a_{k-1}$. Es folgt dann, dass die Dispersionsmenge gleich 1 ist und durch Umschreiben nach Lemma 2.2.2 erhalten wir $p_k = a_k$ und $q_k = r_k = 1$. Dadurch erhalten wir nach (2.7) die Gleichung

$$a_k = f_k - f_{k-1} \tag{2.25}$$

für f_k. Nach (2.5) gilt dann $f_k = s_{k+1}$. Da $q_{k+1} - r_k = 0$ ist, ergibt sich aus dem Fall (2*b*) aus Satz 2.2.6, dass die Gradschranke $\deg(p_k, k) + 1 = \deg(a_k, k) + 1$ ist. Die Koeffizienten für s_k erhalten wir dann, indem wir ein lineares Gleichungssystem mit $\deg(a_k, k) + 1$ Variablen lösen.

Wir betrachten mit Hilfe von MuPAD noch ein konkretes Beispiel und zwar $a_k = k^3 + 4k^2 + 5k + 3$:

```
MuPAD
>> gosper(k^3+4*k^2+5*k+3,k);
Output
k^4/4 + (5*k^3)/6 + (3*k^2)/4 + (7*k)/6 - 3
```

Damit ergibt sich die hypergeometrische diskrete Stammfunktion $s_k = k^4/4 + (5k^3)/6 + (3k^2)/4 + (7k)/6 - 3$.

Um die Antidifferenz für Polynome zu bestimmen, existieren jedoch schnellere Algorithmen als der Gosper-Algorithmus.

Beispiel 2.2.9 (Rationale Funktionen)
Mit dem Gosper-Algorithmus können wir zeigen, dass nicht jede rationale Funktion a_k gospersummierbar ist. Zum Beispiel hat $a_k = 1/k$ keine hypergeometrische Stammfunktion. Wir setzen $p_k = 1,\ q_k = k - 1$ und $r_k = k$. Diese erfüllen (2.8) aus Lemma 2.2.2 und die Gradschranke nach Satz 2.2.6 ist gleich Null. Für f_k erhalten wir eine Konstante c und damit gilt: nach (2.7)

$$1 = ck - ck = 0.$$

Aus dieser Gleichung folgt, dass keine Lösung existiert. Damit haben wir gezeigt, dass die harmonischen Zahlen

$$H_n = \sum_{k=1}^{n} a_k = \sum_{k=1}^{n} \frac{1}{k}$$

keine hypergeometrischen Terme sind.

Falls ein rationaler Term a_k eine rationale hypergeometrische Stammfunktion s_k besitzt, ist dieser gospersummierbar und der Gosper-Algorithmus findet sicher eine Lösung s_k. Wir betrachten dazu das folgende Beispiel:

$$a_k = \frac{k}{(k+1)(k+2)(k+3)}.$$

Anstatt die Summe mit Hilfe einer Teleskopsumme zu bestimmen, werden wir den Gosper-Algorithmus nutzen. Nach Vereinfachung von a_{k+1}/a_k wählen wir $p_k = 1,\ q_k = k^2$ und $r_k = (k-1)(k+3)$. Die Dispersionsmenge ist 1, daher müssen wir die Polynome nach Lemma 2.2.2 umschreiben und wir erhalten: $p_k = k,\ q_k = k$ und $r_k = k + 3$. Als nächstes bestimmen wir die Gradschranke für f_k mit Hilfe von Satz 2.2.6 und erhalten nach Fall (2*b*) die Schranke 2. Wir setzen dann $f = a + bk + ck^2$ in unsere Hauptgleichung (2.7) ein und es ergibt sich:

$$k = (k+1)(a + bk + ck^2) - (k+3)(a + b(k-1) + c(k-1)^2).$$

Wir fassen die Gleichung zusammen und es ergibt sich:

$$(5a - b - 1)k + 3b - 2a - 3c = 0$$

Der Gleichung ist zu entnehmen, dass unser Lösungsraum die Dimension 2 hat, durch Bestimmung der Koeffizienten und indem wir $c = 0$ setzen, erhalten wir die Lösungen $a = -3/2$ und $b = -1$. Diese setzen wir in das Polynom f_k ein und erhalten $f_k = -3/2 - k$. Das Ergebnis für s_k ist nach Gleichung (2.5):

$$s_k = \frac{r_k}{p_k} f_{k-1} a_k = -\frac{(k+3)}{k}\left(\frac{3}{2} + (k-1)\right)\frac{k}{(k+1)(k+2)(k+3)}.$$

Dieses Ergebnis faktorisieren wir und erhalten:

$$s_k = -\frac{(2k+1)}{(2(k+2)(k+1))}.$$

Nun berechnen wir die hypergeometrische diskrete Stammfunktion s_k mit dem implementierten Befehl `gosper(a,k)`. Der Befehl kann zusätzlich weitere Zwischenergebnisse ausgeben, indem wir vor dem Befehl `setuserinfo(gosper,6)` aufrufen[5]. Die Zwischenergebnisse können optional mit einem Wert von `1` bis `6` aufgerufen werden, wobei `1` immer nur die Fehlermeldungen ausgibt, falls Fehler entstehen und der Algorithmus scheitert.

MuPAD

```
>> setuserinfo(gosper,6):
>> setuserinfo(umschreibenpolynome,1):
>> setuserinfo(gradschranke,1):
>> a:=k/(k+2)/(k+1)/(k+3):
>> s:=gosper(a,k);
```

Output

```
Info: a[k+1]/a[k]:=2*k/(k^2 + 4*k) + 1/(k^2 + 4*k) + k^2/(k^2 + 4*k)
Info: Dispersionsmenge:={1}
Info: Teil 2b
Info: Gradschranke:=2
Info: p:=k
Info: q:=k
Info: r:=k + 3
Info: f:=- k - 3/2
-(2*k^2 + 7*k + 3)/(2*(k + 1)*(k + 2)*(k + 3))
```

Damit sehen wir durch den Befehl `gosper`, dass $-(2k+1)/(2(k+2)(k+1))$ eine hypergeometrische Stammfunktion von a_k ist. Durch `Info` werden die oben schon gesehenen Zwischenergebnisse, die während der Berechnung entstehen, ausgegeben.

Wir überprüfen auch gleichzeitig mit `gospertest`, ob das Ergebnis eine hypergeometrische diskrete Stammfunktion ist.

[5]Um Informationen über die Dispersionsmenge und Gradschranke zu erhalten, muss auch `setuserinfo(umschreibenpolynome,1)` und `setuserinfo(gradschranke,1)` eingegeben werden.

```
MuPAD
>> gospertest(a,s,k);
Output
TRUE
```

Beispiel 2.2.10 (Binomischer Satz)
Als nächstes wollen wir testen, ob $a_k = \binom{n}{k}$ gospersummierbar ist. Dazu fassen wir n als unbestimmte Variable auf, arbeiten also über dem Grundkörper $K = \mathbb{Q}(n)$. Wenn a_k gospersummierbar wäre, würden wir für ein beliebiges m einen hypergeometrischen Term für die Formel

$$\sum_{k=0}^{m} \binom{n}{k}$$

finden. Wenn wir den Gosper-Algorithmus auf a_k anwenden, erhalten wir im ersten Schritt

$$\frac{a_{k+1}}{a_k} = \frac{n-k}{k+1}.$$

Damit ergibt sich die endgültige Wahl $p_k = 1,\ q_k = n - k + 1$ und $r_k = k$. Wenn wir diese Parameter als Eingabe für den Algorithmus zur Bestimmung der Gradschranke aus Satz 2.2.6 verwenden, ergibt sich nach Fall (1) für f_k eine Gradschranke von -1. Damit existiert kein f_k, welches der Hauptgleichung (2.7) genügt und daraus folgt: a_k ist nicht gospersummierbar. Durch den implementierten Algorithmus wird dieses Resultat bestätigt:

```
MuPAD
>> setuserinfo(gosper,4):
>> setuserinfo(umschreibenpolynome,1):
>> setuserinfo(gradschranke,1):
>> gosper(binomial(n,k),k);
Output
Info: a[k+1]/a[k]:=n/(k + 1) - k/(k + 1)
Info: Dispersionsmenge:={}
Info: Teil 1
Info: Gradschranke:=-1
Es existiert keine hypergeometrische Antidifferenz s[k], da die
Gradschranke < 0 ist
FAIL
```

Durch den Output `FAIL` wissen wir, dass für diese Eingabe keine hypergeometrische Stammfunktion existiert. Dies bedeutet aber nicht, dass die Summe nicht durch eine andere Methode in eine vereinfachte Form gebracht werden kann. Wir werden dieses Problem im nächsten Kapitel wieder aufgreifen und werden dann sehen, wie wir auch dieses algorithmisch lösen können.

Falls wir den Binomialkoeffizienten mit $(-1)^k$ multiplizieren, sehen wir, dass $a_k = (-1)^k\binom{n}{k}$ gospersummierbar ist. Es gilt nun

$$\frac{a_{k+1}}{a_k} = \frac{k-n}{k+1}.$$

Damit ergibt sich $p_k = 1,\ q_k = k-n-1$ und $r_k = k$. In dem abgeänderten Problem ist die Gradschranke für f_k nach Fall $(2a)$ aus Satz 2.2.6 gleich 0. Indem wir nun $f_k = c$ in unsere Gleichung (2.7) einsetzen, ergibt sich:

$$1 = (k-n)c - kc = -nc.$$

Die Lösung $c = -1/n$ führt dann zu $f_k = -1/n$. Daraus ergibt sich dann nach (2.5):

$$s_k = \frac{r_k}{p_k} f_{k-1} a_k = -\frac{k}{n} a_k = -\frac{k}{n}(-1)^k \binom{n}{k}.$$

Zum Abschluss berechnen wir das Problem noch mit dem Befehl `gosper`. Mit `setuserinfo(gosper,2)` erhalten wir nur den Quotienten a_{k+1}/a_k und die Fehlermeldungen, die auftreten, falls der Algorithmus nicht erfolgreich ist.

MuPAD

```
>> setuserinfo(gosper,2):
>> gosper((-1)^k*binomial(n,k),k);
```

Output

```
Info: a[k+1]/a[k]:=k/(k + 1) - n/(k + 1)
-(-1)^k*k*binomial(n, k)/n
```

Beispiel 2.2.11
Nun zeigen wir noch kurz, dass das Pochhammersymbol eine hypergeometrische Stammfunktion besitzt. Dazu wählen wir $a_k = (k)_n$. Auch hier betrachten wir n als unbestimmte Variable und arbeiten über dem Grundkörper $K = \mathbb{Q}(n)$.

MuPAD

```
>> gosper(pochhammer(k,n),k);
```

Output

```
(pochhammer(k,n)*(k - 1))/(n + 1)
```

Zu erwähnen ist, dass dieses Beispiel bis jetzt nur in der MuPAD Pro Version 4.5 funktioniert, da die Pochhammersymbole in den älteren Versionen noch nicht integriert waren.

Wir werden noch ein weiteres Bespiel betrachten, in dem ein Summand a_k gegeben ist, der in dieser Form üblicherweise im Zeilberger-Algorithmus auftritt, den wir im nächsten Abschnitt kennen lernen werden.

Beispiel 2.2.12
Gegeben sei $a_k = \binom{n}{k} - \frac{1}{2}\binom{n+1}{k}$. Mit dem von mir in MuPAD implementierten Befehl erhalten wir schnell ein Ergebnis.

MuPAD

```
>> gosper((binomial(n, k) - 1/2*binomial(n + 1, k)),k);
```

Output

```
(k*(binomial(n, k) - binomial(n + 1, k)/2))/(n - 2*k + 1)
```

Paule und Schorn haben den sehr bekannten Artikel [PS95] und dazu das Package `zb.m` in Mathematica veröffentlicht. In diesem Package sind sowohl der Gosper-Algorithmus als auch der Zeilberger-Algorithmus, den wir im nächsten Kapitel vorstellen werden, implementiert. Das Interessante ist, dass der Gosper-Algorithmus aus diesem Package nicht alle Probleme, die durch die Gammafunktionen oder Binomialkoeffizienten dargestellt werden, lösen kann. Die Pochhammersymbole werden überhaupt nicht unterstützt. Das oben vorgeführte Beispiel ist solch ein spezielles Problem mit Binomialkoeffizienten, das nicht gelöst werden kann. Der Grund dafür ist, dass der Algorithmus die Eingabeterme nicht vereinfacht[6]. Wenn dies gemacht wird, sind auch mit diesem Befehl alle Probleme lösbar.

Wir wollen uns noch ein Beispiel ansehen, wo mein implementierter `gosper` Befehl mit dem in MuPAD eingebauten `sum` Befehl verglichen wird.

Beispiel 2.2.13
Bei diesem Beispiel geht es weniger um die Resultate, da diese sehr komplex und schlecht auszugeben sind. Wir interessieren uns für die unterschiedlichen Rechenzeiten, die beide Befehle benötigen, um ein Resultat zu erzeugen. Zunächst definieren wir einen Term a_k, der nicht gospersummierbar ist, und wenden beide Algorithmen auf diesen an.

MuPAD

```
>> a:=(4*k+1)*k!/(2*k+1)!/k;
>> t:=time(): gosper(a,k): time = (time()-t)*unit::ms;
>> t:=time(): sum(a,k): time = (time()-t)*unit::ms;
```

Output

```
(k!*(4*k + 1))/(k*(2*k + 1)!)
time = 39*ms
time = 34*ms
```

Es ist zu sehen, dass bei diesem Beispiel beide Algorithmen noch gleich schnell sind. Aber als nächstes substituieren wir $k = k + 20$ in a_k und bilden eine Differenz aus beiden Termen, dadurch erhalten wir den Term $a_{k+20} - a_k$. Wir werden im nächsten Abschnitt genauer auf diesen Fall eingehen, aber wir können vorwegnehmen, dass diese Differenz immer gospersummierbar ist, falls a_k hypergeometrisch ist. Nun wenden wir beide Befehle auf diesen Term an:

MuPAD

```
>> a:=subs(a,k=k+20)-a;
```

[6] Ein Möglichkeit der Vereinfachung ist der `simpcomb` Algorithmus, den wir in den Grundlagen vorgestellt haben.

```
>> t:=time(): gosper(a,k): time = (time()-t)*unit::ms;
>> t:=time(): sum(a,k): time = (time()-t)*unit::ms;
```

Output

```
((k + 20)!*(4*k + 81))/((2*k + 41)!*(k + 20)) -
 (k!*(4*k + 1))/(k*(2*k + 1)!)
time = 3287*ms
time = 7179664*ms
```

An diesem Resultat ist nun deutlich zu sehen, dass der von mir implementierte Befehl viel schneller ist. Obwohl der eingegebene Ausdruck nicht so kompliziert aussieht, braucht `sum` sehr lange. Der Grund dafür ist, dass ich in dem Befehl `gosper` die rationale Faktorisierung verwende und der `sum` Befehl vermutlich eine Resultante, um die Dispersionsmenge zu bestimmen. Wie wir in Beispiel 2.2.4 gesehen haben, ist diese nicht sehr effizient.

Nachdem wir uns nun mit dem Gosper-Algorithmus ausführlich beschäftigt haben, wollen wir im nächsten Teil sehen, wie er sich bei einem Summanden, der aus einer Linearkombination von hypergeometrischen Termen besteht, verhält.

2.3 Linearisierung des Gosper-Algorithmus

In diesem Abschnitt beschäftigen wir uns mit dem Problem der Linearisierung. Wir können leicht zeigen, dass die Menge der hypergeometrische Terme bezüglich der Multiplikation abgeschlossen ist:

$$\frac{a_{k+1}b_{k+1}}{a_k b_k} = \frac{a_{k+1}}{a_k}\frac{b_{k+1}}{b_k}.$$

Wenn zwei Terme hypergeometrisch sind, ist es ihr Produkt auch. Bezüglich der Addition gilt dies nicht. Beispielsweise ist k^2+1 als Term gospersummierbar, aber 2^k+1 ist es nicht, obwohl 2^k und 1 hypergeometrische Terme sind und für beide eine hypergeometrische Stammfunktion existiert. Wir werden dieses Beispiel später noch einmal aufgreifen. Als Resultat erhalten wir, dass der Gosper-Algorithmus nicht linear ist. Dieser Abschnitt basiert auf [PWZ97].

Im oben genannten Beispiel haben wir gesehen, wenn zwei Terme a_k und b_k gospersummierbar sind, muss $a_k + b_k$ nicht unbedingt gospersummierbar sein. Weiter ist es sogar möglich, dass a_k+b_k gospersummierbar ist, aber die einzelnen Terme a_k und b_k müssen es nicht sein. Ein Beispiel hierfür ist $a_k = 1/(k+1)$ und $b_k = -1/k$.

Es ist wünschenswert, eine Möglichkeit zu finden, die eine Linearkombination von hypergeometrischen Termen so einteilt, dass wir für die eingeteilten Klassen entscheiden können, ob eine hypergeometrische Stammfunktion existiert oder nicht. Petkovšek, Wilf und Zeilberger haben einen Weg gefunden den Gosper-Algorithmus zu linearisieren.

Das entscheidende Werkzeug ist das Einteilen der hypergeometrischen Terme in Äquivalenzklassen. Zwei hypergeometrische Terme a_k und b_k werden ähnlich genannt, falls ihr Quotient $a_k/b_k \in K(k)$, also eine rationale Funktion in

k ist. Dadurch können wir die hypergeometrischen Terme in Äquivalenzklassen einteilen.

Wenn wir nun eine Linearkombination von hypergeometrischen Termen gegeben haben, könnnen wir diese so sortieren, dass wir eine Linearkombination von paarweise verschiedenen ähnlichen erhalten. Der von Petkovšek, Wilf und Zeilberger[7] entwickelte Algorithmus zeigt dann, ob für die einzelnen nicht ähnlichen hypergeometrischen Terme eine hypergeometrische Stammfunktion existiert. Wir sind mit diesem Algorithmus also in der Lage eine Linearkombination von hypergeomterischen Termen zu erzeugen, die aus vereinfachten Stammfunktionen und nicht vereinfachten hypergeometrischen Termen besteht. Die nicht vereinfachten hypergeometrischen Terme werden als Summen mit einem hypergeometrischen Summanden zurückgegeben.

Fassen wir die eingegebene Linearkombination als einen Term auf, besitzt dieser Term genau dann eine hypergeometrische Stammfunktion, wenn der Gosper-Algorithmus für jeden der paarweise nicht ähnlichen Terme erfolgreich ist und damit haben wir eine geschlossene Form erhalten.

2.3.1 Äquivalenzklassen von hypergeometrischen Termen

Wir werden uns mit zwei Fragen beschäftigen. Ist ein hypergeometrischer Term a_k gegeben, können wir die Summe $s_k = \sum_k a_k$ als Linearkombination von mehren, aber endlich vielen, hypergeometrischen Termen ausdrücken? Zum Beispiel können wir leicht zeigen, dass $k!$ nicht gospersummierbar ist, daher finden wir für die Summe $\sum_k k!$ keine hypergeometrische diskrete Stammfunktion. Aber eventuell können wir die Summe in eine gleichwertige umschreiben, die aus zwei, drei oder mehr, aber festen von k unabhängigen, hypergeometrischen Termen besteht, so dass diese eine hypergeometrische diskrete Stammfunktion hat.

Als zweites betrachten wir eine Linearkombination c_k von hypergeometrischen Termen. Wie können wir entscheiden, ob die Summe $s_k = \sum_k c_k$ in der gleichen Form als Linearkombination von hypergeometrischen Termen ausgedrückt werden kann, so dass wir eine Stammfunktion finden können? Dabei ist zu beachten, dass die Linearkombination gospersummierbar sein soll, aber die einzelnen Terme es nicht sein müssen. Zum Beispiel ist $a_{k+1} - a_k$ gospersummierbar, wobei a_k ein hypergeometrischer Term ist, der es nicht ist. Einen ähnlichen Fall haben wir schon in Beispiel 2.2.13 betrachtet, indem a_k ein nicht gospersummierbarer hypergeometrische Terme war, aber die Differenz $a_{k+20} - a_k$ war es.

Zum Schluss verfolgen wir in diesem Abschnitt das Ziel eine gegebene Linearkombination von hypergeometrischen Termen in eine Linearkombination umzuschreiben, die aus hypergeometrischen Termen besteht, die weitestgehend in ihre Stammfunktionen vereinfacht sind. Die nicht vereinfachten hypergeometrischen Stammfunktionen werden dann als Summen, mit einem hypergeometrischen Term als Summand, in die Linearkombination geschrieben.

Bei der Summation von hypergeometrischen Termen spielt das Einteilen in Äquivalenzklassen eine sehr wichtige Rolle. Das führt uns zu folgender Defini-

[7] Er ist in [PWZ97] nachzulesen.

tion:

Definition 2.3.1 *Zwei hypergeometrische Terme a_k und b_k sind ähnlich, falls ihr Quotient eine rationale Funktion in k ist. In diesem Fall schreiben wir $a_k \sim b_k$.*

Unter Ähnlichkeit verstehen wir die Äquivalenzrelation verschiedener hypergeometrischen Terme. Eine Äquivalenzklasse besteht zum Beispiel aus der Menge aller rationalen Funktionen.

Satz 2.3.2 *Wenn a_k ein nicht-konstanter hypergeometrischer Term ist, dann ist der hypergeometrische Term $a_{k+1} - a_k$ ähnlich zu a_k.*

Beweis: Es gilt $a_{k+1} - a_k = \big(\frac{a_{k+1}}{a_k} - 1\big)a_k$, dadurch erhalten wir ein rationales Vielfaches von a_k, dieses ist ungleich Null ist, da a_k keine Konstante ist. □

Satz 2.3.3 *Seien a_k und b_k hypergeometrische Terme mit $a_k + b_k \neq 0$, dann ist $a_k + b_k$ ein hypergeometrischer Term genau dann, wenn a_k ähnlich zu b_k ist.*

Beweis: Es seien $f = a_{k+1}/a_k$, $g = b_{k+1}/b_k$, $h = (a_{k+1} + b_{k+1})/(a_k + b_k)$ und $r = a_k/b_k$ (aus Notationsgründen haben wir die Brüche f, g, h und r genannt.). Für a_{k+1}/a_k und b_{k+1}/b_k gilt nach Definition, dass sie rationale Funktionen in k sind. Es gilt

$$h = \frac{a_{k+1} + b_{k+1}}{a_k + b_k} = \frac{fr + g}{r + 1}, \tag{2.26}$$

da

$$\begin{aligned} h = \frac{a_{k+1} + b_{k+1}}{a_k + b_k} &= \frac{\frac{a_{k+1}}{a_k} a_k + \frac{b_{k+1}}{b_k} b_k}{a_k + b_k} = \frac{\frac{a_{k+1}}{a_k}\frac{a_k}{b_k} b_k + \frac{b_{k+1}}{b_k} b_k}{a_k + b_k} \\ &= \frac{(\frac{a_{k+1}}{a_k}\frac{a_k}{b_k} + \frac{b_{k+1}}{b_k})b_k}{(\frac{a_k}{b_k} + 1)b_k} = \frac{fr + g}{r + 1}. \end{aligned}$$

Damit ist h genau dann rational, wenn a_k/b_k es ist, das heißt, wenn a_k und b_k ähnlich sind.

Umgekehrt folgt, wenn $h = a_{k+1}/a_k$ ist, aus (2.26), dass $a_{k+1}/a_k = b_{k+1}/b_k$ ist (Dies folgt aus $hr + h = fr + g$). Daraus folgt, dass a_k und b_k konstante Vielfache voneinander sind und a_k/b_k konstant ist. Wenn andererseits $h \neq a_{k+1}/a_k$ ist, dann gilt mit (2.26):

$$r = \frac{a_k}{b_k} = \frac{g - h}{h - f}.$$

In jedem Fall ist a_k/b_k rational, wenn h es ist. Damit ist der Satz gezeigt, da h genau dann hypergeometrisch ist, wenn a_k und b_k ähnlich sind (a_k/b_k ist rational). □

Satz 2.3.4 *Seien $a_k^{(1)},\ a_k^{(2)}, \ldots, a_k^{(n)}$ hypergeometrische Terme, so dass*

$$\sum_{i=1}^{n} a_k^{(i)} = 0 \tag{2.27}$$

gilt, dann folgt daraus, dass $a_k^{(i)} \sim a_k^{(j)}$ für mindestens ein Paar i und j, mit $1 \leq i < j \leq n$.

Beweis: Wir werden den Satz durch vollständige Induktion über n beweisen.

Ist $n = 2$, gilt dann $a_k^{(1)} \sim a_k^{(2)}$, da ein hypergeometrische Terme nach Definition ungleich Null ist[8].

Sei nun $n > 1$ und $a_{k+1}^{(i)}/a_k^{(i)}$ für $i = 1,\ 2, \ldots, n$. Aus Gleichung (2.27) folgt, dass $\sum_{i=1}^{n} a_{k+1}^{(i)} = 0$ ist, daher gilt weiter

$$\sum_{i=1}^{n} \frac{a_{k+1}^{(i)}}{a_k^{(i)}} a_k^{(i)} = 0. \tag{2.28}$$

Als nächstes multiplizieren wir die Gleichung (2.27) mit $a_{k+1}^{(n)}/a_k^{(n)}$ und subtrahieren (2.28) von der entstanden Gleichung. Dadurch erhalten wir

$$\sum_{i=1}^{n-1} \left(\frac{a_{k+1}^{(n)}}{a_k^{(n)}} - \frac{a_{k+1}^{(i)}}{a_k^{(i)}} \right) a_k^{(i)} = 0. \tag{2.29}$$

Ist nun $(a_{k+1}^{(n)}/a_k^{(n)} - a_{k+1}^{(i)}/a_k^{(i)}) = 0$ für ein i, dann ist $a_k^{(n)}/a_k^{(i)}$ eine Konstante und damit gilt $a_k^{(n)} \sim a_k^{(i)}$. Ansonst sind alle Terme auf der linken Seite von Gleichung (2.29) hypergeometrisch. Mit der Induktionsvoraussetzung gilt dann, dass es i und j gibt, für $1 \leq i < j \leq n-1$, so dass

$$(a_{k+1}^{(n)}/a_k^{(n)} - a_{k+1}^{(i)}/a_k^{(i)}) a_k^{(i)} \sim (a_{k+1}^{(n)}/a_k^{(n)} - a_{k+1}^{(j)}/a_k^{(j)}) a_k^{(j)}.$$

Aber dann gilt auch $a_k^{(i)} \sim a_k^{(j)}$. □

Satz 2.3.5 *Jede Summe, die aus einer endlichen Anzahl von hypergeometrischen Termen besteht, kann in eine Summe, die aus paarweise nicht ähnlichen hypergeometrischen Termen besteht, umgeschrieben werden.*

Beweis: Da eine Summe, die aus zwei ähnlichen hypergeometrischen Termen besteht, entweder Null oder ein hypergeometrischer Term ist, kann dies durch eine Klassenbildung von ähnlichen Termen erreicht werden. Nach Satz 2.3.2 ist jede dieser Klassen ein einzelner hypergeometrischer Term. □

Es stellt sich die Frage, wie wir entscheiden können, ob zwei hypergeometrische Terme ähnlich sind. Es reduziert sich darauf, ob ein gegebener hypergeometrischer Term rational ist oder nicht. In der Praxis wird dies von einer

[8] Die Differenz von zwei ähnlichen hypergeometrischen Termen ist gleich Null oder ein hypergeometrischer Term.

Vereinfachungsroutine[9] entschieden. Da alle Berechnungen von hypergeometrischen Termen in entsprechende Operationen der Repräsentanten von rationalen Funktion umgewandelt werden können, werden wir auch zeigen, wie die Rationalität von gegebenen hypergeometrischen Termen nur durch den Quotienten zwei ähnlicher hypergeometrischer Terme bewiesen werden kann.

Es seien zwei hypergeometrische Terme a_k und b_k gegeben, wie können wir entscheiden, ob sie ähnlich sind (bzw. in der gleichen Äquivalenzklasse liegen)? Wir müssen nur den Quotienten a_k/b_k beider Terme bilden und wenn dieser rational bezüglich der Variablen k ist ($a_k/b_k \in K(k)$), sind beide hypergeometrischen Terme ähnlich. Es ist zu erwähnen, dass ich in meiner Implementierung den Vereinfachungsalgorithmus `simpcomb` verwende. Dieser wandelt die gegebenen Funktionen in Gammaterme um und kann in einigen Fällen nicht entscheiden, ob $a_k/b_k \in K(k)$ gilt. In diesen Fällen findet er nicht heraus, dass a_k und b_k in einer Äquivalenzklasse liegen.

Mit diesen Kenntnissen sind wir in der Lage die Fragen vom Anfang des Abschnittes zu beantworten.

Die erste Frage ist, ob wir einen hypergeometrischen Term als Linearkombination schreiben können, die dann gospersummierbar ist. Das heißt, wir würden $\sum_{k=0}^{n-1} \alpha_k = \alpha_k^{(1)} + \alpha_k^{(2)} + \ldots \alpha_k^{(m)}$ erhalten, wobei $a_k^{(i)}$ hypergeometrische Terme sind. Damit der Term α_k gospersummierbar ist, muss der Gosper-Algorithmus für alle hypergeometrischen Terme $\alpha_k^{(i)}$ auf der rechten Seite erfolgreich sein. Mit Hilfe von Satz 2.3.5 können wir annehmen, dass die Terme auf der rechten Seite, die ungleich Null sind, paarweise nicht ähnlich sind. Durch Satz 2.3.4 wissen wir, dass höchstens ein Term auf der rechten Seite ungleich Null ist. Indem wir nämlich sukzessive die ähnlichen hypergeometrischen Terme zusammenfassen, bleiben am Ende nur zwei hypergeometrische Terme übrig. Daraus folgt, dass m höchstens 2 ist und wenn es 2 ist, dann muss einer der Terme $\alpha_k^{(1)}$, $\alpha_k^{(2)}$ eine Konstante sein. Also folgt die Antwort auf unsere erste Frage aus folgendem Satz.

Satz 2.3.6 *Wenn der Gosper-Algorithmus nicht erfolgreich ist, dann kann die Summe nicht als endliche Linearkombination von hypergeometrischen Termen ausgedrückt werden. In anderen Worten: Die Summe kann nicht in einer geschlossenen Form angegeben werden.*

Wir haben die geschlossene Form im Kapitel der Grundlagen so definiert, dass sie eine Linearkombination von hypergeometrischen Termen ist. Also wenn wir einen hypergeometrischen Term gegeben haben, der nicht gospersummierbar ist, können wir diesen nicht als Linearkombination von hypergeometrischen Termen schreiben, denn diese wären alle ähnlich zueinander.

Ein Beispiel für diesen Satz wäre die Summe $\sum_k k!$. Für den Term $k!$ finden wir keine hypergeometrische Stammfunktion, damit ist er nicht gospersummierbar. Daraus können wir schließen, dass wir $\sum_k k!$ nicht als Linearkombination von hypergeometrischen Termen darstellen können.

Die zweite Frage wird dann durch folgenden Algorithmus beantwortet:

[9] Der `simpcomb`-Algorithmus wird auf a_k/b_k angewendet.

Satz 2.3.7 (Algorithmus) *Bei der Eingabe von hypergeometrischen Termen* $a_k^{(1)}, \ldots, a_k^{(p)}$ *gibt der Algorithmus hypergeometrische diskrete Stammfunktionen* $s_k^{(1)}, \ldots, s_k^{(q)}$ *aus, so dass*

$$\sum_{i=1}^{q} s_k^{(i)} = \sum_{j=1}^{p} a_k^{(j)}.$$

Falls der Algorithmus erfolgreich ist, erhalten wir eine Linearkombination von hypergeometrischen Stammfunktionen.

1. *Eingabe:* $\sum_{j=1}^{p} a_k^{(j)}$.
2. *Schreibe* $\sum_{j=1}^{p} a_k^{(j)} = \sum_{j=1}^{q} u_k^{(j)}$ *um, so dass die* $u_k^{(j)}$ *paarweise nicht ähnlich sind.*
3. *Für* $j = 1, 2, \ldots, q$*:*
   Berechne mit Hilfe des Gosper-Algorithmus die hypergeometrischen diskreten Stammfunktionen $s_k^{(j)}$ *für jeden hypergeometrischen Term* $u_k^{(j)}$.
4. *Falls der Gosper-Algorithmus nicht erfolgreich ist, setze* $s_k^{(j)} = \sum_{n=0}^{k-1} u_n^j$.
5. *Ausgabe:* $\sum_{j=1}^{q} s_k^j$.

Der linearisierte Gosper-Algorithmus ist mächtiger als in diesem Algorithmus beschrieben. Er bietet uns die Möglichkeit eine Linearkombination von hypergeometrischen Termen als Linearkombination von paarweise nicht ähnlichen hypergeometrischen Termen zuschreiben. Auf jeden einzelnen hypergeometrischen Term können wir dann denn Gosper-Algorithmus anwenden. Falls der Term gospersummierbar ist, geben wir ihn als hypergeometrische diskrete Stammfunktion aus, falls nicht, als nicht vereinfachte Stammfunktion. Eine nicht vereinfachte Stammfunktion ist eine Summe mit einem hypergeometrischen Term als Summand. Dadurch erhalten wir eine Linearkombination, die sich aus hypergeometrischen Stammfunktionen und Summen von hypergeometrischen Termen zusammensetzt. Als Resultat erhalten wir also eine Linearkombination von hypergeometrischen Termen, die so weit es zu realisieren ist, in vereinfachter Form (hypergeometrische Stammfunktion) vorliegen.

Der Befehl `lingosper(a,k)` realisiert in MuPAD den oben beschriebenen Algorithmus. Dieser Befehl funktioniert genau wie der Befehl `gosper`, auch hier ist `a` der hypergeometrische Eingabeterm und `k` ist der Summationsindex. Der Unterschied ist nur, dass dieser bei einer Linearkombination von hypergeometrischen Termen sie in Äquivalenzklassen einteilt und dann den Gosper-Algorithmus auf die paarweise verschiedenen hypergeometrischen Terme anwendet. Dadurch ist er mächtiger als der Befehl `gosper`. Der Befehl gibt bei Misserfolg aber nicht `FAIL` aus, sondern die nicht vereinfachte Stammfunktion, in der Form $\sum_k a_k$, wobei a_k ein hypergeometrischer Term ist. Bei dieser Ausgabe kann eine Linearkombination von hypergeometrischen diskreten Stammfunktionen und nicht evaluierten Summen mit hypergeometrischen Termen als Summanden entstehen.

Beispiel 2.3.8
Wir haben bereits erwähnt, dass $a_k = k!$ nicht gospersummierbar ist und wollen dies noch mal in einem Beispiel vorführen. Dieses tun wir mit Hilfe von MuPAD und dem Befehl `lingosper` und setzen `setunserinfo(lingosper,1)`, um Fehlermeldungen auszugeben:

MuPAD

```
>> setuserinfo(lingosper,1):
>> lingosper(k!,k);
```

Output

```
Info: Der Gosper-Algorithmus findet keine hypergeometrische
Stammfunktion s[k].
sum(k!, k)
```

Wir erhalten eine Fehlermeldung und die nicht vereinfachte hypergeometrische Stammfunktion, die aus einer Summe und dem hypergeometrischen Term $a_k = k!$ als Summand besteht. Damit ist a_k nicht gospersummierbar und ein Beispiel für Satz 2.3.6, denn $k!$ kann nicht als Linearkombination von hypergeometrischen Termen ausgedrückt werden.

Wir kommen nun zu dem Beispiel, welches am Anfang des Abschnitts betrachtet wurde.

Beispiel 2.3.9
Wir betrachten $a_k = 2^k + 1$. Beide Terme der Linearkombination sind gospersummierbar, aber die Summe beider nicht. Dieses betrachten wir zuerst mit Hilfe des Befehls `gosper`. Auch hier wollen wir mit `setuserinfo(gosper,1)` die Fehlermeldungen ausgeben.

MuPAD

```
>> setuserinfo(gosper,1):
>> gosper(2^k+1,k);
```

Output

```
Error: Der Algorithmus konnte nicht angewendet werden, da der
Eingabe Term nicht hypergeometrisch ist! [gosper]
```

Durch die `Error` Meldung sehen wir, dass der Eingabeterm nicht hypergeometrisch ist und damit ist die Linearkombination als solche nicht gospersummierbar. Als nächstes verwenden wir den Befehl `lingosper`. Wie wir gesehen haben, können wir auch hier Informationen über die Zwischenergebnisse durch das Aufrufen von `setuserinfo` erhalten[10]. Zu beachten ist nur, dass hier die Zwischenergebnisse aller paarweise verschiedenen hypergeometrischen Terme ausgegeben werden, falls sie in ihre Äquivalenzklassen unterteilt wurden.

MuPAD

[10] Um alle Zwischenergebnisse zu erhalten, müssen wir `setuserinfo(gradschranke,1)`, `setuserinfo(umschreibenpolynome,1)`,`setuserinfo(gosper,6)`,`setuserinfo(lingosper,1)` aufrufen.

```
>> setuserinfo(lingosper,1):
>> s:=lingosper(2^k+1,k,1);
```

Output

```
k + 2^k - 1
```

Auch der Gospertest ist hier erfolgreich:

MuPAD

```
>> gospertest(2^k+1,s,k);
```

Output

```
TRUE
```

So haben wir gesehen, dass a_k, obwohl es als Linearkombination nicht gospersummierbar ist, trotzdem mit Hilfe der linearisierten Form des Gosper-Algorithmus in eine geschlossene Form gebracht werden kann. Dies ist nur möglich, weil die hypergeometrischen Terme der Linearkombination in ihre Äquivalenzklassen eingeteilt werden und der Gosper-Algorithmus auf die einzelnen Klassen angewendet wird.

Nachdem wir nun ein Beispiel für den linearisierten Gosper-Algorithmus gesehen haben, wollen wir uns nun noch einmal genauer mit der Implementierung beschäftigen. Es wird eine Linearkombination von hypergeometrischen Termen eingegeben und diese Terme werden in ihre Äquivalenzklassen eingeteilt. Diese Einteilung erzielt der Algorithmus indem er alle Quotienten a_k/b_k prüft, wobei a_k und b_k verschiedene Terme der eingegebenen Linearkombination sind. Die Terme, die in einer Äquivalenzklasse liegen, werden dann als Linearkombination zusammengefasst.

Unter diesen Äquivalenzklassen existiert eine Klasse, die wir eventuell noch weiter vereinfachen könnten und das ist die Klasse der rationalen Funktionen bezüglich der Summationsvariable k. Wir wissen aus Beispiel 2.2.9, dass nicht jede rationale Funktion gospersummierbar ist und zusätzlich wissen wir aus Beispiel 2.2.8, dass jedes Polynom eine hypergeometrische diskrete Stammfunktion besitzt. Daher können wir versuchen den polynomialen Anteil abzuspalten, um alle möglichen hypergeometrischen Terme zu erhalten, die gospersummierbar sind. Dies tun wir mit dem Ziel, dass der Zähler des rationalen Teils minimal sein soll. Für den implementierten Algorithmus heißt das, dass die hypergeometrischen Terme in ihre Äquivalenzklassen eingeteilt werden und bei der Linearkombination der rationalen Funktionen zusätzlich eine Polynomdivision durchgeführt wird, um den polynomialen Anteil abzuspalten. Dadurch erhalten wir die größte mögliche Anzahl an vereinfachten hypergeometrischen Stammfunktionen und die geringste, der nicht vereinfachten, in unserer Linearkombination.

Ein einfaches Beispiel hierfür ist die Linearkombination $a_k = 1/k + 5$. Die beiden Terme $1/k$ und 5 liegen in der gleichen Äquivalenzklasse (der der rationalen Funktionen) und werden daher als $(5k+1)/k$ zusammengefasst. Dieser Term ist nicht gospersummierbar. Aber wenn wir eine Polynomdivision durchführen, erhalten wir wieder den polynomialen Anteil 5 und den rationalen $1/k$. Dabei

ist die 5 gospersummierbar, wir erhalten also als Ausgabe des Algorithmus

$$5k - 5 + \sum_k \frac{1}{k} \quad \text{statt} \quad \sum_k \frac{(5k+1)}{k}.$$

Keines der beiden Resultate ist falsch. Aber unter dem Aspekt, dass wir die „einfachste Form“ erzielen wollen, das heißt, die Linearkombination soll aus möglichst vielen vereinfachten hypergeometrischen Stammfunktionen bestehen, ist es besser die Polynomdivision durchzuführen.

Damit erhalten wir durch den angepassten linearisierten Gosper-Algorithmus die Linearkombination, die aus vereinfachten hypergeometrischen und nicht vereinfachten Stammfunktionen besteht. Die nicht angepasste Form des linearisierten Algorithmus erzeugt eine Linearkombination, die nur aus hypergeometrischen Stammfunktionen besteht und erzielt dadurch eine geschlossene Form.

Zum Abschluss des Kapitels werden wir uns nun noch ein Beispiel für die Linearisierung des Gosper-Algorithmus ansehen.

Beispiel 2.3.10
Gegeben ist der Eingabeterm $a_k = k^2 + k + 1/k + 1 + 2^k + 2^{k+3} + \binom{n}{k}$. Wir wissen aus den vorangegangenen Beispielen, dass alle Terme hypergeometrisch sind. Die Terme werden in ihre Äuivalenzklassen eingeteilt und als Linearkombination in der jeweiligen Klasse zusammengefasst, dann ergibt sich: $a_k = 2^k + 2^{k+3}$, $b_k = \binom{n}{k}$ und $c_k = (k^3 + k^2 + k + 1)/k$. Wir wissen aus Beispiel 2.2.10, dass b_k nicht gospersummierbar ist. Die Terme 2^k und 2^{k+3} liegen in einer Äquivalenzklasse, da ihr Quotient eine rationale Funktion in k ist (in diesem Fall sogar konstant). Dieser Term ist gospersummierbar. Die Linearkombination der Äquivalenzklasse der rationalen Funktion sieht wie folgt aus: $c_k = (k^3 + k^2 + k + 1)/k$. Diese hat in dieser Form keine vereinfachte hypergeometrische Stammfunktion. Daher spalten wir durch eine Polynomdivision den polynomialen Anteil $k^2 + k + 1$ ab, von dem wir wissen, dass er gospersummierbar ist und haben dann nur noch den rationalen Teil $1/k$ in der Äquivalenzklasse der rationalen Funktionen stehen. Auf diese einzelnen Teile wenden wir dann jeweils den Gosper-Algorithmus an.

Nun sehen wir uns an, welches Ergebnis wir mit dem in MuPAD implementierten Befehl `lingosper` erhalten.

MuPAD

```
>> setuserinfo(lingosper,1):
>> lingosper(binomial(n,k)+k^2+k+1/k+1+2^k+2^(k+3),k);
```

Output

```
Info: Mindestens einer der paarweise verschiedenen
hypergeometrischen Terme ist nicht gospersummierbar!
sum(binomial(n, k), k) + sum(1/k, k) + 9*2^k + k^3/3 - 1 +(2*k)/3
```

Dadurch ergibt sich folgende Lösung:

$$\sum_k \left(k^2+k+1/k+1+2^k+2^{k+3}+\binom{n}{k}\right) = \sum_k \binom{n}{k} + \sum_k \left(\frac{1}{k}\right) + 9\cdot 2^k + \frac{k^3}{3} + \frac{2k}{3} - 1.$$

Wir sehen, dass eine Warnung ausgegeben wird, die besagt, dass mindestens ein Term nicht gospersummierbar ist. Als Ergebnis wird also eine Linearkombination von vereinfachten hypergeometrischen Stammfunktionen und nicht vereinfachten Stammfunktionen ausgegeben. Diese Warnung besagt, dass wir keine geschlossene Form gefunden haben, da mindestens einer der Terme der Linearkombination eine nicht vereinfachte Stammfunktion ist. Dieser wird in Form einer Summe, mit einem hypergeometrischen Term als Summand, dargestellt. Falls wir keine Warnung erhalten, ergibt sich eine Linearkombination von hypergeometrischen Stammfunktionen und damit eine geschlossene Form.

Kapitel 3

Zeilberger-Algorithmus

3.1 Bestimmte Summation

Im vorigen Kapitel haben wir uns mit dem Gosper-Algorithmus beschäftigt. Dieser beantwortet die Frage, ob ein gegebener hypergeometrischer Term unbestimmt summierbar ist oder nicht. Falls $F(k)$ ein gegebener Term ist, wollen wir wissen, ob $F(k) = G(k+1) - G(k)$ für eine diskrete Stammfunktion $G(k)$ erfüllt ist.

In diesem Abschnitt werden wir einen Algorithmus betrachten, der sich mit einer ähnlichen Fragestellung beschäftigt, und zwar mit der bestimmten Summation. Dieser Algorithmus heißt Zeilberger-Algorithmus oder kreative Teleskopsummierung[1]. Der Algorithmus kann nicht nur hypergeometrische Identitäten überprüfen[2], sondern, wie der Fasenmyer-Algorithmus[3], sucht er eine holonome Rekursionsgleichung für eine Summe mit einem hypergeometrischen Term als Summanden. Dabei ist anzumerken, dass der Algorithmus von Zeilberger viel effizienter ist als der von Fasenmyer. Das Kapitel basiert auf den Büchern [Koe97] und [PWZ97].

Wir interessieren uns für eine Summe der Form

$$s_n = \sum_{k \in \mathbb{Z}} F(n,k), \tag{3.1}$$

wobei der Summand von der Summationsvariablen $k \in \mathbb{Z}$ und einer weiteren diskreten Variablen $n \in \mathbb{Z}$ abhängt. Weiter setzen wir voraus, dass die Summe in Wirklichkeit nur endlich viele Summanden besitzt, dass also für jedes $n \in \mathbb{Z}$ (bzw. $n \in \mathbb{N}_{\geq 0}$) nur endlich viele k mit $F(n,k) \neq 0$ existieren. Das heißt, dass der Summand $F(n,k)$ einen endlichen Träger hat. Deswegen können wir annehmen, dass die Summationsgrenzen mit dem Index k die Menge aller ganzen Zahlen sind (das heißt: $k = -\infty \ldots \infty$). Später werden wir sehen, dass diese Annahme problemlos weggelassen werden kann. Wir werden dann auch feste Summationsgrenzen verwenden, aber für den Moment betrachten wir eine Summe über alle ganzen Zahlen.

[1] [Zei90] und [Zei91].
[2] Dies ist mit Hilfe der WZ-Methode möglich [WZ92].
[3] Nachzulesen in [Koe06].

Unser Ziel ist es, s_n als geschlossene Form darzustellen. Solch eine Form liegt vor, falls s_n eine Linearkombination aus einem oder mehreren hypergeometrischen Termen ist. Die Linearkombination erhalten wir aus einer holonomen Rekursionsgleichung mit geeigneten Anfangswerten.

Zunächst definieren wir einen „zulässigen“ hypergeometrischen Term. Dieser hat einen endlichen Träger und die Form $F(n,k) = P(n,k)\frac{Q(n,k)}{R(n,k)}w^k z^k$, wobei $P(n,k)$ ein Polynom (der polynomielle Teil) ist und $Q(n,k),\ R(n,k)$ sind Produkte von Gammafunktionen mit ganzzahligen Argumenten (der faktorielle Teil), nach [Koe97].

Definition 3.1.1 *Ein zulässiger hypergeometrischer Term $F(n,k)$ hat die Form*

$$F(n,k) = P(n,k)\frac{\Gamma(\alpha_1 k + \beta_1 n + c_1)\cdots\Gamma(\alpha_p k + \beta_p n + c_p)}{\Gamma(\gamma_1 k + \delta_1 n + d_1)\cdots\Gamma(\gamma_q k + \delta_q n + d_q)}w^n z^k, \tag{3.2}$$

wobei $P \in K[n,k]$, $\alpha_l,\ \beta_l,\ \gamma_l, \delta_l \in \mathbb{Z}$ und $c_l,\ d_l,\ w,\ z \in \mathbb{Q}$ sind.

Im folgenden Satz ([Koe06]) werden wir sehen, dass s_n dann holonom ist, falls $F(n,k)$ eine k-freie Rekursion erfüllt.

Satz 3.1.2 *Sei $F(n,k)$ ein zulässiger hypergeometrischer Term, dann existiert eine Rekursionsgleichung der Form*

$$\sum_{i=0}^{J}\sum_{j=0}^{J} a_{ij}F(n+j,k+i) = 0 \quad (a_{ij} \in K[n],\ I,\ J \in \mathbb{N}), \tag{3.3}$$

deren Polynomkoeffizienten a_{ij} den Summationsindex k nicht enthalten. Wir nennen eine derartige Rekursion k-frei. Dann ist s_n aus (3.1) holonom (über K).

Beweis: Eine Indexverschiebung zeigt, dass für alle $i,\ j \in \mathbb{Z}$

$$\sum_{k=-\infty}^{\infty} F(n+j,k+i) = s_{n+j}$$

gilt. Indem wir die gegebene Rekursion von $F(n,k)$ über alle ganzen Zahlen summieren, erhalten wir offenbar eine holonome Rekursion über K für s_n. □

Weiter ist $F(n,k)$ ein hypergeometrischer Term bezüglich beider Variablen, mit anderen Worten heißt das, $F(n+1,k)/F(n,k)$ und $F(n,k+1)/F(n,k)$ sind beides rationale Funktionen in n und k:

$$\frac{F(n+1,k)}{F(n,k)} \in K(n,k) \quad \text{und} \quad \frac{F(n,k+1)}{F(n,k)} \in K(n,k).$$

Unser Ziel ist es eine Rekursionsgleichung für die Summe s_n zu finden und dieses erreichen wir, indem wir zuerst eine Rekursionsgleichung für den Summanden $F(n,k)$ finden. Es ist zu beachten, wie sich die Fragestellung von Kapitel 2 unterscheidet. Falls wir eine diskrete Stammfunktion $G(n,k)$ von $F(n,k)$ finden, erhalten wir für die Summe s_n die Gleichung $F(n,k) =$

$G(n,k+1)-G(n,k)$, dann können wir leicht eine Rekursion für s_n finden. Aber in diesem Fall können wir die Summe sogar als Funktion mit einer Variable als obere Grenze ausdrücken. Für den allgemeinen Fall ist dies zu viel erwartet. Es gibt viele Summanden, die nicht unbestimmt summierbar sind, der Gosper-Algorithmus findet also keine Lösung für die Summe. Aber die Summe s_n kann trotzdem in eine geschlossene Form umgeformt werden, wenn der Index k über alle ganzen Zahlen läuft.

Zum Beispiel ist der Binomialkoeffizient $\binom{n}{k}$ für ein festes n nicht über k gospersummierbar. Aber für die uneingeschränkte Summe gilt die einfache Form $\sum_k \binom{n}{k} = 2^n$ und zwar, obwohl die unbestimmte Summe $\sum_{k=0}^{K_0} \binom{n}{k}$ nicht als ein einfacher hypergeometrischer Term in K_0 (und n) ausgedrückt werden kann. Diese Situation verhält sich völlig analog zum Problem der bestimmten und unbestimmten Integration. Hier gibt es das Beispiel der Funktion e^{-t^2}. Diese ist nicht unbestimmt integrierbar, aber auf der anderen Seite gibt es eine Lösungsformel für das bestimmte Integral $\int_{-\infty}^{\infty} e^{-t^2} = \sqrt{\pi}$.

Zunächst zeigen wir, dass die Anwendung der unbestimmten Summation durch den Gosper-Algorithmus nicht zum Ziel führt ([Koe06]). Für ein allgemeines $F(n,k)$ ist es nicht möglich einen hypergeometrischen Term s_n zu finden der ungleich 0 ist.

Satz 3.1.3 *Sei $F(n,k)$ ein hypergeometrischer Term bezüglich n und k, der gospersummierbar bezüglich k ist und eine hypergeometrische diskrete Stammfunktion $s_k = G(n,k)$ besitzt, welche für alle $k \in \mathbb{Z}$ endlich sei. Zusätzlich sei $F(n,k)$ wohldefiniert für alle $n \in \mathbb{N}_{\geq 0}$ und habe einen endlichen Träger. Dann gilt*

$$\sum_{k=-\infty}^{\infty} F(n,k) = 0 \tag{3.4}$$

für alle bis auf höchstens endlich viele $n \in \mathbb{N}_{\geq 0}$. Genauer: Ist $G(n,k) = R(n,k)F(n,k)$ eine hypergeometrische diskrete Stammfunktion, dann gilt (3.4) für alle $n \in \mathbb{N}_{\geq 0}$, für welche der Nenner der rationalen Funktion $R(n,k) \in K(n,k)$ nicht identisch Null ist.

Beweis: Ist $F(n,k)$ gospersummierbar bezüglich k, dann gibt es eine hypergeometrische diskrete Stammfunktion $G(n,k)$. Es gilt dann:

$$F(n,k) = G(n,k+1) - G(n,k).$$

Wenn wir nun die Gleichung über alle $k \in \mathbb{Z}$ aufsummieren, erhalten wir

$$\sum_{k=-\infty}^{\infty} F(n,k) = \sum_{k=-\infty}^{\infty} \Big(G(n,k+1) - G(n,k) \Big) = 0.$$

Da die rechte Summe eine Teleskopsumme ist, erhalten wir Null. Nach Voraussetzung hat $F(n,k)$ einen endlichen Träger und wir erhalten eine endliche Summe

$$\sum_{k=-\infty}^{\infty} F(n,k) = \sum_{k=b}^{a} F(n,k),$$

wobei a und b die natürlichen Grenzen von $F(n,k)$ sind. Es kann passieren, dass $G(n,k)$ an bestimmten Stellen $n \in \mathbb{N}_{\geq 0}$ Singularitäten besitzt. Da $G(n,k) = R(n,k)F(n,k)$ ein rationales Vielfaches von $F(n,k)$ ist, sind die Singularitäten von $G(n,k)$ die Pole von $R(n,k)$. □

Damit haben wir gezeigt, dass die Anwendung des Gosper-Algorithmus auf den Term $F(n,k)$ nie eine hypergeometrische diskrete Stammfunktion $G(n,k)$ finden kann, so dass $\sum_{k=-\infty}^{\infty}(G(n,k+1) - G(n,k))$ ungleich Null ist.

Beispiel 3.1.4
Gegeben ist $a_k = (-1)^k\binom{n}{k}$. Zur Bestimmung der hypergeometrischen diskreten Stammfunktion nutzen wir den Gosper-Algorithmus und wenden den Befehl `gosper(a,k)`, der in MuPAD implementiert ist, an.

MuPAD

```
>> s:=gosper((-1)^k*binomial(n,k),k);
```

Output

```
-(-1)^k*k*binomial(n, k)/n
```

Damit haben wir gezeigt, dass a_k gospersummierbar ist. Wir können jetzt jede beliebige Summe $\sum_{k=b}^{a} a_k = s_{b+1} - s_a$ ausrechnen[4]. Wir betrachten die natürlichen Grenzen und es gilt: $\sum_{k=-\infty}^{\infty} a_k = \sum_{k=0}^{n} a_k$. Dafür ergibt sich dann:

MuPAD

```
>> simplify(subs(s,k=n+1)-subs(s,k=0));
```

Output

```
0
```

Die hypergeometrische diskrete Stammfunktion, die wir erhalten, ist die rationale Funktion $R(n,k) = -k/n$, die abhänging von der Summationsvariablen k und n ist, multipliziert mit dem hypergeometrischen Eingabeterm $F(n,k) = (-1)^k\binom{n}{k}$. Damit ist die Stammfunktion nur für $n \in \mathbb{N}$, aber nicht für $n = 0$ definiert. Wir haben dieses Resultat durch Einsetzen der Grenzen erhalten. Nach Satz 3.1.3 hätten wir für $F(n,k) = a_k$ direkt

$$\sum_{k=-\infty}^{\infty}(-1)^k\binom{n}{k} = \sum_{k=0}^{n}(-1)^k\binom{n}{k} = 0$$

folgern können.

Als nächstes betrachten wir noch eine Bemerkung zu Satz 3.1.3:

Bemerkung 3.1.5 *Sei $F(n,k)$ ein hypergeometrischer Term bezüglich k und n, welcher für alle $n \in \mathbb{N}_{\geq 0}$ wohldefiniert ist und einen endlichen Träger*

[4] Auf die bestimmte Summation mit dem Gosper-Algorithmus werden in Kapitel 5 noch genauer eingehen.

hat. Falls $\sum_{k=-\infty}^{\infty} F(n,k)$ *durch einen von Null verschiedenen hypergeometrischen Term dargestellt werden kann, dann ist* $F(n,k)$ *nicht gospersummierbar bezüglich* k*.*

Beispiel 3.1.6
Wegen

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n,$$

$$\sum_{k=0}^{n} k\binom{n}{k} = n2^{n-1}$$

sowie

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$$

sind $\binom{n}{k}$, $k\binom{n}{k}$ und $\binom{n}{k}^2$ nicht gospersummierbar bezüglich k.

Nun kommen wir zu unserem Summationsproblem (3.1) zurück. Wir haben in Satz 3.1.3 gesehen, dass wir im Allgemeinen nicht in der Lage sein werden, einen Term $G(n,k)$ zu finden, für den $F(n,k) = G(n,k+1) - G(n,k)$ gilt. Dennoch werden wir oft ein $G(n,k)$ finden, so dass

$$F(n+1,k) - F(n,k) = G(n,k+1) - G(n,k) \tag{3.5}$$

gilt. Wenn wir dieses finden, können wir nicht die unbestimmte Summe für F bestimmen, aber wir können beweisen, dass s_n konstant ist. Diese Methode nach Wilf und Zeilberger wird WZ-Methode genannt ([WZ90]), auf diese wollen wir aber nicht weiter eingehen.

Wir können nicht erwarten, dass wir eine Gleichung der Form (3.5) erzielen, aber mit einem allgemeineren Differenzenoperator in n auf der linken Seite von (3.5) können wir den allgemeineren Fall untersuchen. Hierzu definieren wir uns folgende Operatoren:

Definition 3.1.7 *Sei* N *(bzw.* K*) ein Vorwärtsoperator von* n *(bzw.* k*), dann gilt* $Ng(n,k) = g(n+1,k)$ *und* $Kg(n,k) = g(n,k+1)$*.*

Mit diesen Operatoren können wir die Gleichung (3.5) wie folgt ausdrücken: $(N-1)F = (K-1)G$. Wir werden im Folgenden zeigen, dass wir immer in der Lage sind, einen Differenzenoperator in der Form $P(n,N) = \sigma_0(n) + \sigma_1(n)N + \sigma_2(n)N^2 + \ldots + \sigma_J(n)N^J$ zu finden, so dass

$$P(n,N)F(n,k) = (K-1)G(n,k)$$

gilt. Die Koeffizienten $\sigma_i(n)$ $(i = 0,\ldots,J)$ in der Gleichung sind Polynome in n und $G(n,k)/F(n,k)$ ist eine rationale Funktion in n und k, so dass gilt:

$$\sum_{j=0}^{J} \sigma_j(n)F(n+j,k) = G(n,k+1) - G(n,k). \tag{3.6}$$

Der Zeilberger-Algorithmus findet für einen gegebenen Summanden $F(n,k)$ die Rekursionsgleichung (3.6).

Wir versuchen die Summe $s_n = \sum_{k=-\infty}^{\infty} F(n,k)$ zu bestimmen. Auf den folgenden Seiten stellen wir den Zeilberger-Algorithmus vor, dieser findet eine Rekursionsgleichung der Form (3.6) für den Summanden $F(n,k)$ und eine rationale Funktion $R(n,k)$, für die gilt: $G(n,k) = R(n,k)F(n,k)$. Wie hilft dieses bei der Suche nach unserer Summe s_n? Da die Koeffizienten auf der linken Seite von (3.6) unabhängig von k sind, können wir k über alle ganzen Zahlen summieren und da $G(n,k)$ ein endlichen Träger über k für alle n hat, gilt mit dem Argument der Teleskopsumme

$$\sum_{k=-\infty}^{\infty} \sum_{j=0}^{J} \sigma_j(n) F(n+j,k) = \sum_{j=0}^{J} \sigma_j(n) s(n+j) = 0. \tag{3.7}$$

Nun bestehen mehrere Möglichkeiten.

Es kann passieren, dass $J = 1$ ist, das heißt für uns, dass die Gleichung (3.7) eine Rekursion $\sigma_0(n)s_n + \sigma_1(n)s_{n+1} = 0$ erster Ordnung mit polynomialen Koeffizienten ist. Dann haben wir unser Ziel erreicht, denn $s_{n+1}/s_n = -\sigma_0(n)/\sigma_1(n)$ ist eine rationale Funktion in n. Somit ist die Summe s_n mit Sicherheit ein hypergeometrischer Term. Mit Algorithmus 1.1.4 erhalten wir die Form:

$$\frac{s_{n+1}}{s_n} = \frac{(-\sigma_0(j))}{(\sigma_1(j))} = \frac{u_n}{v_n}. \tag{3.8}$$

Der Algorithmus 1.1.6 liefert uns dann den hypergeometrischen Term s_n und mit Hilfe des Anfangswertes s_0 erhalten wir eine geschlossene Form für s_n.

Des Weiteren ist es möglich, dass in unserer Gleichung (3.7) $J > 1$ ist und wir Glück haben, dass alle Koeffizienten $\sigma_j(n)$ $(j = 0, \dots J)$ Konstanten sind. Dann ist unsere Summe leicht zu bestimmen, da wir nur eine lineare Rekursionsgleichung mit konstanten Koeffizienten lösen müssen. Damit sind wir wieder in der Lage, eine explizite „einfache Form“ für die Summe s_n anzugeben.

Wenn keiner der oberen Fälle zutrifft, erhalten wir eine Rekursionsgleichung für die Summe s_n mit polynomialen Koeffizienten und wir wissen nicht, wie und ob wir diese Rekursionsgleichung lösen können. Auch für diesen komplizierteren Fall finden wir mit Hilfe des Algorithmus von Petkovšek, den wir im nächsten Kapitel erläutern, eine Antwort auf unsere Frage. Falls eine Linearkombination aus einer festen Anzahl von hypergeometrischen Termen die Lösung unserer Rekursion (3.7) ist, findet dieser Algorithmus sie. Andernfalls gibt er zurück, dass keine solche Lösung existiert.

Wir können also zu unserer Anfangsfrage zurückkommen: Gegeben ist eine Summe $s_n = \sum_{k=-\infty}^{\infty} F(n,k)$, wobei F ein hypergeometrischer Term in beiden Variablen ist und wir wollen wissen, ob wir für diese Summe eine geschlossene Form finden können. Diese Frage können wir nun komplett algorithmisch beantworten und es ist bewiesen, dass der Algorithmus entweder diese einfache Darstellung findet oder dass für die gegebene Summe keine geschlossene Form existiert.

Das Problem eine geschlossene Form für eine bestimmte hypergeometrische Summe zu finden, ist vergleichbar mit dem Problem der unbestimmten

Integration im Sinn von Liouville und der unbestimmten Summation von hypergeometrischen Termen, wie wir sie in Kapitel 2 kennen gelernt haben. Der Zeilberger-Algorithmus zusammen mit dem Petkovšek-Algorithmus gehören zu der Klasse der klassischen mathematischen Probleme, die komplett algorithmisch lösbar sind.

3.2 Existenz der Rekursionsgleichung

Die Existenz der Rekursion für den Summanden $F(n,k)$ aus der Gleichung (3.6) folgt daraus, dass $F(n,k)$ ein zulässiger hypergeometrischer Term nach Definition 3.1.1 ist. Er ist hypergeometrisch bezüglich beider Variablen und es existiert eine zwei-variablige Rekursion (S. 65 [PWZ97]). Der Abschnitt basiert auf Kapitel 6 aus [PWZ97].

Wir können zuerst eine zwei-variablige Rekursion finden und sie dann wie im Beweis zum folgenden Satz 3.2.1 in eine Rekursion in Teleskopform (3.6) umformen. Aber Zeilberger hat einen viel effizienteren Weg gefunden: Zur Implementierung verwendet er eine abgeänderte Form des Gosper-Algorithmus.

Satz 3.2.1 *Sei $F(n,k)$ ein zulässiger hypergeometrischer Term, wie in 3.1.1 definiert, dann erfüllt $F(n,k)$ eine nicht triviale Rekursion der Form*

$$\sum_{j=0}^{J} \sigma_j(n) F(n+j,k) = G(n,k+1) - G(n,k),$$

in der $G(n,k)/F(n,k)$ eine rationale Funktion in n und k ist.

Beweis: Wir folgen dem Beweis von Wilf und Zeilberger aus [WZ92] und beginnen mit einer zwei-variabligen Rekursion

$$\sum_{i=0}^{I} \sum_{j=0}^{J} \sigma_{i,j}(n) F(n+j,k+i) = 0 \quad \text{für } I,\ J \in \mathbb{N}_0. \tag{3.9}$$

Wir wissen, dass eine solche Rekursion existiert. Mit den Shiftoperatoren K und N aus Definition 3.1.7 gilt dann: $Ku(k) = u(k+1)$ und $Nv(n) = v(n+1)$. Dann kann (3.9) mit Hilfe der Operatoren in der Form $P(N,n,K)F(n,k) = 0$ geschrieben werden. Angenommen wir nehmen das Polynom $P(u,v,w)$ und entwickeln es in eine Potenzreihe in w um den Punkt $w = 1$. Dann erhalten wir

$$P(u,v,w) = P(u,v,1) + (1-w)Q(u,v,w),$$

wobei Q ein Polynom ist. Wenden wir dieses auf (3.9) an, ergibt sich

$$0 = P(N,n,K)F(n,k) = (P(N,n,1) + (1-K)Q(N,n,K))F(n,k),$$

daraus folgt

$$P(N,n,1)F(n,k) = (K-1)Q(N,n,K)F(n,k). \tag{3.10}$$

Die linke Seite der Rekursion variiert nur in n und die rechte Seite ist $G(n, k+1) - G(n, k)$, falls wir $G(n, k) = Q(N, n, k)F(n, k)$ setzen. Dann ist G ein rationales Vielfaches von F. Jeder Shiftoperator, der auf einen hypergeometrischen Term angewendet wird, ist nur eine Multiplikation mit einer rationalen Funktion, das heißt, das Resultat ist wieder hypergeometrisch.

Wir fordern, dass die Rekursion (3.10) nicht trivial ist. Der folgende Teil des Beweises beruht auf Graham, Knuth und Patashnik [GKP89].

Wir wissen, dass es nicht triviale Operatoren $P(N, n, K)$ gibt, die nur abhängig von N, n und K sind, welche $F(n, k)$ annuliert. Sei nun $P = P(N, n, K)$ einer von diesen, der den niedrigsten Grad in K hat. Entwickeln wir bei $K = 1$, erhalten wir

$$P(N, n, K) = P(N, n, 1) - (K - 1)Q(N, n, K),$$

wodurch der Operator Q definiert wird.

Angenommen $P(N, n, 1) = 0$, dann ist $(K - 1)G(n, k) = 0$, das heißt, G ist unabhängig von k. Daher ist G nur ein hypergeometrischer Term in der Variablen n, das bedeutet, dass G eine Rekursion erster Ordnung mit polynomialen Koeffizienten in n erfüllt. Somit existiert ein Operator $H(N, n)$ erster Ordnung, so dass $H(N, n)G(n, k) = 0$ gilt.

Falls $Q = 0$ gilt, ist $P(N, n, K) = P(N, n, 1)$ ein k-freier Operator ($\neq 0$), der unabhängig von K und k ist und $F(n, k)$ annuliert. Ist $Q \neq 0$, dann ist $H(N, n)Q(N, n, K)$ ein k-freier Operator ($\neq 0$), der $F(n, k)$ annuliert.

In beiden Fällen haben wir einen k-freien Operator ($\neq 0$) gefunden, der $F(n, k)$ annuliert und dessen Grad in K kleiner ist als der von $P(N, n, K)$. Dies ist ein Widerspruch zu unserer Voraussetzung, denn P sollte unter allen Operatoren den kleinsten Grad in K haben. □

Daraus folgt, dass eine Rekursion mit einer Teleskopsumme auf der rechten Seite immer existiert. Im nächsten Abschnitt werden wir den Algorithmus vorstellen, der diese Frage untersucht.

3.2.1 Der Zeilberger-Algorithmus

Die Idee von Zeilberger besteht darin, den Gosper-Algorithmus nicht auf $F(n, k)$, sondern für ein geeignetes $J \in \mathbb{N}$ auf den Ausdruck

$$a_k = F(n, k) + \sum_{j=1}^{J} \sigma_j(n) F(n + j, k) \tag{3.11}$$

anzuwenden, wobei die noch zu bestimmenden Variablen $\sigma_j(n)$ von n, aber nicht von k abhängen. Es ist zu beachten, dass der Koeffizient $\sigma_0(n)$ von $F(n, k)$ weggelassen werden kann. Dies ist eine Normierung, wir setzen $\sigma_0(n) = 1$. Denn wenn es eine Lösung $\{\sigma_0(n),\ \sigma_1(n), \ldots, \sigma_J(n)\}$ für (3.7) mit $\sigma_0(n) \neq 0$ gibt, gilt:

$$0 = \sum_{j=0}^{J} \frac{\sigma_j(n)}{\sigma_0(n)} F(n + j, k) = F(n, k) + \sum_{j=1}^{J} \frac{\sigma_j(n)}{\sigma_0(n)} F(n + j, k).$$

Finden wir eine Lösung $\{\sigma_1(n), \ldots, \sigma_J(n)\}$, dann ist die Ordnung J hoch genug gewählt.

Weiter gilt dann:

$$\begin{aligned}\frac{a_{k+1}}{a_k} &= \frac{F(n,k+1) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k+1)}{F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k)} \\ &= \frac{F(n,k+1)}{F(n,k)} \cdot \frac{1 + \sum_{j=1}^{J} \sigma_j(n) \frac{F(n+j,k+1)}{F(n,k+1)}}{1 + \sum_{j=1}^{J} \sigma_j(n) \frac{F(n+j,k)}{F(n,k)}} \in K(n,k). \end{aligned} \tag{3.12}$$

Damit sehen wir, dass a_{k+1}/a_k rational bezüglich k ist und damit ist a_k hypergeometrisch und wird von Algorithmus 2.2.7 aus dem vorigen Abschnitt als Eingabe akzeptiert.

Nun werden wir uns den Gosper-Algorithmus Schritt für Schritt anschauen, wenn er auf den Term a_k angewendet wird. Zuerst bestimmen wir a_{k+1}/a_k und aus diesem rationalen Term gewinnen wir dann die drei Polynome p_k, q_k und r_k, wobei $p_k = 1$, $q_k = u_{k-1}$ und $r_k = v_{k-1}$ gesetzt werden, wie in Lemma 2.2.2. Die Dispersionsmenge enthält auf Grund der Summe immer den Wert 1. Daher enthält p_k nach dem Umschreiben (Lemma 2.2.2) immer die Unbekannten $\sigma_j(n)$ $(j = 1, \ldots, J)$ in linearer Form. Als nächstes wird die Gradschranke M für f_k bestimmt. Diese erhalten wir, indem wir den Algorithmus aus Satz 2.2.6 über $K(n)[k]$ anwenden. Wir ignorieren mögliche Werte von n, für welche die Gradschranke kleiner ist, da wir auf der Suche nach einer Gradschranke sind, die für alle n gültig ist.

Mit der Gradschranke erhalten wir ein allgemeines Polynom, wie folgt

$$f_k = b_0 + b_1 k + b_2 k^2 + \ldots + b_M k^M.$$

Wir setzen nun f_k in Gleichung (2.7) ein. Bei einem Koeffizientenvergleich erhalten wir ein lineares Gleichungssystem mit den Koeffizienten b_l $(l = 0, \ldots, M)$ von f_k und den Unbekannten $\sigma_j(n)$ $(j = 1, \ldots J)$. Zeilbergers entscheidende Beobachtung ist dann, dass wir beim Bestimmen der Polynomlösung gleichzeitig b_l $(l = 0, \ldots, M)$ und auch $\sigma_j(n)$ $(j = 1, \ldots J)$ finden können, wobei $\sigma_j(n) \in K(n)$ ist. Wenn dieser Schritt erfolgreich ist, erhalten wir dadurch

$$G(n,k) = \frac{r_k}{p_k} f_{k-1} a_k,$$

wobei $G(n,k)$ abhängig von n ist und a_k der Form (3.11) entspricht. Weiter finden wir eine Menge von rationalen Funktionen $\sigma_j(n) \in K(n)$, so dass

$$G(n,k+1) - G(n,k) = a_k = F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) \tag{3.13}$$

gilt. Summieren wir jetzt auf, ergibt sich

$$\begin{aligned}\sum_{k=-\infty}^{\infty} a_k &= \sum_{k=-\infty}^{\infty} \left(F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) \right) \\ &= s_n + \sum_{j=1}^{J} \sigma_j(n) s_{n+j} = \sum_{k=-\infty}^{\infty} \Big(G(n,k+1) - G(n,k) \Big) = 0. \end{aligned}$$

Wie wir vorher gesehen haben, ist die rechte Seite gleich Null, da es sich um eine Teleskopsumme handelt. Indem wir mit dem Hauptnenner multiplizieren, erlangen wir eine holonome Rekursionsgleichung der Ordnung J für die Summe s_n. Dies entspricht der Darstellung, die wir in (3.7) angestrebt haben. Wir haben nur einen leicht anderen Ansatz gewählt, indem wir den ersten Koeffizienten $\sigma_0(n)$ normiert haben.

Wir müssen beachten, dass der Gosper-Algorithmus ein Entscheidungsalgorithmus ist, das heißt, er gibt immer ein Ergebnis zurück, falls er erfolgreich ist und bricht ab, falls er nicht erfolgreich ist und dann ist bewiesen, dass keine Lösung existiert. Damit wird er immer erfolgreich sein, falls eine Gleichung der Form (3.13) für $F(n,k)$ gültig ist. Zum Glück von Zeilberger trifft das in fast allen Fällen zu. Es gibt aber keine Garantie, dass wir mit dieser Methode die holonome Rekursionsgleichung niedrigster Ordnung für s_n finden. Wir können aber beweisen, dass der Algorithmus terminiert, indem wir Einschränkungen für den Eingabeterm vornehmen. Im weiteren Verlauf werden wir sehen, dass, wenn wir das beschriebene Verfahren bei $J = 1$ starten und bei Misserfolg um 1 erhöhen, der Algorithmus abbricht.

Als nächstes betrachten wir ein Beispiel für das der Gosper-Algorithmus keine diskrete Stammfunktion findet.

Beispiel 3.2.2
Wir erinnern uns an Beispiel 2.2.10 zurück. Das Ziel ist es eine holonome Rekursionsgleichung erster Ordnung für

$$s_n = \sum_{k=0}^{n} \binom{n}{k}$$

zu finden. Zuerst wird

$$F(n,k) = \binom{n}{k}$$

gesetzt. Der Ansatz für den Zeilberger-Algorithmus ist dann

$$a_k = F(n,k) + \sigma_1(n)F(n+1,k) = \binom{n}{k} + \sigma_1\binom{n+1}{k}.$$

Nun wird der erste Schritt des Gosper-Algorithmus angewendet und es ergibt sich:

$$\frac{a_{k+1}}{a_k} = \frac{(n+1-k)(n-k+\sigma_1(n)n+\sigma_1(n))}{(n+1-k+\sigma_1(n)n+\sigma_1(n))(k+1)}. \tag{3.14}$$

Die Dispersionsmenge enthält dann die 1 und nach dem Umschreiben nach Lemma 2.2.2 ergibt sich dann $p_k = n+1-k+\sigma_1(n)n+\sigma_1(n)$, $q_k = n+2-k$ und $r_k = k$. Die Gradschranke nach Satz 2.2.6 ergibt, dass der Grad für f_k gleich 0 ist. Durch Einsetzen des allgemeinen Polynoms $f_k = b_0$ in die Gleichung (2.7) kommen wir zu der Identität

$$n+1-k+\sigma_1(n)n+\sigma_1(n) = (n+1-k)b_0 - kb_0.$$

Ein Koeffizientenvergleich bezüglich k gibt uns die linearen Gleichungen

$$\begin{aligned} -1 + 2b_0 &= 0, \\ n + 1 + \sigma_1(n)n + \sigma_1(n) - (n+1)b_0 &= 0. \end{aligned}$$

Diese können wir nach den Unbekannten $\{b_0, \sigma_1(n)\}$ auflösen, als Ergebnis erhalten wir

$$\{b_0 = 1/2,\ \sigma_1(n) = -1/2\}.$$

Damit ergibt sich $f_k = 1/2$, aber die wichtige Information, die wir dadurch erhalten, ist, dass $\sigma_1(n) = -1/2$ ist. Durch Einsetzen ergibt sich die Rekursionsgleichung

$$s_n - \frac{1}{2}s_{n+1} = 0.$$

Diese Rekursion können wir auch mit dem implementierten MuPAD Befehl `zeilberger(F,k,s(n))` finden, wobei `F` der Summand, `k` der Summationsindex und `s(n)` der Name der unbekannten Summe ist.

MuPAD

```
>> zeilberger(binomial(n,k),k,s(n));
```

Output

```
2*s(n) - s(n + 1) = 0
```

Der Befehl `zeilberger` kann auch mit weiteren Parametern aufgerufen werden. Als vierter Parameter kann `J` eingegeben werden, der angibt wie oft der Zeilberger-Algorithmus aufgerufen werden soll. Wie beim Befehl `gosper` können durch `setuserinfo`[5] Informationen über Zwischenergebnisse abgefragt werden.

Durch unser Ergebnis erhalten wir $s_{n+1}/s_n = 2$ und zusammen mit dem Anfangswert $s_0 = 1$ ergibt sich aus (3.8) die bekannte Vereinfachung $s_n = 2^n$. Mit MuPAD ist dies schnell gezeigt, der Befehl `rechyper`[6] findet alle hypergeometrischen Lösungen.

MuPAD

```
>> rechyper(2*s(n) - s(n + 1) = 0,s(n));
```

Output

```
{2}
```

Damit haben wir auch mit MuPAD die hypergeometrische Lösung für dieses Problem gefunden.

Wir wollen nun den Zeilberger-Algorithmus nochmals zusammenfassen.

Algorithmus 3.2.3 *Gegeben ist $F(n,k)$, der Algorithmus sucht dann eine holonome Rekursionsgleichung für $s_n = \sum_{k=-\infty}^{\infty} F(n,k)$.*

[5] Genauer müssen alle `setuserinfo` wie beim Befehl `gosper` gesetzt werden und zusätzlich `setuserinfo(zeilberger,5)`.

[6] Wir werden auf diesen Befehl genauer in Kapitel 4.3 eingehen.

1. *Eingabe:* $F(n,k) \neq 0$*, welches ein (rationaler) hypergeometrischer Term bezüglich* k *und* n *ist.*

2. *Setze* $J = 1$.

3. *Setze* $a_k = F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k)$*, mit den zu bestimmenden Variablen* σ_j*, welche von* n *aber nicht von* k *abhängig sind.*

4. *Der angepasste Gosper-Algorithmus wird auf den Term* a_k *angewendet. Im letzten Schritt lösen wir ein Gleichungssystem für die Koeffizienten von* f_k *und zur gleichen Zeit für die Unbekannten* $\sigma_j(n)$ $(j = 1, \ldots, J)$*. Wenn der Algorithmus erfolgreich ist, findet der Gosper-Algorithmus ein* $G(n,k)$ *mit der Eigenschaft*

$$G(n,k+1) - G(n,k) = a_k.$$

   Zu beachten ist, dass es möglicherweise ganzzahlige Nullstellen im Nenner bezüglich n *gibt, wo das rationale Zertifikat*

$$\tilde{R}(n,k) = \frac{G(n,k)}{a_k}$$

   für die resultierende Rekursionsgleichung möglicherweise nicht gültig ist. Die Berechnung bestimmt auch die Funktionen $\sigma_j(n) \in K(n)$ $(j = 1, \ldots, J)$*. Wenn die Prozedur nicht erfolgreich ist, erhöhe* J *um eins und gehe zu Schritt* 3 *zurück.*

5. *Ausgabe: Wir erhalten*

$$s_n + \sum_{j=1}^{J} \sigma_j(n) s_{n+j} = 0$$

   für s_n*. Falls die rechte Seite eine Teleskopsumme ist, insbesondere wenn* $F(n,k)$ *einen endlichen Träger bezüglich* k *hat, ergibt sich durch Multiplikation mit dem Hauptnenner die gewünschte holonome Rekursionsgleichung.*

Für $G(n,k)$ gilt, dass es ein rationales Vielfaches von a_k ist, das heißt: $G(n,k) = R(n,k)a_k$. Da wir in diesem Kapitel nur die bestimmte Summation über alle ganze Zahlen betrachten, müssen wir auf $G(n,k)$ nicht so ein großes Augenmerk legen. In Kapitel 5, in dem wir die bestimmte Summation mit festen Summationsgrenzen betrachten, werden wir genauer auf die Form von $G(n,k)$ eingehen.

Es bleibt noch eine unbeantwortete Frage in Bezug auf den Zeilberger-Algorithmus offen: Können wir garantieren, dass der Algorithmus wirklich terminiert? Wenn der Algorithmus fehlschlägt, erhöhen wir immer weiter die Ordnung J. Weiter wissen wir auch, dass der Zeilberger-Algorithmus nicht immer die Rekursion mit der niedrigsten Ordnung berechnet, sondern eine mit höherem Grad. Wie können wir unter diesen Umständen sicher sein, dass er terminiert? Wir können beweisen, dass der Algorithmus für zulässige hypergeometrische Eingabeterme wirklich terminiert.

In Definition 3.1.1 wurde die zulässigen hypergeometrischen Terme definiert. Für solche Terme existiert nach Satz 3.1.2 eine k-freie holonome Rekursionsgleichung mit Polynomen $\sigma_{i,j}(n)$

$$\sum_{i=0}^{I}\sum_{j=0}^{J}\sigma_{i,j}(n)F(n+j,k+i)=0 \tag{3.15}$$

für ein I und J, die groß genug sind. Eine solche Rekursionsgleichung existiert, wenn

$$J\geq J_0=\sum_{l=1}^{p}|\alpha_l|+\sum_{l=1}^{q}|\gamma_l| \text{ und } I\geq\left(\sum_{l=1}^{p}|\beta_l|+\sum_{l=1}^{q}|\delta_l|-1\right)\cdot J_0\deg P(n,k)$$

gilt.

Der Beweis kann in [Koe97] auf Seite 110 bis 112 nachgelesen werden. Die Existenz der Rekursion haben wir in Abschnitt 3.2 durch Satz 3.2.1 bewiesen. Das führt uns zu folgendem Satz.

Satz 3.2.4 *Für Summen mit zulässigen hypergeometrischen Termen terminiert der Zeilberger-Algorithmus.*

Beweis: Wir haben mit dem Beweis aus Satz 3.2.1 die Existenz der Rekursion bewiesen. Das heißt, für einen zulässigen hypergeometrischen Term $F(n,k)$ nach 3.1.1 hat die Summe des Zeilberger-Algorithmus

$$\sum_{j=0}^{J}\sigma_j(n)F(n+j,k)$$

eine zulässige hypergeometrische Stammfunktion $G(n,k)$ für passende $\sigma_j(n)$ $(j=0,\ldots,J)$ und $J\in\mathbb{N}$. Da der Gosper-Algorithmus ein Entscheidungsalgorithmus ist, findet er ein $G(n,k)$. Die Summation führt dann zur gewünschten Rekursionsgleichung für die Summe, falls $F(n,k)$ einen endlichen Träger hat. □

Dennoch ist zu erwähnen, dass der Zeilberger-Algorithmus nicht für alle hypergeometrischen Terme terminiert, sondern nur für die, die nach Definition 3.1.1 zulässig sind.

Nun werden wir noch einige Beispiele mit Hilfe des Zeilberger-Algorithmus lösen.

Beispiel 3.2.5
Gegeben ist der Summand $F(n,k)=\binom{n}{k}^2$ und wir wollen zunächst per Hand und später mit MuPAD das Problem mit Hilfe des Zeilberger-Algorithmus lösen. Zunächst versuchen wir eine Rekursion für $J=1$ zu finden. Dazu setzen wir $a_k=\binom{n}{k}^2+\sigma_1(n)\binom{n+1}{k}^2$ und es ergibt sich:

$$\frac{a_{k+1}}{a_k}=\frac{(n-k+1)^2}{(k+1)^2}\frac{(k^2+\sigma_1(n)(n^2+2n+1)+n^2-2kn)}{(2n-2k+k^2+\sigma_1(n)(n^2+2n+1)+n^2-2kn+1)}=\frac{u_k}{v_k}.$$

Dann setzen wir $p_k = 1$, $q_k = u_{k-1}$ und $r = v_{k-1}$ wie in Lemma 2.2.2. Da wir wissen, dass die Dispersionsmenge immer die 1 enthält, ergibt sich nach dem Umschreiben von p_k, q_k und r_k, wie im Beweis von Lemma 2.2.2:

$$\begin{aligned} p_k &= (n+1)^2\sigma_1(n) + (k-n-1)^2 \\ q_k &= (k-n-2)^2 \\ r_k &= k^2. \end{aligned}$$

Nun fragen wir uns, ob $\sigma_1(n)$ und eine Polynomlösung f_k für Gleichung (2.7) existieren. Mit Hilfe von Satz 2.2.6 finden wir durch Teil $(2a)$ die Gradschranke 1 für das allgemeine Polynom f_k. Setzen wir nun $f_k = b_0 + b_1 k$ in (2.7) ein, ergibt sich

$$(k-n-1)^2(b_0 + b_1 k) - k^2(b_0 + b_1(k-1)) = (n+1)^2\sigma_1 + (k-n-1)^2.$$

Durch den Koeffizientenvergleich bezüglich k erhalten wir die Resultate

$$b_0 = \frac{(3n+1)}{(4n+2)},\ b_1 = -\frac{1}{(2n+1)} \text{ und } \sigma_1(n) = -\frac{(n+1)}{(4n+2)}.$$

Damit erhalten wir folgende Rekursionsgleichung für $F(n,k) = \binom{n}{k}^2$:

$$F(n,k) - \frac{(n+1)}{(4n+2)}F(n+1,k) = G(n,k+1) - G(n,k).$$

Der Algorithmus hat uns nun eine Rekursion für die Summe $s_n = \sum_k \binom{n}{k}^2$ zurückgegeben und diese können wir dann leicht auswerten. Wenn wir über alle ganzen Zahlen k summieren, wird die rechte Seite zu Null und mit dem Hauptnenner multipliziert ergibt sich:

$$2(2n+1)s_n - (n+1)s_{n+1} = 0.$$

Dieses Ergebnis kann auch mit MuPAD erzielt werden:

MuPAD

```
>> zeilberger(binomial(n,k)^2,k,s(n));
```

Output

```
2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = 0
```

Mit dem Anfangswert $s_0 = 1$ finden wir dann die einfache Form $s_n = \binom{2n}{n}$.

Beispiel 3.2.6
Wir betrachten hier die Summe

$$s_n = \sum_k \binom{3k+1}{k}\binom{3n-3k}{n-k}\frac{1}{(3k+1)}.$$

Unser Ziel ist es eine Rekursion für diese Summe zu finden. Mit Hilfe des in MuPAD implementierten Befehls finden wir eine Rekursion zweiter Ordnung für die Summe s_n:

MuPAD

```
>> setuserinfo(zeilberger,1):
>> zeil:=zeilberger(binomial(3*k+1,k)*binomial(3*n-3*k,n-k)/(3*k+1),
   k,s(n),1);
```

Output

```
Die Ordnung ist nicht hoch genug
FAIL
```

Da wir für diese Ordnung keine Rekursion gefunden haben, erhöhen wir J um 1.

MuPAD

```
>> zeil:=zeilberger(binomial(3*k+1,k)*binomial(3*n-3*k,n-k)/(3*k+1),
   k,s(n),2);
```

Output

```
4*s(n + 2)*(n + 2)*(2*n + 3)*(2*n + 5) - 12*s(n + 1)*(2*n + 3)*
(9*n^2 + 27*n + 22) + 81*s(n)*(n + 1)*(3*n + 2)*(3*n + 4) = 0
```

Wir können diese Rekursion, da sie eine höhere Ordnung als eins hat, nicht mit den bis jetzt bekannten Verfahren lösen. Aber beim Durchsuchen einer Liste von Binomialkoeffizienten, so wie in [GKP89], finden wir folgende Identität:

$$\sum_k \binom{tk+r}{k}\binom{tn-tk+s}{n-k}\frac{r}{tk+r} = \binom{tn+r+s}{n}. \tag{3.16}$$

Wenn wir nun $t = 3,\ r = 1$ und $s = 0$ setzen erhalten wir die spezielle Identität

$$\sum_k \binom{3k+1}{k}\binom{3n-3k}{n-k}\frac{1}{3k+1} = \binom{3n+1}{n}. \tag{3.17}$$

Dies bedeutet, dass unser s_n ein hypergeometrischer Term ist. Indem wir den Term $\binom{3n+1}{n}$ in unsere Rekursion für s_n einsetzen, können wir leicht (3.17) beweisen:

MuPAD

```
>> t := fp::unapply(binomial(3*n+1,n),n):
>> factor(expand(eval(subs(zeil,s   =t))));
```

Output

```
0=0
```

Da s_n auch für $n = 0$ und $n = 1$ gilt, folgt, dass $s_n = \binom{3n+1}{n}$ ist. Weiter ist zu beachten, dass der Term aus (3.16) nicht hypergeometrisch für k oder n ist, falls t eine Variable ist. Aber für jede feste ganze Zahl t ist es ein hypergeometrischer Term in allen verbleibenden Variablen und daher ist auch die rechte Seite von Gleichung (3.16) ein solcher.

Bei dem letzten betrachteten Beispiel hatten wir das Glück, dass wir eine Identität für den Binomialkoeffizienten finden konnten. Aber im Allgemeinen

wissen wir jetzt noch nicht, wie wir eine geschlossene Form für eine Rekursionsgleichung mit einer Ordnung $J \geq 1$ finden. Dieses Problem werden wir im nächsten Kapitel genauer untersuchen.

Kapitel 4

Petkovšek-Algorithmus

4.1 Einführung

In diesem Kapitel werden wir untersuchen, wie wir eine geschlossene Form aus einer Rekursionsgleichung erhalten können. Nach unserer Definition ist die geschlossene Form eine Linearkombination von hypergeometrischen Termen. Für die unbestimmte Summation finden wir direkt mit dem Gosper-Algorithmus eine geschlossene Form, falls eine solche existiert. Im vorigen Kapitel haben wir gesehen, wie wir für die bestimmte Summation eine Rekursion finden. Diese bringt die Summe s_n noch nicht in eine solche Form. Falls die Rekursion erster Ordnung ist, sind wir fertig, denn wir haben gesehen, dass wir einen hypergeometrischen Term erhalten, der in eine geschlossene Form gebracht werden kann. Ist die Ordnung der Rekusionsgleichung ≥ 2, ist noch offen wie wir eine geschlossene Form finden. Im weiteren Verlauf werden wir uns damit beschäftigen, wie wir erkennen können, ob so eine Lösung existiert und wie wir alle Lösungen dieser Form bestimmen können.

Wir werden sehen, dass wir mit dem Petkovšek-Algorithmus in der Lage sein werden, solch eine Form zu finden, falls es eine gibt. Es ist zu beachten, dass dieser Algorithmus völlig unabhängig vom Zeilberger-Algorithmus ist. Die Situation, die wir untersuchen, ist allgemeiner: Wir haben eine holonome Rekursionsgleichung gegeben und wollen alle hypergeometrischen Lösungen bestimmen. Der Petkovšek-Algorithmus ist, wie der Gosper-Algorithmus, ein Entscheidungsalgorithmus. Wir können mit Hilfe dieser Prozedur entscheiden, ob hypergeometrische Lösungen für eine Rekursionsgleichung, die wir durch den Zeilberger-Algorithmus erhalten, existieren oder nicht. Diese können wir dann in eine geschlossene Form umwandeln.

Der Petkovšek-Algorithmus ist in zwei Teile unterteilt. Der eine findet die Polynomlösungen für eine gegebene holonome Rekursionsgleichung. Der andere ist eine Unterroutine, die die hypergeometrischen Termlösungen für eine gegebene holonome Rekursionsgleichung bestimmt.

4.2 Polynomlösungen für eine Rekursionsgleichung

Wir werden den Algorithmus anhand eines allgemeinen Beispiels erläutern. Aus Notationsgründen beschränken wir uns auf eine Rekursionsgleichung zweiter Ordnung, diese sind vom größten Interesse. Dieses Beispiel führt uns dann zum Algorithmus von Petkovšek [Pet92]. Die weiteren Grundlage dieses Abschnittes stammen aus [Koe97].

Beispiel 4.2.1
Angenommen, es ist eine holonome Rekursion zweiter Ordnung für s_n gegeben durch

$$P_n s_{n+2} + Q_n s_{n+1} + R_n s_n = 0, \tag{4.1}$$

wobei $P_n,\ Q_n,\ R_n \in K[n]$ sind.

Unser Ziel ist es nun alle Polynomlösungen $s_n \neq 0$ für diese Gleichung zu bestimmen. Um diese zu finden, brauchen wir nur eine obere Gradschranke für jede Polynomlösung. Sobald wir eine solche Gradschranke haben, können wir ein allgemeines Polynom mit dieser in die Gleichung (4.1) für s_n einsetzen und die Koeffizienten bestimmen, indem wir das entstandene lineare Gleichungssystem für die unbekannten Koeffizienten von s_n lösen. Falls das lineare Gleichungssystem eine nicht triviale Lösung hat, haben wir ein s_n gefunden, andernfalls existiert keine Polynomlösung für (4.1).

Als nächstes werden wir uns damit befassen, wie wir eine obere Gradschranke N für die nicht triviale Polynomlösung

$$s_n = n^N + \delta_1 n^{N-1} + \ldots + \delta_N,$$

mit den unbestimmten Koeffizienten δ_l $(l = 1, \ldots, N)$ bestimmen können. Da die Rekursion linear und homogen ist, ist auch jedes Vielfache der Lösung eine Lösung. Daher reicht es, wenn wir als Lösung normierte Polynome betrachten, das sind die, deren führender Koeffizient gleich Eins ist.

Die gegebenen Polynome aus (4.1) haben die Form

$$P_n = \alpha_0 n^M + \alpha_1 n^{M-1} + \ldots + \alpha_M,$$

$$Q_n = \beta_0 n^M + \beta_1 n^{M-1} + \ldots + \beta_M$$

und

$$R_n = \gamma_0 n^M + \gamma_1 n^{M-1} + \ldots + \gamma_M.$$

Dabei ist M die maximale Gradzahl von $P_n,\ Q_n$ und R_n, genauer: $(\alpha_0,\ \beta_0,\ \gamma_0) \neq (0,\ 0,\ 0)$. Nun kann für jedes $j \in \mathbb{N}$ der Shift s_{n+j} durch den binomischen Lehrsatz ausgedrückt werden

$$\begin{aligned} s_{n+j} &= (n+j)^N + \delta_1 (n+j)^{N-1} + \delta_2 (n+j)^{N-2} \ldots + \delta_N \\ &= n^N + (\delta_1 + jN) n^{N-1} + \big(\delta_2 + j(N-1)\delta_1 + j^2 \frac{N(N-1)}{2}\big) n^{N-2} + \ldots . \end{aligned}$$

Wir setzen nun alle Polynome in (4.1) ein und erhalten:

$$\begin{aligned} 0 =&(\alpha_0 n^M + \alpha_1 n^{M-1} + \ldots)(n^N + (\delta_1 + 2N)n^{N-1} + \ldots) \\ &+ (\beta_0 n^M + \beta_1 n^{M-1} + \ldots)(n^N + (\delta_1 + N)n^{N-1} + \ldots) \\ &+ (\gamma_0 n^M + \gamma_1 n^{M-1} + \ldots)(n^N + \delta_1 n^{N-1} + \ldots). \end{aligned} \tag{4.2}$$

Zunächst bestimmen wir die Koeffizienten von n^{M+N} und es ergibt sich:

$$\alpha_0 + \beta_0 + \gamma_0 = 0. \tag{4.3}$$

Für jede nicht triviale Polynomlösung muss diese Gleichung gültig sein. Ist (4.3) also nicht erfüllt, existiert keine Polynomlösung und wir können abbrechen. Ist (4.3) erfüllt, berechnen wir die Koeffizienten von n^{M+N-1} aus (4.2). Aus Gleichung (4.3) ergibt sich: $\alpha_0 = -\beta_0 - \gamma_0$. Damit kann α_0 durch die beiden anderen ausgedrückt werden und wir erhalten die Eigenschaft

$$\alpha_1 + \beta_1 + \gamma_1 - (\beta_0 + 2\gamma_0)N = 0. \tag{4.4}$$

Nun können zwei Fälle eintreten. Ist $\beta_0 + 2\gamma_0 \neq 0$, dann erlangen wir aus (4.4) eine eindeutige Gradschranke N für s_n. Falls sich keine nicht negative ganze Zahl für N ergibt, können wir aufhören, da keine obere Gradschranke gefunden werden kann. Andernfalls haben wir die gesuchte Gradschranke gefunden.

Ist andererseits

$$\beta_0 + 2\gamma_0 = 0, \tag{4.5}$$

dann gilt weiter mit (4.4)

$$\alpha_1 + \beta_1 + \gamma_1 = 0. \tag{4.6}$$

Wir bestimmen die Koeffizienten für n^{M+N-2} aus (4.2). Mit (4.3), (4.5) und (4.6) als Ersetzungsregeln, ergibt sich die Eigenschaft

$$N^2\gamma_0 - (\beta_1 + \gamma_0 + 2\gamma_1)N + \alpha_2 + \beta_2 + \gamma_2 = 0.$$

Zum Abschluss zeigen wir noch, dass mit dieser Eigenschaft nur noch zwei Möglichkeiten für N existieren. Wir werden überprüfen, ob $\gamma_0 \neq 0$ gilt. Dann nehmen wir an, dass $\gamma_0 = 0$ ist. Es gilt mit Gleichung (4.5) $\beta_0 = 0$ und mit (4.3) $\alpha_0 = 0$, das ist ein Widerspruch zu unserer Wahl M. Dies bedeutet, wir finden eine Gradschranke und dies beweist, dass der Algorithmus alle Polynomlösungen für (4.1) findet. Wir bestimmen die Lösung, indem wir das allgemeine Polynom mit maximalem Grad N in (4.1) einsetzen und einen Koeffizientenvergleich durchführen.

Wir haben in diesem Beispiel gesehen, wie wir die Polynomlösung für eine Rekursion zweiter Ordnung bestimmen. Nun wollen wir den Algorithmus für den allgemeinen Fall kurz zusammenfassen.

Algorithmus 4.2.2 (Polynomlösung einer Rekursionsgleichung) *Der folgende Algorithmus findet alle polynomialen Lösungen einer gegebenen holonomen Rekursionsgleichung falls solche existieren.*

1. *Eingabe: Eine holonome Rekursionsgleichung*

$$\sum_{j=0}^{J} P_j(n) s_{n+j} = 0 \tag{4.7}$$

mit den Polynomen

$$P_j(n) = \sum_{l=0}^{M} \alpha_{jl} n^{M-l} \ \in K[n],$$

so dass mindestens ein $\alpha_{j0} \neq 0$ $(j = 0, \ldots, J)$.

2. *Setze* $m = 0$.

3. *Berechne für alle* $l = 0, \ldots, m$

$$b_{lm} := \sum_{j=0}^{J} j^l \alpha_{j,m-l}.$$

Falls $b_{lm} = 0$ *für alle* $l = 0, \ldots, m$, *erhöhe* m *um eins und wiederhole Schritt* (3).

4. *Sei* $\mathcal{N}$ *die Menge der nicht negativen ganzen Nullstellen* $N \in \mathbb{N}_0$ *der Polynome*

$$\sum_{l=0}^{m} \binom{N}{l} b_{lm}. \tag{4.8}$$

5. *Falls* $\mathcal{N} = \emptyset$ *gib aus, dass keine Polynomlösung existiert.*

6. *Sei* $N = \max \mathcal{N}$. *Wir setzen das allgemeine Polynom mit Grad* N *in die Gleichung ein und führen einen Koeffizientenvergleich durch. Wenn wir das entstandene lineare Gleichungssystem lösen, erhalten wir eine allgemeine Polynomlösung* s_n *für (4.7).*

7. *Ausgabe: Die polynomiale Lösung für* s_n, *die im oberen Schritt bestimmt wurde.*

Wir werden hier diesen Algorithmus nicht beweisen und verweisen auf [Pet92] und S. 148 – 150 [PWZ97]. Aber es ist darauf hinzuweisen, dass die Hauptaufgabe im Beweis des Algorithmus ist, zu zeigen, dass die Iteration in Schritt 3 terminiert. Es stellt sich heraus, dass sie nicht über J Schritte hinausläuft und dass die Formel (4.8) gültig ist. Dies kann in [Pet92] nachgelesen werden.

Zum Abschluss des Abschnitts werden wir uns ein Beispiel mit Hilfe von MuPAD ansehen.

Beispiel 4.2.3
Wir haben die homogene Rekursionsgleichung

$$n(n+1)s(n+2) - 2n(n+7)s(n+1) + (n+6)(n+7)s(n) = 0.$$

gegeben. Mit dem von mir implementierten Befehl `recpoly(eqn,s(n))` können wir die gewünschte Polynomlösung erhalten.

MuPAD

```
>> recpoly(n*(n+1)*s(n+2)-2*n*(n+7)*s(n+1)+(n+6)*(n+7)*s(n),s(n));
```

Output

```
n*(n + 5)*(n + 4)*(n + 3)*(n + 2)*(n + 1)*
(delta2 - 15*delta1 + delta1*n)
```

Damit haben wir eine Polynomlösung erhalten.

Mit der kennengelernten Routine können wir uns im nächsten Abschnitt dem Hauptproblem, der Bestimmung aller hypergeometrischen Lösungen einer holonomen Rekursionsgleichung, widmen.

4.3 Hypergeometrische Lösung einer Rekursion

In diesem Abschnitt erläutern wir genauer, wie wir alle hypergeometrischen Lösungen für eine gegebene holonome Rekursionsgleichung finden können. Er basiert auf [Koe97] S. 149 – 157 und [PWZ97] S. 151 – 155.

Um die hypergeometrischen Lösungen zu finden, brauchen wir eine strengere Version des Lemmas 2.2.2 von Gosper für rationale Funktionen. Diese Modifizierung hat Petkovšek im Rahmen seines Algorithmus vorgenommen.

Lemma 4.3.1 (Gosper-Petkovšek Darstellung von rationalen Funktionen) *Jede rationale Funktion $t_k \in K(k) \setminus \{0\}$ hat eine Darstellung der Form*

$$t_k = C\frac{p_{k+1}}{p_k}\frac{q_{k+1}}{r_{k+1}}, \tag{4.9}$$

wobei $p_k,\ q_k,\ r_k \in K[k]$ normierte Polynome sind und $C \in K$. Bei geschickter Berechnung erfüllen diese zusätzlich die folgenden Eigenschaften:

1. $\mathrm{ggT}(q_k, r_{k+j}) = 1$ *für alle* $j \in \mathbb{N}_0$,
2. $\mathrm{ggT}(p_k, q_{k+1}) = 1$,
3. $\mathrm{ggT}(p_k, r_k) = 1$.

Beweis: Das Lemma und der dazugehörige Beweis bilden auch einen Algorithmus zur Bestimmung von $p_k,\ q_k$ und r_k. Der Beweis verläuft analog zu dem von Lemma 2.2.2. Er kann vollständig in [Koe97] oder in [PWZ97] nachgelesen werden. □

Die Teile 2. und 3. aus dem Lemma unterstreichen, dass in der Darstellung (4.9) sowohl q_{k+1} und p_k als auch p_{k+1} und r_{k+1} zueinander teilerfremd sind. Dadurch ist die Gosper-Petkovšek Darstellung eindeutig. Bei jedem Umschreiben ist es notwendig, immer das kleinste mögliche $j \in \mathbb{N}$ zu wählen.

Wir werden nun den Petkovšek-Algorithmus wieder anhand einer allgemeinen Rekursion zweiter Ordnung als Beispiel vorführen. Die dadurch erlangten Kenntnisse werden uns auch hier zu dem allgemeinen Algorithmus führen [Pet92].

Beispiel 4.3.2 (Hypergeometrische Lösung einer Rekursion)
Wir wählen wie in Beispiel 4.2.1 eine Rekursionsgleichung zweiter Ordnung für s_n:

$$P_n s_{n+2} + Q_n s_{n+1} + R_n s_n = 0 \quad (P_n,\ Q_n,\ R_n \in K[n]). \tag{4.10}$$

In diesem Beispiel wollen wir alle hypergeometrischen Lösungen für $s_n \neq 0$ für die Rekursion finden.

Da wir annehmen, dass s_n ein hypergeometrischer Term ist, ist der Quotient

$$\frac{s_{n+1}}{s_n} = t_n \in K(n) \tag{4.11}$$

rational. Indem wir uns auf Lemma 4.3.1 berufen, wissen wir, dass $C \in K$ und $p_n,\ q_n,\ r_n \in K[n]$ existieren, so dass

$$t_n = C \frac{p_{n+1}}{p_n} \frac{q_{n+1}}{r_{n+1}} \tag{4.12}$$

gilt und es gelten die ggT-Eigenschaften aus Lemma 4.3.1.

Indem wir nun (4.11) in (4.10) einsetzen und die resultierende Gleichung durch s_n teilen, erhalten wir

$$P_n t_{n+1} t_n + Q_n t_n + R_n = 0.$$

Als nächstes setzen wir (4.12) für t_n ein und multiplizieren mit $p_n r_{n+1} r_{n+2}$. Daraus ergibt sich die Gleichung

$$C^2 P_n p_{n+2} q_{n+2} q_{n+1} + C Q_n p_{n+1} q_{n+1} r_{n+2} + R_n p_n r_{n+1} r_{n+2} = 0. \tag{4.13}$$

Nun nutzen wir die ggT-Eigenschaft aus Lemma 4.3.1. Der erste und zweite Summand aus (4.13) haben den gemeinsamen Faktor q_{n+1}. Teilen wir nun durch diesen Term, muss der dritte Term $R_n p_n r_{n+1} r_{n+2}$ durch q_{n+1} teilbar sein. Wir wissen aus Lemma 4.3.1, dass q_{n+1} keine gemeinsamen Teiler mit $p_n,\ r_{n+1}$ und r_{n+2} hat. Daraus folgt: $R_n/q_{n+1} \in K[n]$. Aus dieser Eigenschaft können wir folgern, dass q_n ein normierter Faktor von R_{n-1} ist. Es gibt also nur eine endliche Anzahl von Möglichkeiten, aber es können sehr viele sein, falls R_n viele Teiler hat.

Ähnlich ist es mit dem zweiten und dritten Summanden aus (4.13): Sie haben den gemeinsamen Faktor r_{n+2}. Teilen wir durch diesen, muss also der erste Summand $P_n p_{n+2} q_{n+2} q_{n+1}$ durch r_{n+2} teilbar sein. Da nach Lemma 4.3.1 r_{n+2} keine gemeinsamen Teiler mit $p_{n+2},\ q_{n+2}$ und q_{n+1} hat, muss $P_n/r_{n+2} \in K[n]$ sein. Damit ist r_n einer der normierter Faktoren von P_{n-2}. Die Anzahl dieser Faktoren ist wieder endlich.

Für jede Wahl des Paares $(q_n,\ r_n)$ aus den Faktoren von $(R_{n-1},\ P_{n-2})$, können wir $q_{n+1}r_{n+2}$ aus der Gleichung (4.13) herauskürzen und erhalten die Polynomgleichung

$$C^2\frac{P_n}{r_{n+2}}p_{n+2}q_{n+2} + CQ_np_{n+1} + \frac{R_n}{q_{n+1}}p_nr_{n+1} = 0. \tag{4.14}$$

Nun bestimmen wir C, indem wir den führenden Koeffizienten von n der linken Seite von (4.14) betrachten. Da $p_n,\ p_{n+1}$ und p_{n+2} alle den gleichen Grad haben, entsteht eine quadratische Gleichung für C. Damit gibt es für jede Wahl von q_n und r_n, aus den Faktoren von R_{n-1} und P_{n-2}, maximal zwei Möglichkeiten für die Wahl von $C \in K$.

Für jede feste Wahl von $q_n,\ r_n$ und C können wir Algorithmus 4.2.2 verwenden, um herauszufinden, ob eine Polynomlösung für p_n aus (4.14) existiert oder nicht. Jede solche Lösung liefert uns mit (4.12) eine hypergeometrische Lösung für (4.10). Falls keine weiteren Lösungen gefunden werden können, existiert auch keine weitere hypergeometrische Lösung.

Wir fassen nun noch den allgemeinen Algorithmus von Petkovšek zur Bestimmung von hypergeometrischen Lösungen zusammen. Wir werden ihn nicht beweisen, aber der Beweis kann in [PWZ97] nachgelesen werden.

Algorithmus 4.3.3 (Hypergeometrische Lösungen für eine Rekursionsgleichung) *Der Algorithmus findet alle hypergeometrische Lösungen für eine gegebene holonome Rekursionsgleichung, falls solche existieren.*

1. *Eingabe: Eine holonome Rekursionsgleichung*

$$\sum_{j=0}^{J} P_j(n)s_{n+j} = 0 \tag{4.15}$$

   mit den Polynomen $P_j \in K[n]$.

2. *Setze $L = \{\}$.*

3. *Für alle normierten Faktoren q_n von $P_0(n-1)$ und r_n von $P_J(n-J)$:*

   (a) *Für $j = 0, \ldots, J$ setze*

$$h_j(n) = P_j(n)\prod_{l=1}^{j} q_{n+l}\prod_{l=j+1}^{J} r_{n+l}.$$

   (b) *Sei $M = \max\limits_{0\le j\le J} \deg h_j(n)$ und α_j $(j = 0, \ldots J)$ sind die Koeffizienten von n^M in $h_j(n)$.*

   (c) *Wende für jede rationale Lösung $C \in K$ aus der Polynomgleichung*

$$\sum_{j=0}^{J} \alpha_j C^j = 0 \tag{4.16}$$

den Algorithmus 4.2.2 für die Rekursionsgleichung

$$\sum_{j=0}^{J} C^j h_j(n) p_{n+j} = 0 \tag{4.17}$$

an und finde so alle Polynomlösungen p_n für (4.17). Falls eine Polynomlösung p_n existiert, füge den rationalen Term

$$t_n = C \frac{p_{n+1}}{p_n} \frac{q_{n+1}}{r_{n+1}}$$

zu der Menge L hinzu.

4. *Ausgabe: Gib die Menge L, die alle rationalen hypergeometrischen Termlösungen aus (4.15) enthält, aus.*

Es ist zu erwähnen, dass der Algorithmus eine leere Menge zurückgibt, falls keine hypergeometrischen Lösungen existieren.

Wir sehen, dass die rationale Faktorisierung eine wichtige Rolle im Petkovšek-Algorithmus spielt. Den Begriff der Ähnlichkeit von hypergeometrischen Termen haben wir in Abschnitt 2.3.1 kennen gelernt. Dies finden wir im Petkovšek-Algorithmus wieder, denn er findet nicht nur die Menge aller hypergeometrischen Lösungen, sondern diese Menge bildet eine Basis für alle Linearkombinationen von hypergeometrischen Lösungen. Weiter gilt, dass, wenn der Algorithmus keine Lösung findet, keine Linearkombination von hypergeometrischen Termen als Lösung existieren kann.

Wir können die hypergeometrischen Lösungen der bestimmten Summation vollständig algorithmisch bestimmen, wenn eine Summe

$$s_n = \sum_{k=-\infty}^{\infty} F(n,k),$$

gegeben ist, wobei $F(n,k)$ ein hypergeometrischer Term bezüglich beider Variablen ist und einen endlichen Träger hat. Indem wir zuerst den Zeilberger-Algorithmus anwenden, finden wir eine holonome Rekursionsgleichung für s_n. Der Petkovšek-Algorithmus findet für die resultierende Rekursionsgleichung dann alle hypergeometrischen Lösungen, falls solche hypergeometrischen Terme existieren. Wenn wir nun noch genügend viele Anfangswerte überprüfen, können wir entscheiden, ob s_n durch einen hypergeometrischen Term oder eine Linearkombination von diesen dargestellt werden kann. Damit ist die bestimmte Summation vollständig algorithmisch lösbar und wir können die geschlossene Form angeben, falls eine solche existiert.

Falls die polynomialen Koeffizienten der Rekursionsgleichung einen sehr hohen Grad haben, ist zu beachten, dass die Komplexität des Petkovšek-Algorithmus in der Praxis sehr hoch ist, denn dann müssen sehr viele Möglichkeiten getestet werden.

Zum Abschluss betrachten wir noch ein Beispiel, in dem wir eine hypergeometrische Lösung für eine Summe mit einem hypergeometrischen Summanden suchen. Wir werden in diesem Beispiel die in MuPAD implementierten Befehle `zeilberger` und `rechyper` zu Hilfe nehmen.

Beispiel 4.3.4
Dazu greifen wir noch einmal Beispiel 3.2.6 auf. Gegeben sei also die Summe

$$s_n = \sum_{k=-\infty}^{\infty} \binom{3k+1}{k} \binom{3n-3k}{n-k} \frac{1}{3k+1},$$

wobei der Summand $F(n,k) = \binom{3k+1}{k}\binom{3n-3k}{n-k}\frac{1}{3k+1}$ einen endlichen Träger hat und hypergeometrisch bezüglich k und n ist. Wir wollen das Problem nun vollständig algorithmisch mit Hilfe der implementierten Befehle in MuPAD lösen. Wir werden daher zuerst den Befehl `zeilberger` auf den Summanden anwenden und erhalten die Rekursionsgleichung:

MuPAD

```
>> zeilberger(binomial(3*k+1,k)*binomial(3*(n-k),n-k)*1/(3*k+1),
   k,s(n));
```

Output

```
81*s(n)*(3*n + 2)*(3*n + 4)*(n + 1) - 12*s(n + 1)*(2*n + 3)*
(9*n^2 + 27*n + 22)+ 4*s(n + 2)*(2*n + 3)*(2*n + 5)*(n + 2) = 0
```

Wir erhalten eine Rekursion zweiter Ordnung. Diese können wir nicht mehr ohne den Petkovšek-Algorithmus lösen. Daher nutzen wir den von mir implementierten Befehl `rechyper(eqn,s(n))`, der diesen Algorithmus realisiert und alle hypergeometrischen Lösungen für die Rekursionsgleichung findet, falls solche existiert. In diesem Befehl ist `eqn` die zu berechnende Rekursionsgleichung und `s(n)` ist wieder der Name der unbekannten Summe. Es ergibt sich dann:

MuPAD

```
>> rechyper(81*s(n)*(3*n+2)*(3*n+4)*(n+1)-12*s(n+1)*(2*n+3)*
  (9*n^2+27*n+22)+4*s(n + 2)*(2*n + 3)*(2*n + 5)*(n + 2)= 0,s(n));
```

Output

```
{(27*(n+1))/(2*(2*n+3)),(3*(3*n+4)*(3*n+2))/(2*(2*n+3)*(n+1))}
```

Die Menge, die ausgegeben wird, enthält dann alle hypergeometrischen Lösungen für die Rekursionsgleichung. In diesem speziellen Beispiel erhalten wir zwei hypergeometrische Lösungen für die Summe s_n:

$$\frac{t_{n+1}}{t_n} = \frac{27(n+1)}{2(2n+3)} \quad \text{und} \quad \frac{u_{n+1}}{u_n} = \frac{3(3n+4)(3n+2)}{2(2n+3)(n+1)}.$$

Es ist zu beachten, dass wir nur die Quotienten der hypergeometrischen Terme erhalten.

Dieses Beispiel zum Abschluss des Abschnitts hat uns gezeigt, dass wir für eine bestimmte Summe vollständig algorithmisch entscheiden können, ob eine hypergeometrische Lösung existiert oder nicht. Mit den in MuPAD vorgestellten Befehlen erhalten wir vorerst nur eine Menge der hypergeometrischen Lösungen, die als Quotienten s_{n+1}/s_n ausgegeben werden. Im nächsten Abschnitt werden wir sehen, wie wir aus diesen eine Linearkombination erhalten können.

4.3.1 Linearkombinationen von hypergeometrischen Termen

Wir haben gesehen, wie wir hypergeometrische Lösungen mit Hilfe des Petkovšek-Algorithmus erhalten. Nun beschäftigen wir uns noch damit, wie wir mit Hilfe von ihnen eine geschlossene Form finden können.

Der Algorithmus von Petkovšek gibt uns eine Menge von hypergeometrischen Quotienten der Form $s_{n+1}^{(i)}/s_n^{(i)}$ zurück[1]. Wir betrachten zunächst jeden dieser Quotienten einzeln und faktorisieren den Nenner und den Zähler. Dann können wir die irreduziblen Faktoren durch Potenzen und Pochhammersymbole ausdrücken und wir erhalten den hypergeometrischen Term $s_n^{(i)}$ des jeweiligen Quotienten. Wir bilden eine Linearkombination der hypergeometrischen Terme $s_n^{(i)}$.

Als nächstes bestimmen wir die Ordnung der Rekursionsgleichung, diese nennen wir p. Wir bestimmen dann p Anfangswerte, indem wir $n = 0$ bis p in die eingegebne Summe $\sum_{k=-\infty}^{\infty} a_k$, wobei a_k die Form $F(n,k)$ hat, einsetzen. Dabei ist zu beachten, wenn der führende Koeffizient der Rekursionsgleichung eine Nullstelle bei einer negativen ganzen Zahl m oder Null hat, muss n um $-m+1$ geshiftet werden.

Die Linearkombination unserer hypergeometrischen Terme wird dann mit den Anfangswerten gleichgesetzt und indem wir in die Linearkombination das jeweilige n des Anfangswertes einsetzen, erhalten wir ein lineares Gleichungssystem mit p Gleichungen:

$$s_n = a_1 s_n^{(1)} + a_2 s_n^{(2)} + \ldots + a_l s_n^{(l)} \quad (n = 0, \ldots, p-1), \tag{4.18}$$

wobei p die Anzahl der Anfangswerte ist und l die Anzahl der verschiedenen hypergeometrischen Terme[2]. Indem wir das lineare Gleichungssystem lösen, erhalten wir die l Koeffizienten a_j $(j = 1, \ldots, l)$ und können dann die hypergeometrischen Terme als Linearkombination der Form (4.18) ausgeben. Damit haben wir eine geschlossene Form für die Summe s_n gefunden.

Es ist zu beachten, dass die Anzahl der Anfangswerte größer sein kann als die der hypergeometrischen Terme. Wir erhalten auch nur eine geschlossene Form, wenn für das entstandene Gleichungssystem eine Lösung existiert.

Als nächstes fassen wir den Algorithmus nochmals zusammen.

Algorithmus 4.3.5 *Der Algorithmus wandelt Quotienten vom Typ $s_{n+1}^{(i)}/s_n^{(i)}$ $(i = 1, \ldots, l)$, wobei l die Anzahl der hypergeometrischen Lösungen ist, in eine geschlossene Form*

$$s_n = a_1 s_n^{(1)} + a_2 s_n^{(2)} + \ldots + a_l s_n^{(l)}$$

um, falls eine solche existiert.

1. *Eingabe: Eine Menge von hypergeometrischen Quotienten der Form $s_{n+1}^{(i)}/s_n^{(i)}$ $(i = 1, \ldots, l)$.*

[1] Es gilt: $i = 1, \ldots, l$ und l ist die Anzahl der linear unabhängigen hypergeometrischen Terme.

[2] Diese Gleichung ist ohne Shifts, eventuell muss n um $-m+1$ geshiftet werden.

2. *Setze jedes* $\frac{s_{n+1}^{(i)}}{s_n^{(i)}} = \frac{u_n}{v_n}$.

3. *Führe eine rationale Faktorisierung für* u_n *und* v_n *durch.*

4. *Drücke die Faktoren von* u_n *und* v_n *durch Pochhammersymbole und Potenzfunktionen aus und gebe für jeden Quotienten ein* $s_n^{(i)}$ *aus (Algorithmus 1.1.6).*

5. *Bestimme* p *Anfangswerte für* s_n.

6. *Stelle das lineare Gleichungssystem für*

$$s_n = a_1 s_n^{(1)} + a_2 s_n^{(2)} + \ldots + a_l s_n^{(l)} \quad (n = 0, \ldots, p-1),$$

auf. Löse das Gleichungssystem und bestimme dabei die l *Koeffizienten* a_j $(j = 0, \ldots, l)$. *Falls das Gleichungssystem keine Lösung hat, gib alle* $s_n^{(i)}$ *als Menge aus.*

7. *Ausgabe: Gib die Linearkombination* $s_n = a_1 s_n^{(1)} + a_2 s_n^{(2)} + \ldots + a_l s_n^{(l)}$ *aus.*

Der Algorithmus findet immer eine geschlossene Form, falls das lineare Gleichungssystem für passende Anfangswerte eine eindeutige Lösung hat.

Beispiel 4.3.6
In Beispiel 4.3.4 haben wir nur die Mengen der hypergeometrischen Lösungen durch den Petkovšek-Algorithmus erhalten. Wir wollen nun eine geschlossene Form für die Summe

$$s_n = \sum_{k=-\infty}^{\infty} \binom{3k+1}{k} \binom{3n-3k}{n-k} \frac{1}{3k+1}$$

finden, falls eine existiert. Durch den Zeilberger-Algorithmus ergab sich die Rekursionsgleichung

$$\begin{aligned} &81s(n)(3n+2)(3n+4)(n+1) - 12s(n+1)(2n+3) \\ &(9n^2+27n+22) + 4s(n+2)(2n+3)(2n+5)(n+2) = 0. \end{aligned}$$

Der Petkovšek-Algorithmus liefert uns die Quotienten der hypergeometrischen Terme als Lösungen

$$\frac{t_{n+1}}{t_n} = \frac{27(n+1)}{2(2n+3)} \quad \text{und} \quad \frac{u_{n+1}}{u_n} = \frac{3(3n+4)(3n+2)}{2(2n+3)(n+1)}$$

für s_n. Da die Rekursionsgleichung die Ordnung zwei hat, bestimmen wir zwei Anfangswerte für s_n. Wir erhalten dann $s_0 = 1$ und $s_1 = 4$. Als nächstes wandeln wir die Quotienten mit Hilfe von Algorithmus 1.1.6 in hypergeometrische Terme um und es ergibt sich:

$$t_n = \left(\frac{27}{4}\right)^n \frac{(1)_n}{(\frac{3}{2})_n} \quad \text{und} \quad u_n = \left(\frac{27}{4}\right)^n \frac{(\frac{2}{3})_n (\frac{4}{3})_n}{(1)_n (\frac{3}{2})_n}.$$

Diese Terme können mit Hilfe des von mir implementierten Befehl `tohyper` bestimmt werden:

MuPAD

```
>> tohyper((27*(n + 1))/(2*(2*n + 3)),n);
>> tohyper((3*(3*n + 4)*(3*n + 2))/(2*(2*n + 3)*(n + 1)),n);
```

Output

```
((27/4)^n*pochhammer(1,n))/pochhammer(3/2,n)
((27/4)^n*pochhammer(2/3,n)*pochhammer(4/3,n))/
(pochhammer(1,n)*pochhammer(3/2,n))
```

Nun fassen wir die hypergeometrischen Terme als Linearkombination zusammen und setzen sie mit den Anfangswerten gleich:

$$\alpha_1 t_n + \alpha_2 u_n = s_n \quad \text{für } n = 0, 1.$$

Die Lösungen des Gleichungssystems sind: $\alpha_1 = 0$ und $\alpha_2 = 1$. Damit erhalten wir für s_n die geschlossene Form:

$$s_n = \left(\frac{27}{4}\right)^n \frac{(\frac{2}{3})_n (\frac{4}{3})_n}{(1)_n (\frac{3}{2})_n}.$$

Diese Lösung können wir auch durch die in MuPAD implementierten Befehle erzeugen. Wir erhalten die Linearkombination durch:

MuPAD

```
>> rek:=81*s(n)*(3*n+2)*(3*n+4)*(n+1)-12*s(n+1)*(2*n+3)*
   (9*n^2+27*n+22)+4*s(n+2)*(2*n+3)*(2*n+5)*(n+2)=0:
>> F:=binomial(3*k+1,k)*binomial(3*(n-k),n-k)*1/(3*k+1):
>> t:=((27/4)^n*pochhammer(1,n))/pochhammer(3/2,n):
>> u:=((27/4)^n*pochhammer(2/3,n)*pochhammer(4/3,n))/
   (pochhammer(1,n)*pochhammer(3/2,n)):
>> lincomb(rek,F,[t,u],k,s(n));
```

Output

```
((27/4)^n*pochhammer(2/3,n)*pochhammer(4/3,n))/
  (pochhammer(1,n)*pochhammer(3/2,n))
```

Der Befehl `lincomb` erwartet als Eingaben: Die Rekursionsgleichung, den Summand der Summe, die hypergeometrischen Lösungen in einer Liste und die verwendeten Parameter k und $s(n)$. Im oben vorgeführten Beispiel haben wir die Eingabeterme nicht ausgeben lassen.

Die ganze beschriebene Prozedur kann in einem Befehl aufgerufen werden. Der Befehl `zeilbergersum` findet für einen Summanden, der ein hypergeometrischer Term bezüglich beider Variablen ist, eine geschlossene Form, falls eine solche existiert.

MuPAD

```
>> zeilbergersum(binomial(3*k+1,k)*binomial(3*(n-k),n-k)*1/(3*k+1),
   k,s(n));
```

Output

```
((27/4)^n*pochhammer(2/3,n)*pochhammer(4/3,n))/
(pochhammer(1,n)*pochhammer(3/2,n))
```

Wir überprüfen unser Resultat, indem wir es in die Rekursionsgleichung, welche wir durch den Zeilberger-Algorithmus erhalten haben, einsetzen. Dieses tun wir mit Hilfe von MuPAD:

MuPAD

```
>> u:=fp::unapply(((27/4)^n*pochhammer(2/3,n)*pochhammer(4/3,n))/
   (pochhammer(1,n)*pochhammer(3/2,n)),n):
>> factor(expand(eval(subs(rek,s=u))));
```

Output

```
0=0
```

Nachdem wir nun gesehen haben, wie wir sowohl für die bestimmte als auch für die unbestimmte Summation vollkommen algorithmisch eine geschlossene Form finden können, werden wir uns im nächsten Kapitel mit der Summation von hypergeometrischen Term mit festen Summationsgrenzen beschäftigen.

Kapitel 5

Bestimmte Summation

5.1 Einführung

Wir haben in den letzten Kapiteln Algorithmen zur Summation von hypergeometrischen Termen kennen gelernt. In diesem ist eine Summe

$$S = \sum_{k=a}^{b} a_k \tag{5.1}$$

gegeben, wobei der Summand a_k ein hypergeometrischer Term ist und der Summationsindex ist k. Bei der unbestimmten Summation haben wir keine festen Summationsgrenzen eingesetzt, wir haben nur vorausgesetzt, dass wir nur über ganze Zahlen summieren. Im Fall des Zeilberger-Algorithmus haben wir immer einen Summanden $F(n, k)$ gegeben, der einen endlichen Träger hat. Daher haben wir eine endliche Summe mit natürlichen Grenzen, wenn wir über alle ganzen Zahlen summieren. In diesem Kapitel wollen wir nun untersuchen, wie wir algorithmisch summieren können, wenn die Summationsgrenzen feste ganze Zahlen sind.

5.2 Unbestimmte Summation mit festen Grenzen

In Kapitel 2 haben wir den Gosper-Algorithmus vorgestellt. Gegeben war eine Summe

$$S = \sum_{k=a}^{b} a_k,$$

wobei a_k ein hypergeometrischer Term ist. Unser Ziel war es eine hypergeometrische diskrete Stammfunktion s_k für a_k zu finden. Falls eine existiert, ist die bestimmte Summation von einer Grenze zur anderen trivial, dies haben wir schon in Abschnitt 2.1 gesehen. Denn wir erhalten eine Teleskopsumme

$$S = \sum_{k=a}^{b} a_k = (s_{a+1} - s_a) + (s_a - s_{a-1}) + \ldots + (s_{b+1} - s_b) = s_{b+1} - s_a,$$

wenn wir von a bis b summieren. Indem wir die beiden festen Grenzen einsetzen, erhalten wir die bestimmte Summe von a bis b und das Resultat ist eine

geschlossene Form. Zu beachten ist, dass die Rekursion eventuell nicht für alle Grenzen auswertbar ist. Beispielsweise ist $s_k = -1/k$ nicht für $k = 0$ auszuwerten. Dann müssen die Summationsgrenzen verschoben werden. Wir fassen den Algorithmus zusammen:

Algorithmus 5.2.1 (Gosper-Summation) *Der Algorithmus findet eine Lösung für die bestimmte Summe*

$$S = \sum_{k=a}^{b} a_k,$$

falls eine hypergeometrische Stammfunktion existiert, wobei der Summand a_k hypergeometrisch ist und die Summe von a bis b läuft.

1. *Eingabe: Ein hypergeometrischer Summand a_k und der Summationsindex k von a bis b.*
2. *Wende den Gosper-Algorithmus auf a_k an. Falls er keine Lösung findet, brich ab.*
3. *Berechne $S = s_{b+1} - s_a$. Falls die Rekursion für ein k aus dem Intervall $[a, b]$ nicht auswertbar ist, brich ab.*
4. *Ausgabe: Gib S aus.*

Diese Funktion ist auch in MuPAD implementiert, und zum Abschluss betrachten wir noch einige Beispiele mit MuPAD.

Beispiel 5.2.2
Gegeben sei die Summe

$$S = \sum_{k=1}^{n-1} a_k = \sum_{k=1}^{n-1} \frac{1}{(k+1)(k+2)}.$$

In diesem Beispiel betrachten wir, was passiert, wenn wir von $k = 1$ bis $n - 1$ summieren. Der Gosper-Algorithmus liefert uns das Ergebnis $s_k = -1/(k + 1)$ für die unbestimmte Summation. Der in MuPAD implementierte Befehl `gospersum(a,k=1..n-1)` findet dann die gewünschte Lösung, wobei `a` der hypergeometrische Summand ist und durch `k` die Summationsgrenzen festgelegt werden.

MuPAD

```
>> gospersum(1/(k+1)/(k+2),k=1..n-1);
```

Output

```
1/2 - 1/(n + 1)
```

Damit haben wir das gewünschte Resultat, eine geschlossene Form, für die bestimmte Summation von 1 bis $n - 1$ erhalten. Dabei ist die Unbekannte n eine ganze Zahl. Wir können genauso gut jede andere ganze Zahl eingeben, wie

zum Beispiel $n = 76$. Dann ergibt sich mit MuPAD das Resultat 19/39. Dieser Fall ist aber nicht wirklich interessant. In diesem Beispiel sehen wir auch sehr leicht, dass $s_{k+1} - s_k$ für die Summationsgrenze $k = -1$ nicht definiert ist und MuPAD gibt folgendes zurück:

MuPAD

```
>> setuserinfo(gospersum,1):
>> gospersum(1/(k+1)/(k+2),k=-1..n-1);
```

Output

```
Info: Die diskrete Stammfunktion ist an den Summationsgrenzen
nicht auswertbar!
FAIL
```

Der Gosper-Algorithmus für feste Summationsgrenzen ist in MuPAD auch in den Befehlen `gosper` und `lingosper` implementiert. Wir müssen nur für `k` feste Grenzen eingeben, wie zum Beispiel `k=0..n`. Außerdem können bei allen drei Befehlen die Zwischenergebnisse des Gosper-Algorithmus durch die Eingabe von `setuserinfo` abgefragt werden.

Beispiel 5.2.3
Wir betrachten noch einmal Beispiel 2.3.9, wo wir sehen, dass wir auch eine Linearkombination von hypergeometrischen Termen für bestimmte ganzzahlige Summationsgrenzen summieren können. Dazu summieren wir $a_k = 2^k + 1$ von 1 bis $n - 1$.

MuPAD

```
>> lingosper(2^k+1,k=1..n-1);
```

Output

```
n + 2^n - 3
```

Im nächsten Abschnitt werden wir uns mit der bestimmten Summation durch den Zeilberger-Algorithmus beschäftigen. Hier ist die Summation mit festen Grenzen nicht so trivial wie beim Gosper-Algorithmus, indem eine Teleskopsumme entsteht.

5.3 Bestimmte Summation mit festen Grenzen

Hier beschäftigen wir uns mit einem weitaus komplizierten Fall und zwar der bestimmten Summation, die wir in Kapitel 3 kennen gelernt haben. Die Summe liegt in der Form

$$s_n = \sum_{k=a}^{b} F(n,k)$$

vor, wobei $F(n,k)$ ein hypergeometrischer Term bezüglich beider Variablen ist und einen endlichen Träger hat. Diese Summe wollen wir für die bestimmten Summationsgrenzen $k = a$ bis b auswerten und dann in eine geschlossene Form bringen.

Wir wissen nach (3.13), dass

$$F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) = G(n,k+1) - G(n,k) \tag{5.2}$$

gilt. Wenn wir über die ganzen Zahlen aufsummieren, gilt nach Satz 3.1.3, dass die rechte Seite Null wird, da auf dieser eine Teleskopsumme entsteht. In Kapitel 3 haben wir dann unsere gewünschte Form erhalten, eine homogene Rekursion für den hypergeometrischen Summanden $F(n,k)$. Wir konnten im Folgenden die Koeffizienten der Rekursion bestimmen und haben eine homogene holonome Rekursionsgleichung gefunden. Da $F(n,k)$ einen endlichen Träger hatte, handelte es sich um eine endliche Summe mit natürlichen Grenzen.

In diesem Abschnitt stellen wir uns die Frage, ob es eine Rekursion für die bestimmte Summation gibt, falls wir von festen Summationsgrenzen ausgehen. Das Problem gestaltet sich dann wie folgt: Wir haben eine Summe s_n gegeben und wir können nach Kapitel 3 eine Gleichung der Form (5.2) finden. In diesem Fall summieren wir nicht über alle ganzen Zahlen und daher wird die rechte Seite nicht zu Null.

Aus Notationsgründen bezeichnen wir die linke Seite von (5.2) mit $Re(n)$[1]. Für die rechte Seite gilt: $G(n,k+1) - G(n,k)$, wenn wir dieses für feste Summationsgrenzen auswerten wollen, müssen wir für k die Grenzen einsetzen. Die rechte Seite ist dann nur noch abhängig von n, aber nicht mehr von k. Dadurch entsteht eine Differenz aus zwei hypergeometrischen Termen bezüglich n, diese sind nach Definition 2.3.1 ähnlich. Sie liegen also in einer Äquivalenzklasse und können als ein hypergeometrischer Term aufgefasst werden. Aus diesem Grund wählen wir als Bezeichnung für die rechte Seite[2] $\mathcal{G}(n)$. Weiter gilt, dass $G(n,k)$ ein rationales Vielfaches von $Re(n)$ ist, da $G(n,k) = R(n,k)Re(n)$ gilt[3].

Betrachten wir nun die Summe s_n für $k = a$ bis $k = b$, wobei der Summand $F(n,k)$ ein hypergeometrischer Term bezüglich beider Variablen ist und einen endlichen Träger hat. Es ergibt sich nach (5.2) die inhomogene Rekursionsgleichung

$$Re(n) = F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) = G(n,a+1) - G(n,b) = \mathcal{G}(n) \tag{5.3}$$

für s_n. Für $G(n,k)$ gilt nach (2.5) $G(n,k) = r_k/p_k f_{k-1} Re(n)$.

Da der Summand $F(n,k)$ einen endlichen Träger hat, existieren natürliche Grenzen. Wenn wir diese für k auf der rechten Seite einsetzen, ergibt sich wieder eine homogene Rekursionsgleichung, wie wir sie aus Kapitel 3 kennen, und wir können mit Hilfe des Petkovšek-Algorithmus die hypergeometrischen Lösungen für die Rekursion finden. Durch die natürlichen Summationsgrenzen erhalten wir ein Summationsintervall. Sobald wir ein größeres Intervall einsetzen, das heißt, die untere Grenze a ist kleiner als die natürliche unter Grenze und die

[1] Es handelt sich um eine Rekursionsgleichung.
[2] Es handelt sich um einen hypergeometrischen Term bezüglich n.
[3] In Kapitel 3 haben wir $Re(n)$ mit a_k bezeichnet.

obere Grenze b ist größer als die natürliche obere, erhalten wir mit $\mathcal{G}(n) = 0$ wieder eine homogene Rekursionsgleichung.

Der kritische Fall, den wir in diesem Abschnitt genauer betrachten wollen, tritt ein, wenn wir Grenzen ($[a, b]$) einsetzen, die innerhalb des natürlichen Summationsintervalls liegen. Dann entsteht auf der rechten Seite ein hypergeometrischer Term bezüglich n, der ungleich Null ist, das heißt: $\mathcal{G}(n) \neq 0$.

Wir wollen nun genauer betrachten, wie wir die rechte Seite $\mathcal{G}(n)$ einer inhomogenen Rekursion für $F(n, k)$ bestimmen können. Die Summationsgrenzen haben dabei folgende Eigenschaften (Aus Notationsgründen und um die Abhängigkeit zu n zu zeigen, setzen wir: $a = a_n$ und $b = b_n$):

1. Es existiert ein $j \in (0, \ldots, J)$, so dass für alle $n \in \mathbb{N}_0$ und $\mathcal{A}_n = \{a_{n-j} \mid j = 0, \ldots, J\}$ stets $\max(\mathcal{A}_n) = a_{n-j} = \alpha_n$ ist.

2. Es existiert ein $j \in (0, \ldots, J)$, so dass für alle $n \in \mathbb{N}_0$ und $\mathcal{B}_n = \{b_{n-j} \mid j = 0, \ldots, J\}$ stets $\min(\mathcal{B}_n) = b_{n-j} = \beta_n$ ist.

Wir summieren nun Gleichung (5.2) über k von α_n bis β_n auf, wobei $F(n, k)$ und $G(n, k)$ auf dem entsprechenden Bereich wohldefiniert seien. Es ist zu beachten, dass $\sigma_j(n) \in K(n)$, dass heißt, sie sind rationale Funktionen in n. Haben diese Funktionen Polstellen, dann gilt die Rekursionsgleichung erst für hinreichend große n. Indem wir aufsummieren, erhalten wir:

$$\sum_{k=\alpha_n}^{\beta_n} \sum_{j=0}^{J} \sigma_j(n) F(n + j, k) = \sum_{k=\alpha_n}^{\beta_n} (G(n, k + 1) - G(n, k)).$$

Daraus ergibt sich dann:

$$\sum_{j=0}^{J} \sigma_j(n) \left(s(n + j) - \sum_{k=a_{n-l}}^{\alpha_n - 1} F(n + j, k) - \sum_{k=\beta_n+1}^{b_{n-j}} F(n + j, k) \right) = (G(n, \beta_n + 1) - G(n, \alpha_n)).$$

Als nächstes müssen wir die „Korrekturterme“ auf die rechte Seite bringen und erhalten dadurch eine inhomogene Rekursionsgleichung für die Summe s_n:

$$\sum_{j=0}^{J} \sigma_j(n) s(n + j) = G(n, \beta_n + 1) - G(n, \alpha_n) + \sum_{j=0}^{J} \left(\sum_{k=a_{n-l}}^{\alpha_n - 1} F(n + j, k) - \sum_{k=\beta_n+1}^{b_{n-j}} F(n + j, k) \right). \tag{5.4}$$

Bei der Implementierung der inhomogenen Rekursion ist folgendes zu beachten: Es sollte über den Summationsbereich k von α_n bis β_n summiert werden. Es ist theoretisch denkbar über k von $\min(\mathcal{A}_n)$ bis $\max(\mathcal{B}_n)$ zu summieren, aber das ist problematisch, da $F(n, k)$ (teilweise) außerhalb des Summationsbereiches nicht wohldefiniert ist. Ist dies der Fall, findet der Algorithmus keine Lösung, obwohl die Herleitung einer Rekursion die Summe s_n aus der Rekursion für den Summanden $F(n, k)$ möglich ist.

Dadurch haben wir eine inhomogene Rekursionsgleichung gefunden, diese können wir aber nicht mit Hilfe des Petkovšek-Algorithmus lösen, da dieser nur homogene Rekursionsgleichungen lösen kann. Aber wir können die inhomoge Rekursion in eine homogene umschreiben. Der einzige Nachteil ist, dass die Ordnung der homogenen Gleichung um 1 höher ist, als die der ursprünglichen.

Um die inhomogene Rekursionsgleichung (5.3) in eine homogene umzuschreiben, betrachten wir beide Seiten getrennt. Die linke Seite ist $Re(n)$ und die rechte ist ein hypergeometrischer Term bezüglich n. Wir bezeichnen sie weiter mit $\mathcal{G}(n)$ (und die Grenzen bezeichnen wir wieder mit a und b), das heißt

$$Re(n) = F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) \qquad \text{und}$$
$$\mathcal{G}(n) = G(n,b+1) - G(n,a).$$

Setzen wir auf beiden Seiten $n = n+1$ ein, erhalten wir die beiden Gleichungen

$$Re(n) \quad = \mathcal{G}(n) \qquad \text{und} \tag{5.5}$$
$$Re(n+1) \; = \mathcal{G}(n+1). \tag{5.6}$$

Es gilt dann nach (5.5) und (5.6):

$$\frac{Re(n+1)}{Re(n)} = \frac{\mathcal{G}(n+1)}{\mathcal{G}(n)} = \frac{q_n}{p_n}. \tag{5.7}$$

Da auf der rechten Seite ein Quotient[4] des Termes $\mathcal{G}(n)$, der hypergeometrisch bezüglich n ist, auftritt, können wir diesen mit Hilfe von Algorithmus 1.1.4 vereinfachen und erhalten: $q_n/p_n \in K(n)$, wobei q_n und p_n Polynome aus $K[n]$ sind. Wenn wir (5.7) mit dem Hauptnenner multiplizieren, ergibt sich die homogene Rekursionsgleichung:

$$p_n Re(n+1) - q_n Re(n) = 0. \tag{5.8}$$

Damit haben wir eine homogene Rekursionsgleichung gefunden, deren Ordnung um 1 höher ist als die der ursprünglichen. Aber nun können wir, wie in Kapitel 4 angegeben, den Petkovšek-Algorithmus auf diese Rekursion anwenden und finden dadurch alle hypergeometrischen Lösungen für die Summe s_n, falls solche existieren. Aus diesen können wir dann eine geschlossene Form gewinnen.

Nun fassen wir den Algorithmus zur bestimmten Summation mit festen Grenzen noch mal zusammen.

Algorithmus 5.3.1 *Der Algorithmus findet alle hypergeometrischen Lösungen für die bestimmte Summation von $k = a$ bis $k = b$, falls solche existieren.*

1. *Eingabe:* $s_n = \sum_{k=a}^{b} F(n,k)$.

[4] $\mathcal{G}(n+1)/\mathcal{G}(n)$.

2. *Suche mit Hilfe des Zeilberger-Algorithmus die Rekursionsgleichung*

$$F(n,k) + \sum_{j=1}^{J} \sigma_j(n) F(n+j,k) = G(n,a+1) - G(n,b) = \mathcal{G}(n),$$

falls sie existiert, sonst brich ab.

3. *Falls die Rekursionsgleichung inhomogen ist, wende folgende Unterroutine an, sonst gehe zu Schritt* 4*:*
   (a) *Eingabe: Inhomogene Rekursion:* $Re(n) = \mathcal{G}(n)$.
   (b) *Bilde* $Re(n+1) = \mathcal{G}(n+1)$.
   (c) *Bilde den Quotienten* $\mathcal{G}(n+1)/\mathcal{G}(n)$. *Vereinfache ihn mit Hilfe von Algorithmus 1.1.4 zu* q_n/p_n, *wobei* q_n *und* p_n *Polynome aus* $K[n]$ *sind.*
   (d) *Ausgabe:* $p_n Re(n+1) - q_n Re(n) = 0$.

4. *Nutze den Petkovšek-Algorithmus, um alle hypergeometrischen Lösungen für die gegebene homogene Rekursion zu finden. Falls der Algorithmus keine findet, brich ab.*

5. *Forme die hypergeometrischen Lösung mit Hilfe von Algorithmus 4.3.3 in eine geschlossene Form für die Summe* s_n *um. Falls keine existiert, brich ab.*

6. *Ausgabe: Eine geschlossene Form für* s_n.

Wir haben gesehen, dass auch die bestimmte Summation mit vorgegebenen Grenzen komplett algorithmisch mit den vorgestellten Prozeduren gelöst werden kann. Die Summation mit festen Summationsgrenzen läuft sehr analog zur bestimmten Summation[5]. Der große Unterschied ist, dass wir durch den Zeilberger-Algorithmus in einigen Fällen eine inhomogene Rekursionsgleichung anstatt einer homogenen erhalten.

Wenn wir $G(n,k) = R(n,k)Re(n)$ gefunden haben, ist die Grenzwertbetrachtung der Summationsgrenzen und das Bestimmen der „Korrekturterme“ ein wenig komplizierter. Aber sobald wir diese gefunden haben, können wir die rechte Seite auswerten und erhalten eine inhomogene Rekursion. Die wir dann nur noch in eine homogene umwandeln müssen. Das Anwenden des Petkovšek-Algorithmus funktioniert dann analog zu Kapitel 4 und wir erlangen alle hypergeometrischen Lösungen für die Summe s_n, falls solche existieren. Aus diesen erhalten wir dann die geschlossene Form für s_n.

Zum Abschluss dieses Kapitels werden wir uns noch einige Beispiel ansehen, die wir mit Hilfe der in MuPAD implementierten Befehlen lösen wollen.

Beispiel 5.3.2
Zuerst greifen noch einmal Beispiel 3.2.5 auf, in dem wir die Summe

$$s_n = \sum_{k=-\infty}^{\infty} F(n,k) = \sum_{k=-\infty}^{\infty} \binom{n}{k}^2$$

[5]Kapitel 3 und 4.

gegeben haben. Diesmal wollen wir aber über feste Grenzen summieren. Wir wissen, dass die natürlichen Grenzen von $F(n,k)$ 0 und n sind. Zunächst summieren wir also von $k = 0$ bis n. Zu dieser Berechnung werden wir MuPAD verwenden, denn im implementierten Befehl `zeilberger` besteht die Möglichkeit statt `k` auch die Summationsgrenzen `k=0..n` einzugeben.

```
MuPAD
>> zeilberger(binomial(n,k)^2,k=0..n,s(n));
Output
2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = 0
```

Es zeigt sich, dass wir eine homogene holonome Rekursionsgleichung erhalten. Diese ergibt sich auch, wenn wir ein größeres Summationsintervall als das natürliche wählen:

```
MuPAD
>> zeilberger(binomial(n,k)^2,k=-2..n+1,s(n));
Output
2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = 0
```

Die Erkenntnisse aus diesem Abschnitt haben sich bestätigt und wir erhalten die homogene Rekursionsgleichung

$$2s(n)(2n+1) - s(n+1)(n+1) = 0,$$

falls wir über die natürlichen Grenzen (bzw. über ein größeres Summationsintervall als das natürliche) summieren. Wir erhalten eine Rekursionsgleichung erster Ordnung. Für diese brauchen wir nicht den Petkovšek-Algorithmus aus Kapitel 4 zu verwenden, denn wir können den hypergeometrischen Quotienten für s_n sofort ablesen:

$$\frac{s_{n+1}}{s_n} = \frac{(4n+2)}{(n+1)}$$

Indem wir nun den Quotienten s_{n+1}/s_n mit Hilfe von Algorithmus 1.1.6 in einen hypergeometrischen Term s_n umformen, erhalten wir:

```
MuPAD
>> tohyper(1/(n + 1)*(4*n + 2),n);
Output
(4^n*pochhammer(1/2,n))/pochhammer(1,n)
```

Dieser Befehl schreibt die hypergeometrischen Quotienten der Form s_{n+1}/s_n mit Hilfe von Algorithmus 1.1.6 in einen hypergeometrischen Term s_n um. Mit dem Anfangswert $s_0 = 1$ erhalten wir die geschlossene Form:

$$s_n = 4^n \frac{(1/2)_n}{(1)_n} = \frac{(2n)!}{(n!)^2}.$$

Wir überprüfen dieses Ergebnis, indem wir es in die Rekursionsgleichung einsetzen:

MuPAD

```
>> u:=fp::unapply((2*n)!/n!^2,n):
>> factor(expand(eval(subs(2*s(n)*(2*n+1)-s(n+1)*(n+1)=0,s=u))));
```

Output

```
0=0
```

Nun werden wir uns dem interessanteren Teil dieses Beispiels widmen: Wie verhält sich s_n, wenn wir über Grenzen summieren, die innerhalb der natürlichen liegen? Dazu summieren wir von $k = 1$ bis $n - 1$. Dies werden wir mit Hilfe von MuPAD untersuchen.

MuPAD

```
>> zeilberger(binomial(n,k)^2,k=1..n-1,s(n));
```

Output

```
2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = - 6*n - 2
```

In diesem Fall erhalten wir die inhomogene Rekursionsgleichung

$$2(2n+1)s(n) - s(n+1)(n+1) = -6n - 2 \tag{5.9}$$

als Ausgabe des Zeilberger-Algorithmus. Um die hypergeometrischen Lösungen für diese zu finden, müssen wir sie in eine homogene Rekursionsgleichung umwandeln. Dies erreichen wir, indem wir in (5.9) n durch $n + 1$ substituieren. Dann ergeben sich die beiden Gleichungen

$$\begin{aligned} &2(2n+1)s(n) - s(n+1)(n+1) = -6n - 2 \qquad \text{und} \\ &2(2n+3)s(n+1) - s(n+2)(n+2) = -6n - 8. \end{aligned}$$

Als nächstes bilden wir den Quotienten der beiden rechten Seiten und vereinfachen ihn. Dann ergibt sich nach dem Algorithmus 5.3.1 $p_n = 3n + 1$ und $q_n = 3n + 4$. Nun setzen wir diese beiden in Gleichung (5.8) ein, wobei $Re(n)$ und $Re(n+1)$ jeweils die linken Seiten der beiden oberen Gleichungen sind. Es ergibt sich dann die homogene Gleichung

$$-(3n+1)(n+2)s(n+2) + (15n^2+29n+10)s(n+1) - (3n+4)2(2n+1)s(n) = 0.$$

Wir sehen, dass wir eine homogene holonome Rekursionsgleichung erhalten haben, deren Ordnung um 1 höher ist als die der Ausgangsgleichung. In MuPAD wandelt der implementierte Befehl `inhom2hom(Re,s(n))` eine inhomogene Rekursionsgleichung in eine homogene um, wobei `Re` die Rekursionsgleichung und `s(n)` die Unbekannte repräsentieren.

MuPAD

```
>> inhom2hom(2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = -6*n-2,s(n));
```

Output

```
s(n + 1)*(15*n^2 + 29*n + 10) - 2*s(n)*(2*n + 1)*(3*n + 4)-
s(n + 2)*(n + 2)*(3*n + 1) = 0
```

Da wir nun eine homogene Rekursionsgleichung vorliegen haben, können wir auf diese den Petkovšek-Algorithmus anwenden und wir erhalten alle Quotienten der hypergeometrischen Terme als Lösung, falls solche existieren. Auch dieses werden wir wieder mit MuPAD vornehmen:

MuPAD

```
>> rechyper(s(n + 1)*(15*n^2 + 29*n + 10) - 2*s(n)*(2*n + 1)*
   (3*n + 4)-s(n + 2)*(n + 2)*(3*n + 1) = 0,s(n));
```

Output

```
{1, 1/(n + 1)*(4*n + 2)}
```

Damit sehen wir, dass wir für diese Rekursion, beziehungsweise für die Summe s_n, zwei Quotienten von hypergeometrischen Termen als Lösung finden:

$$\frac{t_{n+1}}{t_n} = 1 \quad \text{und} \quad \frac{u_{n+1}}{u_n} = \frac{(4n+2)}{(n+1)}.$$

Indem wir sie durch Potenzfunktionen und Pochhammersymbole ausdrücken (nach Algorithmus 1.1.6), erhalten wir die hypergeometrischen Terme:

$$t_n = 1^n \quad \text{und} \quad u_n = 4^n \frac{(1/2)_n}{(1)_n}.$$

Nun bestimmen wir noch mit Hilfe von MuPAD die Linearkombination der beiden hypergeometrischen Terme:

MuPAD

```
>> rek:=s(n+1)*(15*n^2+29*n+10)-2*s(n)*(2*n+1)*(3*n+4)
  -s(n+2)*(3*n+1)*(n+2)=0:
>> F:=binomial(n,k)^2:
>> h:=[1,(4^n*pochhammer(1/2,n))/pochhammer(1,n)]:
>> lincomb(rek,F,h,k=1..n-1,s(n));
```

Output

```
(4^n*pochhammer(1/2,n))/pochhammer(1,n) - 2
```

Die Funktion `lincomb` bestimmt bei Eingabe der hypergeometrischen Terme, der Rekursion, dem Summanden von s_n und den Summationsgrenzen die Linearkombination der hypergeometrischen Terme. Die geschlossene Form ist die Linearkombination:

$$s_n = 4^n \frac{(1/2)_n}{(1)_n} - 2 = \frac{(2n)!}{(n!)^2} - 2.$$

Der Befehl `zeilbergersum` bestimmt für eine Summe s_n sofort die geschlossene Form, falls solch eine existiert und gibt durch `setuserinfo` die Zwischenergebnisse der Rechnung aus:

MuPAD

```
>> setuserinfo(zeilbergersum,2):
>> zeilbergersum(binomial(n,k)^2,k=1..n-1,s(n));
```

Output

```
Info: REK:=2*s(n)*(2*n + 1) - s(n + 1)*(n + 1) = - 6*n - 2
Info: Hypergeometrische Qutienten:={2*(2*n + 1)/(n + 1), 1}
Info: Hypergeometrische Terme
s_n:=[1, 4^n/pochhammer(1,n)*pochhammer(1/2,n)]

(4^n*pochhammer(1/2,n))/pochhammer(1,n) - 2
```

Bei diesem Beispiel und im Allgemeinen ist zu beachten, dass die ersten Anfangswerte gleich Null sein können. Daher wählen wir als ersten Anfangswert den ersten der ungleich Null ist. In diesem Beispiel haben wir die Anfangswerte $s_2 = 4$ und $s_3 = 18$ gewählt. Diese Anfangswerte und die dazugehörige Linearkombination bestimmen wir nun noch mit Hilfe von MuPAD:

MuPAD

```
>> s2:=eval(subs(binomial(n,1)^2,n=2));
>> s3:=eval(subs(binomial(n,1)^2+binomial(n,2)^2,n=3));
```

Output

```
4
18
```

MuPAD

```
>> rek1:=a*(4^n*pochhammer(1/2,n))/pochhammer(1,n)+b:
>> lsg:=op(solve([subs(rek1,n=2)=4,subs(rek1,n=3)=18],{a,b}));
>> subs(rek1,lsg);
```

Output

```
[a = 1, b = -2]
(4^n*pochhammer(1/2,n))/pochhammer(1,n) - 2
```

Nach diesem Kapitel sind wir in der Lage sowohl für die unbestimmte als auch für die bestimmte Summation vollkommen algorithmisch zu entscheiden, ob eine geschlossene Form existiert oder nicht. Im nächsten Kapitel werden wir die ganzen kennen gelernten Algorithmen zusammenfassen und damit einen allgemeinen Summationsalgorithmus für Summen mit einem hypergeometrischen Summanden erhalten.

Kapitel 6

Summationsalgorithmus

6.1 Zusammenfassung des Summationsalgorithmus

In diesem Kapitel werden wir nun die Algorithmen zusammenfassen, die wir in den ersten Kapiteln kennen gelernt haben. Unser Ziel ist es eine geschlossene Form für eine Summe s_n zu finden. Hierzu werden wir zuerst ein Beispiel betrachten und nachher mit Hilfe dieses Beispiels den Algorithmus zusammenfassen.

Beispiel 6.1.1
Wir werden ein sehr komplexes Beispiel betrachten, um alle Schritte des Algorithmus aufzuzeigen. Gegeben ist die Summe

$$S_n = \sum_{k=2}^{n-1} a_k = \sum_{k=2}^{n-1} \left(\binom{n}{k} + 2^k + \frac{1}{k} + 1 \right),$$

wobei der Summand a_k eine Linearkombination von hypergeometrischen Termen ist. Im Folgenden werden wir die kennen gelernten Implementierungen aus MuPAD benutzen, um Teilergebnisse zu erzielen.

Zunächst werden wir den linearisierten Gosper-Algorithmus von $k = 2$ bis $n - 1$ auf den Term a_k anwenden. Dieser schreibt die Terme der Linearkombination so um, dass wir eine Linearkombination von paarweise verschiedenen Äquivalenzklassen erhalten. Die gospersummierbaren werden als Linearkombination von hypergeometrischen Stammfunktionen ausgegeben und die anderen als nicht vereinfachte Stammfunktionen, wobei diese als Summe von $k = 2$ bis $n - 1$ dargestellt werden.

MuPAD

```
>> lingosper(binomial(n,k)+2^k +1/k+1,k=2..n-1);
```

Output

```
sum(1/k, k = 2..n - 1) + sum(binomial(n, k), k = 2..n - 1)
+ 2^n - 6 + n
```

Wir sehen, dass der Term 2^k eine hypergeometrische Stammfunktion hat und summiert von 2 bis $n - 1$ ergibt sich für ihn die geschlossene Form $2^n - 4$.

Der konstante Term 1 ist gospersummierbar und wir erhalten $n-2$, wenn wir von $k=2$ bis $n-1$ summieren. Der Term $1/k$ ist nicht gospersummierbar und er wird als nicht vereinfachte Stammfunktion als Summe von 2 bis $n-1$ dargestellt. Da diese Summe nur abhängig von k ist, können wir a priori sagen, dass der Zeilberger-Algorithmus für diese Summe keine Rekursionsgleichung finden wird. Daher können wir diese Summe vorerst außer acht lassen.

Die andere Summe, in der $a_k = \binom{n}{k}$ ist, hat mit n einen weiteren Parameter. Indem wir diesen zu Hilfe nehmen, können wir $a_k = F(n,k)$ setzen und den Zeilberger-Algorithmus auf $\sum_{k=2}^{n-1} F(n,k)$ anwenden.

MuPAD

```
>> zeilberger(binomial(n,k),k=2..n-1,s(n));
```

Output

```
2*s(n) - s(n + 1) = - n - 1
```

Es ergibt sich eine inhomogene Rekursionsgleichung der Ordnung 1, diese schreiben wir in eine homogene der Ordnung 2 um, damit wir den Petkovšek-Algorithmus auf sie anwenden können.

MuPAD

```
>> hom:=inhom2hom(2*s(n) - s(n + 1) = -n - 1, s(n));
```

Output

```
s(n + 1)*(3*n + 4) - 2*s(n)*(n + 2) - s(n + 2)*(n + 1) = 0
```

Auf diese können wir nun den Petkovšek-Algorithmus anwenden, der uns die Quotienten der hypergeometrischen Terme ausgibt, falls solche existieren.

MuPAD

```
>> rechyper(hom, s(n));
```

Output

```
{2, 1/(n + 2)*(n + 3)}
```

Wir erhalten diese zwei hypergeometrischen Quotienten:

$$\frac{t_{n+1}}{t_n} = 2 \quad \text{und} \quad \frac{u_{n+1}}{u_n} = \frac{n+3}{n+2}.$$

Diese formen wir nun zu hypergeometrischen Termen um:

$$t_n = 2^n \quad \text{und} \quad u_n = \frac{(3)_n}{(2)_n} = \frac{2+n}{2}.$$

Nun können wir die Linearkombination der hypergeometrischen Terme bestimmen:

MuPAD

```
>> rek:=s(n + 1)*(3*n + 4) - s(n + 2)*(n + 1) - 2*s(n)*(n + 2) = 0:
>> F:=binomial(n,k):
>> h:=[2^n,(pochhammer(3,n))/pochhammer(2,n)]:
```

```
>> lincomb(rek,F,h,k=2..n-1,s(n));
```

Output

```
2^n - (2*pochhammer(3,n))/pochhammer(2,n)
```

Damit haben wir jeden Term der Linearkombination, die wir durch den linearisierten Gosper-Algorithmus erhalten haben, ausgewertet. Nun müssen wir nur noch die Resultate als Linearkombination zusammenschreiben und es ergibt sich die geschlossene Form für s_n:

$$s_n = 2^n - \frac{2(3_n)}{(2)_n} - 6 + 2^n + n + \sum_{k=2}^{n-1}\left(\frac{1}{k}\right) = (2^{n+1} - 8) + \sum_{k=2}^{n-1}\left(\frac{1}{k}\right).$$

Dieses Resultat hätten wir auch direkt mit dem in MuPAD implementierten Befehl `summe` erhalten können. In dieser Prozedur ist der folgende Algorithmus 6.1.2 implementiert.

MuPAD

```
>> summe(binomial(n,k)+2^k +1/k+1,k=2..n-1);
```

Output

```
n + sum(1/k, k = 2..n - 1) - (2*pochhammer(3,n))/pochhammer(2,n)
+ 2*2^n - 6
```

Bei diesem Befehl ist der dritte Parameter intern auf `s(n)` gesetzt, daher braucht man ihn nicht eingeben. Aber damit kann der Befehl den Zeilberger-Algorithmus nur für ein $F(n,k)$ mit $n \in \mathbb{Z}$ als eine weitere diskrete Variable aufrufen. Durch eine Eingabe des dritten Parameters kann dies geändert werden.

Damit erhalten wir eine Linearkombination von hypergeometrischen Termen, diese sind vereinfachte hypergeometrische oder nicht vereinfachte Stammfunktionen. In unserem Beispiel ist $1/k$ der einzige Term, der als nicht vereinfachte Stammfunktion vorliegt. Das heißt aber nicht, dass er nicht durch einen anderen Algorithmus in eine „einfache Form“ gebracht werden kann, diese ist aber nach unserer Definition keine geschlossene.

Wir fassen nun den im Beispiel kennen gelernten Algorithmus zusammen:

Algorithmus 6.1.2 (Summationsalgorithmus) *Gegeben ist*

$$S_n = \sum_{k=a}^{b} a_k,$$

wobei der Eingabeterm a_k eine Linearkombination von Termen ist, die bezüglich der Summationsvariable k hypergeometrisch sind. Der Algorithmus findet für die Summe eine geschlossene Form, falls solch eine existiert. Falls er keine geschlossene Form findet, gibt er eine Linearkombination aus nicht vereinfachten Stammfunktionen und hypergeometrischen Termen zurück. Die nicht vereinfachten werden als Summen über k dargestellt.

1. *Eingabe: Eine Linearkombination von hypergeometrischen Termen* $\sum_{j=1}^{p} a_k^{(j)}$ *und eine Summationsvariable* k *(Falls ein zweiter Parameter benötigt wird, muss dieser als* $s(n)$ *eingegeben werden).*

2. *Wende den linearisierten Gosper-Algorithmus 2.3.7 auf* $\sum_{j=1}^{p} a_k^{(j)}$ *an. Dieser gibt eine Linearkombination von hypergeometrischen Stammfunktionen und Summen von hypergeometrischen Termen* $\sum_{j=1}^{q} s_k^{(j)} + \sum_{j=1}^{r} \sum_k b_k^{(j)}$ *aus.*

3. *Schreibe die verschiedenen Terme in Listen um:*

   (a) *Schreibe alle hypergeometrischen Stammfunktionen in eine Liste:*

   $$Lsg = \{s_k^{(j)}\} \ (j = 1, \ldots, q).$$

   (b) *Schreibe alle anderen Terme, die wir in Form von Summen erhalten haben, in eine Liste:*

   $$Nlsg = \{\sum_k b_k^{(j)}\} \ (j = 1, \ldots, r).$$

4. *Wende auf jeden einzelnen Summanden* $b_k^{(j)}$ *der Liste* $\{\sum_k b_k^{(j)}\}$ $(j = 1, \ldots, r)$*, die Unterroutine Zeilbergersumme an:*

   (a) *Falls* $b_k^{(j)}$ *nur abhänging von* k *ist, brich ab.*

   (b) *Wähle eine zweite Variable neben* k *aus* $b_k^{(j)}$ *und wende den Zeilberger-Algorithmus auf den Summanden* $b_k^{(j)}$ *an (Zeilberger-Algorithmus 3.2.3 oder 5.3.1). Falls er nicht erfolgreich ist, brich ab.*

   (c) *Falls die Ausgabe eine inhomogene Rekursionsgleichung ist, schreibe sie in eine homogene um.*

   (d) *Wende auf die homogene Rekursionsgleichung den Petkovšek-Algorithmus 4.3.3 an.*

   (e) *Falls dieser erfolgreich ist, schreibe die hypergeometrischen Lösungen in eine geschlossene Form* s_n *um und gib sie als Linearkombination aus (Algorithmus 4.3.5) und streiche* $\sum_k b_k^{(j)}$ *aus der Liste* $Nlsg$*.*

5. *Schreibe* s_n *in die Liste* Lsg*.*

6. *Schreibe die Einträge der Liste* Lsg *und* $NLsg$ *als Linearkombination und addiere sie zusammen.*

7. *Ausgabe: Eine geschlossene Form oder eine Linearkombination von hypergeometrischen Termen und nicht vereinfachten Stammfunktionen.*

Bei diesem Algorithmus ist zu beachten, dass er nur für die Summation mit festen Grenzen $k = a$ bis b Sinn macht. Denn wenn wir unbestimmt summieren, erhalten wir als Ausgabe eine Linearkombination, in der hypergeometrische Terme stehen, die abhängig von dem Summationsindex k sind.

In diesem Kapitel haben wir alle Algorithmen zusammengeschrieben, die wir in den vorangehenden Kapiteln vorgestellt haben. Damit haben wir die komplette hypergeometrische Summation abgeschlossen. Die vorgestellten Algorithmen finden für jeden hypergeometrischen Eingabeterm eine geschlossene Form, falls eine solche existiert.

Im letzten Kapitel werden wir zum Abschluss einen Ausblick geben und Optimierungsvorschläge für die implementierten Algorithmen vorstellen.

Kapitel 7

Zusammenfassung und Ausblick

In dem letzten Kapitel wollen wir noch einmal alle wichtigen Ergebnisse zusammenfassen und einen Ausblick geben, welche Teile der Algorithmen verbessert werden können.

Wir haben uns mit der algorithmischen Summation von hypergeometrischen Termen beschäftigt. Unser Ziel war es, die Summe in eine geschlossene Form zu bringen, das heißt, sie als Linearkombination von einem oder mehreren hypergeometrischen Termen auszudrücken.

Die vorgestellten Algorithmen sind ein sehr mächtiges Instrument der algorithmischen Summation. Der Gosper-Algorithmus ist nicht nur interessant, weil er geschlossene Formen finden kann, falls eine solche existiert. Er spielt durch die Einbettung in den Zeilberger-Algorithmus auch eine sehr große Rolle bei der unbestimmten Summation. Der Petkovšek-Algorithmus findet als Anschluss an den Zeilberger-Algorithmus die geschlossene Form der entstandenen Rekursionsgleichung, falls eine solche existiert.

Zuerst haben wir gesehen, dass für eine Summe mit dem Summationsindex k und einem Summanden, der hypergeometrisch bezüglich k ist, mit Hilfe des Gosper-Algorithmus entschieden werden kann, ob eine hypergeometrische Stammfunktion existiert oder nicht. Wenn wir eine solche Stammfunktion gefunden haben, war die bestimmte Summation über k für geeignete Summationsgrenzen trivial. Da die hypergeometrische diskrete Stammfunktion ein rationales Vielfaches des Summanden a_k ist, kann es passieren, dass für einige Werte k die rationale Funktion, die mit a_k multipliziert wird, nicht definiert ist.

Der Gosper-Algorithmus ist nicht abgeschlossen bezüglich der Addition. Aber wir können eine Linearkombination von hypergeometrischen Termen in Äquivalenzklassen einteilen, so dass eine Linearkombination von paarweise verschiedenen ähnlichen hypergeometrischen Termen entsteht. Auf diese Terme können wir dann den Gosper-Algorithmus anwenden und erhalten eine Linearkombination, die aus vereinfachten hypergeometrischen und nicht vereinfachten Stammfunktionen besteht. Besteht diese nur aus vereinfachten hypergeometrischen Stammfunktionen, also nur aus hypergeometrischen Termen, haben wir eine geschlossene Form gefunden.

Danach betrachten wir Summen, die abhängig von der Summationsvariablen $k \in \mathbb{Z}$ und einer weiteren diskreten Variablen $n \in \mathbb{Z}$ sind und einen endlichen Träger haben. Somit betrachten wir Summen, die nur endlich viele Summanden $F(n, k)$ haben, die ungleich Null sind. Der Summand ist hypergeometrisch bezüglich beider Variablen. In diesem Fall können wir die Summe nicht sofort in eine geschlossene Form umwandeln. Aber wir sind in der Lage eine Rekursion für diese Summe zu finden, die unabhängig von der Summationsvariablen k ist. Für die Summation über Summationsgrenzen, die außerhalb der natürlichen liegen, finden wir eine homogene Rekursionsgleichung mit Hilfe des Zeilberger-Algorithmus. Falls wir Summationsgrenzen wählen, die innerhalb der natürlichen Grenzen liegen, erhalten wir eine inhomogene Rekursionsgleichung für die Summe s_n.

Es ist erwähnenswert, dass der Zeilberger-Algorithmus nur in wenigen Fällen eine Rekursion höherer Ordnung erzeugt, obwohl eine niedrigerer Ordnung existiert.

Um für diese Rekursion eine geschlossene Form zu finden, müssen wir den Petkovšek-Algorithmus auf die durch den Zeilberger-Algorithmus entstandene Rekursion anwenden. Dieser findet für eine homogene Rekursion als Lösung alle Quotienten der hypergeometrischen Terme, falls solche existieren. Falls wir eine inhomogene Rekursion erhalten, schreiben wir diese in eine homogene um. Der Grad der Ordnung erhöht sich dabei um 1.

Die durch den Petkovšek-Algorithmus erhaltenen hypergeometrischen Terme können wir dann als Linearkombination ausdrücken. Dadurch können wir auch eine geschlossene Form für die bestimmte Summation finden.

Zum Abschluss haben wir noch alle Algorithmen zusammengefasst. Mit Hilfe dieses Algorithmus kann jede Summe, die einen Summanden hat, der aus einer Linearkombination von hypergeometrischen Termen besteht, in eine Linearkombination von hypergeometrischen Termen und nicht vereinfachten Stammfunktionen gebracht werden. Eine geschlossene Form erhalten wir, wenn die Linearkombination nur aus hypergeometrischen Termen besteht.

Im Verlaufe der Arbeit haben wir gesehen, dass die Summation von hypergeometrischen Termen vollständig algorithmisch entschieden werden kann. Wir haben alle Algorithmen vorgestellt, um mit Hilfe von mechanischen Prozeduren systematisch eine Antwort zu erhalten.

Wir werden nun noch einige Schwächen der Algorithmen aufdecken und Verbesserungsvorschläge angeben.

Zuerst betrachten wir die bestimmte Summation. Mit Hilfe des Zeilberger-Algorithmus können wir eine holonome Rekursionsgleichung für eine Summe s_n finden. Bei der Implementierung dieses Algorithmus haben wir intern eine Schranke für die Ordnung J eingebaut. Die Ordnung ist auf 5 gesetzt, das heißt, der Algorithmus wird 5 mal durchlaufen und wenn er dann nicht erfolgreich ist, bricht er ab und gibt aus, dass die Ordnung nicht groß genug ist. Der Nutzer kann sie aber durch Eingabe eines vierten Parameters in den Befehl `zeilberger` manuell verändern und neu setzen.

Wir haben in Abschnitt 3.2.1 uns mit der Frage beschäftigt, ob der Zeilberger-Algorithmus terminiert ([WZ92] und [GKP89]) und gesehen, dass für „zulässige“ hypergeometrische Terme $F(n,k)$ eine Schranke J a priori bestimmt werden kann. Wenn diese Prozedur in den Zeilberger-Algorithmus eingebaut wird, kann er vollständig algorithmisch operieren und bei einer Eingabe eines Summanden und der zugehörigen Variablen findet er eine Rekursion für die Summe, falls eine solche existiert.

Aber bei der Berechnung von Rekursionen $J > 5$, werden die Ausdrücke so komplex, dass eine effiziente algorithmische Bestimmung nicht mehr möglich ist. Die Berechnung wird sehr zeitaufwendig und die resultierenden Rekursionen sehr komplex.

Wenn wir eine homogene Rekursion mit Hilfe des Zeilberger-Algorithmus erhalten haben, wenden wir den Petkovšek-Algorithmus auf diese an und erhalten alle hypergeometrischen Lösungen für diese Rekursionsgleichung. Dieser Algorithmus ist sehr langsam, weil der Algorithmus zur Bestimmung der Polynomlösung exponentiell (bezüglich des Grades der Eingabepolynome) oft aufrufen werden muss. Dieser Algorithmus gibt die richtigen Ergebnisse aus, aber es gibt einen viel effizienteren Algorithmus, der die gleiche Fragestellung behandelt. Der Algorithmus ist von van Hoeij und ist in [CvH06] zu finden. Er bestimmt auch alle hypergeometrischen Lösungen einer homogenen Rekursionsgleichung. Falls wir eine inhomogene Rekursion durch den Zeilberger-Algorithmus erhalten, müssen wir diese immer erst noch in eine homogene umformen. Aber der van Hoeij-Algorithmus ist viel effizienter im Lösen von Rekursionsgleichungen.

In den drei großen Algorithmen von Gosper, Zeilberger und Petkovšek läuft es am Ende immer auf das Lösen eines linearen Gleichungssystems hinaus. In den implementierten Algorithmen werden immer die eingebauten `solve`-Prozeduren verwendet. Diese sind aber in unserem Fall nicht sehr effizient. Denn wir wissen, dass wir ein lineares Gleichungssystem lösen müssen und dafür gibt es effizientere Möglichkeiten, die nur dazu da sind, um lineare Gleichungssysteme zu lösen. Indem wir unser Wissen ausnutzen, dass es sich um ein lineares System handelt und eine Prozedur einbauen, die speziell lineare Gleichungssysteme löst, können die Algorithmen beschleunigt werden und dadurch werden sie effizienter.

Der implementierte Zeilberger-Algorithmus ist für den inhomogenen Fall sehr langsam. Das liegt einerseits daran, dass die Grenzwertberechnung, die durchgeführt werden muss, sehr lange dauert. Der andere Grund ist, dass ich die Terme mit Hilfe des `simpcomb` Befehls in Gammafunktion umschreibe und diese Terme haben nicht die „einfachste“ Darstellung bezüglich der Gammafunktionen, daher muss ich auf diese den Befehl `simplify` anwenden. Dieser ist einer der zeitaufwendigsten Verinfachungsbefehle. Aber eine vollständige Vereinfachung ist durch andere Befehle nicht gewährleistet. Das Problem ist, wenn die Gammafunktionen nicht in der „einfachsten“ Form vorliegen, findet der Befehl `limit` nicht den richtigen Grenzwert. Daher kann der Algorithmus durch eine verbesserte Vereinfachungsprozedur beschleunigt werden.

Zur Vereinfachung ist noch anzumerken: Einige meiner implementierten Befehle geben Pochhammersymbole aus, diese sind nicht bis zum letzten verein-

facht (beispielsweise: $(1)_n = n!$) und könnten daher noch mit einigen Sonderregeln vereinfacht werden.

In den implementierten Prozeduren arbeiten wir mit vielen Polynomen. Wir können die Algorithmen beschleunigen, indem wir alle polynomiellen Ausdrücke mit `poly(f)` in Polynome des Kern-Domains `DOM_POLY` umwandeln.

Zum Abschluss ist noch zu erwähnen, dass die von mir implementierten Befehle größtenteils mit Hilfe der Entwicklerversion MuPAD Pro 4.5 entwickelt worden sind. Daher ist es möglich, dass einige Befehle in MuPAD Pro 4.0 nicht fehlerfrei laufen. Ein Beispiel hierfür sind die Pochhammersymbole, die erst ab Version 4.5 unterstützt werden.

Literaturverzeichnis

[Böi98] H. Böing. Theorie und Anwendungen zur q-hypergeometrischen Summation. Diplomarbeit, Freie Universität Berlin, 1998.

[CvH06] T. Cluzeau and M. van Hoeij. Computing hypergeometric solutions of linear recurrence equations. *Applicable Algebra in Engineering, Communication and Computing*, 17:83–115, 2006.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashink. *Concrete mathematics.* Addison Wesley, Reading, MA, 1989.

[Gos78] R. W. Gosper Jr. *Desicion procedure for indefinite hypergeometric summation.* Proc. Natl. Acad. Sci. USA 75, 1978.

[Koe97] W. Koepf. *Hypergeometric Summation.* Advanced Lectures in Mathematics. Vieweg, 1997.

[Koe06] W. Koepf. *Computeralgebra. Eine algorithmisch orientierte Einführung.* Springer, 2006.

[Koo93] T. H. Koornwinder. On zeilberger's algorithm and its q-analogue: a rigorous description. *J. of Comput. and Appl. Math.*, 48:91–111, 1993.

[MW94] Y. K. Man and F. J. Wright. Fast polynomial dispersion computation and its application to indefinite summation. *Proc. of ISSAC 94*, 1994.

[Pet92] M. Petkovsek. Hypergeometric solutions of linear recurrences with polynomial coefficients. *J. Symbolic Computation*, 14:243–264, 1992.

[PS95] P. Paule and M. Schorn. A mathematica version of zeilberger's algorithm for proving binomial coefficient identities. *J. Symbolic Computation*, 20:673–698, 1995.

[PWZ97] M. Petkovsek, H. S. Wilf, and D. Zeilberger. *A=B.* B&T, April 1997.

[Ris70] R. H. Risch. The solution of the problem of integration in finite terms. *Bull. Amer. Math. Soc.*, 76:605–608, 1970.

[WZ90] H. S. Wilf and D. A. Zeilberger. Rational functions certify combinatorial identies. *J. Amer. Math. Soc.*, 3:147–158, 1990.

[WZ92] H. S. Wilf and D. A. Zeilberger. An algorithmic proof theory for hypergeometric (ordinary and "q") multisum/integral identities. *Inventiones Mathematics*, 108:575–633, 1992.

[Zei90] D. A. Zeilberger. A fast algorithm for proving terminating hypergeometric identities. *Disrete Math.*, 80:207–211, 1990.

[Zei91] Doron A. Zeilberger. The methode of creative telescoping. *J. Symbolic Computation*, 11:195–204, 1991.