Diplomarbeit

Automatische Berechnung von Grenzwerten und Implementierung in Mathematica

von
Udo Richter

18. Mai 2005

Betreuer: Prof. Dr. Wolfram Koepf
Universität Kassel

Bis zur Unendlichkeit und noch viel weiter...

Vorwort

Grenzwertberechnung ist ein unbeliebtes Gebiet der Mathematik. Jeder Schüler hasst es. Das liegt daran, dass es kein universelles Kochrezept gibt, das einen automatisch zur Lösung führt. Statt dessen muss man verschiedenste Ansätze daraufhin überprüfen, ob sie einen einer Lösung näher bringen. Computeralgebra leidet unter dem gleichen Problem, denn Computer lieben Kochrezepte ebenfalls. Entsprechend haben manche Computeralgebrasysteme auch heute noch starke Probleme mit Grenzwerten.

1996 stellte Dominik Gruntz in seiner Dissertation [Gru96] „On Computing Limits in a Symbolic Manipulation System“ einen Algorithmus vor, der eine Vielzahl komplexer Grenzwertaufgaben souverän und schnell lösen kann und der dennoch durch seine Einfachheit und Überschaubarkeit besticht. Ziel dieser Diplomarbeit ist es, den Algorithmus von Dominik Gruntz vorzustellen und im Computeralgebrasystem Mathematica zu implementieren.

Das erste Kapitel führt in die Problematik der Grenzwertberechnung ein und zeigt Ansätze, wie auch Computeralgebrasysteme automatisiert Grenzwerte berechnen können.

Das zweite Kapitel soll eine Übersicht über den MrvLimit-Algorithmus von Dominik Gruntz vermitteln, ohne jedoch alle Details schon vollständig zu erklären.

Das dritte Kapitel führt in die algebraischen Grundlagen des Hardykörpers ein und entwickelt präzise Werkzeuge zum Vergleich vom Wachstum von Funktionen. Dieses Kapitel ist das am stärksten mathematisch geprägte Kapitel.

Das vierte Kapitel erklärt den Algorithmus schließlich in allen Details und begründet, warum der Algorithmus tatsächlich seine Aufgabe vollenden kann. Dieses Kapitel wendet sich eher an Informatiker.

Das fünfte Kapitel widmet sich der Implementierung in Mathematica. Einerseits wird in die Benutzung des MrvLimit-Pakets eingeführt, andererseits zeigt ein ausführlicher Blick hinter die Kulissen, wie der Algorithmus in Mathematica umgesetzt wurde.

Das sechste Kapitel widmet sich schließlich der Anwendung und zeigt anhand von Beispielen die Benutzung des Pakets. Vergleiche zu Mathematicas Limit-Funktion und Auslotungen der Grenzen des Algorithmus runden das Kapitel ab.

Der Text dieser Diplomarbeit steht in Druck- und Bildschirmversion unter
http://urichter.cjb.net/MrvLimit/ zum Download bereit, ebenso das Mathematica-Paket MrvLimit sowie die dazu gehörigen Quelltexte.

Web: http://urichter.cjb.net/MrvLimit/
Mail: udo_richter@gmx.de

Inhaltsverzeichnis

1. Einführung

Die Grenzwertberechnung ist eine häufig auftretende Problemstellung in der Mathematik, die zugleich auch zu den schwierigsten Aufgaben zählt: Es gibt kein universelles Lösungsschema, nur verschiedene Ansätze, die in bestimmten Situationen zum Erfolg führen können. Gleichzeitig muss man bei der Lösung sehr sorgfältig vorgehen, da die Intuition hier oft trügerisch ist und die präzisen Randbedingungen meist entsprechend komplex ausfallen.

Seit es Rechenmaschinen gibt, existiert daher auch der Wunsch, diese ‚lästige' Aufgabe den Rechenmaschinen zu überlassen. Leider stellen sich dabei die meisten Computer als nicht viel besser heraus, als ihre menschlichen Pendants.

1.1. Numerische Ansätze

Jeder Schüler hat schon einmal mit seinem Taschenrechner ausprobiert, welches grobe Ergebnis eine Grenzwertaufgabe wohl hat. Wie vollkommen falsch man damit liegen kann, verdeutlicht folgendes Beispiel:

$$\lim_{x\to 0} x^{1-\ln\ln\ln\ln\frac{1}{x}}$$

Tippt man diese Funktion in einen normalen Taschenrechner mit $x = 10^{-99}$ ein, so ergibt sich ein ‚Grenzwert' von $1.12851 \cdot 10^{-47}$, was mehr als stark auf einen Grenzwert von 0 hindeutet. Jeder Schüler und viele Nicht-Mathematiker wären jetzt bereits überzeugt.

Moderne Computeralgebraprogramme bieten natürlich eine höhere Rechengenauigkeit und können diese Funktion auch noch für $x = 10^{-100000}$ berechnen. Damit ergibt sich der Funktionswert $1.0669 \cdot 10^{-7836}$. Also ist der Grenzwert doch 0?

Lassen wir zum Abschluss noch einmal Mathematicas NLimit-Funktion ihr Glück versuchen:

```
In[1]:= Needs["NumericalMath`NLimit`"]

In[2]:= NLimit[x^(1 - Log[Log[Log[Log[1/x]]]]), x → 0,
          Scale → 1/10]
Out[2]= 8.13782 × 10^-7 + 8.85665 × 10^-7 i
```

Auch Mathematica scheint also eher an 0 als Lösung zu denken.

Erfahrenere Mathematiker riechen natürlich schon die Gefahr: Der Logarithmus ist dafür bekannt, ein sehr langsames, aber dennoch beständiges Wachstum gegen Unendlich zu besitzen. Die Verkettung mehrerer Logarithmen verstärkt diesen Effekt derart, dass das Wachstum von $\frac{1}{x}$ für $x \to 0$ extrem lange unterdrückt wird. Erst bei etwa $10^{-579203}$ fängt die Funktion wieder an zu wachsen, bei etwa $10^{-1656521}$ wächst der verkettete Logarithmus über 1 und das Vorzeichen im Exponenten kippt. Danach beginnt das langsame Wachstum gegen Unendlich.

Numerische Verfahren sind trotz moderner Mathematiksoftware immer nur zu einer begrenzten Genauigkeit in der Lage. Doch selbst einfach wirkende Aufgaben können diese Genauigkeit bereits überfordern.

1.2. Grenzwert einer Funktion

Natürlich kennt die Mathematik eine eindeutige Definition des Grenzwertes. Hier noch mal die Kompaktfassung einer der übersichtlicheren Definitionen:

Definition 1.1. ***Grenzwert einer Funktion.*** *Eine Funktion $f(x)$ mit Definitionsbereich $D(f)$ hat für $x \to x_0$ den Grenzwert a,*

$$\lim_{x \to x_0} f(x) = a,$$

wenn x_0 ein Häufungspunkt von $D(f)$ ist, und wenn für jedes $\epsilon > 0$ ein δ existiert, so dass für alle x aus $D(f) \cap U_\delta$ die Ungleichung $|f(x) - a| < \epsilon$ gilt. Dabei ist $U_\delta = \{x \in \mathbb{R} \mid 0 < |x - x_0| < \delta\}$, wenn $x_0 \in \mathbb{R}$ ist und $U_\delta = \{x \in \mathbb{R} \mid x > \delta\}$, wenn $x_0 = \infty$ ist. Außerdem gilt $\lim_{x \to -\infty} f(x) := \lim_{x \to \infty} f(-x)$.

Definition 1.2. ***Grenzwert im Unendlichen.*** *Eine Funktion $f(x)$ mit Definitionsbereich $D(f)$ hat für $x \to x_0$ den Grenzwert $+\infty$,*

$$\lim_{x \to x_0} f(x) = +\infty,$$

wenn x_0 ein Häufungspunkt von $D(f)$ ist und wenn für jedes $C \in \mathbb{R}$ ein δ existiert, so dass für alle x aus $D(f) \cap U_\delta$ die Ungleichung $f(x) > C$ gilt. Dabei ist $U_\delta := \{x \in \mathbb{R} \mid 0 < |x - x_0| < \delta\}$, wenn $x_0 \in \mathbb{R}$ ist, und $U_\delta := \{x \in \mathbb{R} \mid x > \delta\}$, wenn $x_0 = \infty$ ist. Entsprechend gilt $\lim_{x \to x_0} f(x) = -\infty$, wenn $\lim_{x \to x_0} -f(x) = \infty$ gilt. Außerdem gilt wieder $\lim_{x \to -\infty} f(x) := \lim_{x \to \infty} f(-x)$.

Wenn das Definieren von Grenzwerten so einfach ist, wo ist dann das Problem? Das Problem ist, dass es sich nicht um eine konstruktive Definition handelt. Die Definition sagt uns nicht, wie wir den Grenzwert finden, und das Kriterium, das zur Überprüfung des Grenzwertes erforderlich ist, eignet sich eher für theoretische Beweise, als für praktische Tests.

1.3. Heuristische Lösungsverfahren

Dem Mathematiker stehen zur Grenzwertberechnung neben der reinen Definition auch große Tabellen mit fertig berechneten Grenzwerten für bestimmte Funktionen und ein Sortiment aus

Sätzen und Regeln zur Verfügung. Mit Erfahrung und Intuition lässt sich so meist eine Herleitung des Grenzwertes finden. Sucht man zum Beispiel

$$y_0 = \lim_{x \to x_0} f(x)$$

und kann $f(x)$ darstellen als $g(h(x))$, so kann der Grenzwert in zwei Schritten berechnet werden:

$$z_0 = \lim_{x \to x_0} h(x)$$

$$y_0 = \lim_{z \to z_0} g(z)$$

Das klappt gut, selbst wenn z_0 nicht endlich ist. Wichtig ist aber, dass g nur von z abhängt und nicht zusätzlich von x.

Eine weitere bekannte Regel ist die Regel von Bernoulli / De l'Hôpital. Sie wird angewendet, wenn in einem Ausdruck ein Term folgender Form auftritt:

$$\lim_{x \to x_0} \frac{f(x)}{g(x)}$$

der sich beim Grenzübergang verhält wie $\frac{0}{0}$ oder $\frac{\infty}{\infty}$. Das Grenzwertverhalten lässt sich dann unter bestimmten Bedingungen durch das Grenzverhalten von $\frac{f'(x)}{g'(x)}$ ersetzen. Allerdings ist die Gültigkeit dieser Regel an einige Randbedingungen geknüpft, die präzise eingehalten werden müssen. Insbesondere die Bedingung, dass $g'(x) \neq 0$ in einer ganzen Umgebung von x_0 gelten muss, ist für ein Computeralgebrasystem schwer zu überprüfen, zumal schon eine harmlos wirkende Vereinfachung diese Störung verdecken kann.

Eine weitere Technik ist z.B. das Squeeze-Theorem: Dazu muss man die Funktion wie in einer Art Trichter zwischen zwei anderen Funktionen einsperren. Haben beide Funktionen einen gemeinsamen Grenzwert, so muss auch die eigentliche Funktion in diesem Grenzwert gefangen sein.

Es gibt viele solche Tricks zum Umgang mit Grenzwerten. All diesen Verfahren gemein ist jedoch die Tatsache, dass sie keine klare Strategie besitzen. Die Anwendung einer bestimmten Regel kann zur Lösung führen, kann aber auch eine Sackgasse sein. Ein Mathematiker kann das Ergebnis intuitiv einschätzen und beurteilen, ob dieser Schritt sinnvoll ist. Ein Computerprogramm verirrt sich dagegen leicht im Labyrinth der Möglichkeiten.

1.4. Potenzreihenentwicklung

Für Algorithmen der Computeralgebra ist ein Umdenken erforderlich. Klare Strategien sind nötig, die zuverlässig zum Ziel führen. Dafür kann man die Vorteile der Computeralgebra nutzen: Komplexe Terme und extrem lange Rechenwege sind kein Problem.

Um den Grenzwert einer rationalen Funktion zu berechnen, bietet sich eine Potenzreihenentwicklung als Hilfsmittel an: Jede rationale Funktion lässt sich als Laurentreihe darstellen:

$$f(x) \in \mathbb{R}(x) \qquad \Rightarrow \qquad f(x) = \sum_{k=k_0}^{\infty} a_k \left(x - x_0\right)^k \qquad \left(k_0 \in \mathbb{Z}\right)$$

(Man beachte, dass im Gegensatz zur Taylorreihe k_0 auch negativ sein kann.) Für Funktionen $f \neq 0$ kann außerdem $a_{k_0} \neq 0$ als Normalform angenommen werden.
Ist man nun an $\lim_{x \to x_0} f(x)$ interessiert, kann das Ergebnis an k_0 und a_{k_0} abgelesen werden:

$$\begin{array}{lcll}
\lim_{x \to x_0} f(x) & = & 0 & \text{, wenn } k_0 > 0 \text{ ist,} \\
\lim_{x \to x_0} f(x) & = & a_0 & \text{, wenn } k_0 = 0 \text{ ist,} \\
\lim_{x \to x_0^+} f(x) & = & \mathrm{Sign}(a_{k_0}) \cdot \infty & \text{, wenn } k_0 < 0 \text{ ist,} \\
\lim_{x \to x_0^-} f(x) & = & \mathrm{Sign}(a_{k_0}) \cdot \infty & \text{, wenn } k_0 < 0 \text{ und } k_0 \text{ gerade ist,} \\
\lim_{x \to x_0^-} f(x) & = & -\mathrm{Sign}(a_{k_0}) \cdot \infty & \text{, wenn } k_0 < 0 \text{ und } k_0 \text{ ungerade ist.}
\end{array}$$

Schauen wir uns das in der Praxis an:

```
In[1]:= f = (x^3 - 2x^2 - 9x + 18)/(x^3 + x);

        Series[f, {x, 0, 3}]
```

$$\text{Out[1]=} \quad \frac{18}{x} - 9 - 20\,x + 10\,x^2 + 20\,x^3 + O[x]^4$$

Am Ergebnis kann man direkt ablesen, dass das Grenzverhalten für $x \to 0$ genau dem von $18\frac{1}{x}$ entspricht. Dieser Term der Reihe dominiert das Wachstum, alle positiven Potenzen liefern sogar einen Null-Beitrag für $x \to 0$.

Mit dem klassischen Verfahren zur Ermittlung von Taylorreihen kommt man hier allerdings nicht weit: Schon zur Ermittlung des ersten Terms der Taylorreihe benötigt man $f(x_0)$ und damit genau den Grenzwert, der eigentlich gesucht ist.

Computeralgebrasysteme sind allerdings in der Lage, Potenzreihenentwicklungen von Funktionen bis zu einem vorgegebenen Grad sehr effizient zu berechnen: Potenzreihen von Grundfunktionen werden fest integriert, und komplexere zusammengesetzte Funktionen werden aus den Potenzreihen ihrer Teilausdrücke durch Transformationen entwickelt. Ableitungen und Grenzwerte sind dafür (zum Glück) nicht mehr erforderlich.

Die Potenzreihenentwicklung ist daher für die Grenzwertberechnung ein attraktives Verfahren, das sich geradezu anbietet zur Implementierung in Computeralgebra-Systemen: Das Erzeugen einer Potenzreihe zu einer Funktion ist ein Standardverfahren, und das Ergebnis kann schnell daran abgelesen werden. Der Algorithmus terminiert immer und liefert immer ein korrektes Ergebnis ab. Wo also ist der Haken?

Betrachten wir die folgende Funktion:

$$f(x) = e^{-\frac{1}{x^2}}$$

Zur Ermittlung der zugehörigen Taylorreihe für $x_0 = 0$ benötigt man deren Ableitungen:

$$\begin{aligned} f'(x) &= \frac{2}{x^3} e^{-\frac{1}{x^2}} \\ f''(x) &= \left(\frac{4}{x^6} - \frac{6}{x^4}\right) e^{-\frac{1}{x^2}} \\ f'''(x) &= \left(\frac{8}{x^9} - \frac{36}{x^7} + \frac{24}{x^5}\right) e^{-\frac{1}{x^2}} \end{aligned}$$

Offensichtlich taucht die ursprüngliche Funktion als Faktor in ihren Ableitungen wieder auf, mit einem rationalen Ausdruck als Vorfaktor. Da das Wachstum der Exponentialfunktion jede rationale Funktion dominiert und die Funktion selbst gegen $e^{-\infty} = 0$ strebt, gilt für alle Ableitungen $f^{(k)}(0) = 0$, und damit gilt für die Taylorreihe ebenfalls $a_k = 0$ für alle k, was der Taylorreihe der 0-Funktion entspricht. Offensichtlich ist eine Potenzreihe also keine geeignete Repräsentation für diese Funktion.

Die Exponentialfunktion hat in diesem Fall eine essenzielle Singularität, denn sie hat eine Nullstelle, die so stark ist, dass $x^{-k} f(x)$ für kein k stark genug ist, um die Nullstelle zu ‚heben'. Entsprechend kann keine endliche Potenz der Potenzreihe die Funktion repräsentieren.

Nur Funktionen, deren Nullstellen und Unendlich-Stellen hebbar sind, d.h. durch Multiplizieren mit x^k verschwinden, lassen sich durch Potenzreihen sinnvoll darstellen. Trotzdem ist das Grenzverhalten der Exponentialfunktion überschaubar. Die Potenzreihen-Analyse lässt sich problemlos auf das Argument $-\frac{1}{x^2}$ anwenden, $\lim_{x\to 0} -\frac{1}{x^2}$ ergibt $-\infty$. Durch das Anwenden der Exponentialfunktion ergibt sich dann ein Grenzwert von 0.

Verfolgen wir die Strategie an einem komplexeren Beispiel weiter, diesmal für $x \to \infty$:

```
In[2]:=  f = (3 + 2 x^3 E^(2-x^2)) / (1 + 3 x^2 E^(5-x^2));
```

Diesmal treten zwei auf den ersten Blick verschiedene Exponential-Funktionen auf, die jedoch ein sehr ähnliches asymptotisches Grenzverhalten besitzen: $\frac{e^{2-x^2}}{e^{5-x^2}} = e^{-3}$.

Die Behandlung des Arguments der Exponentialfunktion fällt wieder leicht, denn es ist auch für den Computer leicht einzusehen, dass $\lim_{x\to\infty} e^{2-x^2} = 0$ ist, und dass der Grenzwert exponentiell gegen 0 strebt. Ersetzt man nun in der Funktion e^{2-x^2} durch ω, und e^{5-x^2} durch $e^3\omega$, ergibt sich folgende Funktion:

```
In[3]:=  f = f /. {E^(2-x^2) → ω, E^(5-x^2) → E^3 ω}

Out[3]=  (3 + 2 x^3 ω) / (1 + 3 E^3 x^2 ω)
```

In dieser Funktionsdarstellung strebt ω exponentiell gegen 0, wenn x linear gegen ∞ strebt.

Das Wachstumsverhalten von x können wir daher vorübergehend ignorieren, da es durch das Wachstumsverhalten von ω dominiert wird. Ein stärkeres Wachstum als das von ω tritt außerdem nicht auf. Daher können wir eine Potenzreihenentwicklung in ω, entwickelt in $\omega_0 = 0$, durchführen:

```
In[4]:= Series[f, {ω, 0, 2}]
Out[4]= 3 + (-9 e^3 x^2 + 2 x^3) ω + (27 e^6 x^4 - 6 e^3 x^5) ω^2 + O[ω]^3
```

Und wieder kann das Ergebnis direkt abgelesen werden: $\lim_{x\to\infty} f(x) = 3$. Alle anderen Terme bestehen aus einem polynomiell wachsenden Faktor und einer Potenz von ω. Da ω exponentiell wächst, dominiert ω das Wachstum der polynomiellen Faktoren, und da ω gegen 0 strebt, bleibt nur der 3-Term übrig.

Im Wesentlichen haben wir damit die Strategie des MrvLimit-Algorithmus bereits erreicht. Im nächsten Kapitel verschaffen wir uns daher einen ersten Überblick.

2. Überblick über den Algorithmus

Der Kern des MrvLimit-Algorithmus ist die Potenzreihenentwicklung. Da die Potenzreihenentwicklung jedoch nur mit polynomiellem Wachstum zuverlässig umgehen kann, muss die Funktion so angepasst werden, dass sie aus Sicht der Potenzreihenentwicklung nur polynomiell wächst. Langsameres Wachstum wird dazu vorübergehend ausgeblendet.

Bevor wir uns in Kapitel 3 mit den mathematischen Details intensiver beschäftigen, werfen wir aber schon mal einen Blick auf die grobe Strategie des MrvLimit-Algorithmus für eine Funktion $f(x)$.

(1) Transformation der allgemeinen Grenzwertaufgabe $\lim_{x \to x_0} f(x)$ in eine Grenzwertaufgabe der Form $\lim_{x \to \infty} f(x)$.

(2) Isolieren desjenigen Teilausdrucks der Funktion, der das größte asymptotische Wachstumsverhalten besitzt.

(3) Bestimmen aller Teilausdrücke, die das gleiche Wachstumsverhalten besitzen wie der ausgewählte Teilausdruck.

(4) Darstellen all dieser Teilausdrücke mit Hilfe einer einzelnen Variable ω, die das größte Wachstumsverhalten repräsentiert. Alle verbleibenden Terme müssen signifikant schwächeres Wachstum haben.

(5) Ermitteln der Potenzreihe in ω und Ermittlung des führenden Terms der Potenzreihe. Dabei kann x als konstant betrachtet werden.

(6) Falls erforderlich, weitere Analyse des führenden Terms durch rekursive Grenzwertberechnungen.

(7) Abschließende Bestimmung des Grenzwerts durch Analyse des führenden Terms.

Dieser Überblick ist natürlich sehr knapp ausgefallen. Deswegen werden im Folgenden ein paar Themen noch detaillierter besprochen.

2.1. Transformation

Um den Algorithmus überschaubar zu gestalten, werden grundsätzlich nur Grenzwerte für $x \to \infty$ betrachtet. Andere Grenzwerte lassen sich leicht durch Transformation ermitteln:

$$\begin{aligned} \lim_{x \to x_0^+} f(x) &= \lim_{x \to \infty} f\left(x_0 + \tfrac{1}{x}\right) \\ \lim_{x \to x_0^-} f(x) &= \lim_{x \to \infty} f\left(x_0 - \tfrac{1}{x}\right) \\ \lim_{x \to -\infty} f(x) &= \lim_{x \to \infty} f(-x) \end{aligned}$$

Beidseitige Grenzwerte löst man, indem der Grenzwert aus beiden Richtungen separat ermittelt und verglichen wird.

2.2. Wachstumsklassen

Wie in der Computeralgebra üblich, gehen wir davon aus, dass unsere zu analysierende Funktion explizit in Form eines Funktionsausdrucks in Baumstruktur vorliegt. Vorläufig begrenzen wir auch die Funktionsvielfalt auf Konstanten, die Variable x sowie die beliebige Anwendung der Grundrechenoperationen Addition, Subtraktion, Multiplikation, Division, Potenzfunktion, Exponentialfunktion und Logarithmus. Später wird dieses Grundgerüst um weitere Funktionen ergänzt, die sich in der Praxis als leicht behandelbar erweisen.

Liegt eine Funktion als Baumstruktur vor, ist es einfach, sämtliche Teilausdrücke zu betrachten. Diese Teilausdrücke müssen dann anhand ihrer asymptotischen Wachstumseigenschaften geordnet werden. Wie genau das erfolgt, wird Kapitel 3 zeigen. Es sei auch schon mal darauf hingewiesen, dass im Folgenden oft auch gegen Null strebendes Grenzverhalten der Einfachheit halber als Wachstum bezeichnet wird. Für unsere Fälle hat also sowohl e^x als auch e^{-x} für $x \to \infty$ exponentielles Wachstum, im zweiten Fall jedoch exponentielles Wachstum gegen 0.

Ist der am stärksten wachsenden Teilausdruck gefunden, kann er als ω zu einem Symbol substituiert werden, für die spätere Potenzreihenentwicklung. Vorher müssen aber alle Terme, die einen relevanten Beitrag zu dieser Potenzreihe liefern können, ebenfalls mittels ω dargestellt werden. Terme, deren Wachstum kleiner als jedes $\omega^n \quad (n \neq 0)$ sind, leisten keinen Beitrag zur Potenzreihe und werden vorerst als konstant betrachtet.

2.3. Bedingungen an das Wachstum

Sollte ω langsamer als exponentiell wachsen, kann es im Algorithmus zu Problemen kommen, die zu unendlichen Rekursionen führen. Der Algorithmus gestaltet sich dagegen erheblich einfacher, wenn angenommen werden kann, dass ω mindestens exponentielles Wachstum besitzt.

Abhilfe: Ersetzt man in *f(x)* jedes Vorkommen von *x* durch e^x, ändert sich der Grenzwert selbst nicht, nur das Wachstumsverhalten. Durch wiederholtes Ersetzen kann so jede Funktion (innerhalb unserer Einschränkung der Funktionsvielfalt, siehe oben) derart verschärft werden, dass sie mindestens exponentiell gegen Unendlich strebt.

Der Algorithmus soll später eine Potenzreihe in $\omega = 0$ bilden. Damit das funktioniert, muss auch der gewählte Teilausdruck ω für $x \to \infty$ gegen 0^+ streben. Zum Glück ist das kein kritisches Problem: Sollte $\omega \to \infty$ gelten, kann statt dessen $1/\omega$ verwendet werden. Der Fall $lim_{x\to\infty}\omega \in \mathbb{R}, lim_{x\to\infty}\omega \neq 0$ kann dank des Mindestwachstums zum Glück nicht mehr auftreten.

2.4. Ersetzung von Teilausdrücken

Beim Umschreiben der Funktion in eine Funktion in ω muss man sehr umsichtig vorgehen. Es ist durchaus möglich, dass ein Teilausdruck der höchsten Wachstumsklasse selbst einen weiteren Teilausdruck der selben Klasse beinhaltet. In dem Fall muss immer der größere Teilausdruck vor den kleineren Teilausdrücken behandelt werden.

Kritisch sind auch die nötigen Umformungen. An manchen Stellen ist es wünschenswert, dass Teilausdrücke nach dem Umschreiben vereinfacht werden, an anderen Stellen jedoch auch höchst problematisch: Durch manche Umformungen können erneut Teilausdrücke mit höherer Wachstumsklasse entstehen. Mathematicas fest eingebaute Funktion zur automatischen Vereinfachung wird sich dabei als sehr problematisch erweisen.

2.5. Potenzreihenermittlung

Nach der Isolation des stärksten Wachstums in ω wird die Potenzreihe der Funktion ermittelt. In Computeralgebra-Systemen wird üblicherweise die Potenzreihe durch konstruktive Verfahren aus der Struktur der Funktion aufgebaut, und nicht klassisch durch die Definition der Taylorreihe. Um den vollen Umfang der unterstützten Funktionen beizubehalten, ist eine Darstellung als Generalisierte Potenzreihe (Generalized Power Series) erforderlich. Eine Generalisierte Potenzreihe ist eine Potenzreihe der Form $\sum_{k=1}^{\infty} a_k x^{e_k}$, wobei die e_k eine streng monoton wachsende Folge von reellen Zahlen bilden, und $a_k \neq 0$ für Funktionen ungleich der Nullfunktion gilt. Es dürfen also durchaus auch irrationale Exponenten in der Potenzreihe auftreten.

Es existieren Algorithmen, die solche Potenzreihen als abgebrochene Potenzreihen mit Restterm ermitteln können. Dabei sind die $a_k x^{e_k}$ nur bis zu einem Summenindex n bekannt, und der Algorithmus kann bei Bedarf diesen Abbruch-Index weiter hinaus schieben. Die Unvollständigkeit der abgebrochenen Potenzreihe ist nicht kritisch, da der für die Analyse wichtige Term der Potenzreihe sowieso der führende Term ist. Leider existiert noch keine Implementierung eines solchen Algorithmus in Mathematica, so dass diese Diplomarbeit mit den eingeschränkten Mitteln des Series-Kommandos von Mathematica auskommen muss.

2.6. Analyse des führenden Terms

Im Allgemeinen hat der führende Term die Gestalt $C(x) \cdot \omega^n$, und $C(x)$ wächst dabei schwächer als jedes ω^k für $k \neq 0$.

Falls n sich als 0 ergibt, war die ursprüngliche Annahme, dass ω das Wachstum dominieren würde, falsch. Tatsächlich hat sich in diesem Fall das Wachstum aller mit ω vergleichbaren Terme gegenseitig aufgehoben, und der wirklich dominante Term ist in einer niedrigeren Wachstumsklasse zu finden. Durch die Potenzreihenentwicklung wurde die Wachstumsklasse von ω eliminiert, und der gesuchte Grenzwert kann nun durch weitere Analyse von $C(x)$ gefunden werden. Ist $n \neq 0$, ist das Ergebnis entweder 0 oder $\pm\infty$. Das Vorzeichen kann durch weiteres Analysieren von $C(x)$ gefunden werden.

Durch rekursives Abarbeiten wird so die Funktion jedes mal um eine Wachstumsklasse vereinfacht, bis letztlich $C(x)$ konstant ist. Zeichnet man diese Historie auf, gewinnt man einen guten Einblick in die mit einander konkurrierenden Wachstumsprozesse und kann sogar eine Asymptote zur Funktion bestimmen, die das Grenzverhalten der Funktion exakt nachbildet.

2.7. Null-Test

Im Verlauf des Algorithmus wird es wiederholt nötig sein, einen Term daraufhin zu überprüfen, ob er Null ist, oder welches Vorzeichen er besitzt. So ist das Ermitteln des führenden Terms der Potenzreihe zum Scheitern verurteilt, wenn die Funktion selbst die Null-Funktion ist: In diesem Fall gibt es keine von Null verschiedenen Terme in der Potenzreihe. Es genügt sogar bereits, wenn die Funktion in einer Umgebung um den Grenzübergangspunkt identisch der Nullfunktion ist.
Aber auch in anderen Teilen des Algorithmus treten solche Nulltests auf, z.B. beim Vergleichen des Wachstumsverhaltens verschiedener Funktionen.

Auf den ersten Blick scheint es einfach zu sein, zu überprüfen, ob eine Funktion identisch Null ist. In der Computeralgebra hat sich jedoch leider herausgestellt, dass dies ein sehr schwieriges, wenn nicht gar unlösbares Problem ist. [Har10] und [Sha04] beschäftigen sich eingehender mit dem Thema.

So ist zum Beispiel in unserem eingeschränkten Funktionsraum ein zweifelsfreier Nulltest theoretisch machbar: Es existieren Abschätzungen, wie viele Nullstellen eine solche Funktion maximal haben kann. (vgl. [Ric69] und [Mac80]) Findet man mehr Nullstellen, muss die Funktion insgesamt identisch Null sein. Leider ist der Aufwand zur Ermittlung dieser Abschätzung sehr groß, und auch die Anzahl der zu überprüfenden Nullstellen ist meist so groß, das das Verfahren in der Praxis zu langsam ist.

In dieser Diplomarbeit ist es daher auch wieder einfachen Mathematica-Abfragen überlassen, die Frage des Nulltests zu beantworten. Eine gewisse Fehlerwahrscheinlichkeit muss hier wieder in Kauf genommen werden.

2.8. Rekursionen

Auf dem Weg zur Berechnung des Grenzwerts wird es immer wieder erforderlich sein, andere Grenzwerte ebenfalls zu berechnen. Solche Grenzwertberechnungen sind z.B. nötig, wenn das Wachstumsverhalten von Termen analysiert wird. In solchen Situationen wird sich der MrvLimit-Algorithmus selbst rekursiv aufrufen. Dabei muss sorgfältig darauf geachtet werden, dass die rekursiv berechneten Grenzwertaufgaben immer einfacherer Natur sind, als der eigentlich zu berechnende Grenzwert. Andernfalls würde sich der Algorithmus immer wieder mit immer schwierigeren Problemen selbst aufrufen und niemals ein Ergebnis liefern. Solange aber die Grenzwertaufgaben zunehmend einfacher werden, wird der Algorithmus irgendwann bei den einfachsten Aufgaben ankommen und die Grenzwerte von x oder von Konstanten berechnen - dessen Lösungen direkt fest in den Algorithmus integriert werden können.

3. Mathematische Grundlagen

Nachdem wir nun eine grobe Vorstellung der Strategie des Algorithmus haben, wird es Zeit, einige Begriffe mathematisch präzise zu definieren: Wann wachsen zwei Funktionen gleich schnell, und wie kann schnelleres und langsameres Wachstum unterschieden werden? Wann ist das Wachstum eines Teilausdrucks dem Wachstum von ω ähnlich genug, um in der Potenzreihenentwicklung einen Beitrag zu leisten? Und wie können all diese Kriterien sinnvoll im Algorithmus geprüft werden?

Beginnen wir also mit dem Elementarsten. Dem Verhalten von Funktionen nahe Unendlich.

3.1. Hardykörper

Wir interessieren uns hauptsächlich für das Verhalten von Funktionen für $x \to \infty$. Daher macht es Sinn, Funktionen als äquivalent zu betrachten, wenn sie sich in einem Intervall $\left[x_0, \infty\right)$ nicht unterscheiden. Diese Äquivalenzrelation führt direkt zur Definition des Hardykörpers:

Sei $\mathcal{K}$ ein Körper von Funktionen $\left[x_0, \infty\right) \to \mathbb{R}, x_0 \in \mathbb{R}$, d.h. von Funktionen, die in einer Umgebung um Unendlich definiert sind.
Auf $\mathcal{K}$ sei die Äquivalenzrelation $\sim$ definiert wie folgt: Für $f, g \in \mathcal{K}$ gelte $f \sim g$ genau dann, wenn es ein $x_0 \in \mathbb{R}$ gibt, so dass $f(x) = g(x)$ für $x > x_0$ gilt. Damit bildet die Restklasse $\mathcal{K}/\sim$ mit den kanonischen Operatoren $+, \cdot$ einen Körper. Diese Äquivalenzklasse $f/\sim$ wird auch als Keim (germ) der Funktion f bezeichnet. Alle Funktionen einer Äquivalenzklasse sind identisch im Verhalten nahe Unendlich, wohingegen sie sich in jedem endlichen Intervall, sei es auch noch so nahe an Unendlich, unterscheiden können. Ein gemeinsames Verhalten aller Funktionen in einem konkreten Intervall $\left[x_0, \infty\right)$ gibt es in der Regel nicht.

Ist eine Funktion $f \in \mathcal{K}$ differenzierbar in einem Intervall $\left[x_0, \infty\right)$, so ist jede Funktion der Äquivalenzklasse ebenfalls in einem geeigneten Intervall differenzierbar, und die Ableitungen sind zu einander äquivalent. So kann auf $\mathcal{K}/\sim$ eine kanonische Differentiation definiert werden.

Definition 3.1. ***(Hardykörper) (vgl. [Har10])***
Ein Körper $\mathcal{H} \subseteq \mathcal{K}/\sim$ heißt Hardykörper, wenn er abgeschlossen ist bezüglich der Differentiation. $\mathcal{H}_0$ sei dabei die Menge $\mathcal{H} \setminus \{0\}$. Jede Funktion eines Hardykörpers ist damit stetig und beliebig oft differenzierbar in einer Umgebung um Unendlich.

Formal sei darauf hingewiesen: Im Folgenden wird oft von Funktionen $f \in \mathcal{H}$ gesprochen, obwohl natürlich eine beliebige Funktion f der Klasse $f_\sim \in \mathcal{H}$ gemeint ist. Insbesondere be-

ziehen sich Aussagen über Intervalle $[x_0, \infty)$ auf unterschiedliche Intervalle, je nach gewähltem Vertreter der Klasse, und nicht auf ein gemeinsames Intervall der gesamten Klasse.

Ist $\mathcal{H}$ ein Hardykörper und $f \in \mathcal{H}_0$, so existiert wegen der Körpereigenschaft $1/f$ und ist ebenfalls in einer Umgebung um Unendlich differenzierbar. Daher muss $f(x) \neq 0$ in einer ganzen Umgebung von Unendlich gelten.

Damit ist jede Funktion $f \in \mathcal{H}$ in einer geeigneten Umgebung um Unendlich immer ganz positiv, ganz negativ oder identisch 0. Da das gleiche für f' gilt, ist jede Funktion $f \in \mathcal{H}_0$ in einer Umgebung von Unendlich streng monoton, und für $f \in \mathcal{H}$ gilt: $\lim_{x\to\infty} f(x)$ existiert und ist entweder eine reellwertige Konstante oder $\pm\infty$.

$\mathbb{R}(x)$, der Körper der rationalen Funktionen, ist ein Hardykörper, und bildet man den Abschluss bezüglich der Funktionen $f \to \exp(f)$ und $f \to \log|f|$, so erhält man auch einen Hardykörper, genannt $\mathcal{L}$-Körper, der Körper der exp-log Funktionen. (vgl. [Har10])

3.2. Wachstumsklassen

Für die Analyse von Termausdrücken auf ihr Grenzwertverhalten benötigen wir einen Vergleichsmaßstab, um Funktionen nach ihrem Wachstumsverhalten zu ordnen. Im Folgenden werden gleich zwei Äquivalenzklasseneinteilungen eingeführt, um damit das Wachstum von Termen zu vergleichen. Die erste Klasseneinteilung wird dabei abgeschlossen bezüglich der Multiplikation mit Konstanten sein, die zweite wird abgeschlossen bezüglich der Potenzierung ihrer Funktionen sein und damit abgeschlossen bezüglich der Verkettung mit bestimmten rationalen Funktionen.

Definition 3.2.
Für $a, b \in \mathcal{H}_0$ gelte $a \approx b$ genau dann, wenn $\lim_{x\to\infty} a(x)/b(x) \in \mathbb{R} \setminus \{0\}$ ist. Für $a \in \mathcal{H}_0$ schreiben wir $v(a)$ für die Äquivalenzklasse von a und $\Upsilon = \{v(a) | a \in \mathcal{H}_0\}$ für die Menge aller Äquivalenzklassen.

Theorem 3.3. *(vgl. [Ros83], Th.4 und [Gru96] Th.3.4)*
Mit $v(a) + v(b) = v(ab)$ bildet $(\Upsilon, +)$ eine Abelsche Gruppe, die mittels

$$v(a) > v(b) :\Leftrightarrow \lim_{x\to\infty} a(x)/b(x) = 0$$

vollständig geordnet ist. Außerdem gilt für $a, b \in \mathcal{H}_0$:

(1) $v(1) = 0$.

(2) $v(a^{-1}) = -v(a)$.

(3) $v(a^n) = n \cdot v(a)$ $, n \in \mathbb{Z}$ *(mit der Multiplikation auf Υ definiert als Summierung).*

(4) $v(a) > 0$ *genau dann, wenn* $\lim_{x\to\infty} a(x) = 0$.

(5) $v(a) = 0$ *genau dann, wenn* $\lim_{x\to\infty} a(x) \in \mathbb{R} \setminus \{0\}$.

(6) $v(a) < 0$ *genau dann, wenn* $\lim_{x\to\infty} a(x) = \pm\infty$.

(7) *Wenn* $a + b \in \mathcal{H}_0$ *ist, dann gilt* $v(a + b) \geq \min(v(a), v(b))$.

(8) *Wenn* $a + b \in \mathcal{H}_0$ *und* $v(a) \neq v(b)$ *ist, dann gilt* $v(a + b) = \min(v(a), v(b))$.

(9) *Wenn* $a + b \in \mathcal{H}_0$ *und* $v(a + b) > \min(v(a), v(b))$ *ist, gilt* $\lim_{x\to\infty} a(x)/b(x) = -1$ *(und damit* $v(a) = v(b)$*).*

(10) *Wenn* $v(a) \neq 0 \neq v(b)$*, dann* $v(a) \geq v(b)$ *genau dann, wenn* $v(a') \geq v(b')$.

(11) *Wenn* $v(b) \neq 0$*, dann* $v(a) > v(b)$ *genau dann, wenn* $v(a') > v(b')$.

(12) *Wenn* $v(a) \neq 0 \neq v(b)$*, dann* $v(a) = v(b)$ *genau dann, wenn* $v(a') = v(b')$.

(1)-(9) folgen leicht aus der Definition. (10)-(12) folgen aus dem Satz von de l'Hôpital.

(7), (8) und (9) kann auf ganz $\mathcal{H}$ ausgedehnt werden mit der Definition $v(0) = +\infty$. □

Beachte: $v(x^2) < v(x) < v(1)$, was der intuitiven Bedeutung von ‚kleiner' nicht entspricht. (7) bedeutet umgangssprachlich also: a+b wächst langsamer oder gleich schnell, als die schneller wachsende der beiden Funktionen, a oder b. (8) beschreibt die Dominanz des stärkeren Wachstums, und (9) den Fall, dass sich das Wachstum von a und b gegenseitig aufhebt.

Zum Abschluss noch ein paar Beispiele für Wachstumsklassen und deren Anordnung:

$$\begin{aligned} &v(1/x) > v(1) > v(\ln x) > v\left((\ln x)^{10}\right) > v\left(x^{0.1}\right) > v(x) \\ &> v(x \ln x) > v\left(x(\ln x)^{10}\right) > v\left(x^{1.1}\right) > v\left(x^{10}\right) > v\left(e^{0.1x}\right) > v(e^x) > v\left(e^{10x}\right) \end{aligned}$$

Man sieht an diesen Beispielen bereits, dass die Funktionen $x, e^x, \ln x$ eigene kleine Gruppen bilden, die sich auch durch das Potenzieren mit beliebigen positiven Zahlen nicht überschneiden. Gemischte Terme wie $x(\ln x)^k$ können diese Gruppe unterbrechen und bilden darin gleich wieder eine eigene Gruppe.

Diese Einteilung in Wachstumsklassen ist noch ungeeignet, um das Wachstum von Termen jenseits von Unendlich ausreichend einschätzen zu können. Versuchen wir also, das Gruppenverhalten, das x^n von e^x trennt, durch einen weiteren Wachstumsbegriff zu erfassen.

3.3. Noch einmal Wachstumsklassen

Wir kommen zur zweiten Äquivalenzklasse. Ziel ist, die Äquivalenzklasse abgeschlossen bezüglich rationaler Operationen zu gestalten. Ist $f \in \mathcal{H}_0$, und $g(x) = \sum_{k=1}^{n} a_k x^k \quad (a_n \neq 0)$, so ergibt sich $v(g(f)) = n \cdot v(f)$. Das Potenzieren vervielfacht also die Wachstumsklasse, und damit kann kein Vielfaches von $v(x)$ jemals $v(e^x)$ erreichen. Deswegen zielt die nächste Definition darauf, Funktionen, die sich in der ersten Wachstumsklasse durch endliche Vielfache unterscheiden, zusammen zu fassen. Darüber hinaus ist es sinnvoll, das Wachstum von f und $1/f$ ebenfalls zu einer Klasse zusammen zu fassen.

Definition 3.4.
Zwei Funktionen $f, g \in \mathcal{H}_0$ sind in der gleichen Wachstumsklasse, geschrieben $f \asymp g$, genau dann, wenn $m, n \in \mathbb{N}$ existieren mit $m|v(f)| \geq |v(g)|$ und $n|v(g)| \geq |v(f)|$. Wir schreiben $\gamma(f)$ für die Äquivalenzklasse von f.
Außerdem gelte $f \succ g :\Leftrightarrow \gamma(f) > \gamma(g)$ genau dann, wenn $n|v(g)| < |v(f)|$ für alle $n \in \mathbb{N}$ gilt.

Theorem 3.5.
Für $f, g \in \mathcal{H}_0$ gilt:

(1) $f \succeq g \Leftrightarrow \exists n \in \mathbb{N} : n|v(f)| \geq |v(g)|$.

(2) $v(f) = v(g) \Rightarrow \gamma(f) = \gamma(g)$.

(3) $\gamma(f^n) = \gamma(f)$ *für alle* $n \in \mathbb{Z} \setminus \{0\}$.

(4) $\gamma(f) = \gamma(1) \Leftrightarrow v(f) = 0$.

(5) $\gamma(1) \leq \gamma(f)$ *für alle* $f \in \mathcal{H}_0$.

(6) Für $f, g \to +\infty$ gilt $f \asymp g$ genau dann, wenn $v(f^m/g) \leq 0$ und $v(g^n/f) \leq 0$ ist.

(7) $\gamma(fg) \leq \max\{\gamma(f), \gamma(g)\}$.

(8) Für $\gamma(f) \neq \gamma(g)$ gilt: $\gamma(fg) = \max\{\gamma(f), \gamma(g)\}$.

(1) - (5) folgen fast unmittelbar aus der Definition.
Zu (6): Es gilt $v(f) < 0, v(g) < 0$. Damit ist $v(f^m/g) = -m|v(f)| + |v(g)| \leq 0$ genau dann, wenn $m|v(f)| \geq |v(g)|$ gilt.

Zu (7): Sei o.B.d.A. $\gamma(f) \geq \gamma(g)$. Dann gibt es nach (1) ein $n \in \mathbb{N}$ mit $n|v(f)| \geq |v(g)|$. Damit folgt: $|v(fg)| = |v(f) + v(g)| \leq |v(f)| + |v(g)| \leq (n+1)|v(f)|$, und daraus folgt $\gamma(f) \geq \gamma(fg)$.

(8) beweist man am einfachsten unter Zuhilfenahme von Lemma 3.8 weiter unten. □

Definition 3.6. *(Wachstumsverhältnis)*
Sei $f, g \in \mathcal{H}_0$, $v(f) \neq 0 \neq v(g)$. Dann heißt der Wert

$$R(f, g) := \lim_{x \to +\infty} \frac{\ln|f(x)|}{\ln|g(x)|}$$

die Wachstumsverhältnis von f und g.

Dieser Wert eignet sich sehr gut, um das Wachstumsverhalten zweier Funktionen zu vergleichen, wie wir im folgenden Theorem sehen werden. Gleichzeitig ist die Funktion einfach genug zu berechnen, und damit sehr gut für Computeralgebrazwecke zu verwenden.

Theorem 3.7. *(vgl. [Gru96] Th. 3.5 und Lem. 3.6)*
Sei $f, g \in \mathcal{H}_0$, $v(f) \neq 0 \neq v(g)$. Dann gilt:

$$\begin{aligned} f \asymp g &\Leftrightarrow R(f,g) \in \mathbb{R} \setminus \{0\} \\ f \prec g &\Leftrightarrow R(f,g) = 0 \\ f \succ g &\Leftrightarrow R(f,g) = \pm\infty \end{aligned}$$

Wegen der Bedeutung dieses Zusammenhangs hier der Beweis:

Zunächst kann man o.B.d.A davon ausgehen, dass $f \geq 0, g \geq 0$ gilt, da alle Aussagen vorzeichenunabhängig sind. Weiterhin kann man von $f, g \to +\infty$ und damit von $v(f) < 0, v(g) < 0$ ausgehen: Falls $f \to 0$ oder $g \to 0$, kann der Beweis mit $1/f$ oder $1/g$ geführt werden.
Dann gilt folgende Äquivalenz:

$$\begin{aligned} & mv(f) < v(g) \\ & \Leftrightarrow v(f^m/g) < 0 \\ & \Leftrightarrow \lim f^m/g = +\infty \\ & \Leftrightarrow \lim e^{m \ln f - \ln g} = +\infty \\ & \Leftrightarrow \lim m \ln f - \ln g = +\infty \\ & \Leftrightarrow \lim \ln(g) \left(m \tfrac{\ln f}{\ln g} - 1 \right) = +\infty \end{aligned}$$

Analog folgt $nv(g) < v(f) \Leftrightarrow \lim \ln(f) \left(n \tfrac{\ln g}{\ln f} - 1 \right) = +\infty$.

Gibt es nun m,n mit $m|v(f)| \geq |v(g)|$ und $n|v(g)| \geq |v(f)|$, so kann im Gleichheitsfalle einfach m oder n erhöht werden, und die Aussage gilt damit auch für ‚>‘. Da $v(f) < 0, v(g) < 0$ ist, kann die Betragsbildung ersetzt werden, und es gilt $mv(f) < v(g)$ und $nv(g) < v(f)$ und damit, wie oben gezeigt, $\lim \ln(g) \left(m \tfrac{\ln f}{\ln g} - 1 \right) = +\infty$ und $\lim \ln(f) \left(n \tfrac{\ln g}{\ln f} - 1 \right) = +\infty$. Da bereits $\ln f, \ln g \to +\infty$ gilt, muss $\lim m \tfrac{\ln f}{\ln g} \geq 1$ und $\lim n \tfrac{\ln g}{\ln f} \geq 1$ gelten. Das ist aber nur möglich, wenn $\lim \tfrac{\ln f}{\ln g}$ weder 0 noch $\pm\infty$ ist.

Gilt umgekehrt $\lim \tfrac{\ln f(x)}{\ln g(x)} \in \mathbb{R}^+ \setminus \{0\}$, so gilt für hinreichend großes m,n $\lim m \tfrac{\ln f}{\ln g} > 1$ und $\lim n \tfrac{\ln g}{\ln f} > 1$, und der obige Rechenweg lässt sich umkehren.

Für $f \prec g$ ergibt sich analog $\lim \ln(g) \left(n \tfrac{\ln f}{\ln g} - 1 \right) = -\infty$ für alle $n \in \mathbb{N}$. Dazu muss $n \tfrac{\ln f}{\ln g} - 1 < 0$ für alle n gelten, beziehungsweise $\lim \tfrac{\ln f}{\ln g} < \tfrac{1}{n}$. Das ist aber nur für $\lim \tfrac{\ln f}{\ln g} = 0$ erfüllt. □

Die Funktion $R(f, g)$ ist in doppelter Hinsicht sehr interessant. Zum einen bietet sie ein einfaches Kriterium, um das Wachstum von Funktionen zu vergleichen, ohne dabei auf die Existenz oder Nichtexistenz von Abschätzungskonstanten angewiesen zu sein.

Zum anderen stellt der Wert von $R(f, g)$ eine Ordnungsbeziehung innerhalb einer Wachstumsklasse dar. Da für $f, g, h \in \mathcal{H}_0$ mit $f \asymp g \asymp h$ die Beziehung $R(f, h) = R(f, g) \cdot R(g, h)$ gilt, kann man durch Auswahl eines Repräsentanten f_0 jeder Funktion $f \in \gamma(f_0)$ die Zahl $R(f, f_0)$ zuordnen und so alle Funktionen von $\gamma(f_0)$ anordnen.

Unter Zuhilfenahme von Theorem 3.3 (10) und (11) kann die Aussage von Theorem 3.7 auch elegant zusammengefasst werden:

Lemma 3.8.
Sei $f, g \in \mathcal{H}_0$, $v(f) \neq 0 \neq v(g)$. Dann gilt:

$$\begin{aligned} f \asymp g &\Leftrightarrow v(f'/f) = v(g'/g) \\ f \prec g &\Leftrightarrow v(f'/f) > v(g'/g) \end{aligned}$$

Diese Aussage ist inhaltlich vergleichbar mit Theorem 3.7, bietet aber bessere Möglichkeiten für mathematische Beweise durch die Einbeziehung der Ableitung.

Beweis: Benutze den Zusammenhang $(\log|f|)' = f'/f$ zusammen mit Theorem 3.3 (10) und (11). □

Abschließend noch die formale Einschätzung der Wirkung der Exponential- und Logarithmusfunktion in unserem Wachstumsmodell:

Lemma 3.9.
Sei $f \in \mathcal{H}_0$, $v(f) \neq 0$, dann gilt:

(1) $\gamma(\log|f|) < \gamma(f)$

(2) $\gamma(f) < \gamma\left(e^f\right)$, *wenn* $f \to \pm\infty$.

(3) $\gamma(f) > \gamma\left(e^f\right)$, *wenn* $f \to 0$.

Alle drei Aussagen folgen durch direkte Anwendung von Lemma 3.8 und der Erkenntnisse $v(\log|f|) < 0$ im Falle von (1) und $v(f) < 0$ bzw. > 0 im Falle von (2) und (3). □

Verglichen mit den Beispielen des vorigen Kapitels ergibt sich dieses Bild:

$$\begin{aligned} &1/x \succ 1 \prec \ln x \asymp (\ln x)^{10} \prec x^{0.1} \asymp x \\ &\asymp x \ln x \asymp x(\ln x)^{10} \asymp x^{1.1} \asymp x^{10} \prec e^{0.1x} \asymp e^x \asymp e^{10x} \end{aligned}$$

Betrachten wir noch ein paar Beispiele aus der Welt jenseits von e^x:

$$x \asymp x + e^{-x} \prec e^{-x} \asymp e^x \asymp x + e^x \asymp e^{x+e^{-x}} \prec e^{x \ln x} \prec e^{x^2} \prec e^{x+e^x}$$

Interessant anzumerken ist, dass $x + e^{-x} \prec x + e^x$ gilt, anderseits aber $e^{-x} \asymp e^x$. Diese Klassenbildung ist also nicht verträglich mit der Addition.

3.4. Termanalyse

Während wir bisher von Funktionen im mathematischen Sinne ausgegangen sind, betrachten wir nun Funktionen, wie sie von Computeralgebrasystemen gesehen werden: Als rekursive Baumstruktur mit Zahlen, Konstanten und Variablen als Blätter und Operationen als innere Knoten.

Der Algorithmus sieht vor, die am stärksten wachsenden Teilausdrücke durch ω darzustellen. Welche das sind, wird im Folgenden festgelegt. Vorher bezeichnen wir noch mit SubExp(f) die Menge aller Teilausdrücke von f, inklusive f selbst. Damit ergibt sich zum Beispiel:

$$\text{SubExp}\left(5x^2 + 7\right) = \left\{5x^2 + 7, 5x^2, 5, x^2, x, 2, 7\right\}$$

Nun kann die Menge der am stärksten wachsenden Teilausdrücke (Most Rapidly Varying, MRV) definiert werden:

Definition 3.10. ***(MrvSet)***
Sei $f(x)$ eine Funktion in Form einer Baumstruktur. Dann ist MrvSet(f) *wie folgt definiert:*

$$\text{MrvSet}(f) := \{\, g \in \text{SubExp}(f) \mid \forall h \in \text{SubExp}(f) : h \preceq g \,\}$$

Wie man an der Definition leicht sieht, sind alle Elemente von MrvSet(f) in der gleichen Wachstumsklasse, und kein anderer Teilausdruck von f besitzt ein stärkeres Wachstum.

Da alle Elemente von MrvSet(f) in der gleiche Wachstumsklasse liegen, macht es Sinn, solche Mengen und andere Funktionen direkt mit $\prec, \asymp, \succ$ zu vergleichen. Es reicht dabei vollkommen, den Vergleich mit einem Vertreter der Menge durchzuführen.

Als nützlich erweist sich auch eine Hilfsfunktion zur Vereinigung solcher Mengen:

$$\begin{aligned}
&\text{MrvMax}(A, B) = A && \text{, wenn } A \succ B,\\
&\text{MrvMax}(A, B) = B && \text{, wenn } A \prec B,\\
&\text{MrvMax}(A, B) = A \cup B && \text{, wenn } A \asymp B
\end{aligned}$$

Damit ist der Weg frei für eine konstruktivere Analyse des Wachstums einer Funktion:

Lemma 3.11.
Ist f eine exp-log Funktion in Baumstruktur, dann gilt:

(1) Wenn $x \notin$ SubExp(f) *ist, dann ist* MrvSet(f) = SubExp(f).

(2) Wenn $f = x$ ist, dann ist MrvSet(f) = $\{x\}$.

(3) Wenn $f = g \cdot h$ ist, dann ist MrvMax(MrvSet(g), MrvSet(h)) $\subseteq$ MrvSet(f).

(4) Wenn $f = g + h$ ist, dann ist MrvMax(MrvSet(g), MrvSet(h)) $\subseteq$ MrvSet(f).

(5) Wenn $f = g^c$ ($c \in \mathbb{R}$) ist, dann ist MrvSet(g) $\subseteq$ MrvSet(f).

(6) Wenn $f = \log g$ ist, dann ist MrvSet(f) = MrvSet(g).

(7) Wenn $f = e^g$ und $g \to \pm\infty$ ist, dann ist MrvSet(f) = MrvMax($\{e^g\}$, MrvSet(g)).

(8) Wenn $f = e^g$ und $g \to c \in \mathbb{R}$ ist, dann ist MrvSet(g) $\subseteq$ MrvSet(f).

In (1) haben alle Teilausdrücke das minimale Wachstum $\gamma(1)$. (2) ist offensichtlich.
Bei (3)-(8) ist die Frage entscheidend, ob die jeweilige Operation ein stärkeres Wachstum erzeugen kann, als es die Teilausdrücke einzeln können.
(3) ergibt sich aus Theorem 3.5 (7). (4) erweist sich als schwieriger: So haben im Fall $(-x + e^{-x}) + x$ beide Teile der Summe das Wachstum $\gamma(x)$, die Summe jedoch durch Elimination das Wachstum $\gamma(e^x)$. Trotzdem muss das stärkste Wachstum durch einen Teilausdruck von g oder h erzeugt worden sein. (5) folgt aus Theorem 3.5 (3). (6)-(8) folgen im Wesentlichen direkt aus Lemma 3.9, jedoch muss hier zusätzlich bedacht werden, dass g auch in $\gamma(1)$ liegen kann, wodurch auch der Gesamtausdruck in $\gamma(1)$ liegt. □

In allen obigen Fällen bedeutet $\subseteq$ außerdem effektiv, dass höchstens noch der Gesamtausdruck f selbst zusätzlich in der Menge liegen kann. Bei der späteren Umsetzung im Algorithmus kann in diesen Fällen der Gesamtausdruck f ausgelassen werden.

3.5. Ersetzung

Ein wichtiges Thema wurde noch nicht ausreichend betrachtet: Die Ersetzung eines Terms der höchsten Wachstumsklasse durch einen Term, der das Wachstum in der Variablen ω isoliert. Angenommen, ein Term f soll durch einen Term g mit $f \asymp g$ dargestellt werden. Wie bereits in Kapitel 2.3 dargelegt, wird das Wachstum von Termen künstlich bis auf exponentielles Wachstum angehoben werden. Wie die Implementierung zeigen wird, kann sogar davon ausgegangen werden, dass $f = e^s$ und $g = e^t$ ist. Als Ansatz versuchen wir f darzustellen als $f = A \cdot g^c$, wobei c konstant und weder 0 noch ∞ sein soll, und A kleineres Wachstum als f und g haben sollte: $A \prec g$.

Instinktiv scheint es eine gute Idee zu sein, c so zu wählen, dass $v(f) = v(g^c)$ ist - wenn dies denn möglich ist. A ergibt sich dann automatisch als $A = f/g^c = e^{s-ct}$.

Wollen wir $A \prec g$ erzwingen, können wir Theorem 3.7 zu Hilfe nehmen:

$$\begin{aligned} A \prec g &\Leftrightarrow \lim_{x\to+\infty} \tfrac{\ln|A|}{\ln|g|} = 0 \\ &\Leftrightarrow \lim_{x\to+\infty} \tfrac{s-ct}{t} = 0 \\ &\Leftrightarrow \lim_{x\to+\infty} s/t - c = 0 \\ &\Leftrightarrow c = \lim_{x\to+\infty} s/t \end{aligned}$$

c erweist sich dabei als das bereits bekannte Wachstumsverhältnis: $c = R(f, g)$. Damit ist auch gleich die Existenz von c gesichert, dank Theorem 3.7 angewendet auf $f \asymp g$.

4. Der MrvLimit-Algorithmus

Nachdem wir nun eine Vorstellung der Wachstumsprozesse im Unendlichen haben und bereits einige Bruchstücke des Algorithmus kennen, wird es jetzt Zeit, den MrvLimit-Algorithmus vollständig zu betrachten. Der Fokus wechselt mit diesem Kapitel weg vom mathematischen Hintergrund, hin zu den Algorithmen und Strategien.

Während wir den Ablauf des Algorithmus Schritt für Schritt beobachten, werden wir auch gleich überlegen, welche weiteren Funktionen problemlos in das Funktionsmodell übernommen werden können.

Außerdem werden wir einen kritischen Blick darauf werfen, ob die vorkommenden rekursiven Aufrufe des Algorithmus tatsächlich nicht zu unendlichen Rekursionen führen können. Die entsprechenden Nachweise sind anspruchsvoll, jedoch zum Verständnis des Algorithmus nicht zwingend erforderlich, lassen sich also beim ersten Lesen bequem überspringen.

4.1. Transformation

Wie schon in Kapitel 2.1 dargelegt, ist der Algorithmus darauf spezialisiert, Grenzwertaufgaben für $x \to \infty$ zu lösen. Zunächst muss die Funktion also geeignet transformiert werden: Für $x \to {x_0}^+$ ersetze x durch $x_0 + \frac{1}{x}$, für $x \to {x_0}^-$ ersetze x durch $x_0 - \frac{1}{x}$ und für $x \to -\infty$ ersetze x durch -x.

Die Behandlung von beidseitigen Grenzwerten wird in Computeralgebrasystemen unterschiedlich gehandhabt: Mathematica scheint bevorzugt Grenzwerte von oben (Direction → -1) zu berechnen, Maple liefert *undefined*, falls der Grenzwert nicht eindeutig ist. Im reellen Fall bleibt noch die Möglichkeit, beide Grenzwerte auszugeben, falls sich unterschiedliche Grenzwerte ergeben. Abgesehen von Mathematicas unvorsichtiger Herangehensweise muss man bei beidseitigen Grenzwerten aber immer beide einseitigen Grenzwerte ermitteln und vergleichen.

4.2. Erweitertes Funktionsmodell

Der ursprüngliche Algorithmus beschränkt sich auf exp-log Funktionen, d.h. Konstanten, Potenzen von x, Grundrechenarten, Exponentialfunktion und Logarithmus. Betrachtet man den Algorithmus aber im Detail, werden hauptsächlich vier Arten von Funktionen unterschieden: Funktionen mit stetigem Grenzübergang, Funktionen mit hebbaren polynomiellen Polstellen, Funktionen mit logarithmischen Polstellen und Funktionen mit nicht hebbaren (essenziellen)

Polstellen, jeweils natürlich auf den Punkt des Grenzüberganges bezogen. Grenzübergänge von stetigen Funktionen können direkt berechnet werden, hebbare Polstellen werden durch Potenzreihenanalyse betrachtet. Logarithmische Polstellen werden dabei automatisch von ω-Termen zu x-Termen abgebaut. Nur die nicht hebbaren Polstellen der Exponentialfunktion werden algorithmisch erfasst und analysiert.

Daher spricht auch nichts dagegen, weitere Funktionen in das Funktionsmodell aufzunehmen, solange die Funktionen höchstens hebbare Polstellen besitzen bzw. der Grenzübergang an unkritischen Stellen der Funktion auftritt und solange die Funktion in eine Potenzreihe entwickelbar ist.

Unter anderem kann so der Funktionsraum um trigonometrische Funktionen und die Gammafunktion erweitert werden:

$\sin(x), \cos(x), \tan(x)$	für $x \in \mathbb{R}$
$\arcsin(x), \arccos(x)$	für $x \in \mathbb{R}$
$\arctan(x)$	für $-\infty \leq x \leq +\infty$
$\Gamma(x)$	für $x \in \mathbb{R}$
$\Gamma_s(x) = \log(\Gamma(x))$	für $0 \leq x \leq +\infty$
$\psi(x) = \Gamma'(x)/\Gamma(x)$	für $-\infty < x \leq +\infty$
$\psi^{(n)}(x) = \frac{d^n}{dx^n}\psi(x)$	für $-\infty < x \leq +\infty$

(Weitere Funktionen werden in [Gru96] Kapitel 5.1 aufgeführt.)

Falls solche Funktionen in der Grenzwertaufgabe auftreten, muss vor der weiteren Grenzwertberechnung überprüft werden, ob das Argument der Funktion existiert und die jeweiligen Einschränkungen erfüllt werden. Das macht man am besten in Form einer Vorverarbeitung.

4.3. Vorverarbeitung

Die Vorverarbeitung dient dazu, die Struktur der Funktion zu überprüfen und anzupassen. Nicht unterstützte Funktionen müssen erkannt werden, bei manchen Funktionen muss das Argument der Funktion auf Gültigkeit geprüft werden. Der dazu erforderliche rekursive Aufruf des Grenzwertalgorithmus ist nicht kritisch, da der rekursive Aufruf ja nur mit einem Teilausdruck der Funktion erfolgt. Unendliche Rekursionen sind so nicht möglich.

Falls ein nicht unterstützter Ausdruck auftritt, muss die Grenzwertberechnung mit einer entsprechenden Fehlermeldung abgebrochen werden. Andere Ausdrücke müssen für die weitere Verarbeitung erst in eine geeignete Darstellung transformiert werden:

a^b	$\rightarrow$	$e^{\log(a)\cdot b}$	es sei denn, $a = e$ oder b ist konstant.
$\Gamma(x)$	$\rightarrow$	$e^{\Gamma_s(x)}$	falls $x \rightarrow \infty$

Gleichzeitig bietet es sich an, redundante Funktions-Schreibweisen, wie z.B. bei den Trigono-

metrischen Funktionen, auf die Grundformen zu beschränken:

$$\begin{aligned}
\sec(x) &\to 1/\cos(x) \\
\csc(x) &\to 1/\sin(x) \\
\cot(x) &\to 1/\tan(x) \\
\operatorname{arcsec}(x) &\to \arccos(1/x) \\
\operatorname{arccsc}(x) &\to \arcsin(1/x) \\
\operatorname{arccot}(x) &\to \arctan(1/x)
\end{aligned}$$

Nach der Vorverarbeitung muss sichergestellt sein, dass die Funktion keine unerwarteten Teilausdrücke mehr enthält. Nur mit einem klar begrenzten Funktionsumfang kann der Algorithmus zuverlässig arbeiten.
Als Folge sind ab hier auch automatische Vereinfachungssysteme der Computeralgebraprogramme mit äußerster Vorsicht zu benutzen, da so schnell wieder Funktionen auftauchen, die nicht innerhalb des Modells liegen.

Ein weiterer Fall, den man besser vorab behandelt, sind konstante Funktionen, die nicht von x abhängen. Der Algorithmus scheitert an Funktionen ohne jedes Wachstum daran, das Wachstum auf Mindestniveau anzuheben. Da hier aber nichts zu berechnen ist, sollte man die weitere Verarbeitung frühzeitig abbrechen.

Schlimmer noch sind Funktionen, die in einer ganzen Umgebung um Unendlich identisch Null sind, wie zum Beispiel $1-(x-C)/\sqrt{(x-C)^2}$, die identisch Null ist für $x > C$. Solche Funktionen sind praktisch nicht sicher erkennbar, sprengen aber das Funktionsmodell des Algorithmus und führen so zu Programmfehlern und Abbrüchen.

4.4. Rekursion und Terminierung

Immer wieder kommt es im Laufe des Algorithmus zu rekursiven Aufrufen der Grenzwertberechnung. So kann eine einzige Grenzwertaufgabe durchaus eine Kaskade von mehreren Hundert weiteren Grenzwertberechnungen auslösen. Wird dabei bei der Berechnung von $\lim_{x\to\infty} f(x)$ irgendwann ein rekursiver Aufruf nötig, der wieder $f(x)$ oder gewisse Variationen davon enthält, ist eine unendliche Schleife nicht mehr zu verhindern.

Um sicher zu gehen, dass solche Schleifen nicht auftreten, sollte bei jedem rekursiven Aufruf eine Art ‚Fortschritt' bei der Bewältigung der Aufgabe erkennbar sein. In unserem Fall wird der Fortschritt messbar, indem wir in zwei Schritten die Komplexität einer Funktion berechenbar machen:

```
function ComplexitySet(t : Term)
    if (x not in t)
        return {}
    if (t = x)
        return  x
    if (t = _f_ + _g_)
        return ComplexitySet(f) ∪ ComplexitySet(g)
    if (t = _f_ * _g_)
        return ComplexitySet(f) ∪ ComplexitySet(g)
    if (t = _g_ ^ _c_ and x not in c)
        return ComplexitySet(g)
    if (t = Log(_g_))
        return { Log(g) } ∪ ComplexitySet(g)
    if (t = e ^ _g_)
        return { e^g } ∪ ComplexitySet(g)
    if (t = PolyGamma(_n_,_g_))
        return ComplexitySet(g)

    if (t = _f_(_g_))
        if (f in { Sin, Cos, Tan, ArcSin, ArcCos,
                   ArcTan, Gamma, LogGamma } )
            return ComplexitySet(g)

end function

function Complexity(t : Term)
    return SizeOf(ComplexitySet(t))
end function
```

_t_ dient dabei als Platzhalter für beliebige Teilausdrücke, die in Folge dann als t referenziert werden. Complexity(t) wird auch kurz als $C(t)$ bezeichnet.

Wie man leicht sieht, gilt $C(e^f) = C(\log f) = C(f) + 1$, wohingegen für die Grundrechenarten nur gilt: $\max(C(f), C(g)) \leq C(f + g) = C(f \cdot g) \leq C(f) + C(g)$.

Um unendliche Rekursionen sicher auszuschließen, wird die Forderung aufgestellt, dass bei jedem rekursiven Aufruf die Komplexität der rekursiv gelösten Aufgabe niedriger ist, als die Komplexität der ursprünglichen Funktion.

Leider ist auch das nicht haltbar, deshalb werden in Kapitel 4.11 noch zwei Fälle untersucht, in denen eine rekursive Berechnung von gleicher Komplexität erforderlich sein kann. Die rekursiven Aufrufe sind jedoch von ausrechend spezieller Natur, um sicher zu stellen, dass die Komplexität in späteren rekursiven Aufrufen sinken wird.

4.5. Stärkstes Wachstum

Nun ist der Weg frei, die Funktion auf ihr grobes Wachstumsverhalten zu untersuchen. Wir gehen dabei analog zu Kapitel 3.4 vor.

Bei MrvSet, der Menge aller Teilausdrücke der stärksten Wachstumsklasse, weichen wir jedoch

von der mathematischen Sichtweise in einigen Details ab:

- Während formal das MrvSet einer konstanten Funktion aus allen Teilausdrücken besteht, liefert die Implementierung die leere Menge zurück. Da konstante Teilausdrücke keiner besonderen Aufmerksamkeit bedürfen, ist das eine sinnvolle Vereinfachung.
- Wenn das Wachstum eines Ausdrucks ausschließlich durch seine Teilausdrücke bestimmt wird, wie z.B. bei $f+g$ oder $f \cdot g$, genügt es, das Wachstum der Teilausdrücke zu behandeln und den Gesamtausdruck unverändert zu belassen. Werden später die am stärksten wachsenden Teilausdrücke ersetzt, ist damit auch der Gesamtausdruck ausreichend behandelt.

Sehen wir uns die Funktion MrvSet im Detail an:

```
function MrvSet(t : Term)
    if (x not in t)
        return {}
    if (t = x)
        return  x
    if (t = _f_ + _g_)
        return MrvMax(MrvSet(f),MrvSet(g))
    if (t = _f_ * _g_)
        return MrvMax(MrvSet(f),MrvSet(g))
    if (t = _g_ ^ _c_ and x not in c)
        return MrvSet(g)
    if (t = Log(_g_))
        return MrvSet(g)
    if (t = e ^ _g_)
        if (|Limit(g,x→inf)| < inf)
            return MrvSet(g)
        if (Limit(g,x→inf)=+/-inf)
            return MrvMax({ e^g },MrvSet(g))
    if (t = PolyGamma(_n_,_g_))
        return MrvSet(g)

    if (t = _f_(_g_))
        if (f in { Sin, Cos, Tan, ArcSin, ArcCos,
                   ArcTan, Gamma, LogGamma } )
            return MrvSet(g)

end function
```

_t_ dient dabei wieder als Platzhalter für beliebige Teilausdrücke, die in Folge dann als t referenziert werden. Die Maximum-Mengenvereinigung MrvMax ist bereits in Kapitel 3.4 beschrieben und wird gleich noch im Detail dargelegt.

Im Falle der Exponentialfunktion wird erstmals ein rekursiver Aufruf des Grenzwertalgorithmus erforderlich. Da aber der rekursive Aufruf nur mit dem Exponent erfolgt, die Komplexität also um eins sinkt, ist eine unendliche Rekursion hier nicht zu befürchten.

Die trigonometrischen und anderen Funktionen am Ende benötigen keine besondere Aufmerksamkeit mehr, da der Grenzübergang nur an unkritischen Stellen der Funktion erfolgt. Kritische

Stellen wurden bereits von der Vorverarbeitung aufgelöst oder abgewiesen.

Für den MrvLimit-Algorithmus ist es von entscheidender Bedeutung zu erkennen, welcher Art die von MrvSet zurückgelieferten Mengen sind:

- MrvSet liefert eine Menge von Termen zurück, die alle in der gleichen Wachstumsklasse liegen.
- MrvSet(t) liefert nur Teilausdrücke von t zurück.
- Die Menge ist leer, oder enthält nur Elemente der Form x oder $e^{f(x)}$.
- Enthält die Menge Terme der Wachstumsklasse $\gamma(x)$, so enthält sie auch x selbst.
- Alle Elemente haben den Grenzwert 0 oder ∞ für $x \to \infty$.
- Kein Element ist konstant.
- Für $x > 0$ sind alle Elemente größer als 0.

Auf diese Eigenschaften wird später im Algorithmus aufgebaut.

Es lohnt noch anzumerken, dass die Wachstumsklasse $\gamma(x)$ in der Regel, aber nicht immer, als $\{x\}$ zurückgegeben wird. Andernfalls muss ein Term die Form $e^{f(x)}$ haben und dieses $f(x)$ muss logarithmisch oder langsamer wachsen. Die meisten solchen Terme fallen vorher bereits Mathematicas automatischer Vereinfachung zum Opfer.

Es bleibt der Funktionsaufruf von MrvMax zu erklären. Da die beiden übergebenen Mengen jeweils nur Elemente einer Wachstumsklasse enthalten, genügt es, je einen Stellvertreter jeder Menge auszusuchen und deren Wachstum zu vergleichen:

```
function MrvMax(s1,s2 : Set of Term)
    if (s1 = {})
        return s2
    if (s2 = {})
        return s1
    if (s1[1] = s2[1])
        return Union(s1,s2)

    t1 = Log(s1[1])
    t2 = Log(s2[1])
    if (t1 = Log(e^_f_)
        t1 = f
    if (t2 = Log(e^_f_)
        t2 = f
```

```
    if (Limit(t1/t2,x→inf) = inf)
        return s2
    if (Limit(t1/t2,x→inf) = 0)
        return s1

    return Union(s1,s2)
end function
```

Der Fall der leeren Mengen (d.h. konstanten Terme) wird vorab erledigt. Danach werden die Stellvertreter beider Mengen anhand von Theorem 3.7 verglichen. Sind die Stellvertreter vom Typ $e^{f(x)}$, kann $\log(e^{f(x)})$ sofort zu $f(x)$ vereinfacht werden.

Da wieder rekursive Aufrufe der Limit-Funktion auftreten, muss wieder mit Vorsicht vorgegangen werden. Dazu muss beobachtet werden, in welchen Situationen MrvMax aus MrvSet heraus aufgerufen wird.

Stößt MrvSet auf einen Term der Form $e^{g(x)}$, kann es zu einem Aufruf MrvMax($e^{g(x)}$, MrvSet($g(x)$)) kommen. Das erste Argument kann dabei schlimmstenfalls die Funktion selbst sein, hat also maximal die Komplexität $C(e^{g(x)})$. Durch die Vereinfachung sinkt die Komplexität aber auf $C(g(x)) = C(e^{g(x)}) - 1$.

Das zweite Argument hat maximal die Komplexität $C(g(x))$, ist aber in jedem Fall ein Teilausdruck von $g(x)$. Die Komplexität des rekursiven Aufrufs ist daher begrenzt mit $C(t_1/t_2) \leq C(e^{g(x)}) - 1$.

Es bleiben die Aufrufe von MrvMax bei $f(x) + g(x)$ und $f(x) \cdot g(x)$. Schlimmstenfalls ist dabei die ursprüngliche Funktion direkt $t = f(x) \cdot g(x)$, und der Wachstumsvergleich findet tatsächlich für die Funktionen $f(x)$ und $g(x)$ statt. Entsprechendes gilt für $f(x) + g(x)$.

Der Fall $f(x) = g(x) = x$ wird vorab behandelt und führt nicht zu einer Rekursion. Ist o.B.d.A. $g(x) = x$ und $f(x) \neq x$, so bleibt für $f(x)$ nur die Form $f(x) = e^{f_1(x)}$, und es kommt zum rekursiven Aufruf $\lim_{x\to\infty} f_1(x)/\log x$. Die Komplexität dieses Ausdrucks ist möglicherweise gleich der von $f(x) \cdot g(x) = e^{f_1(x)} \cdot x$. Den Beweis, dass der Grenzwertaufruf $\lim_{x\to\infty} f_1(x)/\log x$ nicht zu einer unendlichen Rekursion führt, wird auf das Kapitel 4.11 aufgeschoben.

Ist schließlich $f(x) = e^{f_1(x)}$ und $g(x) = e^{g_1(x)}$, so erfolgt der rekursive Aufruf $\lim_{x\to\infty} \frac{f_1(x)}{g_1(x)}$ mit einer garantiert niedrigeren Komplexität als $f(x) \cdot g(x)$, die Komplexität sinkt also.

4.6. Anhebung der Wachstumsklasse

Nachdem wir nun das schlimmst mögliche Wachstumsverhalten der Funktion ermittelt haben, wird es Zeit, sich eines unangenehmen Sonderfalls zu entledigen: Wachstum der Klasse $\gamma(x)$, d.h. Funktionen, die polynomiell oder langsamer wachsen.

Wie schon im Kapitel 2.3 angeschnitten, wird solange x durch e^x ersetzt, bis das Wachstum der Funktion nicht mehr in die Klasse $\gamma(x)$ fällt. Danach enthält die Funktion garantiert Teil-

ausdrücke von exponentiellem Wachstum, die klar von Teilausdrücken niedriger Ordnung getrennt werden können. Als Nebeneffekt werden durch das Anheben Terme mit logarithmischem Wachstum so weit angehoben, dass sie entweder exponentiell sind oder vorerst von anderen exponentiellen Termen überschattet werden.

Das Anheben der Wachstumsklasse ist allerdings nicht ohne Folgen. Die Komplexität der Funktion nimmt durch das Ersetzen von x durch e^x zu, deswegen muss sehr vorsichtig auf eventuelle unendliche Rekursionen geachtet werden! Das MrvSet der angehobenen Funktion erneut zu berechnen, kann bereits zu einer Schleife führen. Zum Glück ist das jedoch nicht immer erforderlich. Die Komplexität wird auch in Grenzen gehalten, wenn konsequent jedes Vorkommen von $\log(x)$ nicht durch $\log(e^x)$, sondern gleich durch x ersetzt wird:

```
Ω = MrvSet(f)
ScaleUp = 0
while (x in Ω)
    ScaleUp = ScaleUp + 1
    f = Replace(f, Log(x) → x, x → e^x)
    Ω = Replace(Ω, Log(x) → x, x → e^x)

    for each ω in Ω
        if (ω not in f)
            Ω=Delete(Ω,ω)
    end for

    if (Ω = {})
        Ω = MrvSet(f)
end while
```

Es genügt, auf das Vorkommen von x in Ω zu achten, da das MrvSet der Klasse $\gamma(x)$ immer auch x enthält. ScaleUp zählt mit, wie häufig die Funktion in eine höhere Klasse angehoben wurde, damit die Ergebnisse später wieder in die ursprüngliche Klasse abgesenkt werden können. Ist man nur am schlichten numerischen Ergebnis interessiert und nicht an Asymptoten, kann das auch eingespart werden.

Als nächstes werden in f und Ω synchron alle x und $\log(x)$ Terme durch ihre angehobenen Varianten ersetzt. Ω ist damit fast schon wieder identisch zum neuen MrvSet(f). Der einzige Fall, in dem die Ersetzung in f und in Ω unterschiedlich erfolgt, ist ein Auftreten von $\log(x)$ in f.
Tritt in f ein Term $\log(x)$ auf, so ist dessen Repräsentant in Ω der Term x. Daher wird in Ω die Ersetzung $x \to e^x$ angewendet, und in f wird $\log(x) \to x$ ersetzt. Dadurch enthält Ω ein e^x, das möglicherweise nicht in f vorkommt.

Deswegen wird zunächst erst einmal jedes Element von Ω entfernt, das nicht mehr in f vorkommt. Ist Ω jetzt nicht leer, so gibt es in f mindestens einen Teilausdruck mit exponentiellem Wachstum. Alle Teilausdrücke von f mit exponentiellem Wachstum waren vorher mit polynomiellem Wachstum in Ω vertreten und wurden in Ω und f identisch transformiert. Es gilt also wieder Ω=MrvSet(f).

Ist Ω dagegen leer, so kann in f nur die Ersetzungsregel $\log(x) \to x$ angewendet worden sein, und f enthält immer noch keinen Term mit exponentiellem Wachstum. Da die bisherige Wachstums-

klasse vollständig eliminiert wurde, muss die nächst niedrigere, bisher logarithmische Wachstumsklasse erneut ermittelt werden.

Insgesamt ist also nun folgendes passiert:

- Ist das stärkste Wachstum stärker als polynomiell, so ist die Funktion unverändert geblieben.
- Ist das stärkste Wachstum polynomiell gewesen, so wurde genau ein mal die Ersetzung $x \to e^x$ durchgeführt. Die Komplexität ist entsprechend maximal um 1 gestiegen.
- Ist das stärkste Wachstum logarithmisch gewesen, wurde ein- oder mehrmals $\log(x) \to x$ ersetzt und abschließend ein mal $x \to e^x$. Die Komplexität sank dadurch zunächst, stieg aber im letzten Schritt maximal wieder um 1 an.

Der rekursive Aufruf von MrvSet ist zum Glück unkritisch, da die Ersetzungsregel $\log(x) \to x$ die Komplexität von f vorher gesenkt hat. Bei den noch folgenden rekursiven Aufrufen muss jedoch mit um so mehr Sorgfalt vorgegangen werden, da mit der schlimmstenfalls um 1 gestiegenen Komplexität kalkuliert werden muss.

4.7. Wahl des Repräsentanten

Als nächstes wird ein Repräsentant ω der Menge Ω gewählt, der als Substitutionsvariable ω und Repräsentant des stärksten Wachstums dienen soll. Dabei gibt es ein später entscheidendes Kriterium zu berücksichtigen: Kein anderes Mitglied von Ω darf als Teilausdruck innerhalb von ω auftreten. Wenn also zum Beispiel $\Omega = \{e^{-x}, e^{x+e^{-x}}\}$ ist, so ist $\omega = e^{x+e^{-x}}$ eine schlechte Wahl, da darin e^{-x} auftritt. Das Problem kann umgangen werden, wenn man als ω das Element von Ω mit der geringsten Komplexität auswählt.

Der Vertreter ω muss zusätzlich die Eigenschaft $\lim_{x\to\infty} \omega = 0$ erfüllen, damit später die Potenzreihenentwicklung in ω=0 durchgeführt werden kann. Da aber die Mitglieder von Ω bereits alle in der Form $e^{g(x)}$ vorliegen und als Grenzwert nur noch 0 oder ∞ möglich ist (s. Kapitel 4.5), ist die Forderung $\omega \to 0$ leicht zu erfüllen: Wenn $\lim_{x\to\infty} g(x) = \infty$, dann verwende $\omega = e^{-g(x)}$. Diese Vorzeichenänderung hat keine Auswirkungen auf die Stabilität des Algorithmus oder die Komplexität der Funktion.

Interessanter ist da schon die Frage, woher man weiß, ob $\lim_{x\to\infty} g(x) = \infty$ gilt. Dafür einen rekursiven Aufruf des Grenzwertalgorithmus starten könnte zu unendlichen Rekursionen führen.

Zum Glück ist das aber nicht notwendig, denn es kommen nur zwei ‚Verursacher' als Quelle für ω in Frage:

- Der Aufruf von MrvSet(x, x), der mittlerweile zu e^x angehoben wurde.
- Der Aufruf von MrvSet$(e^{h(x)}, x)$ mit einer Funktion $h(x) \to \pm\infty$.

Im ersten Fall ist das Wachstumsverhalten offensichtlich, im zweiten Fall kann es anhand des damals sowieso berechneten Grenzwerts von $h(x)$ ermittelt werden. Es bietet sich daher an, entweder den Grenzwert von $g(x)$ durch exaktes Zurückverfolgen zum Ausgangspunkt zu bestimmen, oder, besser noch, das Wachstumsverhalten von vornherein bei der Bestimmung von Ω mit zu bestimmen und aufzubewahren.

4.8. Umschreiben der Funktion

Nun wird die Funktion so umgeschrieben, dass alle Teilausdrücke des stärksten Wachstums durch ω repräsentiert werden. Den dafür nötigen Ansatz liefert Kapitel 3.5: Ersetze jedes Vorkommen von $g \in \Omega$ in f durch $A \cdot \omega^c$. Nimmt man $g = e^s$ und $\omega = e^t$ an, so ergibt sich die Konstante $c = \lim_{x\to\infty} s/t$ und $A = e^{s-ct}$. Bei der Ersetzung sollte ω gleich als symbolische Konstante eingesetzt werden und nicht wieder durch den Term ersetzt werden, den ω repräsentiert.

Nach der Ersetzung muss sichergestellt sein, dass MrvSet$(f) = \{\omega\}$ gilt. Um das zu garantieren, müssen alle Vorkommen von Ω in f ersetzt werden, und durch die Ersetzung dürfen keine neuen Terme eingeführt werden, die in die gleiche Wachstumsklasse fallen.

Alle Vorkommen von Elementen von Ω werden direkt ersetzt, als Quelle für neue Vorkommen kommt daher nur der Ausdruck A in Frage. A selbst hat nachweislich eine niedrigere Wachstumsklasse, es bleiben aber noch Teilausdrücke von A zu berücksichtigen. Diese bestehen wiederum aus Teilausdrücken von ω und von $g \in \Omega$. Es geht also darum, wann die Elemente von Ω selbst wiederum Teilausdrücke enthalten, die ebenfalls in Ω sind.

Ordnet man die Elemente von Ω nach ihrer Komplexität, wird die Situation überschaubar: Elemente von höherer Komplexität können nur Elemente niedrigerer Komplexität als Teilausdrücke enthalten. Ersetzt man nun zuerst die Elemente hoher Komplexität, so werden in den nachfolgenden Ersetzungen ebenfalls alle neu eingeführten kritischen Teilausdrücke niedrigerer Komplexität mit ersetzt. Hat man ω als das Element mit niedrigster Komplexität gewählt, so kann auch der von ω abstammende Teil in A keine neuen kritischen Teilausdrücke mehr einführen, was andernfalls zu rekursiven Ersetzungsschleifen führen könnte.

Eine weitere Quelle von Problemen sind Term-Optimierungen. Vereinfacht man $A = e^{s-ct}$ unvorsichtigerweise zu $A = e^s \cdot e^{-ct}$, ist man fast wieder am Ausgangspunkt angekommen, und der Algorithmus scheitert. Je nach Computeralgebrasystem muss man daher sicher stellen, dass auf automatische Optimierungen weitgehend verzichtet wird.

Schließlich bleibt der rekursive Aufruf des Grenzwertalgorithmus. Die Gefahr einer unendlichen Rekursion ist auch hier wieder gegeben. Im Kapitel 4.12 wird ein Trick aufgezeigt, mit dem der

rekursive Aufruf komplett vermieden werden kann.

Da diese Arbeit nicht auf diesen Trick zurück greift, hier der Beweis, dass auch der rekursive Aufruf terminiert: Berechnet wird der Grenzwert $c = \lim_{x\to\infty} s/t$, ausgehend von den Ausdrücken e^s und e^t bzw. e^{-t} aus Ω und damit Teilausdrücken von f. Falls die Wachstumsklasse nicht angehoben wurde, treten e^s und $e^{\pm t}$ direkt im ursprünglichen f auf, und s/t hat eine um mindestens 1 niedrigere Komplexität.

Es bleibt der Fall, dass mindestens ein mal die Wachstumsklasse angehoben wurde. Solange dabei nur $\log x \to x$ ersetzt wurde, kann die Komplexität höchstens gesunken sein. Beim letzten Anheben wurde aber genau ein mal $x \to e^x$ ersetzt, wodurch die Komplexität wieder um eins stieg.

Da nun auch garantiert $e^x \in \Omega$ ist, liegt $t = -x$ bereits fest, und es gilt $C(s) = C(s/t)$, was schlimmstenfalls gleich der Komplexität der ursprünglichen Funktion f ist.

Außerdem muss e^x in e^s enthalten sein, d.h. entweder ist $s = x$, oder e^x ist in s enthalten. Im ersten Fall ergibt sich (bei geeigneter Optimierung) $s/t = x/(-x) = -1$, wodurch keine rekursiven Probleme entstehen. Andernfalls bleibt ein Grenzwertaufruf $\lim_{x\to\infty} s/-x$, wobei s den Teilausdruck e^x enthält. Dass dieser Aufruf nicht zu einer unendlichen Rekursion führt, wird im Kapitel 4.11 gezeigt.

4.9. Potenzreihe

Die Hauptarbeit des Algorithmus ist damit geschafft. Die Funktion wurde in eine Darstellung überführt, bei der die Teilausdrücke größter Wachstumsklasse nur in der Gestalt des Symbols ω auftreten, und alle anderen auftretenden Terme einer niedrigeren Klasse angehören.

ω hat die nötige Eigenschaft $\omega \to 0$ für $x \to \infty$, also können die Anfangsterme einer generalisierten Potenzreihe für f in $\omega = 0^+$ ermittelt werden. Das „Wie“ würde den Rahmen dieser Diplomarbeit endgültig sprengen und sei daher anderen überlassen. So viel sei gesagt, es gibt Algorithmen, die in unserem Fall die abgebrochene Potenzreihe garantiert finden.

Der Algorithmus sollte eine abgebrochene Potenzreihendarstellung für f liefern, mit der Gestalt $f = a_0 \cdot \omega^{e_0} + O[\omega^{e_1}]$, mit einem reellen Exponenten $e_0 < e_1$ und einem Faktor $a_0 \neq 0$, der konstant oder von niedrigerer Wachstumsklasse als ω ist und der eine niedrigere Komplexität als die ursprüngliche Funktion f hat.

Ein Hinweis muss der Potenzreihenentwicklung noch auf den Weg gegeben werden. Falls im Laufe der Potenzreihenentwicklung die Singularitäten $\log \omega$ oder $\log 1/\omega$ auftreten, so können diese direkt vereinfacht werden: Sie gehören einer niedrigeren Wachstumsklasse an und liefern keinen Beitrag zu dieser Potenzreihe. Da ω immer die Form $\omega = e^t$ hat, ist es ein leichtes, die Beziehungen $\log \omega = t$ und $\log 1/\omega = -t$ aufzustellen und die auftretenden Singularitäten aufzulösen.

Umgekehrt kann es je nach Algorithmus vorkommen, dass Terme der Form ω oder $1/\omega$ in ihrer nach x aufgelösten Form auftreten. Diese sollten dann von ihrer Darstellung mittels x zurück

transformiert werden zur Darstellung mit ω, damit der Algorithmus sie bei der Potenzreihenentwicklung korrekt berücksichtigt.

4.10. Ergebnisanalyse

Es bleibt nur noch wenig zu tun. Als nächstes sollte die Anhebung der Wachstumsklasse aus Kapitel 4.6 rückgängig gemacht werden, indem genau so oft wie damals die umgekehrte Ersetzung $e^x \to x$, $x \to \log x$ auf ω und auf a_0 angewendet wird. Wie schon gesagt, falls man nur an dem numerischen Ergebnis interessiert ist, kann dieser Schritt auch entfallen.

Nach der Potenzreihenentwicklung sind die Wachstumskomponenten der Klasse $\gamma(\omega)$ konzentriert in Termen ω^{e_k} mit $e_k \neq 0$, alle Komponenten niedrigerer Wachstumsklassen konzentrieren sich auf die Faktoren a_k. Daher gilt: Ist $e_0 > 0$, so dominiert $\omega \to 0$ alle anderen Terme, der Grenzwert ist also 0. Ist $e_0 < 0$, so dominiert $1/\omega \to \infty$ alle anderen Terme, das Ergebnis ist $\text{Sign}(\lim_{x\to\infty} a_0) \cdot \infty$. Ist hingegen $e_0 = 0$, so heben sich alle Wachstumskomponenten der Klasse $\gamma(\omega)$ für $x \to \infty$ gegenseitig auf, und der Grenzwert ist $\lim_{x\to\infty} a_0$.

Eventuell ist also ein rekursiver Grenzwertaufruf für $\lim_{x\to\infty} a_0$ erforderlich. Da aber die Komplexität durch die Potenzreihenentwicklung gesunken ist, bereitet der rekursive Aufruf in der nächst niedrigeren Wachstumsklasse keine Probleme. Außerdem wird durch jede Potenzreihenentwicklung eine vollständige Wachstumsklasse abgebaut, so lange, bis alle Wachstumsklassen aufgelöst sind und a_0 nicht mehr von x abhängt. Da der ursprüngliche Funktionsausdruck endliche Komplexität hat, können nur endlich viele Wachstumsklassen auftreten, der Algorithmus terminiert also.

Als zusätzliches Bonbon kann der Algorithmus auch eine Asymptote zur Funktion bestimmen, die alle Wachstumsklassen geordnet aufführt: Protokolliert man die Ergebnisse der Potenzreihenentwicklungen $a_{(0,i)}\omega_i^{e_{(0,i)}}$, so ergibt sich am Ende folgende Asymptote:

$$A(x) = a_{(0,n)} \cdot \omega_{n-1}{}^{e_{(0,n-1)}} \cdot \omega_{n-2}{}^{e_{(0,n-2)}} \cdot \ldots \cdot \omega_1{}^{e_{(0,1)}}$$

Alle ω_i streben gegen Null, sind positiv, haben für zunehmendes i eine fallende Wachstumsklasse, und der erste Term mit $e_{(0,i)} \neq 0$ ist dominant. Außerdem gilt $\lim_{x\to\infty} f(x)/A(x) = 1$. Der Algorithmus kann auch solange fortgesetzt werden, dass $a_{(0,n)}$ immer konstant ist. War der ursprüngliche Grenzwert nicht von der Form $x \to \infty$, so muss natürlich auch hier noch eine Rücktransformation durchgeführt werden.

4.11. Noch einmal Terminierung

Zwischenzeitlich hatten wir in zwei Fällen einen rekursiven Aufruf zugelassen, obwohl die Komplexität des rekursiven Aufrufs schlimmstenfalls gleich der der ursprünglichen Funktion war. Um trotzdem sicher zu stellen, dass es nicht zu einer unendlichen Rekursion kommen kann, müssen wir nun zeigen, dass die nächsten rekursiven Aufrufe nicht wieder auf eine solche Ausnahme hinaus laufen.

Im ersten Fall aus Kapitel 4.5 gilt es zu beweisen, dass $\lim_{x\to\infty} f_1(x)/\log x$ sicher terminiert, wobei dieser Grenzwert bei der Bestimmung des MrvSet von $f(x) \cdot g(x)$ entstand mit $e^{f_1(x)} \in \text{MrvSet}(f(x))$ und $x \in \text{MrvSet}(g(x))$.

Beim rekursiven Aufruf kommt es bei der Bestimmung des MrvSet wieder zu einer solchen Situation, bei der $\text{MrvSet}(f_1(x))$ mit $\text{MrvSet}(\log x) = \{x\}$ verglichen wird. Stellt sich dabei heraus, dass $e^{f_2(x)} \in \text{MrvSet}(f_1(x))$ ist, kommt es erneut zu einem rekursiven Aufruf ähnlicher Art. In diesem Aufruf wird aber durch den Wegfall des Logarithmus die Komplexität um eins sinken, der nächste rekursive Aufruf findet also wieder mit verringerter Komplexität statt.

Im zweiten Fall aus Kapitel 4.8 geht es um den Grenzwert $c = \lim_{x\to\infty} s(e^x)/-x$. Er entstand aus der Umschreibung eines Terms $e^{s(e^x)}$ in die Form $A \cdot (e^{-x})^c$. Durch die Anhebung der Wachstumsklasse war für problemlose rekursive Aufrufe mindestens $C(s(e^x)/-x) < C(e^{s(x)})$ erforderlich.

Bei der MrvSet-Bestimmung wird diesmal jedoch der Teilausdruck e^x sofort dominant hervortreten, eine weitere Anhebung der Wachstumsklasse ist nicht erforderlich. Dadurch ist eine direkte Schleife hier ebenfalls nicht möglich.

Eine Kopplung der beiden Ausnahmen ist noch denkbar, wenn Terme gleicher Komplexität abwechselnd durch die erste Ausnahme und die zweite Ausnahme in tiefere rekursive Aufrufe gelangen. Um auch dies auszuschließen, muss der Aufruf $s(e^x)/-x$ der zweiten Ausnahme in den MrvSet-Aufruf der ersten Ausnahme verfolgt werden. Tatsächlich kann es zu einem weiteren rekursiven Aufruf gleicher Komplexität kommen, wenn $s(e^x)$ auch darstellbar ist als $e^{s_1(x)}$, so dass es beim MrvSet zum problematischen Aufruf mit $s_1(x)/\log x$ kommen kann. Die Schleife wird durchbrochen, wenn $s_1(x)$ noch immer e^x enthält, da so keine Anhebung der Wachstumsklasse erforderlich ist, und es nicht wieder zu Ausnahme 2 kommen kann. Andernfalls muss $s_1(x) = x$ sein, der rekursive Aufruf in MrvSet ist damit $x/\log x$, was zur angehobenen Form e^x/x führt. ω wird als e^{-x} gewählt, und der einzige bezüglich Ausnahme 2 interessante Grenzwertaufruf ist $c = \lim_{x\to\infty} x/-x$, der aber schon im Kapitel 4.8 als unkritisch erkannt wurde.

4.12. Alternativer Ansatz

Bei der Konstante c aus Kapitel 4.8 handelt es sich um das Wachstumsverhältnis $R(g, \omega)$ aus Definition 3.6. Berechnet wurden solche Verhältnisse bereits bei der Konstruktion von Ω mittels MrvMax. Bewahrt man diese Ergebnisse zusammen mit Ω auf, so kann daraus das Wachstumsverhältnis aller Terme in Ω relativ zu ω bestimmt werden.

Am einfachsten verfährt man dabei so: Die erste Funktion $\omega = \Omega[1]$ jedes MrvSet dient als Referenz und ihr wird die Zahl r=1 zugeordnet. Wann immer eine Funktion g mittels MrvMax in die Menge aufgenommen wird, wird dessen relatives Wachstumsverhältnis $r = R(g, \omega)$ für den Klassenvergleich bereits berechnet und kann dann zusammen mit der Funktion g im MrvSet aufbewahrt werden.

Soll eine andere Funktion ω_2 die erste Funktion ω ersetzen, so sind alle gespeicherten Verhältnisse anzupassen mit Hilfe der Regel $r_{neu} = r_{alt} \cdot R(\omega, \omega_2)$. Wie schon im Anschluss an Theorem 3.7 gesehen, gilt ja $R(g, \omega_2) = R(g, \omega) \cdot R(\omega, \omega_2)$. Normalerweise ist auch dieses Wachstumsverhältnis

bekannt, ein weiterer Grenzwertaufruf ist nicht nötig.

Entsprechend muss im Fall der Vereinigung bei MrvMax eine der beiden Mengen an die Referenz der anderen Menge angepasst werden, indem zum Beispiel in Ω_2 jedes Wachstumsverhältnis mit $R(\Omega_2[1], \Omega_1[1])$ multipliziert wird.

Die Anhebung der Wachstumsklasse hat auf das Wachstumsverhältnis genauso wenig einen Einfluss, wie auf den Grenzwert der Funktion selbst, hier ist also nichts zu beachten. Schließlich bleibt noch die endgültige Wahl von ω. Diese ist genauso zu behandeln, wie auch die bisherigen Wechsel der Referenzfunktion. Sollte das Wachstumsverhalten von ω durch Ersetzung $\omega \to 1/\omega$ gedreht worden sein, ergibt sich daraus eine weitere Multiplikation für alle Wachstumsverhältnisse mit -1.

Der Lohn der Mühe ist, dass bei der Ersetzung im Kapitel 4.8 das Wachstumsverhältnis c bereits bekannt ist und nicht durch einen weiteren rekursiven Aufruf ermittelt werden muss. Dadurch treten in der Hauptschleife des Algorithmus rekursive Aufrufe nur noch bei der Bestimmung von Ω sowie bei der Parameterprüfung am Anfang auf. Insbesondere treten aber nach der Anhebung der Wachstumsklasse keine rekursiven Aufrufe mehr auf, was die Beweisführung weiter vereinfacht.

In der weiteren Diplomarbeit wird dieser alternative Ansatz jedoch nicht impementiert.

5. Details der Implementierung

5.1. Paketdokumentation

Die Paketdokumentation ist Teil des Mathematica-Paketes MrvLimit und steht auf der Projekt-Webseite zum Download bereit. Es existiert auch eine komplett in Englisch verfasste Version der Paketdokumentation.

5.1.1. Laden des Pakets

Das Paket wird wie jedes andere Mathematica-Paket mit `Needs` oder `Get` geladen.

Ist die Datei `MrvLimit.m` in einem Verzeichnis des Mathematica-Suchpfads `$Path`, kann das Paket direkt geladen werden:

```
In[1]:= Needs["MrvLimit`"]
```

```
MrvLimit v1.0 © 2005 by Udo Richter
mail : udo_richter(at)gmx.de
web : http://urichter.cjb.net/MrvLimit
Released under the GNU General Public License.
```

Andernfalls muss beim Laden der komplette Pfad zur Datei angegeben werden:

```
In[2]:= Needs["MrvLimit`","Pfad/Zur/Datei/MrvLimit.m"]
```

```
MrvLimit v1.0 © 2005 by Udo Richter
mail : udo_richter(at)gmx.de
web : http://urichter.cjb.net/MrvLimit
Released under the GNU General Public License.
```

Wird das Paket nur innerhalb eines anderen Pakets benötigt, kann es auch ‚stillschweigend' geladen werden:

```
In[3]:= MrvLimit`MrvLimitLoadSilent = True;

        Needs["MrvLimit`"]
```

5.1.2. Einfache Grenzwertberechnungen

Einfache Grenzwertberechnungen können analog zu Mathematicas `Limit`-Funktion durchgeführt werden:

```
In[4]:= f = x^2 / ((x + 1) (x - 1));

        MrvLimit[f, x → ∞]
Out[4]= 1
```

Falls ein eindeutiger Grenzwert nicht existiert, kann die Richtung der Annäherung vorgegeben werden:

```
In[5]:= MrvLimit[f, x → 1, Direction → -1]
Out[5]= ∞
```

Dabei bedeutet -1 eine Annäherung von oben, +1 eine Annäherung von unten und 0 eine beidseitige Annäherung. Falls `MrvLimit` im beidseitigen Modus auf eine Unstetigkeit trifft, werden beide einseitigen Grenzwerte als Liste ausgegeben:

```
In[6]:= MrvLimit[f, x → 1]
Out[6]= {-∞, ∞}
```

Als optisch ansprechende Variante kann die Richtung der Annäherung durch ein hochgestelltes + oder - am Grenzwert angegeben werden:

```
In[7]:= MrvLimit[f, x → 1^+]
Out[7]= ∞
```

Sollte `MrvLimit` einen Grenzwert nicht korrekt berechnen können, wird eine entsprechende Warnmeldung ausgegeben, und die Grenzwertaufgabe wird unausgewertet zurück geliefert:

```
In[8]:= MrvLimit[Sin[1/x], x → 0]
        MrvLimit : : UnsupportedArgument : MrvLimit does not support Sin[x] for x → ∞,
         appearing in Sin[x]
Out[8]= MrvLimit[Sin[1/x], x → 0]
```

Die Spezialität des `MrvLimit`-Algorithmus liegt dabei insbesondere bei Funktionen mit verschachtelten Exponential- und Logarithmusfunktionen:

```
In[9]:= f = e^x (-e^(1/x - e^-e^x) + e^(e^-x + e^-x^2 + 1/x));

        MrvLimit[f, x → ∞]
```

```
Out[9]=  1

In[10]:= Limit[f, x → ∞]

Out[10]= Limit[e^x (-e^(-e^(-e^x)+1/x) + e^(e^(-x)+e^(-x^2)+1/x)), x → ∞]
```

MrvLimit unterstützt die Grundrechenarten +, -, ·, /, Potenz und Wurzel und die Funktionen `Log, Gamma, LogGamma, PolyGamma, Sin, Cos, Tan, ArcSin, ArcCos, ArcTan, Sec, Csc, Cot, ArcSec, ArcCsc` und `ArcCot`.

5.1.3. Fortgeschrittene Optionen

MrvLimit unterstützt einige fortgeschrittene Optionen. Diese Optionen können in beliebiger Reihenfolge angegeben werden.

Ausgabeformat

Mittels `Output→Limit`, `Output→LeadTerm` und `Output→SignedZero` kann die Ausgabe von MrvLimit verändert werden. Dabei ist `Output→Limit` die Standardvorgabe.

`Limit[`*f*`,`*x*`→`*lim*`, Output→SignedZero]` gibt den Grenzwert normal aus, wenn der Grenzwert nicht 0 ist. Ist der berechnete Grenzwert 0, wird entweder `Zero` oder `-Zero` ausgegeben, je nachdem, ob in der Umgebung des Grenzwertes die Funktion positiv oder negativ ist.

`Limit[`*f*`,`*x*`→`*lim*`, Output→LeadTerm]` gibt eine bei der Grenzwertberechnung ermittelte Asymptote aus. Die Ausgabe hat folgendes Format:

`LeadTerm[`*const*`,{pwr[`$base_1, exp_1$`],pwr[`$base_2, exp_2$`],... pwr[`$base_n, exp_n$`]}]`

const ist der nicht von x abhängige Teil der Funktion,
$base_i$ sind exponentielle Basisfunktionen,
exp_i sind die reellen, konstanten Exponenten, die die Ordnung repräsentieren, in der die Basisfunktionen auftreten.

Die zurückgegebenen Terme sind in interner Darstellung belassen, um deren spezielle Struktur zu erhalten. Durch Anwendung der Funktion `FromInternal[`*f*`,`*x*`]` kann die allgemeine Mathematica-Darstellung zurück gewonnen werden. Mehr zur internen Darstellung im Kapitel über Hilfsfunktionen.

Alle Basisfunktionen sind Exponentialfunktionen, sind positiv und haben für $x \to lim$, $x > 0$ den Grenzwert 0.

Die Basisfunktionen sind dabei nach fallender Wachstumsordnung geordnet. Die erste Funktion mit $exp_i \neq 0$ ist dominant und bestimmt das Wachstumsverhalten der Funktion.

Die Asymptote ist dabei die Funktion $const \cdot {base_1}^{exp_1} \cdot {base_2}^{exp_2} \cdot \ldots \cdot {base_n}^{exp_n}$

```
In[11]:= f = -e^(1/x - e^-e^x) + e^(e^-x + e^-x^2 + 1/x);

         lt = MrvLimit[f, x → ∞, Output → LeadTerm]
Out[11]= LeadTerm[1, {pwr[e^-e^x, 0], pwr[e^-x^2, 0], pwr[e^-x, 1], pwr[e^-Log[x], 0]}]
```

Alle Terme, abgesehen von e^{-x}, treten nur scheinbar auf und spielen für den Grenzwert keine Rolle.

Die Ausgabe erfolgt hier in der internen Funktionsdarstellung von MrvLimit und nicht in normaler Mathematica-Darstellung. Durch die Aufhebung der internen Darstellung wird $e^{-\log(x)}$ wieder automatisch zu $\frac{1}{x}$ vereinfacht:

```
In[12]:= lt = FromInternal[lt, x]
Out[12]= LeadTerm[1, {pwr[e^-e^x, 0], pwr[e^-x^2, 0], pwr[e^-x, 1], pwr[1/x, 0]}]
```

Durch Ausführung der Potenzen eliminieren sich die scheinbaren Basisfunktionen:

```
In[13]:= lt = lt/.pwr → Power
Out[13]= LeadTerm[1, {1, 1, e^-x, 1}]
```

Multiplikation aller Terme ergibt die eigentliche Asymptote:

```
In[14]:= lt = lt[[1]] * Apply[Times, lt[[2]]]
Out[14]= e^-x
```

Die Überprüfung zeigt, dass die Asymptote gefunden ist:

```
In[15]:= MrvLimit[f/lt, x → ∞]
Out[15]= 1
```

Cachekontrolle

Durch Setzen der Option `ClearCache→True` wird vor der Grenzwertberechnung eine Leerung des Algorithmus-internen Ergebniscache veranlasst. Jeder einmal berechnete Grenzwert wird darin aufbewahrt, um wiederholte Berechnungen des gleichen Grenzwertes zu verhindern. Eine Leerung des Caches ist erforderlich, wenn sich Rahmenbedingungen verändern, die Auswirkungen auf die Grenzwertberechnung haben:

```
In[16]:= Unprotect[Sign]; Sign[a] = 1; Protect[Sign];

         MrvLimit[e^a x, x → ∞]
Out[16]= ∞
```

```
In[17]:= Unprotect[Sign]; Sign[a] = -1; Protect[Sign];

         MrvLimit[e^(a x), x → ∞]
Out[17]= ∞
```

Dieses Ergebnis ist falsch, da das angenommene Vorzeichen von `a` Auswirkung auf den Grenzwert hat, und damit das gecachete Ergebnis keine Gültigkeit mehr hat.

```
In[18]:= MrvLimit[e^(a x), x → ∞, ClearCache → True]
Out[18]= 0

In[19]:= Unprotect[Sign]; Sign[a] = .; Protect[Sign];
```

Alternativ kann der Cache durch Aufruf von `MrvLimitClearCache[]` geleert werden.

`MrvLimitGetCacheStats[]` liefert eine statistische Ausgabe über den Nutzungsgrad des Caches:

```
In[20]:= MrvLimitGetCacheStats[]
Out[20]= {CacheHits → 52, CacheMisses → 13}
```

`CacheHits` gibt dabei an, wie viele Grenzwertberechnungen durch gecachete Ergebnisse vermieden werden konnten, `CacheMisses` gibt an, wie viele Grenzwertberechnungen tatsächlich ausgeführt wurden.

Debuginformationen

Mit `Debug→`*n* wird die Debug-Ausgabe des MrvLimit-Algorithmus aktiviert. Dabei gibt *n* an, bis zu welcher rekursiven Schachtelungstiefe die Debuginformationen ausgegeben werden sollen. `Debug→∞` gibt Debuginformationen in beliebiger Schachtelungstiefe aus.

```
In[21]:= MrvLimit[e^x, x → ∞, Debug → 1, ClearCache → True]
```

```
Enter Level 1 Call MrvLimitInf [e^x, x, Debug → ∞]
Calculating limit of e^x
Calculating Mrv set of e^x
Ω = {MrvF [e^x, 3, ∞] }
ω = e^x
Replacing ω → 1/ω : ω = e^-x
Prepare to rewrite g = e^x to A * ω^c
c = -1 A = 1
Rewriting e^x → 1/ω
 f = 1/ω
Calculating power series : f = 1/ω
```

```
Series expansion : 1/ω + O[ω]^2
Dominant term : 1/ω , ω = e^-x
Dominant asymptotic : 1 1/e^-x
Leave Level 1 Call MrvLimitInf[e^x, x, Debug → ∞] == ∞
Out[21]= ∞
```

5.1.4. Hilfsfunktionen

ToInternal / FromInternal

MrvLimit verwendet intern eine Funktionsdarstellung, in der bestimmte Funktionen durch alternative Schreibweisen ersetzt werden. Durch diese Ersetzung werden störende Effekte durch automatische Transformationen vermieden. MrvLimit übersetzt die Eingabe und Ausgabe automatisch in die interne Darstellung und zurück. Nur bei einigen fortgeschrittenen Eingriffsmöglichkeiten trifft man auf die interne Darstellung.

Funktionen und ihre interne Darstellung in `MrvLimit`:

Funktion	Darstellung
`Power`	`power`
`Log`	`log`
`Sin`	`sin`
`Cos`	`cos`
`Tan`	`tan`
`ArcSin`	`arcsin`
`ArcCos`	`arccos`
`ArcTan`	`arctan`
`Gamma`	`gamma`
`LogGamma`	`loggamma`
`PolyGamma`	`polygamma`

Die Anzeige der Funktionen durch Mathematica entspricht jeweils der normalen Anzeige:

```
In[22]:= power[e, x] + Power[e, x]
Out[22]= e^x + e^x
```

Zur Konvertierung zwischen Mathematica-Darstellung und normaler Darstellung dienen die Hilfsfunktionen `ToInternal[`f`,`x`]` und `FromInternal[`f`,`x`]`:

```
In[23]:= f = ToInternal[e^x * Sin[x]^c, x]
Out[23]= e^x Sin[x]^c

In[24]:= f = f/.c → 0
Out[24]= e^x Sin[x]^0
```

```
In[25]:= FromInternal[f, x]
Out[25]= ℯ^x
```

`ToInternal` verarbeitet nur Funktionen, die für den Algorithmus als bekannt gelten:

```
In[26]:= ToInternal[Foo[x], x]
  MrvLimit :: UnsupportedFunction : MrvLimit does not support this function :
  Foo[x]
Out[26]= ToInternal[Foo[x], x]
```

5.1.5. Erweiterungsschnittstellen

Potenzreihen

`MrvLimit` ist abhängig von einem guten Algorithmus zur Potenzreihenentwicklung, kann selbst aber nur auf die beschränkten Möglichkeiten des `Series`-Kommandos von Mathematica zurückgreifen. Oft genügt aber schon eine kleine Modifikation an der auftretenden Funktion, um die Potenzreihe zu ermitteln. Oder es steht ein gänzlich anderer Algorithmus zur Potenzreihenentwicklung zur Verfügung. Über die Variable `$MrvSeriesHead` kann eine solche Methode eingebunden werden:

```
In[27]:= MrvLimit[(ℯ^x x)^(1/x), x → ∞]

 MrvLimit :: SeriesFail : Series failed at ℯ^(log[x/ω]/x)
Out[27]= MrvLimit[(ℯ^x x)^(1/x), x → ∞]
```

```
In[28]:= SeriesHelper[f_, x_, ω_, logω_, dprint_] := Module[{},
            If[FromInternal[f, x] === (x/ω)^(1/x),
              Return[{ToInternal[ℯ^(1+Log[x]/x), x], 0}];
            ];
            MrvSeriesHead[f, x, ω, logω, dprint]
          ];

          $MrvSeriesHead = SeriesHelper;

          res = MrvLimit[(ℯ^x x)^(1/x), x → ∞, ClearCache → True];

          $MrvSeriesHead = MrvSeriesHead;

          res
Out[28]= ℯ
```

Für die genaue Implementierung der Schnittstelle sollte der Quelltext der Funktion `MrvSeriesHead` herangezogen werden.

Nulltests

Genau wie das Series-Kommando von Mathematica sind auch die Nulltests eine kritische Schwäche des Algorithmus. Auch sie können einfach ersetzt werden. `$MrvTestZero` verweist auf eine Funktion, die zum Testen von Termen auf den Wert 0 verwendet wird. Die Standardvorgabe ist `$MrvTestZero = MrvTestZero` mit:

```
MrvTestZero[term_,x_] := Simplify[FromInternal[term,x] == 0];
```

Alternativ bietet sich die schnellere, aber ungenauere folgende Variante an:
`FromInternal[term,x] == 0`,
oder die gründlichere, aber erheblich langsamere folgende Variante :
`Simplify[FromInternal[term, x] == 0, Element[x, Reals]]`
Mathematica steuert außerdem noch die folgende Variante bei:
`Developer'ZeroQ[FromInternal[term, x]]`

Um Funktionen in der Umgebung eines Grenzwertes (normalerweise ∞) auf die Nullfunktion zu testen, wird `$MrvTestZeroInterval` verwendet. Die Standardvorgabe ist hierbei `$MrvTestZeroInterval = MrvTestZeroInterval`, definiert als:

```
MrvTestZeroInterval[term_,x_→∞] := Simplify[FromInternal[term, x]
== 0];
```

Neben den bereits genannten Varianten kann hier auch die konkrete Umgebung berücksichtigt werden, zum Beispiel mit `Simplify[FromInternal[term, x] == 0, x > 10^1000]`. Leider ist auch dieser Aufruf mit unkalkulierbaren Laufzeiten verbunden, weshalb als Standard nur ein einfacher Nulltest ohne Bezug auf x durchgeführt wird.

Debugausgabe

Falls man an einer anderen Ausgabeform der `Debug`-Option interessiert ist, kann man diese über `$DebugPrint` bekannt machen:

```
In[29]:= MyPrint[Indent_, Text__] := Print["Debug[", Indent, "] : ", Text];

         $DebugPrint = MyPrint;

         MrvLimit[e^x, x → ∞, Debug → 1];

         $DebugPrint = IndentPrint;

         Debug[0] : Enter Level 1 Call MrvLimitInf[e^x, x, Debug → 1]

         Debug[0] : Taking cached result

         Debug[0] : Leave Level 1 Call MrvLimitInf[e^x, x, Debug → 1] == ∞
```

Ausgabesteuerung

Da die Ausgabe von Funktionsteilen und die Debugausgabe sehr umfangreich sein kann, ist es u.U. sinnvoll, solche Ausgaben zu filtern. Mit der Variable `$OutputProcessor` kann jede Debugausgabe und jede Meldungsausgabe nachbearbeitet werden:

In[30]:=

```
f = e^(e^(h-x)+h-x/(1+h)) /. h- -> 1/(1+e^-x);

$OutputProcessor = (FromInternal[#, x] /. {1/(1+e^-x) -> h} &);

MrvLimit[f, x -> ∞, Debug -> 1, ClearCache -> True];

$OutputProcessor = Identity;
```

```
Enter Level 1 Call MrvLimitInf[e^(e^(-x+h)+h-x/(1+h)), x, Debug -> 1]
Calculating limit of e^(e^(h-x)+h-x/(1+h))
Calculating Mrv set of e^(e^(h-x)+h-x/(1+h))
Ω = {MrvF[e^-x, 5, 0], MrvF[e^(h-x), 15, 0], MrvF[e^(e^(h-x)+h-x/(1+h)), 43, 0]}
ω = e^-x
Prepare to rewrite g = e^(e^(h-x)+h-x/(1+h)) to A * ω^c
c = 1/2 A = e^(e^(h-x)+h+x/2-x/(1+h))
Rewriting e^(e^(h-x)+h-x/(1+h)) → e^(e^(h-x)+h+x/2-x/(1+h)) √ω
 f = e^(e^(h-x)+h+x/2-x/(1+h)) √ω
Prepare to rewrite g = e^(h-x) to A * ω^c
c = 1 A = e^h
Rewriting e^(h-x) → e^h ω
 f = e^(h+x/2-x/(1+h)+e^h ω) √ω
Prepare to rewrite g = e^-x to A * ω^c
c = 1 A = 1
Rewriting e^-x → ω
 f = e^(x/2+e^(1/(1+ω)) ω+1/(1+ω)-x/(1+1/(1+ω))) √ω
Calculating power series : f = e^(x/2+e^(1/(1+ω)) ω+1/(1+ω)-x/(1+1/(1+ω))) √ω
Series expansion : e √ω + O[ω]^(3/2)
Dominant term : e √ω , ω = e^-x
Dominant asymptotic : e √(e^-x)
Leave Level 1 Call MrvLimitInf[e^(e^(-x+h)+h-x/(1+h)), x, Debug -> 1] == 0
```

5.2. Quelltextdokumentation

Kommen wir zum primären Ergebnis der Diplomarbeit, dem Mathematica-Paket MrvLimit. Der folgende Quelltext steht selbstverständlich zum Download bereit.

Die hier wiedergegebene Version 1.0 des MrvLimit-Pakets enthält neben den normalen Kommentaren noch zusätzliche weiterführende Hinweise.

5.2.1. Paketanfang

```
(* MrvLimit v1.0 (c) 2005 by Udo Richter          *)
(* mail:udo_richter(at)gmx.de                     *)
(* web:http://urichter.cjb.net/MrvLimit           *)
(* Released under the GNU General Public License. *)
(* "$Id: MrvLimit.m 97 2005-05-11 00:57:18Z udo $" *)


BeginPackage["MrvLimit`"];
```

Mit BeginPackage beginnt der öffentliche Teil des Pakets. Alle nun folgenden Symbole stehen nach dem Laden des Pakets zur Verfügung. Deswegen folgt als nächstes eine Aufzählung aller öffentlichen Symbole. Um bei Namensähnlichkeiten keine Warnungen auszulösen, werden vorübergehend die General::spell Warnungen deaktiviert.

```
Off[General::spell1];
Off[General::spell];
{power,pwr,log,sin,cos,tan,arcsin,arccos,arctan,gamma,loggamma,polygamma,
    Output,output,direction,term,Limit,LeadTerm,SignedZero,Debug,Automatic,
    Direction,ClearCache,MrvLimitGetCacheStats,MrvLimitClearCache,CacheHits,
    CacheMisses,ω,MrvF,MrvSeriesHead,$MrvSeriesHead,$DebugPrint,
    IndentPrint,MrvTestZero,$MrvTestZero,
    MrvTestZeroInterval,$MrvTestZeroInterval,$OutputProcessor,
    MrvLimitLoadSilent,Zero,FromInternal,ToInternal};
On[General::spell1];
On[General::spell];
```

Es folgen die Hilfetexte für die exportierten Funktionen. Die Hilfetexte können mit ?MrvLimit usw. abgefragt werden.

```
ToInternal::usage=
    "ToInternal[f,x] translates f[x] to internal package form, making sure \
that no critical automatic transformations will be done on f any more. If \
ToInternal succeeds, return value will be free of ToInternal calls (check \
```

```
using FreeQ). If ToInternal fails, an UnsupportedFunction error is issued, \
and the failing ToInternal call will be encapsulated in HoldForm.";

FromInternal::usage=
    "FromInternal[f,x] translates internal representation of f[x] back to \
general Mathematica representation. See ToInternal[f,x] for details.";

MrvLimit::usage=
    "MrvLimit[f,x->lim,options] calculates the limit of f for x approaching \
lim. If MrvLimit succeeds, the limit value will be returned. In  case of \
two-sided limits, a list of two results may be returned, in case that they \
are different. If MrvLimit fails, an error message will be issued, and \
MrvLimit will return the whole call encapsulated in HoldForm. \nPossible \
Options:\n Direction: One of -1, +1, 0 or Automatic. Defaults to Automatic. \
+1 will calculate limits from below, -1 from above, 0 both sided. Automatic \
does both sided limits for finite limit processes, and matching sides for \
infinite.\n Output: Limit, LeadTerm or SignedZero. Default Limit. LeadTerm \
will return a LeadTerm[] object describing the found asymptotic. See LeadTerm \
help. SignedZero will return Zero or -Zero if the limit is 0.\n Debug: \
positive integer n or Automatic. Default Automatic. Enables debugging output \
down to nested level n in recursive calls. Automatic mode will detect \
recursive calls and inherit predecessor's debug level minus 1.";


LeadTerm::usage =
    "Output->LeadTerm is an option to MrvLimit, to return the found asymptotic \
behavior of the function. To preserve its structure, the result will be \
in internal form, see ToInternal.\nGeneral result form is LeadTerm[c,a], \
with a of form {pwr[ω1, e1], pwr[ω2, e2], ..}.\ωi is an asymptotic with \
ωi → 0, ωi > 0 for x → ∞. ei is the exponent order of the asymptotic, and may be \
0. The a list is ordered by dominance, dominating asymptotics first. c is \
the leading term constant.\nThe general asymptotic behavior is \
c*Apply[Times,a]\.pwr->Power.";
```

Es folgen die Texte der diversen Fehlermeldungen, die mittels Message ausgegeben werden.

```
MrvLimit::"LimitFail"="Limit failed at ‘1‘ on ‘2‘";
MrvLimit::"UnknownSign"="Cannot determine sign: ‘1‘";
MrvLimit::"SeriesFail"="Series failed at ‘1‘";
MrvLimit::"SeriesNoTerms"=
    "Series failed at ‘1‘: No leading term found, possibly undetected Zero \
function";
MrvLimit::"ZeroFunction"="Unexpected Zero function detected.";
MrvLimit::"UnknownOutput"="Unknown output type Output→‘1‘";
MrvLimit::"MrvSetFailed"="MrvSet failed at ‘1‘";
MrvLimit::"UnsupportedFunction"=
    "MrvLimit does not support this function: ‘1‘";
MrvLimit::"UnsupportedArgument"=
    "MrvLimit does not support ‘1‘ for ‘2‘, appearing in ‘3‘";
MrvLimit::"RecursiveCall"="Recursive Limit call encountered on ‘1‘";
MrvLimit::"LostContext"=
```

```
    "Lost Link to aborted MrvLimit context detected. Skipping.";
MrvLimit::"UnknownOption"="Unknown Option ‘1‘";
MrvLimitInf::"UnknownOption"="Unknown Option ‘1‘";
MrvLimit::"ConditionCheck"="Cannot check important condition ‘1‘";
MrvLimit::"Assert"="Assertion check failed: ‘1‘ on ‘2‘";

Begin["‘Private‘"];
```

Damit endet der öffentliche Teil des MrvLimit-Pakets. Alle folgenden Deklarationen sind Teil des privaten Symbolkontextes und werden durch das Laden des Pakets nicht direkt verfügbar. Zunächst folgen wieder einige Symboldeklarationen, wobei General::spell Warnungen wieder deaktiviert sind.

```
Off[General::spell1];
Off[General::spell];
{Res,ResA,ResB};
On[General::spell1];
On[General::spell];

(* Greetings *)
If[!(MrvLimitLoadSilent===True)
    (*then*),
    Print[
        "MrvLimit v1.0 (c) 2005 by Udo Richter\n mail: \
udo_richter(at)gmx.de\n web: http://urichter.cjb.net/MrvLimit\n Released \
under the GNU General Public License."];
];(* end if *)
```

Die Begrüßungsnachricht wird beim Laden des Pakets ausgegeben, es sei denn, die Variable MrvLimitLoadSilent wurde vorab gesetzt.

5.2.2. Interne Darstellung

Es folgt der erste echte Code. Die Funktion ToInternal dient dazu, Formelterme in die ‚sichere‘ interne Darstellung zu überführen. Das bedeutet in der Regel, einige kritische Mathematica-Symbole durch gleichlautende Symbole in Kleinbuchstaben zu ersetzen. Weitere Parametertests finden hier nicht statt.

ToInternal nutzt Mathematicas Spezialität, Funktionen für bestimmte Muster in den übergebenen Parametern separat zu definieren. Damit wird der Aufruf von ToInternal rekursiv auf die Argumente weiter gereicht, nachdem jeweils die äußerste Funktion durch die interne Darstellung ersetzt wurde.

```
(* Translate term to internal restricted function space *)

ToInternal[f_,x_]:=f/;FreeQ[f,x];
ToInternal[x_,x_]:=x;
ToInternal[g_*h_,x_]:=ToInternal[g,x]*ToInternal[h,x];
ToInternal[g_+h_,x_]:=ToInternal[g,x]+ToInternal[h,x];
ToInternal[Power[g_,c_],x_]:=
    power[ToInternal[g,x],ToInternal[c,x]]/;FreeQ[c,x];
ToInternal[Power[E,g_],x_]:=power[E,ToInternal[g,x]];
ToInternal[Power[f_,g_],x_]:=power[E,log[ToInternal[f,x]]*ToInternal[g,x]];
ToInternal[Log[g_],x_]:=log[ToInternal[g,x]];

ToInternal[Sin[f_],x_]:=sin[ToInternal[f,x]];
ToInternal[Cos[f_],x_]:=cos[ToInternal[f,x]];
ToInternal[Tan[f_],x_]:=tan[ToInternal[f,x]];
ToInternal[Sec[f_],x_]:=1/cos[ToInternal[f,x]];
ToInternal[Csc[f_],x_]:=1/sin[ToInternal[f,x]];
ToInternal[Cot[f_],x_]:=1/tan[ToInternal[f,x]];

ToInternal[ArcSin[f_],x_]:=arcsin[ToInternal[f,x]];
ToInternal[ArcCos[f_],x_]:=arccos[ToInternal[f,x]];
ToInternal[ArcTan[f_],x_]:=arctan[ToInternal[f,x]];
ToInternal[ArcSec[f_],x_]:=arccos[1/ToInternal[f,x]];
ToInternal[ArcCsc[f_],x_]:=arcsin[1/ToInternal[f,x]];
ToInternal[ArcCot[f_],x_]:=arctan[1/ToInternal[f,x]];

ToInternal[Gamma[f_],x_]:=gamma[ToInternal[f,x]];
ToInternal[LogGamma[f_],x_]:=loggamma[ToInternal[f,x]];
ToInternal[PolyGamma[n_,f_],x_]:=polygamma[n,ToInternal[f,x]];

(* Pass-through if already in internal form: *)
ToInternal[power[f_,g_],x_]:=power[ToInternal[f,x],ToInternal[g,x]];
ToInternal[log[f_],x_]:=log[ToInternal[f,x]];
ToInternal[sin[f_],x_]:=sin[ToInternal[f,x]];
ToInternal[cos[f_],x_]:=cos[ToInternal[f,x]];
ToInternal[tan[f_],x_]:=tan[ToInternal[f,x]];
ToInternal[arcsin[f_],x_]:=arcsin[ToInternal[f,x]];
ToInternal[arccos[f_],x_]:=arccos[ToInternal[f,x]];
ToInternal[arctan[f_],x_]:=arctan[ToInternal[f,x]];
ToInternal[gamma[f_],x_]:=gamma[ToInternal[f,x]];
ToInternal[loggamma[f_],x_]:=loggamma[ToInternal[f,x]];
ToInternal[polygamma[n_,f_],x_]:=polygamma[n,ToInternal[f,x]];

(* Catch all remaining: *)
ToInternal[f_,x_]:=(
      Message[MrvLimit::"UnsupportedFunction",f];
      HoldForm[ToInternal[f,x]]
);
```

Die abschließende allgemeine Definition für ToInternal wird immer dann angewendet, wenn keine der vorherigen spezielleren Formen auf den Term passt. Da für alle unterstützten Terme

eine spezielle Form vorliegt, tritt diese Definition nur für nicht unterstützte Funktionsaufrufe in Kraft.

Als Reaktion wird eine Warnmeldung ausgegeben und der Aufruf ‚nicht ausgewertet' zurück gegeben, d.h. in HoldForm eingebettet, um den sofortigen rekursiven Aufruf zu verhindern. Um zu testen, ob ein Aufruf von ToInternal erfolgreich war, empfiehlt sich, mit FreeQ auf das Vorhandensein des Symbols ToInternal zu testen.

```
(* Remove all internal stuff: *)
FromInternal[f_,x_]:=
    f/.{log→Log,power→Power,sin→Sin,cos→Cos,
        tan→Tan,arcsin→ArcSin,arccos→ArcCos,
        arctan→ArcTan,gamma→Gamma,loggamma→LogGamma,
        polygamma→PolyGamma
    };
```

FromInternal ist das Gegenstück zu ToInternal. Auf eine komplexe Analyse wird hier verzichtet, da die übergebenen Terme normalerweise nicht vom Anwender stammen und auch keine unbekannten Symbole behandelt werden müssen.

```
(* Some very basic simplification rules *)
SimpleSimplifications={
      power[z_,0]→1,
      power[z_,1]→z,
      power[E,log[z_]]→z,
      log[power[E,z_]]→z,
      power[power[z_,e_],-1]→power[z,-e],
      power[-power[z_,e_],-1]→-power[z,-e]
};
```

Da in interner Darstellung keine der normalen Vereinfachungen Gültigkeit hat, bleiben selbst Terme wie x^0, x^1, $e^{\log x}$, $\log e^x$, $(x^k)^{-1}$ und $(x^{-1})^k$ unverändert. Diese sehr einfachen Transformationen kann man in geeigneten Momenten durch manuelles Anwenden der SimpleSimplifications erreichen.

Als nächstes wird die visuelle Darstellung der internen Funktionen verbessert. Damit bei Fehlermeldungen und der Debugausgabe nicht power[E,x], sondern e^x ausgegeben wird, muss ein geeigneter Aufruf für MakeBoxes definiert werden, der die Darstellung der internen Funktionsnamen erzeugt.

Um das Rad nicht neu erfinden zu müssen, wird der Aufruf einfach an den MakeBoxes-Aufruf für das öffentliche Pendant der Funktion weiter gereicht.

```
(* Display internal functions like public functions *)

log/:MakeBoxes[log[x_],form_]:=MakeBoxes[Log[x],form];
power/:MakeBoxes[power[x_,y_],form_]:=MakeBoxes[Power[x,y],form];

Unprotect[Times];
Times/:MakeBoxes[HoldPattern[Times[t___]],form_]:=Module[{h,i},
        h=HoldComplete[MakeBoxes[Times[t],form]];
        Do[
          If[h[[1,1,i,0]]===power,
              h[[1,1,i,0]]=Power;
          ];(*end if*)
          (*do for*),
          {i,1,Length[{t}]}
        ];
        ReleaseHold[h]
]/;MemberQ[{t},power[___]];
Protect[Times];
```

Eine Besonderheit bleibt: Mathematica stellt bekanntlich Brüche intern als Multiplikation mit Kehrwerten dar. $\frac{a \cdot b}{c \cdot d}$ hat so die interne Darstellung Times[a,b,Power[c,-1],Power[d,-1]]. Damit dessen Darstellung nicht ebenfalls $a \cdot b \cdot \frac{1}{c} \cdot \frac{1}{d}$ ist, wird bereits bei der Times-Darstellung auf eventuell vorhandene Potenzen geachtet. Deswegen muss auch für ‚power' eine Darstellung von Times definiert werden.

Erschwert wird das Ganze durch die Tatsache, dass MakeBoxes unter HoldComplete-Bedingungen aufgerufen wird. Das übergebene Argument muss daher sorgfältig in HoldComplete aufbewahrt werden, Veränderungen können nur durch Ersetzungsregeln oder direkten Zugriff erfolgen. Keinesfalls darf es zur mathematischen Auswertung des Times-Terms oder der tiefer gelegenen Terme kommen.

Abschließend wird noch über die Bedingung festgelegt, dass diese Definition nur dann Gültigkeit hat, wenn mindestens ein power-Term in Times direkt vorkommt.

```
sin/:MakeBoxes[sin[x_],form_]:=MakeBoxes[Sin[x],form];
cos/:MakeBoxes[cos[x_],form_]:=MakeBoxes[Cos[x],form];
tan/:MakeBoxes[tan[x_],form_]:=MakeBoxes[Tan[x],form];
arcsin/:MakeBoxes[arcsin[x_],form_]:=MakeBoxes[ArcSin[x],form];
arccos/:MakeBoxes[arccos[x_],form_]:=MakeBoxes[ArcCos[x],form];
arctan/:MakeBoxes[arctan[x_],form_]:=MakeBoxes[ArcTan[x],form];
gamma/:MakeBoxes[gamma[x_],form_]:=MakeBoxes[Gamma[x],form];
loggamma/:MakeBoxes[loggamma[x_],form_]:=MakeBoxes[LogGamma[x],form];
polygamma/:MakeBoxes[polygamma[n_,x_],form_]:=MakeBoxes[PolyGamma[n,x],form];
```

5.2.3. Tools

Es folgen einige allgemeine nützliche Mathematica-Hilfsfunktionen.

```
(* General Tools *)

(* Extract context part of module variables *)
GetSymbolModuleContext[a_Symbol]:=Module[{n,p},
      n=SymbolName[Unevaluated[a]];
      p=StringPosition[n,"$"];
      If[p=={},"",StringDrop[n,p[[1,1]]-1]]
];
SetAttributes[GetSymbolModuleContext,HoldFirst];

(* Extract variable part of module variables *)
GetSymbolModuleName[a_Symbol]:=Module[{n,p},
      n=StringJoin[Context[Unevaluated[a]],SymbolName[Unevaluated[a]]];
      p=StringPosition[n,"$"];
      If[p=={},n,StringTake[n,p[[1,1]]-1]]
];
SetAttributes[GetSymbolModuleName,HoldFirst];
```

GetSymbolModuleContext erlaubt es, den Kontext-Teil von lokalen Variablen innerhalb von Module[] oder von Variablen von Unique[] zu extrahieren und als String zurück zu geben. Der Kontext-Teil hat dann die Form ‚$nnn‘. Da das Attribut HoldFirst gesetzt wird, kann der Variablen sogar bereits ein Wert zugewiesen worden sein.

GetSymbolModuleName ist das Gegenstück zur vorherigen Funktion und liefert den Symbolnamen ohne den Kontext-Teil zurück.

```
(* Translate module symbol name to global symbol name *)
Public[a_Symbol]:=Symbol[GetSymbolModuleName[Unevaluated[a]]];
SetAttributes[Public,HoldFirst];

(* Change a symbol's module context to a different one *)
SetModuleContext[a_Symbol,context_]:=
  Symbol[StringJoin[GetSymbolModuleName[Unevaluated[a]],context]]
SetAttributes[SetModuleContext,HoldFirst];
```

Public ist eine Anwendung der vorherigen Funktionen. Public erzeugt zu einem lokalen Modulsymbol ein passendes globales Symbol gleichen Namens. Dies ist zum Beispiel dann sinnvoll, wenn in einem Modul ein Term ausgegeben werden soll, der lokale Symbole beinhaltet, welche andernfalls mit sichtbarem $nnn-Teil ausgegeben würden.

SetModuleContext ersetzt den Kontext-Teil eines lokalen Modulsymbols durch einen anderen Kontext. Dadurch wird der Zugriff auf lokale Symbole anderer Modul-Kontexte ermöglicht. Als Beispiel kann der folgende Code-Schnipsel dienen:

```
(* Nicht Teil des Quelltextes! *)

        Module[Local,Link,
            Local = "Modul 1";
            Link = GetSymbolModuleContext[Local];
            Module[Local,
                Local = "Modul 2";
                Print[Local];
                (* Gibt "Modul 2" aus *)
                Print[SetModuleContext[Local, Link]];
                (* Gibt "Modul 1" aus *)
            ];
        ];
```

Im zweiten Modul überdeckt zunächst das lokale Symbol Local das gleichnamige Symbol des ersten Moduls. Da aber die Kontextkennung des ersten Moduls über Link bekannt ist, kann auch das zweite Modul auf die Variablen des ersten Moduls lesend zugreifen. Ein Schreibzugriff ist so allerdings nicht möglich. Wichtig ist auch, dass sämtliche lokalen Variablen am Ende des Moduls ihre zugewiesenen Werte wieder verlieren.

Sinnvoll ist eine solche Verbindung bei rekursiven Aufrufen. Hat ein vorheriger Aufruf seine Modulkennung in einer globalen Variable zurück gelassen, können rekursive Aufrufe des gleichen Moduls auf die lokalen Symbole des Vorgängers lesend zugreifen.

Es folgen einige Hilfsfunktionen, um Funktionen mit optionalen Parametern der Form Name → Wert zu unterstützen:

```
(* Variable options handler *)
(* Handles optional parameters of name→value form *)
(* Example: *)
(* MyFunction::"UnknownOption"=                              *)
(*   "MyFunction does not support this option: ‘1‘";         *)
(* Options[MyFunction]={Foo→True,Bar→False};                 *)
(* MyFunction[Baz_,Options___] := Module[{MyFoo,MyBar},      *)
(*   ReadOptions[MyFunction,{Foo→MyFoo,Bar→MyBar},Options]; *)
(*   Print["Foo=",MyFoo," Bar=",MyBar];                      *)
(* ];                                                        *)

Clear[ReadOptions];
ReadOptions[Caller_,OptionList_,OptionSeq___]:=Module[{FullOptions,pos},
      (* List of Option,Destination,Default *)
      FullOptions=Map[{#[[1]],Null,#[[2]]}&,Options[Caller]];

      (* Add Destinations *)
      Map[
        (FullOptions[[Position[FullOptions,{#[[1]],_,_}][[1,1]],2]]=#[[2]])&
        ,OptionList
      ];
```

```
      (* Process sequence of options, replace defaults if found *)
      Map[
        (
          If[!MatchQ[#,_→_],
            Message[Caller::"UnknownOption",#];
            (*else*),
            pos=Position[FullOptions,{#[[1]],_,_}];
            If[Length[pos]==0,
              Message[Caller::"UnknownOption",#]
              (*else*),
              FullOptions[[pos[[1,1]],3]]=#[[2]]
            ];
          ];
        )&,
        {OptionSeq}
      ];

      (* Assign results *)
      Map[(Evaluate[#[[2]]]=#[[3]])&,
        Select[FullOptions,!MatchQ[#,{_,Null,_}]&]];
];
```

Erster Parameter des Aufrufs von ReadOptions ist der Funktionsname. Dieser wird für den Zugriff auf die Standardoptionen und für den Text der Fehlermeldung verwendet. Deshalb sollte vor der Benutzung für jede Option in Options[Funktionsname] ein Default-Wert hinterlegt werden. Mehr dazu in der Mathematica-Hilfestellung zu Options[]. Außerdem sollte ein Fehlertext in der Form MyFunction:: „UnknownOption“ = „Fehlertext“ angelegt werden. Mehr dazu in der Mathematica-Hilfestellung zu Message[]. Diese Fehlermeldung wird immer dann ausgegeben, wenn eine Option angegeben wurde, die nicht in Options[] aufgeführt ist.

Zweiter Parameter ist eine Liste von Optionszuweisungen zu (lokalen) Symbolen in der Form {Optionsname→LokalerName, ...}. Nicht jeder optionale Parameter muss hier aufgeführt sein, entscheidend für die Anerkennung als gültige Option ist das Vorhandensein eines Default-Wertes in Options[]. ReadOptions wird in den angegebenen lokalen Symbolen die angegebenen Optionen oder die Default-Optionen hinterlegen. Den lokalen Symbolen dürfen vor dem Aufruf von ReadOptions noch keine Werte zugewiesen worden sein!

Ab dem dritten Parameter folgen die auszuwertenden Optionen. Wurde die aufrufende Funktion mit Options__ oder Options___ deklariert, kann diese Sequenz direkt an ReadOptions weitergegeben werden.

```
(* Remove some options from a sequence of options *)
Clear[DropOptions];
DropOptions[DropOpts_,Options___]:=Module[{opt,i},
      opt={Options};
      Do[
        opt=Select[opt,(!MatchQ[#,DropOpts[[i]]→_])&];
        (*do on*),
        {i,1,Length[DropOpts]}
      ];
      Apply[Sequence,opt]
];
```

Diese Hilfsfunktion dient dazu, bestimmte Optionen aus einer Sequenz herauszufiltern, um dann beispielsweise die verbleibenden Optionen an eine andere Funktion weiterzugeben.

Erster Parameter ist eine einfache Liste von Optionsnamen, die entfernt werden sollen. Alle weiteren Parameter werden als Optionen betrachtet und gefiltert. Zurückgegeben wird wieder eine Sequenz von Optionen.

```
(* Debug Print function *)
(* IndentPrint allows to indent print output *)

IndentPrint[Spacing_,Text__]:=
    Print[DisplayForm[AdjustmentBox[ToBoxes[SequenceForm[Text]],BoxMargins->
                {{Spacing,0}, {0, 0}}]]];

(* Interface to set own debug printing routines *)

$DebugPrint=IndentPrint;

(* Interface to modify debug and message output *)

$OutputProcessor=Identity;
```

Dies ist der Standard-Handler für Debugausgaben. IndentPrint akzeptiert als ersten Parameter einen Einrückwert, um den die Print-Ausgabe eingerückt wird. Dies wird in MrvLimit die Rekursionsstufe sein. Die weiteren Parameter werden wie bei Print[] gehandhabt.

Für $OutputProcessor ist kein besonderer Standard-Handler vorgesehen.

```
(* Global Zero Test code *)

MrvTestZero[term_,x_]:=Simplify[FromInternal[term,x]==0];

$MrvTestZero=MrvTestZero;
(* test if constant is 0 *)

MrvTestZeroInterval[term_,x_→∞]:=
    Simplify[FromInternal[term,x]==0];

$MrvTestZeroInterval=MrvTestZeroInterval;
(* test if function is identical 0 in interval (x0,∞) for some x0. *) \

(* Sorry, no idea how to check this properly. *)
```

Dies sind die Standard-Handler für Nulltests, siehe Kapitel 5.1.5.

5.2.4. MrvSet

Als nächstes folgt die Implementierung der Funktion MrvSet, vgl. Kapitel 4.5.

MrvSet liefert eine Liste von Objekten der Form MrvF[f, lc, lim] zurück, die die Teilausdrücke der stärksten Wachstumsklasse repräsentieren. f stellt den entsprechenden Teilausdruck der Funktion dar, lc ist gleich LeafCount[f], und lim ist das Grenzverhalten für x→∞. Die Liste ist immer nach lc sortiert, so dass das MrvF-Objekt mit dem kleinsten lc zuerst kommt.

Alle Funktionen rund um MrvSet liefern im Fehlerfall Null zurück, der Aufrufer kann aber davon ausgehen, dass die Ursache für den Fehler bereits durch eine Meldung bekannt gegeben wurde.

Zunächst kommen einige Hilfsfunktionen.

```
(* MrvSet is a helper function to calculate the set of most varying sub \
expressions. *)
(* Call MrvSet[f,
      x] with function f in internal representation and variable x. *)

(* Result is a list of MrvF[f,lc,lim] objects. *)
(* f represents one of the most varying sub expressions. f is either x,
  or of e^_ form. *)
(* lc is LeafCount[f]. *)
(* lim is the limit behavior of f for x→∞.
         This can only be +∞ or 0. *)
(* The resulting list is always sorted by lc, lower (simpler) results first. *)

(* Compare two terms regarding varying class.
      Optimized for results of MrvSet *)
MrvCompareMrvSet[exp1_,exp2_,x_]:=Module[{l,e1,e2},
      If[exp1===exp2,Return[0]];
      (* This esp. catches exp1=x, exp2=x *)

      If[exp1===x ,
        (*then*)
        e1=log[x],
        (*else*)
        If[!MatchQ[exp1,power[E,_]],
          Message[MrvLimit::"Assert","Should be exp:",exp1]];
        e1=exp1[[2]];
      ];

      If[exp2===x ,
        (*then*)
        e2=log[x],
        (*else*)
        If[!MatchQ[exp2,power[E,_]],
          Message[MrvLimit::"Assert","Should be exp:",exp2]];
        e2=exp2[[2]];
      ];
```

```
      (*calculate limit of Log[exp1]/Log[exp2] recursively*)
      l=MrvLimitInf[e1*power[e2,-1],x];
      If[l===Null,
        Message[MrvLimit::"LimitFail",MrvCompareMrvSet,e1/e2];
        Return[Null];
      ];
      If[$MrvTestZero[l,x],Return[1]];
      If[Abs[l]==∞,Return[-1]];
      Return[0];
];
```

Die Hilfsfunktion MrvCompareMrvSet vergleicht die Wachstumsklasse zweier Terme durch rekursive Grenzwertaufrufe. Die Funktion ist dabei nur für Terme geeignet, die die Form $e^{f(x)}$ haben oder direkt x sind.

Zurückgegeben wird -1,0,1 je nach Vergleichsergebnis oder Null im Fehlerfall. 1 bedeutet, die Wachstumsklasse des zweiten Arguments ist größer, -1 bedeutet, die Wachstumsklasse des ersten Arguments ist größer. 0 bedeutet, die Wachstumsklassen sind gleich.

Wichtig ist auch die erste Zeile. Der Vergleich exp1===exp2 fängt den trivialen Fall ab, dass exp1=exp2=x ist, der stur nach Algorithmus zu einer unendlichen Rekursion führen würde.

```
(*Calculate union of two MrvSet sets.
      Drop all elements that are not of max varying class. *)

Clear[MrvSetMax];
MrvSetMax[Set1_,Set2_,x_]:=Module[{l,e1,e2},
      If[Head[Set1]=!=List||Head[Set2]=!=List,
        Return[Null];
      ];

      If[Set1==={},Return[Set2]];
      If[Set2==={},Return[Set1]];

      (* Now both sets have at least one member *)

      Switch[MrvCompareMrvSet[Set1[[1,1]],Set2[[1,1]],x],
        1   ,Set2,
        -1  ,Set1,
        0   ,Sort[Union[Set1,Set2],(#1[[2]]<#2[[2]])&],
        Null,Null (* error indicator *)
      ]
];
```

MrvSetMax vergleicht das Wachstumsverhalten von zwei MrvSet-Ergebnismengen. Zum Vergleich wird, falls vorhanden, das erste Element der Menge herangezogen. Da die Mengen nach LeafCount sortiert sind, wird so auch der einfachste Term der Menge zum Vergleich benutzt.

Zurückgegeben wird die stärker wachsende Menge bzw. die Vereinigung der Mengen, falls beide gleich stark wachsen. Die Menge wird natürlich korrekt sortiert zurückgegeben. Im Fehlerfall wird wieder Null zurückgegeben. Insbesondere behandelt MrvSet den Fall, dass eine oder beide Mengen keine Mengen, sondern Null-Fehlermeldungen von rekursiven Aufrufen sind.

```
(*
  MrvSetRules:
    Apply simple rules to Mrv Set while preserving sort structure etc.
        Only apply rule sets that dont change the limiting behavior!
  *)

MrvSetRules[MrvF[f_,lc_,lim_],Rules_List]:=Module[{f2},
      (* Apply rule and re-calculate leaf count *)
      f2=f/.Rules;
      MrvF[f2,LeafCount[f2],lim]
];

MrvSetRules[S_List,Rules_List]:=
    (* Apply rules and re-sort *)

    Sort[
      Map[MrvSetRules[#,Rules]&,S],
      (#1[[2]]<#2[[2]])&
    ];
```

MrvSetRules ist eine Hilfsfunktion, um einfache Regeln auf alle Funktionen eines MrvSet anzuwenden. Nach der Regelanwendung wird der LeafCount erneut ermittelt und die Liste erneut sortiert. Die Regeln dürfen das Wachstumsverhalten (0 oder ∞) nicht beeinflussen, da diese Information ungeprüft übernommen wird.

Es folgt die Implementierung von MrvSet selbst, wieder indem MrvSet separat für verschiedene Funktionsaufrufmuster definiert wird.

```
(* Calculate set of most rapidly varying sub expression, main code *)

Clear[MrvSet];
MrvSet[f_,x_]:={}/;FreeQ[f,x];
MrvSet[x_,x_]:={MrvF[x,1,∞]};
MrvSet[g_*h_,x_]:=MrvSetMax[MrvSet[g,x],MrvSet[h,x],x];
MrvSet[g_+h_,x_]:=MrvSetMax[MrvSet[g,x],MrvSet[h,x],x];
MrvSet[power[g_,c_],x_]:=MrvSet[g,x]/;FreeQ[c,x];
MrvSet[log[g_],x_]:=MrvSet[g,x];
MrvSet[power[E,g_],x_]:=Module[{l,m},
      l=MrvLimitInf[g,x];
      If[l==Null,
        Message[MrvLimit::"LimitFail","MrvSet",g];
        Return[Null];(* error indicator *)
      ];

      If[FreeQ[l,DirectedInfinity],
```

```
        (* Seems const. Pass through Mrv set. *)
        Return[MrvSet[g,x]];
      ];

      If[l===∞,
        (* tends to e^∞. May contribute to Mrv set *)

        m=MrvF[
            power[E,g],
            LeafCount[power[E,g]],
            ∞
          ];
        Return[MrvSetMax[{m},MrvSet[g,x],x]];
      ]; (* end if l===∞ *)


      If[l===-∞,
        (* tends to e^-∞. May contribute to Mrv set *)
        m=MrvF[
            power[E,g],
            LeafCount[power[E,g]],
            0
          ];
        Return[MrvSetMax[{m},MrvSet[g,x],x]];
      ]; (* end if l===-∞ *)


      Message[MrvLimit::"UnknownSign",l];
      Return[Null];
]; (* end MrvSet[power[E, g_], x_] *)
```

Die Implementierung der Grundfunktionen erfolgt recht geradlinig. Der Test auf $\pm\infty$ ist nicht sehr elegant, da er durch die Tatsache erschwert wird, dass $+\infty$ Mathematica-intern als DirectedInfinity[1] dargestellt wird und $-\infty$ als DirectedInfinity[-1]. Der Test auf „enthält nicht Unendlich“ (FreeQ[l,∞]) wäre deshalb falsch. Sicherheitshalber wird hier auch mit der Möglichkeit von komplexer Unendlichkeit oder anderen unbekannten Situationen gerechnet.

```
(* Functions with no essential singularities may pass through: *)
(* Unsupported cases need to be filtered out, of course. *)
MrvSet[sin[f_],x_]:=MrvSet[f,x];
MrvSet[cos[f_],x_]:=MrvSet[f,x];
MrvSet[tan[f_],x_]:=MrvSet[f,x];
MrvSet[arcsin[f_],x_]:=MrvSet[f,x];
MrvSet[arccos[f_],x_]:=MrvSet[f,x];
MrvSet[arctan[f_],x_]:=MrvSet[f,x];
MrvSet[gamma[f_],x_]:=MrvSet[f,x];
MrvSet[loggamma[f_],x_]:=MrvSet[f,x];
MrvSet[polygamma[n_,f_],x_]:=MrvSet[f,x];
```

```
(* Catch all remaining *)
MrvSet[f_,x_]:=(
      Message[MrvLimit::"MrvSetFailed",f];
      Null
);
```

Die restlichen Definitionen für MrvSet sind wieder unspektakulär.

```
(* Calculate the complexity of a term *)

Clear[MrvComplexitySet];
MrvComplexitySet[f_,x_]:={}/;FreeQ[f,x];
MrvComplexitySet[x_,x_]:={x};
MrvComplexitySet[g_*h_,x_]:=
    Union[MrvComplexitySet[g,x],MrvComplexitySet[h,x]];
MrvComplexitySet[g_+h_,x_]:=
    Union[MrvComplexitySet[g,x],MrvComplexitySet[h,x]];
MrvComplexitySet[power[g_,c_],x_]:=MrvComplexitySet[g,x]/;FreeQ[c,x];
MrvComplexitySet[log[g_],x_]:=Append[MrvComplexitySet[g,x],log[g]];
MrvComplexitySet[power[E,g_],x_]:=Append[MrvComplexitySet[g,x],power[E,g]];

(* Functions with no essential singularities may pass through: *)
MrvComplexitySet[sin[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[cos[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[tan[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[arcsin[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[arccos[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[arctan[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[gamma[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[loggamma[f_],x_]:=MrvComplexitySet[f,x];
MrvComplexitySet[polygamma[n_,f_],x_]:=MrvComplexitySet[f,x];

(* catch all remaining *)
MrvComplexitySet[f_,x_]:=(
      Message[MrvLimit::"UnsupportedFunction",f];
      HoldForm[MrvComplexitySet[f,x]]
);

MrvComplexity[f_,x_]:=Module[{c},
      c=MrvComplexitySet[f,x];
      If[!FreeQ[c,MrvComplexitySet],Return[∞]];
      Return[Length[c]];
];
```

Der Programmcode für die Bestimmung der Komplexität eines Terms im Sinne der Beweisführung für die Terminierung rekursiver Aufrufe ist hier nur der Vollständigkeit halber angegeben. Benutzt wird er derzeit nicht, zumal Mathematicas Series-Kommando nicht unbedingt nach den Regeln des Beweises spielt...

5.2.5. MrvSeriesHead

Es folgt die Standard-Implementierung von $MrvSeriesHead, die auf das Series-Kommando von Mathematica zurückgreift.

```
(* Calculate head of power series of f, dominant variable ω,
  secondary variable x. *)
(* logω is a replacement term for Log[ω] and can be substituted \
at any time *)

(* Returns {u,ex} if leading term is u*ωˆex *)
(* May return u=0 only if f is identical 0. *)
```

Zunächst wird aber die Hilfsfunktion SeriesHead definiert, die die Ausgabe des Series[]-Kommandos analysiert und aufbereitet. Normalerweise liefert Series als Ergebnis ein SeriesData-Objekt ab. Gelegentlich handelt es sich beim Ergebnis aber auch um eine Konstante, ein komplettes Polynom (ohne Abbruch-Potenz) oder auch Summe oder Produkt von Polynomterm mit einem SeriesData-Objekt.

SeriesHead ermittelt in all diesen Fällen den führenden Term der Potenzreihe. Zurückgegeben wird eine Menge {c,e}, die den Term $c \cdot x^e$ repräsentiert, oder im Fehlerfall ein relativ unveränderter Term, der noch das Symbol SeriesHead enthält.

```
(* SeriesHead *)
(* Tries hard to interprete the output of the Series[] command. *)
(* Returns {c,e} for leading term c*xˆe *)

Clear[SeriesHead];
SeriesHead[HoldPattern[SeriesData[x_,0,c_List,nmin_,nmax_,den_]],x_]:=
    If[$MrvTestZero[c[[1]],x],
        SeriesHead[SeriesData[x,0,Drop[c,1],nmin+1,nmax,den]]
        (*else*),
        {c[[1]],nmin/den}
        (*undetermined*),
        {c[[1]],nmin/den}
    ]/;Length[c]>0;

SeriesHead[f_,x_]:={f,0}/;FreeQ[f,x];
SeriesHead[c_.*x_ˆe_.,x_]:={c,e}/;FreeQ[c,x];&&FreeQ[e,x];
```

Bei SeriesData-Objekten bestimmt SeriesHead direkt den führenden Term. Sollte dieser sich als 0 herausstellen, wird er aus dem SeriesData-Objekt entfernt, und ein rekursiver Aufruf ermittelt den nächsten führenden Term. Die Bedingung am Ende stellt dabei sicher, dass das SeriesData-Objekt stehen bleibt, wenn die bekannten Terme aufgebraucht sind.

Die nächsten zwei Definitionen leiten Konstanten und einfache polynomielle Terme durch SeriesHead durch.

```
SeriesHead[f_*g_,x_]:=Module[{sf,sg},
      sf=SeriesHead[f,x];
      sg=SeriesHead[g,x];
      If[!FreeQ[sf,SeriesHead]||!FreeQ[sg,SeriesHead],
        Return[sf*sg]
      ];
      Return[{sf[[1]]*sg[[1]],sf[[2]]+sg[[2]]}];
];

SeriesHead[f_+g_]:=Module[{sf,sg},
      sf=SeriesHead[f,x];
      sg=SeriesHead[g,x];
      If[!FreeQ[sf,SeriesHead]||!FreeQ[sg,SeriesHead],
        Return[sf+sg]
      ];
      If[sf[[2]]<sg[[2]],Return[sf]];
      If[sf[[2]]>sg[[2]],Return[sg]];
      If[sf[[2]]==sg[[2]],Return[{sf[[1]]+sg[[1]],sf[[2]]}]];
      Return[sf+sg];
];
```

Diese beiden Definitionen dienen zum Auflösen von Produkten und Summen. Bei Produkten werden die führenden Terme multipliziert, bei Summen gewinnt jeweils die kleinere Ordnung bzw. die Summe bei gleicher Ordnung.

Es folgt der eigentliche Code von MrvSeriesHead.

Der allgemeine Aufruf von MrvSeriesHead lautet MrvSeriesHead[f, x, ω, logω, dprint], und dabei ist:

- f die zu untersuchende Funktion,
- x die Variable, die nur die niedrigeren Wachstumsklassen repräsentiert und im Sinne unserer Potenzreihenentwicklung nur logarithmische Singularitäten darstellt,
- ω die Variable, die die relevante Wachstumsordnung repräsentiert und in der die Potenzreihe ermittelt werden soll (in $\omega = 0$ mit $\omega > 0$),
- logω der Wert von $\ln \omega$ als Term in x und
- dprint ein Verweis auf die Debugausgabe-Funktion von MrvLimit für Debugmeldungen.

MrvSeriesHead liefert wie SeriesHead eine Liste {c,e} zurück, die den Term $c \cdot \omega^e$ repräsentiert oder im Fehlerfall Null. Die Komponenten c,e sind im Gegensatz zu SeriesHead aber in interner Darstellung.

```
Clear[MrvSeriesHead];
MrvSeriesHead[f_,x_,ω_,logω_,dprint_]:=
    Module[{ser,serh,i,u,ex},

      (* Define some local auto simplifications *)
      Unprotect[Log];Unprotect[Power];
      Log[ω]=FromInternal[logω,x];
      Log[1/ω]=-Log[ω];
      Power[E,Log[ω]]=ω;
      Power[E,Log[1/ω]]=1/ω;
      Protect[Log];Protect[Power];
```

Durch das direkte Definieren von $\ln \omega$, $\ln 1/\omega$, $e^{log\omega}$ und $e^{1/log\omega}$ wird erreicht, dass während der Potenzreihenentwicklung eventuell auftretende logarithmische Singularitäten von ω in Terme in x abgebaut werden können und exponentielle Terme in x korrekt zu Termen in ω aufgebaut werden können, wobei letzteres bisher nur im Zusammenhang zur PolyGamma-Funktion aufgetreten ist.

Als nächstes wird die Series-Funktion bemüht. Da vorab keine Ordnung abschätzbar ist, wird solange die Abbruchordnung erhöht, bis ein brauchbarer führender Term entsteht. Leider ist das ein problematisches Verfahren ohne brauchbare Alternative: Hat der führende Term extrem niedrige Ordnung, ermittelt der erste Aufruf viel zu viele unnötige Terme, ist die Ordnung dagegen extrem hoch, wird nie ein brauchbares Ergebnis gefunden. Handelt es sich gar um eine versteckte Nullfunktion, so wird nie ein führender Term zu finden sein, wobei man aber nie sicher sein kann, ob nicht doch noch ein Term kommt.

```
      (* Call Series with increasing order to find terms *)
      (* Ugly, but not to avoid. *)
      Do[
        ser=Series[FromInternal[f,x],{ω,0,i}];
        serh=SeriesHead[ser,ω];

        (* check for series term that run out of precision *)
        If[FreeQ[serh,HoldPattern[SeriesData[_,_,{},_,_,_]]],
          (* No: done. *)
          Break[];
        ];

        (* Increase precision and continue *)

        (* do for *)
        ,{i,1,30}
      ]; (* end do *)
```

Die Schleife bricht bei 30 ab. Eine gute Methode, den Algorithmus zum stolpern zu bringen, ist also, eine Ordnung größer als 30 zu konstruieren...

In der Schleife wird getestet, ob ein SeriesData-Objekt vorliegt, das keine Terme enthält. So ein Objekt entsteht, wenn SeriesHead alle Terme aus SeriesData extrahiert hat, oder SeriesData schon keine Terme liefern konnte. Falls solch ein Objekt vorhanden ist, wird die Ordnung weiter erhöht. Andernfalls wird die Schleife beendet.

```
      (* Remove local auto simplifications *)
      Unprotect[Log];Unprotect[Power];
      Power[E,Log[ω]]=.;
      Power[E,Log[1/ω]]=.;
      Log[ω]=.;
      Log[1/ω]=.;
      Protect[Log];Protect[Power];

      dprint["Series expansion: ",$OutputProcessor[ser]];

      If[!FreeQ[serh,SeriesHead],
        (* SeriesHead could not interprete the result *)
        If[!FreeQ[serh,HoldPattern[SeriesData[_,_,{},_,_,_]]],
          (* No leading terms found *)
          Message[MrvLimit::"SeriesNoTerms",$OutputProcessor[f]];
          Return[Null];
        ];
        Message[MrvLimit::"SeriesFail",$OutputProcessor[f]];
        Return[Null];
      ]; (* end if *)
```

Der Test auf das Vorhandensein eines SeriesHead-Terms deutet auf einen generellen Fehler der SeriesHead-Funktion hin. Es geht im Folgenden dann nur noch darum, welche passende Fehlermeldung ausgegeben werden soll.

```
      {u,ex}=serh;

      (* Check if Series succeded in eliminating ω *)
      If[!FreeQ[u,ω],
        Message[MrvLimit::"SeriesFail",$OutputProcessor[f]];
        Return[Null];
      ];

      u=ToInternal[u,x];
      If[!FreeQ[u,ToInternal],
        Return[Null];
      ];
      ex=ToInternal[ex,x];
      If[!FreeQ[ex,ToInternal],
        Return[Null];
      ];
      {u,ex}
]; (* end MrvSeriesHead *)

$MrvSeriesHead=MrvSeriesHead;
```

Der restliche Code führt noch ein paar Kontrollen durch, konvertiert das Ergebnis in interne Darstellung und liefert das Ergebnis schließlich zurück.

5.2.6. MrvLimit Tools

Die Funktion MrvLimitOutput analysiert LeadTerm-Objekte, wie sie von MrvLimit mit der Option Limit → LeadTerm ausgegeben werden. Diese Objekte werden auch intern von MrvLimit verwendet. Zuvor aber noch eine Hilfsfunktion:

```
(* Helper that interprets leading term results *)

MrvLimitLeadTermDominant[asympt_List]:=Module[{i},
      i=1;
      While[i≤Length[asympt] && asympt[[i,2]]==0,i++];
      If[i>Length[asympt],Return[Null]];
      Return[i];
];
```

Diese Funktion sucht in einer Liste von pwr-Objekten das erste mit Ordnung ungleich 0 heraus oder liefert Null, wenn alle die Ordnung 0 haben.

```
Clear[MrvLimitOutput];
MrvLimitOutput[LeadTerm[const_,asympt_List],Output→Limit]:=
    Module[{i,ex},
      i=MrvLimitLeadTermDominant[asympt];

      If[i===Null,Return[const]]; (* All exponents are 0,
        const is the result. *)
      ex=asympt[[i,2]]; (* Exponent of leading term *)
      If[ex>0,Return[0]];(* tends to 0 in all cases *)
      If[ex<0 ,
         (* limit result is +/- ∞ *)
        s=Sign[const];
        If[s===0,
          Message[MrvLimit::"ZeroFunction"];
          Return[Null];
        ];
        Return[s*∞]
      ]; (* end if ex < 0 *)
      Message[MrvLimit::"UnknownSign",ex];
]; (* end MrvLimitOutput *)
```

Diese Fassung von MrvLimitOutput (man beachte die festgeschriebene Option Output → Limit) berechnet den konkreten Grenzwert aus dem LeadTerm-Objekt.

Die folgenden zwei Varianten interpretieren die beiden anderen Ausgabevarianten bzw. lassen sie unverändert passieren.

```
MrvLimitOutput[LeadTerm[const_,asympt_List],Output→LeadTerm]:=
    LeadTerm[const,asympt];

MrvLimitOutput[LeadTerm[const_,asympt_List],Output→SignedZero]:=
    Module[{l},
      l=MrvLimitOutput[LeadTerm[const,asympt],Output→Limit];
      If[l===0,Return[Sign[const]*Zero]];
      Return[l];
];
```

Der folgende Code dient zur Leerung des MrvLimit-Caches. Auf die genaue Funktionsweise des Caches wird im Hauptalgorithmus noch eingegangen.

```
(* Mrv limit remember cache *)
MrvLimitClearCache[]:=Module[{},
      Clear[MrvLimitCache];
      MrvLimitCacheHits=0;
      MrvLimitCacheMisses=0;
];
MrvLimitClearCache[];
MrvLimitGetCacheStats[]:={CacheHits->MrvLimitCacheHits,
    CacheMisses->MrvLimitCacheMisses}
```

5.2.7. MrvLimitPreProcess

Wir hatten bereits die Funktion ToInternal kennengelernt, die Funktionsterme in interne Darstellung überführt. Diese Funktion führte aber keine Parameterprüfungen und keine Umformungen durch, da das nicht unbedingt in jedem Anwendungsfall erforderlich ist.

Für den eigentlichen Grenzwertaufruf ist dagegen eine strengere Darstellung erforderlich. Einige Funktionen müssen in spezielle Darstellungen überführt werden, bei vielen Funktionen müssen die Argumente auf Gültigkeit geprüft werden. Dazu dient die Funktion MrvLimitPreProcess.

Zunächst wird aber die Hilfsfunktion TestLimitArgQ definiert. TestLimitArgQ[f, x, test] ermittelt den Grenzwert von f für $x \to \infty$ und prüft danach die Bedingung test[#] mit dem Ergebnis der Grenzwertberechnung als Argument. Entsprechend des Ergebnisses der Testfunktion wird True oder False zurück geliefert. Falls die Grenzwertberechnung fehlschlägt, wird ebenfalls False zurück geliefert.

Liefert der Test kein klares True oder False, wird eine Warnmeldung und ebenfalls False ausgegeben. Ruft man TestLimitArgQ jedoch mit dem optionalen Parameter Undetermined → True | False auf, wird keine Meldung ausgegeben, und der Test liefert stattdessen entsprechend True oder False.

```
(* Pre-Processing:
      Transform expression into processable form and check for limit fail \
conditions. *)

TestLimitArgQ[f_,x_,test_,Undetermined→undet_]:=Module[{ff},
      (* test if limit exists and matches a test criteria *)
      ff=MrvLimitInf[f,x];
      If[ff===Null,Return[False]];
      If[test[ff],
        (*then*)
        Return[True];
        ,(*else*)
        Return[False];
        ,(*undetermined*)
        If[undet===True,Return[True]];
        If[undet===False,Return[False]];
        Message[MrvLimit::"ConditionCheck",test[ff]];
        Return[False];
        ];
      ];
TestLimitArgQ[f_,x_,test_]:=TestLimitArgQ[f,x,test,Undetermined→-1];
```

Die eigentliche Funktion MrvLimitPreProcess ist wieder als Funktion für bestimmte Muster in den Parametern definiert:

```
Clear[MrvLimitPreProcess];
MrvLimitPreProcess[f_,x_]:=f/;FreeQ[f,x];
MrvLimitPreProcess[x_,x_]:=x;
MrvLimitPreProcess[g_*h_,x_]:=
    MrvLimitPreProcess[g,x]*MrvLimitPreProcess[h,x];
MrvLimitPreProcess[g_+h_,x_]:=
    MrvLimitPreProcess[g,x]+MrvLimitPreProcess[h,x];
MrvLimitPreProcess[power[g_,c_],x_]:=
    power[MrvLimitPreProcess[g,x],MrvLimitPreProcess[c,x]]/;FreeQ[c,x];
MrvLimitPreProcess[power[E,g_],x_]:=power[E,MrvLimitPreProcess[g,x]];
MrvLimitPreProcess[power[f_,g_],x_]:=
    power[E,log[MrvLimitPreProcess[f,x]]*MrvLimitPreProcess[g,x]];
```

Allgemeine Terme der Form f^g werden in die Form $e^{\log(f)\cdot g}$ gebracht, es sei denn, f ist gleich e, oder g ist konstant.

```
MrvLimitPreProcess[log[f_],x_]:=log[MrvLimitPreProcess[f,x]];
MrvLimitPreProcess[sin[f_],x_]:=
    If[TestLimitArgQ[f,x,FreeQ[#,DirectedInfinity]&],
      sin[MrvLimitPreProcess[f,x]],
      Message[MrvLimit::"UnsupportedArgument",Sin[x],x→∞,
        Sin[f]];
      HoldForm[MrvLimitPreProcess[sin[f],x]]
    ];
```

Im Falle von sin[f] sieht man erstmals die Anwendung von TestLimitArgQ: Gilt FreeQ[MrvLimit[sin[f]], DirectedInfinity], wird die sin-Funktion durch MrvLimitPreProcess hindurch geleitet und nur das Argument weiter analysiert. Ist die Testbedingung dagegen falsch, so wird eine Fehlermeldung ausgegeben und die Grenzwertberechnung in Folge abgebrochen.

Die meisten weiteren Funktionen werden ähnlich behandelt, nur mit wechselnden Bedingungen:

```
MrvLimitPreProcess[cos[f_],x_]:=
    If[TestLimitArgQ[f,x,FreeQ[#,DirectedInfinity]&],
      cos[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",Cos[x],x→∞,
        Cos[f]];
      HoldForm[MrvLimitPreProcess[cos[f],x]]
    ];
MrvLimitPreProcess[tan[f_],x_]:=
    If[TestLimitArgQ[f,x,FreeQ[#,DirectedInfinity]&],
      tan[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",Tan[x],x→∞,
        Tan[f]];
      HoldForm[MrvLimitPreProcess[tan[f],x]]
    ];
MrvLimitPreProcess[arcsin[f_],x_]:=
    If[TestLimitArgQ[f,x,(Abs[#]≤1)&],
      arcsin[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",ArcSin[x],x→∞,
        ArcSin[f]];
      HoldForm[MrvLimitPreProcess[arcsin[f],x]]
    ];
MrvLimitPreProcess[arccos[f_],x_]:=
    If[TestLimitArgQ[f,x,(Abs[#]≤1)&],
      arccos[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",ArcCos[x],x→∞,
        ArcCos[f]];
      HoldForm[MrvLimitPreProcess[arccos[f],x]]
    ];
MrvLimitPreProcess[arctan[f_],x_]:=If[TestLimitArgQ[f,x,(True)&],
      arctan[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",ArcTan[x],x,ArcTan[f]];
      HoldForm[MrvLimitPreProcess[arctan[f],x]]
    ];
MrvLimitPreProcess[gamma[f_],x_]:=
    If[TestLimitArgQ[f,x,FreeQ[#,DirectedInfinity]&],
      gamma[MrvLimitPreProcess[f,x]]
      (*else*),
      If[TestLimitArgQ[f,x,(#===∞)&],
        power[E,loggamma[MrvLimitPreProcess[f,x]]]
        (*else*),
```

```
        Message[MrvLimit::"UnsupportedArgument",Gamma[x],
          x→MrvLimitInf[f,x],Gamma[f]];
        HoldForm[MrvLimitPreProcess[gamma[f],x]]
      ]
    ];
```

Bei der Gamma-Funktion tritt erstmals wieder eine Besonderheit auf: Endliche Argumente dürfen wieder wie üblich passieren. Im Fall von f → ∞ jedoch wird eine Transformation zu $e^{\mathrm{LogGamma}[f]}$ durchgeführt, um die essenzielle Singularität der Gamma-Funktion zu kompensieren.

```
MrvLimitPreProcess[loggamma[f_],x_]:=
    If[TestLimitArgQ[f,x,(0≤#≤∞)&],
      loggamma[MrvLimitPreProcess[f,x]]
      (*else*),
      Message[MrvLimit::"UnsupportedArgument",LogGamma[x],
        x→MrvLimitInf[f,x],LogGamma[f]];
      HoldForm[MrvLimitPreProcess[loggamma[f],x]]
    ];

MrvLimitPreProcess[polygamma[n_,f_],x_]:=Module[{},
      If[!IntegerQ[n]||n<0,
        Message[MrvLimit::"UnsupportedArgument",PolyGamma[k,x],k==n,
          PolyGamma[n,f]];
        Return[HoldForm[MrvLimitPreProcess[polygamma[n,f],x]]];
      ];
      If[TestLimitArgQ[f,x,(-∞<#≤∞)&],
        Return[polygamma[n,f]];
      ];
      Message[MrvLimit::"UnsupportedArgument",PolyGamma[n,x],
        x→MrvLimitInf[f,x],PolyGamma[n,f]];
      Return[HoldForm[MrvLimitPreProcess[polygamma[n,f],x]]];
    ];
```

Bei der PolyGamma-Funktion wird zusätzlich darauf getestet, ob der n-Parameter ganzzahlig und nicht negativ ist.

```
(* catch all remaining *)
MrvLimitPreProcess[f_,x_]:=(
      Message[MrvLimit::"UnsupportedFunction",f];
      HoldForm[MrvLimitPreProcess[f,x]]
);
```

Die abschließende ‚catch all'-Funktion fängt wie üblich alle verbleibenden ungültigen Aufrufe kontrolliert ab.

5.2.8. MrvLimitInf

Damit sind fast alle Vorbereitungen abgeschlossen, der Hauptalgorithmus kann beginnen. Der Algorithmus ist implementiert in zwei Teilen, der internen Funktion MrvLimitInf und der öffentlichen Funktion MrvLimit. Im Gegensatz zu MrvLimit berechnet MrvLimitInf nur Grenzwerte gegen Unendlich, und im Fehlerfall wird Null zurück geliefert.

```
(* MrvLimitInf: Main algorithm *)

(* Reset Context links *)
MrvLimitCurrentContext=Null;
Clear[MrvLimitRunningTasks];
MrvLimitDefaultDebugLevel=0;

Clear[MrvLimitInf];
Options[MrvLimitInf]={Output→Limit,Debug→Automatic};
MrvLimitInf[ff_,x_,Options___]:=Module[
      {Ω,dprint,f,scaleup,w,g,A,s,t,c,a,ex,asympt,i,LastContext,
        NestLevel,DebugLevel,output},

      ReadOptions[MrvLimitInf,{Output→output,Debug→DebugLevel},
        Options];

      If[!MemberQ[{Limit,LeadTerm,SignedZero},output],
        Message[MrvLimit::"UnknownOutput", output];
        output=Limit;
      ];
```

Am Anfang erfolgt einfache Optionsverarbeitung und Prüfung auf Gültigkeit.

```
      (* Some quick exits for really simple terms *)
      If[FreeQ[ff,x],
        If[output===LeadTerm,Return[LeadTerm[ff,{}]]];
        Return[ff];
      ];
```

Konstante Funktionen werden hier sehr frühzeitig abgebrochen.

```
      (* Check if previous MrvLimit context is still alive *)
      If[MrvLimitCurrentContext=!=Null,
        If[!NameQ[
                StringJoin[GetSymbolModuleName[LastContext],
                  MrvLimitCurrentContext]],
            (* Last context died unexpected. *)
            Message[MrvLimit::"LostContext"];
            MrvLimitCurrentContext=Null;
            Clear[MrvLimitRunningTasks];
        ];
      ]; (* end if *)
```

```
(* Connect to last context if present. *)
(* From here on, clean exit code is requred. *)
LastContext=MrvLimitCurrentContext;
MrvLimitCurrentContext=GetSymbolModuleContext[LastContext];
```

Über MrvLimitCurrentContext nimmt MrvLimitInf zu früheren rekursiven Instanzen Kontakt auf, um auf dessen Parameter und Schachtelungstiefe zugreifen zu können. Um die Gültigkeit des durch MrvLimitCurrentContext referenzierten Aufrufs zu prüfen, wird getestet, ob in diesem Kontext die Variable LastContext existiert. Wurde das Modul des referenzierten Aufrufs bereits beendet, zum Beispiel durch einen unkontrollierten Abbruch, ist das Symbol LastContext in diesem Kontext nicht mehr existent. Das wird mit einer Warnmeldung quittiert.

Schließlich wird der vorherige Kontextlink in der lokalen Variable LastContext gespeichert und MrvLimitCurrentContext auf den aktuellen Kontext umgesetzt. Bevor diese Instanz von MrvLimitInf beendet wird, muss auf jeden Fall der ursprüngliche Wert von MrvLimitCurrentContext wieder hergestellt werden.

```
Catch[
    (*
      Catch/Throw error handling from here.
          Throw[] uses clean exit code. *)
```

Um ein einfaches kontrolliertes Beenden von MrvLimitInf zu ermöglichen, ist die verbleibende Funktion in Catch[] eingehüllt. Innerhalb dieses Bereichs kann Throw[] ersatzweise wie Return[] verwendet werden, wobei nötige Aufräumarbeiten vor dem Beenden automatisch ausgeführt werden.

```
    (* Get MrvLimit nest level *)

    NestLevel=
      If[LastContext===Null,1,
        SetModuleContext[NestLevel,LastContext]+1];

    (* Get Debug level *)
    If[DebugLevel===Automatic,

      DebugLevel=
          If[LastContext===Null,MrvLimitDefaultDebugLevel,
            SetModuleContext[DebugLevel,LastContext]];
    ];
```

Mit diesem Code ermittelt MrvLimitInf die Schachtelungstiefe des rekursiven Aufrufs und den Debug Level des vorherigen Aufrufs, falls dieser nicht bereits durch die Option Debug → n gesetzt wurde.

```
(* Prepare dprint based on DebugLevel *)
If[DebugLevel≥NestLevel
  (*then*),
  dprint[s__]:=$DebugPrint[NestLevel-1,s];

  dprint["Enter Level ",NestLevel,
    " Call ",$OutputProcessor[
      HoldForm[MrvLimitInf[ff,x,Options]]]];
  (*else*),
  dprint[s__]=Null;
];(*end if*)
SetAttributes[dprint,HoldAll];
```

Ist der Debug Level höher oder gleich der Schachtelungstiefe, so wird unter dem Namen dprint die Funktion $DebugPrint hinterlegt und damit die Debugausgabe aktiviert. Danach wird die Einstiegsmeldung für den rekursiven Aufruf ausgegeben.

Ist der Debug Level niedriger, so wird dprint mit einer leeren Funktion initialisiert. Damit Mathematica keine unnötige Zeit mit dem Bestimmen der Argumente verliert, wird dprint schließlich noch auf HoldAll gesetzt.

```
(* Check for cached result *)
asympt=MrvLimitCache[ff];
If[Head[asympt]=!=MrvLimitCache,
  (*found in cache*)

  MrvLimitCacheHits++;
  dprint["Taking cached result"];

  Throw[MrvLimitOutput[asympt,Output→output]];

  (*else if Length[i]>0*),
  MrvLimitCacheMisses++;
]; (* end if Length[i]>0 *)

(* Check for recursive calls *)
If[MrvLimitRunningTasks[ff],
  Message[MrvLimit::"RecursiveCall",ff];
  Return[Null];
];

(* Mark this task 'in progress' for recursive call checking *)
MrvLimitRunningTasks[ff]=True;
```

Im Symbol MrvLimitCache[f] werden bereits ermittelte Funktionsergebnisse abgelegt. Dabei kommt grundsätzlich das LeadTerm-Format zum Einsatz. Ist ein solches Ergebnis bereits hinterlegt, so wird dieses Ergebnis an das Ausgabeformat angepasst und zurück geliefert.

Ist kein passendes Ergebnis im Cache, wird zusätzlich MrvLimitRunningTasks[f] geprüft. Dieses Symbol ist auf True definiert, solange die Berechnung läuft. Sollte es zu einem rekursiven Aufruf mit exakt dieser Funktion kommen, so würde das dadurch bemerkt und unterbunden. Dies ist kein perfekter Schutz gegen unendliche Rekursionen, aber besser als nichts.

```
(**********************************)
(* Ok, lets start with the real work *)
(**********************************)

f=ff;
If[$MrvTestZeroInterval[f,x→∞],
  (* Function is identical 0 around ∞.
          Dont start to calculate crap with it... *)
  Throw[0];
];
```

Hier werden Funktionen abgefangen, die in der Umgebung um Unendlich identisch 0 sind. Sie sprengen das Funktionsmodell und führen zu nicht behebbaren Ausnahmebedingungen im Algorithmus, deswegen müssen solche Funktionen frühzeitig erkannt werden.

```
asympt={};
(* Result is f*Apply[Times,asympt]/.pwr→Power ,
  while asympt is a dominance ordered list of pwr[w,ex],
  w tends to 0 for x→0.
  *)
```

Ab hier gilt, dass LeadTerm[f,asympt] der aussichtsreichste Kandidat für die Lösung der Grenzwertaufgabe ist, mit der Ausnahme, dass f bisher noch von x abhängt.

```
While[!FreeQ[f,x],
  (* While f depends on x: Do Mrv limit calculation *)
```

Damit beginnt die Hauptschleife, die jeweils die stärkste Wachstumsklasse bestimmt und auflöst. Dies wird solange fortgesetzt, bis f konstant ist.

```
  dprint["Calculating limit of ",$OutputProcessor[f]];

  (* First, do pre-processing for some functions *)
  f=MrvLimitPreProcess[f,x];
  If[!FreeQ[f,MrvLimitPreProcess],
    dprint["Pre Processing failed  :",$OutputProcessor[f]];
    Throw[Null];
  ];
```

Als Vorbereitung wird die Funktion in die für MrvLimit erforderliche Struktur transformiert, siehe auch Kapitel 4.3. Dies muss bei jedem Schleifendurchlauf erneut geschehen, da die Funktion MrvSeriesHead die Struktur eventuell verändert hat.

```
(* Now do 'simple simplifications' *)
f=f/.SimpleSimplifications;

(* Calculate Mrv set *)
dprint["Calculating Mrv set of ",$OutputProcessor[f]];
Ω=MrvSet[f,x];
If[Ω===Null,Throw[Null]];
If[Ω==={},
  Message[MrvLimit::"MrvSetFailed",$OutputProcessor[f]];
  Throw[Null];
];
dprint["Ω=",$OutputProcessor[Ω]];
```

Als nächstes folgt die Bestimmung der stärksten Wachstumsklasse inklusive elementarer Fehlerbehandlung, siehe Kapitel 4.5.

Danach wird die Funktion bei Bedarf in eine ausrechend hohe Wachstumsklasse angehoben. Die Strategie ist aus Kapitel 4.6 bekannt.

```
(* Scale up? *)
scaleup=0;
While[Ω[[1,1]]===x,
  (* x, if present in Ω,
    will always be the first one *)
  scaleup++;
  f=f/.{log[x]→x,x→power[E,x]};
  Ω=
    MrvSetRules[Ω,{log[x]→x,
        x→power[E,x]}];

  (*
    in f and Ω, all occurences of eˆx got replaced the same way.
    also, all inner occurences of log[x] got replaced the same way.
    Only problem: The x in Ω may have been present in f as log[x],
    transforming x→eˆx in Ω and log[x]→x in f. But since all members
    of Ω now have comparability class eˆx, this x term is of lower
    class. But if none of the terms in Ω is left in f (all
    occurences of x in f were log[x]), then there is no term of
    eˆx order in f, and we have to search again for the next lower class.
  *)

  Ω=Select[Ω,!FreeQ[f,#[[1]]]&];
  (* eliminate non-occuring terms *)

  If[Ω==={},Ω=MrvSet[f,x]];
  (* If all eliminated, scan for next lower class *)


  dprint["Scale Up: f=",$OutputProcessor[f],"\n",
    "Ω=",$OutputProcessor[Ω]];
];
```

Damit ist die Wachstumsklasse bis über polynomielles Wachstum angehoben, der Algorithmus kann fortgesetzt werden.

```
(* All Mrv's must be of e^_ form. Assert if not. *)
If[!Apply[And,Map[MatchQ[#[[1]],power[E,_]]&,Ω]],

  Message[MrvLimit::"Assert",
    "Ω non-exponential",Ω];
  Throw[Null];
];
```

Es folgt ein schlichter Test, ob Ω die geforderte Struktur hat und nur exponentielle Terme enthält. Es gibt keinen Grund, weshalb dieser Fehler auftreten sollte.

```
(* ω is the first one *)
w=Ω[[1,1]];
```

Der Algorithmus fordert eigentlich die Ordnung von Ω nach der Komplexität. Tatsächlich genügt aber auch die Ordnung nach LeafCount, die bereits vorliegt.

```
dprint["ω=",$OutputProcessor[w]];

(* We need ω→0. If ω→∞,
   substitute. *)
(* We know, ω is of e^f_ form *)

If[Ω[[1,3]]===∞,
  (*
    w→∞.
    Turn around sign inside e^_ to get w→0
  *)
  w[[2]]=-w[[2]];

  dprint["Replacing ω→1/ω: \
         ω=",$OutputProcessor[w]];
];
```

Falls $\omega \to \infty$ gilt, wird das Wachstumsverhalten abgeändert, wie im Kapitel 4.7 dargelegt.

Es folgt die Ersetzungsschleife, die die Terme der stärksten Wachstumsklasse in die Darstellung mit ω transferiert. Siehe Kapitel 4.8 für Details.

```
(* replace to ω in f step by step *)
Do[
  g=Ω[[i,1]];
  (* Substitute all occurences of this subexpr *)
  (* see chapter 4.8 or Lemma 3.18 [Gru96] for details *)

  dprint["Prepare to rewrite g=",$OutputProcessor[g],
    " to A*ωˆc"];

  c=FromInternal[g[[2]]/w[[2]],x];
  (* We'll take some auto-optimizations for this *)
  c=ToInternal[c,x];
  If[!FreeQ[c,ToInternal],
    c=Null;
  ];
  (* Now fine again *)
```

Für den Term c lassen wir hier kurz Mathematica seine automatischen Vereinfachungen durchführen. Der Verlust der Struktur von MrvLimitPreProcess ist kein Problem, da c als nächstes an MrvLimitInf übergeben wird und dann sowieso ein neuer Aufruf von MrvLimitPreProcess erfolgt.

```
  c=MrvLimitInf[c,x];
  If[c===Null || c===0 || !FreeQ[c,DirectedInfinity],

    Message[MrvLimit::"LimitFail","MrvLimit rewrite",
      g[[2]]/w[[2]]];
    Throw[Null]
  ];
```

Wie im Kapitel 4.12 dargelegt, kann dieser Grenzwertaufruf auch eingespart werden.

Nun lässt sich der Term A bestimmen und die Ersetzung durchführen:

```
  A=power[E,g[[2]]-c w[[2]]]//.SimpleSimplifications;

  dprint["c=",$OutputProcessor[c]," A=",$OutputProcessor[A]];

  f=f/.{g→A*power[ω,c]}; (* and replace. *)

  dprint[
    "Rewriting ",$OutputProcessor[g],
    "→",$OutputProcessor[A*power[ω,c]],
    "\n","f=",$OutputProcessor[f]
  ];

  (* do for *)
  ,{i,Length[Ω],1,-1}
]; (* end do *)

(* Done rewriting *)
```

Damit ist der Umschreibeprozess fertig, es kommt die Potenzreihenentwicklung, vgl. Kapitel 4.9:

```
dprint["Calculating power series: f=",$OutputProcessor[f]];

(* Calculate power series head *)
ser=$MrvSeriesHead[f,x,ω,w[[2]],dprint];

If[ser===Null,Throw[Null]];
{f,ex}=ser;
```

Das Ergebnis der Potenzreihenanalyse, der Vorfaktor des führenden Terms, ersetzt bereits hier die alte Funktion f. Der Rest wird durch ω und ex ausreichend repräsentiert.

Schließlich kann die Anhebung der Wachstumsklasse wieder rückgängig gemacht werden, wie in Kapitel 4.10 geschildert.

```
While[scaleup>0,
  scaleup--;
  f=f/.{power[E,x]→x,x→log[x]};
  w=w/.{power[E,x]→x,x→log[x]};

  dprint["Scale Down: f=",$OutputProcessor[f],
    " , ω=",$OutputProcessor[w]];
];
```

Die Wachstumsklasse ω ist eliminiert, es bleibt, den führenden Term der Potenzreihe weiter zu analysieren.

```
dprint["Dominant term: ",$OutputProcessor[f*power[ω,ex]],
  " , ω=",$OutputProcessor[w]];

If[$MrvTestZeroInterval[f,x→∞],
  (* This explicitly means that f has been identical zero *) (*
    for x→∞ when entering this loop.
          This is fatal, *)
  (* since this should have been detected before entering *)
  (* the loop, or while in the previous loop run. *)
  (* Either the function was identical 0 from the beginning, *)
  (* or one of the series heads returned has been identical 0 *)
  (* without detecting it. Either way, we failed miserably. *)

  Message[MrvLimit::"ZeroFunction"];
  Throw[Null];
];
```

Das wäre ein fataler Fehler: Sollte sich jetzt herausstellen, dass der führende Term 0 ist, so war die Funktion schon vor dem Aufruf von MrvSeriesHead in der Umgebung um ∞ die Nullfunktion, denn MrvSeriesHead hätte andernfalls weitere führende Terme ermitteln müssen.

Die Funktion war entweder von Anfang an die Nullfunktion in der Umgebung um ∞, oder in einem vorherigen Schleifendurchlauf wurde in der Bestimmung des führenden Terms der Potenz-

reihe ein Fehler gemacht. Dieses Problem derart zurück zu rollen, dass die Ursache gefunden werden kann, ist fast unmöglich. In jedem Fall hat ein Nulltest versagt.

```
  AppendTo[asympt,pwr[w,ex]];

  (* End of this limit round. Continue on lower Mrv level? *)
]; (* While !FreeQ[f,x] *)
```

Das ist das Ende der Schleife, in der jeweils eine Wachstumsklasse abgebaut und an asympt angehängt wird. An dieser Stelle ist die Analyse beendet und der Grenzwert damit bekannt.

```
(* Result is f*Apply[Times,asympt]/.pwr→Power ,
  while asympt is a dominance ordered list of pwr[w,ex],
  w tends to 0 for x→0
*)


dprint["Dominant asymptotic: ",$OutputProcessor[f],
  Apply[Sequence,$OutputProcessor[asympt/.pwr→power]]];


(* Add this result to cache *)
MrvLimitCache[ff]=LeadTerm[f,asympt];

Throw[MrvLimitOutput[LeadTerm[f,asympt],Output→output]];
```

Das Ergebnis wird noch im Cache gespeichert, in die korrekte Ausgabeform gebracht und per Throw abgeliefert.

```
(* end Catch *)
] // (
  (* Begin pure function *)
  (* Exit code *)
```

Dies ist das Ende des mittels Catch[] gekapselten Hauptalgorithmus, und alle Ergebnisse des Algorithmus durchlaufen diesen Code. Mit dem letzten ‚(‘ beginnt eine namenlose Funktion (siehe Mathematica-Hilfestellung unter Function), die die Ausgabe von Catch[] verarbeitet, d.h. das mit Throw[] gemeldete Ergebnis. Dieses Ergebnis ist jetzt in der Variablen # gespeichert.

Es bleibt nur noch, den Funktionsaufruf als beendet zu markieren, eine passende Debug-Meldung über das Ergebnis abzusetzen und den alten Inhalt von MrvLimitCurrentContext wieder herzustellen.

```
            (* Clear 'in progress' state *)
            If[MrvLimitRunningTasks[ff],MrvLimitRunningTasks[ff]=.];


            dprint["Leave Level ",NestLevel,
              " Call ",$OutputProcessor[
                HoldForm[MrvLimitInf[ff,x,Options]==#]]];

            MrvLimitCurrentContext=LastContext;
```

Schließlich wird das Ergebnis in # durchgereicht und als Funktionsergebnis zurückgegeben.

```
            # (* Pass through result *)
          )& (* end pure function *)
]; (* end MrvLimitInf *)

(* force cache clear on code change *)
MrvLimitClearCache[];
```

Die letzte Zeile soll nur sicherstellen, dass bei allen Änderungen am Code auch der Cache nochmals geleert wird. Damit ist der Algorithmus vollständig, abgesehen vom eigentlichen Interface-Code.

5.2.9. MrvLimit

Der Interface-Code von MrvLimit hat keine komplizierten Aufgaben mehr zu bewältigen. Er erledigt die Transformation zu einer Aufgabe für $x \to \infty$ und übersetzt die Funktion in die interne Darstellung.

```
(* MrvLimit interface code *)

Clear[MrvLimit];
Options[MrvLimit]={Output→Limit,Debug→Automatic,
      Direction→Automatic,ClearCache→False};
MrvLimit[ff_,x_→lim_,Options___]:=
    Module[{Res,ResA,ResB,f,z,direction,lim1,ChildOptions,clearcache},
      ReadOptions[
        MrvLimit,{Direction→direction,ClearCache→clearcache},
        Options];

      (* Remove Direction→ and ClearCache→ options *)
      ChildOptions=DropOptions[{Direction,ClearCache},Options];
```

Die Optionen Direction und ClearCache werden vom Interface-Code behandelt und werden daher nicht an MrvLimitInf weiter gereicht.

```
If[clearcache,
  MrvLimitClearCache[];
];

(* Delete lost contexts *)
MrvLimitCurrentContext=Null;
Clear[MrvLimitRunningTasks];
```

Rekursive Aufrufe führen niemals durch MrvLimit, immer durch MrvLimitInf. Deswegen kann hier sicher der Kontext zurück gesetzt werden, falls eine vorherige Grenzwertberechnung unerwartet beendet wurde.

```
f=ToInternal[ff,x];
If[!FreeQ[f,ToInternal],
  Return[HoldForm[MrvLimit[ff,x→lim,Options]]];
];
```

Übersetzung des Funktionsterms in interne Darstellung.

```
lim1=lim;
If[direction===Automatic,
  (* Set auto direction *)
  Which[
      Head[lim]===Global‘SuperPlus,(direction=-1;lim1=lim[[1]]),
      Head[lim]===Global‘SuperMinus,(direction=1;lim1=lim[[1]]),
      (*
        hell knows why I have to explicitly request global context.
            Wont work otherwise.*)
      lim===∞,direction=1,
      lim===-∞,direction=-1,
      True,direction=0
  ];
];
```

Falls die Option Direction nicht angegeben wurde, hat die Variable direction den Wert Automatic. In diesem Modus wird bei Grenzwerten im Unendlichen automatisch die korrekte Richtung gewählt, während bei endlichen Grenzwerten der beidseitige Modus aktiviert wird. Außerdem wird die Schreibweise mit SuperPlus und SuperMinus (hochgestelltes Plus/Minus) ausgewertet, siehe Kapitel 5.1.2.

```
If[direction≤0,
  (* limit from above *)
  If[Abs[lim1]==∞,
    ResA=MrvLimitInf[f/.x→-x,x,ChildOptions];
```

```
    ResA=ResA/.{x→-x};
    (* else *),
    ResA=MrvLimitInf[
        f/.{x→power[x,-1]+lim1}//.SimpleSimplifications
        ,x,ChildOptions];
    ResA=ResA/.{x→power[x-lim1,-1]}//.SimpleSimplifications;
  ];

  If[ ResA===Null,Return[HoldForm[MrvLimit[ff,x→lim,Options]]]];
];
```

Grenzwerte von oben: Je nachdem, ob endlich oder unendlich, wird die passende Transformation angewendet, der Grenzwert ermittelt und die Transformation (für Output → LeadTerm) wieder rückgängig gemacht.

```
If[direction≥0,
  (* limit from below *)
  If[Abs[lim1]==∞,
    ResB=MrvLimitInf[f,x,ChildOptions];
    (* else *),
    ResB=MrvLimitInf[
        f/.{x→-power[x,-1]+lim1}//.SimpleSimplifications
        ,x,ChildOptions];
    ResB=ResB/.{x→power[lim1-x,-1]}//.SimpleSimplifications;
  ];

  If[ ResB===Null,Return[HoldForm[MrvLimit[ff,x→lim,Options]]]];
];
```

Grenzwerte von unten werden analog behandelt, nur wird das Ergebnis diesmal in ResB gespeichert. Beidseitige Grenzwerte durchlaufen beide Blöcke.

```
(* Final result processing depending on direction *)

If[direction==0,
  (* two sided limit, check continuity *)
  If[ResA===ResB ,
      Res=ResA;
      , (* else *)
      Res={ResB,ResA};
  ];
];
```

Im Fall von beidseitigen Grenzwerten wird nun auf Stetigkeit geprüft. Falls die Funktion stetig am Grenzübergang ist, wird nur ein Ergebnis zurück geliefert, andernfalls werden beide Ergebnisse zurück geliefert.

```
If[direction<0,
  Res=ResA;
];
```

```
      If[direction>0,
        Res=ResB;
      ];

      If[Head[Res]=!=LeadTerm,Res=FromInternal[Res,x]];
```

Im Fall von einseitigen Grenzwerten wird natürlich nur dieses eine Ergebnis zurück geliefert.

Mit Ausnahme der Option Limit → LeadTerm wird das Ergebnis noch in die offizielle Mathematica-Schreibweise zurück übersetzt.

```
      Res
]; (* end MrvLimit *)
```

Ende der Funktion MrvLimit...

```
End[];
EndPackage[];
```

... und Ende des Pakets MrvLimit.

6. Praxistests

Nachdem wir nun genug Algorithmen und Quelltexte studiert haben, wird es Zeit, das Ganze einmal im praktischen Einsatz zu beobachten. Schauen wir, wie weit der Algorithmus tragfähig ist und wie oft die Kompromisse der Implementierung uns dennoch scheitern lassen.

Soweit nicht anders vermerkt, kam für die Tests Mathematica 5.0 auf einem handelsüblichen 2.4GHz-PC zum Einsatz.

6.1. Einfache Auslöschung

Beginnen wir mit einem Beispiel aus [Gru96], Beispiel 3.21 auf Seite 47. Die Funktion lautet:

```
In[1]:= f = Hold[(e^H e^(-x/(1 + H)) e^e^(-x + H))/(H^2) -
            e^x + x];

        f = ReleaseHold[f/.Power → power]
```

$$\text{Out[1]=}\quad x - e^x + \frac{e^H\, e^{e^{H-x}}\, e^{-\frac{x}{1+H}}}{H^2}$$

Schon um die Funktion unverändert eingeben zu können, muss man etwas tricksen, sonst schlägt die automatische Vereinfachung von Mathematica zu. Durch das Kapseln in Hold bleibt die Struktur solange erhalten, bis Mathematicas eigene Power-Funktion durch die interne Darstellung von MrvLimit ersetzt ist. Danach kann der Haltezustand aufgehoben werden. Optisch sieht die Ausgabe wie eine normale Funktion aus, doch das täuscht: Mathematica selbst kann mit dieser Funktion nicht viel anfangen. Es bleibt, noch H zu erklären:

```
In[2]:= h = e^(-x/(1 + e^ - x))
```

$$\text{Out[2]=}\quad e^{-\frac{x}{1+e^{-x}}}$$

```
In[3]:= f = f/.H → h
```

$$\text{Out[3]=}\quad x - e^x + \frac{e^{e^{-\frac{x}{1+e^{-x}}}}\, e^{e^{e^{-\frac{x}{1+e^{-x}}}-x}}\, e^{-\frac{x}{1+e^{-\frac{x}{1+e^{-x}}}}}}{\left(e^{-\frac{x}{1+e^{-x}}}\right)^2}$$

Für bessere Lesbarkeit wird der Debugausgabe beigebracht, den h-Term symbolisch darzustellen:

```
In[4]:= $OutputProcessor = ((#/.{ToInternal[h, x] → H})&);
```

Damit ist der Weg frei für die Berechnung des Grenzwertes für $x \to \infty$:

```
In[5]:= MrvLimit[f, x → ∞, Debug → 1]
```

```
Enter Level 1 Call
  MrvLimitInf[x - e^x + e^(e^(-x+H)) e^H e^(-x/(1+H)) / H^2, x, Debug → 1]

Calculating limit of x - e^x + e^H e^(e^(H-x)) e^(-x/(1+H)) / H^2

Calculating Mrv set of x - e^x + e^H e^(e^(H-x)) e^(-x/(1+H)) / H^2

Ω = {MrvF[e^x, 3, ∞], MrvF[e^(-x), 5, 0],
     MrvF[H, 14, 0], MrvF[e^(H-x), 20, 0], MrvF[e^(-x/(1+H)), 23, 0]}
```

MrvSet hat 5 verschiedene Teilausdrücke gefunden, die allesamt exponentielles Wachstum besitzen, und hat diese auch gleich nach Komplexität geordnet: e^x, e^{-x}, h, e^{h-x} und $e^{-\frac{x}{1+h}}$. e^x strebt dabei gegen ∞, die anderen streben gegen 0.

Es sei noch mal darauf hingewiesen, dass der Term H nur in der Ausgabe auftaucht, der Algorithmus selbst arbeitet dagegen immer mit dem kompletten Funktionsterm.

```
ω = e^x
Replacing ω → 1/ω : ω = e^(-x)
```

Als ω wurde der einfachste Term e^x gewählt. Da aber immer ein Term mit Grenzwert 0 gewählt werden muss, wird stattdessen e^{-x} verwendet, der diesmal zufälligerweise auch in Ω liegt.

```
Prepare to rewrite g = e^(-x/(1+H)) to A * ω^c
c = 1 A = e^(x - x/(1+H))
Rewriting e^(-x/(1+H)) → e^(x - x/(1+H)) ω^1
f = x - e^x + e^H e^(e^(H-x)) e^(x - x/(1+H)) ω^1 / H^2
```

Der Algorithmus hat sich den komplexesten Term $e^{-\frac{x}{1+h}}$ ausgesucht und schreibt ihn in einen Term $A \cdot \omega^c$ um. Der statt dessen eingesetzte Term $e^{x-\frac{x}{1+h}}$ hat eine niedrigere Wachstumsklasse, erst das außerdem auftauchende ω^1 erzeugt wieder das alte Wachstum.

Man sieht auch, dass durch die Ersetzung ein Term h hinzugekommen ist, der dann später ersetzt wird. Deswegen ist die Reihenfolge der Ersetzung so entscheidend.

Die weiteren Ersetzungen erfolgen analog:

```
Prepare to rewrite g = e^(H-x) to A * ω^c
```

c = 1 A = e^{H}

Rewriting $e^{H-x} \to e^{H}\,\omega^{1}$

$$f = x - e^{x} + \frac{e^{H}\,e^{x-\frac{x}{1+H}}\,e^{e^{H}\omega^{1}}\,\omega^{1}}{H^{2}}$$

Prepare to rewrite g = H to A * ω^ c

c = 1 A = $e^{x-\frac{x}{1+e^{-x}}}$

Rewriting $H \to e^{x-\frac{x}{1+e^{-x}}}\,\omega^{1}$

$$f = x - e^{x} + \frac{e^{e^{e^{x-\frac{x}{1+e^{-x}}}\omega^{1}}\omega^{1}}\,e^{e^{x-\frac{x}{1+e^{-x}}}\omega^{1}}\,e^{x-\frac{x}{1+e^{x-\frac{x}{1+e^{-x}}}\omega^{1}}\omega^{1}}}{\left(e^{x-\frac{x}{1+e^{-x}}}\,\omega^{1}\right)^{2}}$$

Prepare to rewrite g = e^{-x} to A * ω^ c

c = 1 A = 1

Rewriting $e^{-x} \to \omega^{1}$

$$f = x - e^{x} + \frac{e^{e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}\,e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\,e^{x-\frac{x}{1+e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}}{\left(e^{x-\frac{x}{1+\omega^{1}}}\,\omega^{1}\right)^{2}}$$

Prepare to rewrite g = e^{x} to A * ω^ c

c = − 1 A = 1

Rewriting $e^{x} \to \frac{1}{\omega}$

$$f = x - \frac{1}{\omega} + \frac{e^{e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}\,e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\,e^{x-\frac{x}{1+e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}}{\left(e^{x-\frac{x}{1+\omega^{1}}}\,\omega^{1}\right)^{2}}$$

Damit ist die Funktion vollständig transformiert. Das exponentielle Wachstum konzentriert sich in ω, alle anderen, nicht von ω abhängigen Teilausdrücke haben niedrigere Ordnung. Der Weg ist frei für die Potenzreihenentwicklung:

Calculating power series : f =

$$x - \frac{1}{\omega} + \frac{e^{e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}\,e^{e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\,e^{x-\frac{x}{1+e^{x-\frac{x}{1+\omega^{1}}}\omega^{1}}\omega^{1}}}{\left(e^{x-\frac{x}{1+\omega^{1}}}\,\omega^{1}\right)^{2}}$$

Series expansion : $2 + \left(3 + \frac{3x^{2}}{2}\right)\omega + O[\omega]^{2}$

Dominant term : $2\,\omega^{0}$, $\omega = e^{-x}$

Damit zerfällt die scheinbar so komplexe Funktion zu nichts. Der führende Term hat die Ordnung 0, das Wachstum von e^{-x} löscht sich also vollständig aus. Der Restterm niedrigerer Wachstumsklasse ist konstant 2, deswegen sind keine weiteren Berechnungen erforderlich, das Endergebnis ist 2:

Dominant asymptotic : $2\,(e^{-x})^{0}$

```
Leave Level 1 Call
   MrvLimitInf[x - e^x + e^(e^(-x+H)) e^H e^(-x/(1+H)) / H^2, x, Debug → 1] == 2
Out[5]= 2
```

6.2. Überlagerung unterschiedlichen Wachstums

Als nächstes folgt ein typisches Beispiel für das Auftreten verschiedener Wachstumsklassen in einem Term und deren stückweise Abarbeitung. Das Beispiel stammt von [Gru96], Beispiel 3.5 von Seite 60, geht aber auf [RSSH96] zurück. Die Debugausgabe ist hier nur in gekürzter Form wiedergegeben.

```
In[1]:= f = Log[Log[x e^(x e^x) + 1]] - e^e^(Log[Log[x]] + 1/x)
Out[1]= -x^(e^(1/x)) + Log[Log[1 + e^(e^x x) x]]

In[2]:= MrvLimit[f, x → ∞, Debug → 1]
Enter Level 1 Call MrvLimitInf[log[log[1 + x e^(x e^x)]] - e^(log[x] e^(1/x)), x, Debug → 1]
Ω = {MrvF[e^(x e^x), 7, ∞]}
```

Der Term mit höchster Wachstumsklasse ist gefunden, und er tritt sogar nur einmal in der Funktion auf. Wir überspringen hier das Umschreiben der Funktion und verfolgen als nächstes die Potenzreihenentwicklung:

```
Calculating power series : f = log[log[1 + x/ω]] - e^(log[x] e^(1/x))
Dominant term : (log[log[x] + x e^x] - e^(log[x] e^(1/x))) ω^0 , ω = e^(-x e^x)
```

Trotz dass der ω-Term genau einmal in der Funktion auftritt, sein Beitrag zum Grenzwert löscht sich aus und ist von der Ordnung 0. Der Koeffizient hängt aber weiterhin von x ab, es kommt also zu einer weiteren Grenzwertberechnung.

```
Calculating limit of log[log[x] + x e^x] - e^(log[x] e^(1/x))
Ω = {MrvF[e^x, 3, ∞]}
```

Das nächst schwächere Wachstumsverhalten ist e^x und taucht auch nur einmal in der Funktion auf. Wir springen wieder direkt zur Potenzreihe:

```
Calculating power series : f = log[log[x] + x/ω] - e^(log[x] e^(1/x))
Dominant term : (x + log[x] - e^(log[x] e^(1/x))) ω^0 , ω = e^(-x)
```

Auch diese Wachstumsklasse ist von der Ordnung 0, eine dritte Grenzwertberechnung wird nötig.

```
Calculating limit of x + log[x] - e^(log[x] e^(1/x))
Ω = {MrvF[x, 1, ∞], MrvF[e^(log[x] e^(1/x)), 10, ∞]}
Scale Up : f = x + e^x - e^(x e^(e^(-x)))
 Ω = {MrvF[e^x, 3, ∞], MrvF[e^(x e^(e^(-x))), 11, ∞]}
ω = e^x
Replacing ω → 1/ω : ω = e^-x
```

Diesmal bleibt als Wachstumsklasse nur x übrig, sowie ein weiterer Term mit polynomiellem Wachstumsverhalten: $e^{\log(x)e^{1/x}}$. Obwohl auf den ersten Blick eine Exponentialfunktion, wächst der Term doch insgesamt asymptotisch nur polynomiell.

Da der Algorithmus auf exponentiellem Wachstum aufbaut, ist jetzt erstmals eine Anhebung der Wachstumsklasse erforderlich gewesen. Die Ersetzung war erfolgreich, beide Terme treten weiterhin in der Funktion auf und sind nun exponentiell.

```
Calculating power series : f = x + 1/ω - (e^(-x) + x e^(e^-x))/ω
Series expansion : (-1/2 - x^2/2) ω + O[ω]^2
Scale Down : f = -log[x]/2 - log[x]^2/2 , ω = e^-log[x]
Dominant term : ω^1 (-log[x]/2 - log[x]^2/2) , ω = e^-log[x]
```

Diesmal tritt die Wachstumsklasse real auf und wird nicht durch Auslöschungseffekte eliminiert. Nachdem die Anhebung der Wachstumsklasse rückgängig gemacht wurde, bleibt der etwas seltsame Term $\omega = e^{-\log x}$ übrig, was natürlich nichts anderes als $1/x$ ist. Die Funktion hat also das Wachstumsverhalten von x^{-1}.

Eigentlich ist damit bereits alles klar, die Funktion muss den Grenzwert 0 haben, auch der noch ungeklärte Koeffizient $-\frac{\log x}{2} - \frac{\log(x)^2}{2}$ kann daran nichts mehr ändern. Der Algorithmus analysiert die Funktion aber trotzdem vollständig bis zum Ende weiter.

```
Calculating limit of -log[x]/2 - log[x]^2/2
Ω = {MrvF[x, 1, ∞]}
Scale Up : f = -x/2 - x^2/2
 Ω = {MrvF[x, 1, ∞]}
Scale Up : f = -e^x/2 - (e^x)^2/2
 Ω = {MrvF[e^x, 3, ∞]}
ω = e^x
```

```
Replacing ω → 1/ω : ω = e^-x
```

Damit war zu rechnen: Die Wachstumsklasse x war bereits eliminiert, also sind diesmal mindestens zwei Anhebungen der Wachstumsklasse erforderlich. Die erste Anhebung führt dabei gleich wieder zu dem Term, der in der Vorrunde aus der Potenzreihenentwicklung hervorging.

```
Calculating power series : f = -1/(2 ω) - 1/2 (1/ω)^2
Series expansion : -1/(2 ω^2) - 1/(2 ω) + O[ω]^2
Scale Down : f = -1/2 , ω = e^-log[x]
Scale Down : f = -1/2 , ω = e^-log[log[x]]
Dominant term : -1/(2 ω^2) , ω = e^-log[log[x]]
```

Damit sind alle Wachstumsklassen abgearbeitet. Übrig bleibt $-\frac{1}{2}(e^{-\log\log x})^{-2}$, oder einfacher $-\frac{1}{2}\log(x)^2$. Es folgt nur noch die Schlussbetrachtung:

```
Dominant asymptotic : -1/2 (e^-x e^x)^0 (e^-x)^0 (e^-log[x])^1 1/(e^-log[log[x]])^2
Leave Level 1 Call MrvLimitInf[log[log[1 + x e^x e^x]] - e^log[x] e^(1/x), x, Debug → 1] == 0
Out[2]= 0
```

Die ersten beiden Funktionen, e^{-xe^x} und e^{-x}, haben starkes Wachstum, gehen in die asymptotische Grenzwertentwicklung jedoch nicht ein, deswegen lautet die Asymptote nur $-\frac{1}{2}x^{-1}\log(x)^2$. Der Grenzwert ist, wie schon vorher klar war, 0. Als zusätzliche Erkenntnis bleibt noch zu erwähnen, dass der Grenzwert sich von unten an die 0 annähert, da das Vorzeichen des konstanten Faktors negativ ist.

6.3. Große Erfolge

In diesem Kapitel untersuchen wir die Kompatibilität und Performance des Algorithmus in unterschiedlichen Mathematica-Versionen. Als Vergleich darf sich auch Mathematicas eingebaute Limit-Funktion abmühen. [Gru96] gibt dazu in Kapitel 8 eine stattliche Liste von schwierig zu lösenden Grenzwertaufgaben vor:

8.1 $\lim_{x\to\infty} e^x \left(e^{1/x-e^{-x}} - e^{1/x}\right) = -1$

8.2 $\lim_{x\to\infty} e^x \left(e^{e^{-x}+e^{-x^2}+1/x} - e^{1/x-e^{-e^x}}\right) = 1$

8.3 $\lim_{x\to\infty} e^{\frac{e^x-e^{-x}}{1-1/x}} - e^{e^x} = \infty$

8.4 $\lim_{x\to\infty} \exp\left(\exp\left(\frac{e^x}{1-1/x}\right)\right) - \exp\left(\exp\left(\frac{e^x}{-\ln(x)^{-\ln(x)}-1/x+1}\right)\right) = -\infty$

8.5 $\lim_{x\to\infty} \frac{\exp\left(\exp\left(e^{x+e^{-x}}\right)\right)}{\exp(\exp(e^x))} = \infty$

8.6 $\lim_{x\to\infty} \frac{\exp(\exp(e^x))}{\exp\left(\exp\left(\exp\left(x-e^{-e^x}\right)\right)\right)} = \infty$

8.7 $\lim_{x\to\infty} \frac{\exp(\exp(e^x))}{\exp\left(\exp\left(\exp\left(x-e^{-e^x}\right)\right)\right)} = 1$

8.8 $\lim_{x\to\infty} \frac{e^{e^x}}{\exp\left(\exp\left(x-e^{-e^x}\right)\right)} = 1$

8.9 $\lim_{x\to\infty} \frac{\ln(x)^2\, e^{\sqrt{\ln(x)}\,\ln(\ln(x))^2\, e^{\sqrt{\ln(\ln(x))}\,\ln(\ln(\ln(x)))^3}}}{\sqrt{x}} = 0$

8.10 $\lim_{x\to\infty} \frac{x\,\ln(x)\ln\left(x\,e^x-x^2\right)^2}{\ln\left(\ln\left(x^2+2\,e^{e^3\,x^3\,\ln(x)}\right)\right)} = \frac{1}{3}$

8.11 $\lim_{x\to\infty} \left(e^{(x\,e^{-x})/\left(e^{-x}+e^{-\frac{2\,x^2}{x+1}}\right)} - e^x\right)/x = -e^2$

8.12 $\lim_{x\to\infty} (3^x + 5^x)^{1/x} = 5$

8.13 $\lim_{x\to\infty} x/\ln\left(x^{\ln(2)/\ln(x)}\right) = \infty$

8.14 $\lim_{x\to\infty} \frac{\exp\left(\exp\left(2\,\ln\left(x^5+x\right)\,\ln(\ln(x))\right)\right)}{\exp(\exp(10\,\ln(x)\,\ln(\ln(x))))} = \infty$

8.15 $\lim_{x\to\infty} \frac{4\exp\left(\exp\left(\frac{5}{2}\,x^{-5/7}+\frac{21}{8}\,x^{6/11}+\frac{54}{17}\,x^{49/45}+2x^{-8}\right)\right)^8}{9\ln\left(\ln\left(-\ln\left(\frac{4}{3}\,x^{-5/14}\right)\right)\right)^{(7/6)}} = \infty$

8.16 $\lim_{x\to\infty} \frac{\exp\left(\frac{4\,x\,e^{-x}}{(e^x)^{-1}+\left(\exp\left((2\,x^2)/(x+1)\right)\right)^{-1}}\right)-e^x}{(e^x)^4} = 1$

8.17 $\lim_{x\to\infty} \frac{\exp\left(\frac{x\,e^{-x}}{e^{-x}+\exp\left(-\frac{2}{x+1}\,x^2\right)}\right)}{e^x} = 1$

8.18 $\lim_{x\to\infty} x - e^x + \frac{e^{e^{-\frac{x}{1+e^{-x}}}}\; e^{-\frac{x}{1+e^{-\frac{x}{1+e^{-x}}}}}\; e^{e^{e^{-\frac{x}{1+e^{-x}}}}-x}}{\left(e^{-\frac{x}{1+e^{-x}}}\right)^2} = 2$

8.19 $\lim_{x\to\infty} \frac{(\ln(\ln(x))+\ln(\ln(x)))-\ln(\ln(x)))\;\ln(x)}{\ln(\ln(x)+\ln(\ln(\ln(x))))} = 1$

8.20 $\lim_{x\to\infty} e^{\frac{\ln\left(\ln\left(x+e^{\ln(x)\,\ln(\ln(x))}\right)\right)}{\ln(\ln(\ln(x+e^x+\ln(x))))}} = e$

8.21 $\lim_{x\to\infty} e^x\,\left(\sin\left(e^{-x}+1/x\right)-\sin\left(e^{-x^2}+1/x\right)\right) = 1$

8.22 $\lim_{x\to\infty} e^{e^x}\,\left(e^{\sin\left(e^{-e^x}+1/x\right)}-e^{\sin(1/x)}\right) = 1$

8.26 $\lim_{x\to\infty} e^x\,(\Gamma(x+e^{-x})-\Gamma(x)) = \infty$

8.27 $\lim_{x\to\infty} e^{\Gamma(x-e^{-x})\,e^{1/x}} - e^{\Gamma(x)} = \infty$

8.28 $\lim_{x\to\infty} \frac{\Gamma\left(x+\frac{1}{\Gamma(x)}\right)-\Gamma(x)}{\ln(x)} = 1$

8.29 $\lim_{x\to\infty} x\,\left(-\Gamma(x)+\Gamma\left(x-\frac{1}{\Gamma(x)}\right)+\ln(x)\right) = \frac{1}{2}$

8.30 $\lim_{x\to\infty} \left(\frac{\Gamma\left(x+\frac{1}{\Gamma(x)}\right)-\Gamma(x)}{\ln(x)}-\cos(1/x)\right)\,x\,\ln(x) = -\frac{1}{2}$

8.31 $\lim_{x\to\infty} \frac{\Gamma(x+1)}{\sqrt{2\,\pi}} - e^{-x}\,\left(x^{x+\frac{1}{2}}+\frac{1}{12}\,x^{x-\frac{1}{2}}\right) = \infty$

8.32 $\lim_{x\to\infty} \frac{\ln(\Gamma(\Gamma(x)))}{e^x} = \infty$

8.33 $\lim_{x\to\infty} \frac{e^{e^{\psi(\psi(x))}}}{x} = \frac{1}{\sqrt{e}}$

8.34 $\lim_{x\to\infty} \frac{e^{e^{\psi(\ln(x))}}}{x} = \frac{1}{\sqrt{e}}$

8.35 $\lim_{x\to\infty} \frac{e^{e^{e^{\psi(\psi(\psi(x)))}}}}{x} = 0$

Jeweils unter Mathematica 3.0, Mathematica 4.2 und Mathematica 5.0 wurde jede dieser Grenzwertaufgaben einmal mit Mathematicas eigener Limit-Funktion und einmal mit MrvLimit berechnet. Um die Rechenzeit in Grenzen zu halten, wurde die Berechnung nach 60 Sekunden abgebrochen. Um Chancengleichheit zu gewähren, wurden außerdem vor jeder Berechnung eventuell gespeicherte Ergebnisse vorheriger Durchläufe gelöscht. Der Aufruf hat damit für Mathematicas Limit-Funktion folgende Form:

```
Developer‘ClearCache[];
Timing[TimeConstrained[Limit[f, x→∞], 60, Timeout]];
```

Der Aufruf von MrvLimit hatte die folgende Form:

```
Developer‘ClearCache[];
MrvLimitClearCache[];
Timing[TimeConstrained[MrvLimit[f, x→∞], 60, Timeout]];
```

Hier die Ergebnisse der Testläufe:

Aufgabe	MrvLimit			Limit		
	Math. 5.0	Math. 4.2	Math. 3.0	Math. 5.0	Math. 4.2	Math. 3.0
8.1	Ok 0.1s	Ok 0.1s	Ok 0.3s	Ok 0.1s	Ok 0.8s	– 1.9s
8.2	Ok 1.1s	Ok 0.5s	Ok 1.0s	– 5.8s	Ok 41.1s	– 0.0s
8.3	Ok 1.2s	Ok 1.0s	Ok 1.6s	Ok 0.8s	Ok 0.7s	– 0.0s
8.4	Ok 3.6s	Ok 2.5s	Ok 5.3s	Ok 7.3s	Ok 0.6s	– 0.3s
8.5	Ok 1.4s	– 1.0s	– 1.8s	Ok 0.1s	Ok 0.0s	– 0.0s
8.6	Ok 0.8s	Ok 0.7s	Ok 1.2s	– 0.8s	– 0.1s	– 0.1s
8.7	Ok 0.9s	Ok 0.9s	Ok 1.6s	– 0.7s	– 0.4s	– 0.1s
8.8	Ok 0.6s	Ok 0.6s	Ok 1.0s	– 0.5s	Ok 0.1s	– 0.0s
8.9	Ok 0.4s	Ok 0.4s	Ok 0.8s	F 0.8s	Ok 4.4s	– 0.1s
8.10	Ok 0.4s	Ok 0.3s	Ok 0.7s	– 17.7s	– 50.3s	– 3.4s
8.11	Ok 0.6s	Ok 0.5s	Ok 0.8s	– 5.0s	– 1.8s	– 2.3s
8.12	Ok 0.0s	– 0.1s	– 0.0s	Ok 0.1s	– 0.8s	– 1.9s
8.13	Ok 0.0s	Ok 0.0s	Ok 0.0s	Ok 0.0s	Ok 0.0s	Ok 0.0s
8.14	Ok 0.6s	Ok 0.4s	Ok 0.9s	– 1.1s	– 0.4s	– 0.0s
8.15	Time out	Time out	Time out	F 42.3s	F 6.7s	– 0.4s
8.16	Ok 0.8s	Ok 0.9s	Ok 1.0s	Ok 1.3s	Ok 1.6s	– 2.3s
8.17	Ok 0.3s	Ok 0.2s	Ok 0.4s	Ok 0.3s	Ok 0.3s	– 2.2s
8.18	Ok 7.1s	Ok 2.3s	Ok 2.1s	– 25.0s	– 0.5s	– 0.3s
8.19	Ok 0.0s	Ok 0.0s	Ok 0.1s	Ok 0.5s	– 1.1s	– 0.1s
8.20	– 0.0s	– 0.3s	– 0.6s	– 8.8s	– 6.4s	– 11.9s
8.21	Ok 0.3s	Ok 0.3s	Ok 0.6s	– 6.2s	Ok 2.0s	– 1.9s
8.22	Ok 0.4s	Ok 0.3s	Ok 0.7s	F 1.0s	– 0.6s	– 2.4s
8.26	Ok 0.9s	– 0.1s	– 0.3s	Time out	– 0.1s	– 0.1s
8.27	Ok 2.5s	– 0.1s	– 0.4s	Time out	– 0.0s	– 0.1s
8.28	Ok 0.5s	– 0.1s	– 0.3s	– 47.5s	– 0.0s	– 0.1s
8.29	Ok 0.6s	– 0.1s	– 0.4s	– 44.4s	– 0.1s	– 0.2s

Aufgabe	MrvLimit			Limit		
	Math. 5.0	Math. 4.2	Math. 3.0	Math. 5.0	Math. 4.2	Math. 3.0
8.30	Ok 0.7s	– 0.1s	– 0.4s	Time out	– 0.1s	– 0.4s
8.31	Ok 0.4s	– 0.2s	– 0.4s	– 1.6s	– 1.2s	– 0.0s
8.32	Ok 0.2s	– 0.1s	– 0.3s	– 12.5s	– 0.0s	– 0.1s
8.33	Ok 1.3s	– 0.0s	– 0.1s	F 1.4s	– 0.0s	– 0.0s
8.34	Ok 0.3s	– 0.0s	– 0.1s	F 2.8s	F 0.0s	– 0.3s
8.35	Ok 10.4s	– 0.1s	– 0.3s	F 27.7s	– 0.0s	– 0.0s

Ok:	Grenzwert wurde korrekt berechnet
–:	Kein Ergebnis gefunden
F:	Falsches Ergebnis gefunden
Time out:	Maximal zulässige Zeit (60s) überschritten

Am katastrophalsten schneidet Mathematica 3.0 ab. Dass wenigstens Aufgabe 8.13 erfolgreich gelöst wird, liegt an der Tatsache, dass die Funktion bei Eingabe sofort zu $x/\ln(x^{\ln 2})$ vereinfacht wird.

Mathematica 4.2 kommt mit einigen Aufgaben schon erheblich besser zurecht, scheitert aber noch an zu vielen Aufgaben, insbesondere an allen Aufgaben, die die Gamma-Funktion betreffen. In diesem Bereich gibt sich dann Mathematica 5.0 sehr viel Mühe, leider auch mit wenig Erfolg. Im Bereich der normalen Funktionen ist dagegen nur eine Verschiebung, aber kaum eine Verbesserung zu bemerken.

Schwer wiegt auch, dass sich Mathematica mit seiner Limit-Funktion falsche Ergebnisse erlaubt. Eine Aufgabe als nicht lösbar abzuweisen ist wenigstens ehrlich, ein falsches Ergebnis wird dagegen nur allzu oft als wahr hingenommen.

MrvLimit schneidet dagegen erwartungsgemäß gut ab. Unter Mathematica 5.0 trüben nur zwei Fehlschläge das Bild. Eine genauere Fehleranalyse findet im folgenden Kapitel statt. Unter Mathematica 4.2 und 3.0 ergibt sich ein identisches Bild: Zwei weitere Grenzwerte können nicht ermittelt werden, außerdem fallen sämtliche Aufgaben, die die Gamma-Funktion betreffen, aus. Schuld daran ist, dass das Series-Kommando erst seit Version 5.0 brauchbar mit der Gamma-Funktion umgehen kann.

Bleibt noch ein abschließender Blick auf die Performance: Die Rechenzeit von MrvLimit schneidet fast immer gut ab, lange Denkzeiten sind selten. Fast immer ist MrvLimit auch schneller als Limit.

Ungewöhnlich ist auf den ersten Blick jedoch, dass MrvLimit unter Mathematica 4.2 schneller zu sein scheint, als unter Mathematica 5.0. Der Grund dafür ist der geringere Arbeitseifer des Simplify-Kommandos, das für die Nulltests des Algorithmus verantwortlich ist. Die Performance des gesamten Algorithmus hängt tatsächlich hauptsächlich von der Performance des Series-Kommandos und des Simplify-Kommandos ab.

6.4. Fehlschläge

Nicht alles lief einwandfrei, deswegen werfen wir einen detaillierten Blick auf die 4 Fälle, in denen MrvLimit versagt hat.

6.4.1. Aufgabe 8.15

In Fall von Aufgabe 8.15 legt MrvLimit auf allen Plattformen eine Denkpause epischer Länge ein und kommt auch nach dreistündiger Rechenzeit noch zu keinem Ergebnis. Verfolgt man den Ablauf, so kommt es in der dritten Rekursionsstufe zu folgendem Aufruf:

```
Calculating power series : f =
```

$$\frac{\frac{2}{\left(\frac{1}{\omega}\right)^8}+\frac{5}{2\left(\frac{1}{\omega}\right)^{5/7}}+\frac{21}{8}\left(\frac{1}{\omega}\right)^{6/11}+\frac{54}{17}\left(\frac{1}{\omega}\right)^{49/45}}{-\frac{2}{\left(\frac{1}{\omega}\right)^8}-\frac{5}{2\left(\frac{1}{\omega}\right)^{5/7}}-\frac{21}{8}\left(\frac{1}{\omega}\right)^{6/11}-\frac{54}{17}\left(\frac{1}{\omega}\right)^{49/45}}$$

Offensichtlich unterscheiden sich Zähler und Nenner nur durch das entgegengesetzte Vorzeichen. Mathematica bemerkt das aber nicht und wird davon derart verwirrt, dass die Rechenzeit explodiert. Hilft man Mathematica über diese Hürde hinweg, ist die Aufgabe korrekt lösbar:

```
In[1]:= Unprotect[Series];

        Series[(2/(1/ω_)^8 + 5/(2 (1/ω_)^(5/7)) + 21/8 (1/ω_)^(6/11) + 54/17 (1/ω_)^(49/45))/
          (-2/(1/ω_)^8 - 5/(2 (1/ω_)^(5/7)) - 21/8 (1/ω_)^(6/11) - 54/17 (1/ω_)^(49/45)),
          {ω_, 0, _}] = -1;

        Protect[Series];

In[2]:= MrvLimit[StandardExamples[[15, 2]], x → ∞]
Out[2]= ∞
```

6.4.2. Aufgabe 8.20

Bei Aufgabe 8.20 scheitert MrvLimit auch auf allen Plattformen, diesmal mit einer konkreten Fehlermeldung:

```
In[1]:= MrvLimit[StandardExamples[[20, 2]], x → ∞, Debug → 1]
```

(Gekürzt...)

```
Calculating power series : f = e^(log(⌈x/ω⌉)/x)
Series expansion : (x/ω)^(1/x)
MrvLimit::SeriesFail : Series failed at e^(log(⌈x/ω⌉)/x)
```

Schauen wir doch mal, was Mathematica aus dieser Funktion macht, bevor sie an Series weitergegeben wird:

```
In[2]:= e^(Log(⌈x/ω⌉)/x)
Out[2]= (x/ω)^(1/x)
```

Da hat die automatische Vereinfachung von Mathematica mal wieder ganze Arbeit geleistet. Wir können Mathematica aber zwingen, die Potenzreihenentwicklung zuerst auf den Exponenten anzuwenden, um so die Vereinfachung zu unterdrücken.

```
In[3]:= Unprotect[Series];

Series[e^(Log(⌈x/ω_⌉)/x), {ω_, 0, k_}] := e^Series[Log(⌈x/ω⌉)/x, {ω, 0, k}];

Protect[Series];

In[4]:= MrvLimit[StandardExamples[[20, 2]], x → ∞]
Out[4]= e
```

Na, warum denn nicht gleich so? Dieses Ergebnis ist jedenfalls korrekt.

6.4.3. Aufgabe 8.5

Aufgabe 8.5 wird von Mathematica 5.0 fehlerfrei gelöst, von Mathematica 4.2 und 3.0 jedoch nicht. Woran liegt es?

```
In[1]:= MrvLimit[StandardExamples[[5, 2]], x → ∞, Debug → 1]
Calculating power series : f = e^(-e^(e^x) - (e^(e^x - e^(e^x+e^-x)))^(-1+e)/(1-e) + e^(e^x+e^-x)) ω^((-1+e)/(1-e))
Series expansion : Series[e^(-e^(e^x) + e^(e^(e^-x)+x) - (e^(e^x - e^(e^-x)+x))^(-1+e)/(1-e)) ω^((-1+e)/(1-e)), {ω, 0, 1}]
```

```
MrvLimit :: SeriesFail : Series failed at e^(-e^(e^x) - ((-1+e) (e^<<1>> - e^<<1>>))/(1-e) + e^(e^(x+e^-x))) ω^((-1+e)/(1-e))
```

Offensichtlich bereitet der Term $\omega^{\frac{-1+e}{1-e}}$ Mathematica 4.2 erheblich Bauchschmerzen. Auf den ersten Blick scheint das ein irrationaler Exponent zu sein, und irrationale Exponenten verkraftet das Series-Kommando nicht. Würde Mathematica etwas genauer hinschauen, wäre vielleicht aufgefallen, dass der Exponent schlicht -1 ist...

Doch warum scheitert Mathematica 5.0 nicht an dieser Stelle? Die gleiche Situation tritt jedenfalls auch auf:

```
Calculating power series : f = e^(-e^(e^x) - ((-1+e) (e^(e^x) - e^(e^(x+e^-x))))/(1-e) + e^(e^(x+e^-x))) ω^((-1+e)/(1-e))

Series expansion : e^(-e^(e^x) + e^(e^(-x)+x) - ((-1+e) (e^(e^x) - e^(e^(-x)+x)))/(1-e)) ω^((-1+e)/(1-e))
```

Auch Mathematica 5.0 erkennt den Exponenten nicht als -1, erkennt aber sehr wohl, dass es ein polynomieller Term mit Exponent ≤ 1 ist und reicht ihn direkt unmodifiziert durch Series hindurch, auf das der Aufrufer sehe, was er damit anfange...

Es bleibt danach für das Series-Kommando nur noch die Nullfunktion übrig, wodurch die eigentliche Potenzreihe gleich vollständig verschwindet. Daher taucht auch kein O[ω] auf.

6.4.4. Aufgabe 8.12

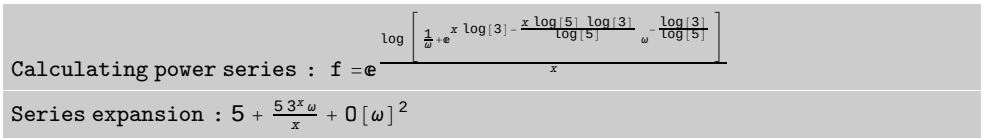
Auch mit Aufgabe 8.12 hat Mathematica 5.0 mehr Erfolg, als seine Vorgänger. Werfen wir wieder einen Blick auf das Problem:

```
In[1]:= MrvLimit[StandardExamples[[12, 2]], x → ∞, Debug → 1]

Calculating power series : f = e^(log[1/ω + e^(x log[3] - (x log[5] log[3])/log[5])] ω^(-log[3]/log[5]))/x

Series expansion : Series[(3^x + 1/ω)^(1/x), {ω, 0, 1}]

MrvLimit :: SeriesFail : Series failed at e^(log[1/ω + e^<<1>> ω^<<1>>])/x
```

Hier lohnt wieder ein Blick auf die Funktion, wie Mathematica sie vereinfacht, bevor das Series-Kommando seine Arbeit beginnt:

```
In[2]:= e^(Log[1/ω + e^(x Log[3] - (x Log[5] Log[3])/Log[5])] ω^(-Log[3]/Log[5]))/x

Out[2]= (1/ω + ω^(-Log[3]/Log[5]))^(1/x)
```

Die Ähnlichkeit zum Problem von 8.20 ist kaum zu übersehen. Vermutlich würde der damalige Trick auch hier funktionieren. Diesmal kommt Mathematica 5.0 aber an gleicher Stelle besser mit der Funktion zurecht:



6.4.5. Schlussfolgerung

Alle vier Beispiele zeigen deutlich, dem Algorithmus ist kein Vorwurf zu machen. Der MrvLimit-Algorithmus scheitert nur an den Nahtstellen zu Mathematica. Natürlich sind die Beispiele von vornherein so ausgesucht, dass sie vom Algorithmus auch bewältigt werden können. Doch der Vergleich zur Limit-Funktion zeigt deutlich die Überlegenheit, wenn es um Funktionen dieser Kategorie geht.

Zwei Fehlerquellen treten derzeit noch deutlich hervor: Zum einen ist Mathematicas Angewohnheit, jeden Term ungefragt umzuschreiben, hier ein ernstzunehmendes Problem. Kann das Umschreiben Algorithmus-intern noch umgangen werden, so ist spätestens bei der Übergabe an Series die mühsam entwickelte Funktionsstruktur oft wieder hinfällig. Mathematica wäre gut beraten, zumindest die Möglichkeit vorzusehen, bestimmte Vereinfachungen vorübergehend zu deaktivieren.

Die zweite Fehlerquelle ist das Series-Kommando. Seine Beschränkung auf Puiseux-Reihen, deren Exponenten rational mit gemeinsamen Hauptnenner sein müssen, erfüllt im Extremfall schon nicht die Anforderungen des MrvLimit-Algorithmus. Eine Implementierung eines Series-Kommandos für generalisierte Potenzreihen wäre sicher nicht nachteilig, insbesondere wenn Funktionen ebenfalls in Algorithmus-interner Darstellung übergeben werden könnten, anstatt sie zuerst in allgemeine Mathematica-Terme zurück zu übersetzen.

6.5. Grenzen

In diesem Kapitel geht es um die Grenzen des Algorithmus und wie man sie mit ein paar Handgriffen vielleicht umgehen kann, um doch noch zu einer Lösung zu kommen. Eine solche Grenze ist die Verwendung von symbolischen Konstanten.

Schon eine einfache Aufgabe wie $\lim_{x\to\infty} x^c$ hat keine allgemeine Lösung, sondern hängt direkt vom Vorzeichen von c ab. Genauso problematisch ist $\lim_{x\to\infty}(-1)^c \cdot x$, bei dem die Lösung sogar vom konkreten Wert von c abhängt. Kann man solche Aufgaben überhaupt mit Computeralgebra lösen?

Im Laufe der Tests trat folgende Aufgabe auf:

```
In[1]:= Δ = √(b² - 4 a c);
```

```
In[2]:= f = Δ^n * Pochhammer[d/(2 * a) - (2 * a * e - b * d)/(2 * a * Δ), n]

        /((-a)^n * Pochhammer[n - 1 + d/a, n])
```

$$\text{Out[2]}= \frac{(-a)^{-n}\,(b^2-4\,a\,c)^{n/2}\,\text{Pochhammer}\left[\frac{d}{2\,a}-\frac{-b\,d+2\,a\,e}{2\,a\sqrt{b^2-4\,a\,c}},n\right]}{\text{Pochhammer}\left[-1+\frac{d}{a}+n,n\right]}$$

```
In[3]:= f = f/.Pochhammer[x_, n_] → Gamma[x + n]/Gamma[x]
```

$$\text{Out[3]}= \frac{(-a)^{-n}\,(b^2-4\,a\,c)^{n/2}\,\text{Gamma}\left[-1+\frac{d}{a}+n\right]\,\text{Gamma}\left[\frac{d}{2\,a}-\frac{-b\,d+2\,a\,e}{2\,a\sqrt{b^2-4\,a\,c}}+n\right]}{\text{Gamma}\left[\frac{d}{2\,a}-\frac{-b\,d+2\,a\,e}{2\,a\sqrt{b^2-4\,a\,c}}\right]\,\text{Gamma}\left[-1+\frac{d}{a}+2\,n\right]}$$

Gesucht ist der Grenzwert für $a \to 0$. Wir beschränken uns hier auf $a > 0$, der Grenzwert bei Annäherung von unten verläuft ähnlich. Außerdem beschränken wir uns auf den interessanten Fall $d > 0$, damit die Gamma-Funktion $\Gamma(-1 + d/a + n)$ gegen $+\infty$ läuft.

Um Mathematica mitzuteilen, dass $d > 0$ sein soll, genügt es, diese Bedingung direkt in das Sign-Kommando zu integrieren:

```
In[4]:= Unprotect[Sign]; Sign[d] = 1; Protect[Sign];
```

```
In[5]:= MrvLimit[f, a → 0, Direction → -1]
```

```
MrvLimit :: UnsupportedArgument : MrvLimit does not support Gamma[a] for
```

$$a \to \infty\ \text{Sign}\left[\frac{d}{2}+\frac{\sqrt{b^2}\,d}{2\,b}\right],\ \text{appearing in Gamma}\left[\frac{a\,d}{2}+n-\frac{a\left(-b\,d+\frac{2\,e}{a}\right)}{2\sqrt{b^2-\frac{4\,c}{a}}}\right]$$

Nun, das funktioniert noch nicht so, wie geplant. Offensichtlich ist das Vorzeichen von $\frac{d}{2}+\frac{\sqrt{b^2}d}{2b}$ von entscheidender Rolle. Da aber

$$\text{Sign}\left[\frac{d}{2}+\frac{\sqrt{b^2}d}{2b}\right]=\text{Sign}\left[\frac{d}{2}\right]\cdot\text{Sign}\left[1+\frac{\sqrt{b^2}}{b}\right]$$

gilt, ist das kritische Vorzeichen das von b. Ist $b > 0$, so ist der letzte Term 2, für $b < 0$ dagegen 0.

Beschäftigen wir uns zunächst mit $b > 0$. Wieder weisen wir das Vorzeichen von b direkt Sign zu. Außerdem werden wir den Term $\sqrt{b^2}$ direkt zu b vereinfachen, um auch diese Falle gleich zu entschärfen:

```
In[6]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = 1; Protect[Sign];

        Unprotect[Power]; √b^2 = b; Protect[Power];
```

```
In[7]:= MrvLimit[f, a → 0, Direction → -1, Debug → 1]
Ω = {MrvF[a, 1, ∞]}
```

$\omega = e^{a}$

Replacing $\omega \to 1/\omega$: $\omega = e^{-a}$

Series expansion : $(-\omega)^{-n}\left((b^2)^{n/2} + \left(-2\,(b^2)^{-1+\frac{n}{2}}\,c\,n + (b^2)^{n/2}\left(\frac{c\,n}{b^2} + \frac{n}{d} - \frac{e\,n}{b\,d} - \frac{n^2}{d}\right)\right)\omega + O[\omega]^2\right)$

MrvLimit :: SeriesFail : Series failed at

$$e^{\text{LogGamma}\left[-1+n+\frac{d}{\omega}\right] - \ll 1 \gg - \ll 1 \gg + \text{LogGamma}\left[n+\frac{d}{\ll 1 \gg} - \frac{-b\,d+\frac{\ll 1 \gg}{\ll 1 \gg}}{2\,\omega\sqrt{\ll 1 \gg - \ll 1 \gg}}\right]}\left(-\frac{1}{\frac{1}{\omega}}\right)^{-n}\left(b^2 - \frac{4\,c}{\frac{1}{\omega}}\right)^{n/2}$$

Die Ausgabe von Series enthält hier einen Faktor $(-\omega)^{-n}$. Das genaue Verhalten dieses Faktors ist natürlich abhängig vom Vorzeichen von n, deswegen scheitert der Algorithmus hier. Die einzige Alternative ist, den Faktor von Hand aus der Aufgabe zu entfernen. ω ist in diesem Fall e^a, allerdings wurde der Term einmal in eine höhere Wachstumsklasse angehoben, und auch die Richtung $a \to 0^+$ spielt eine Rolle. Der richtige Faktor, um $(-\omega)^{-n}$ zu neutralisieren, lautet insgesamt $(-a)^n$.

```
In[8]:= MrvLimit[f * (-a)^n, a → 0, Direction → -1]
Out[8]= (b^2)^(n/2)
```

Damit ist im Fall $b > 0$ der Grenzwert:

$$\lim_{a\to 0^+} f = b^n \lim_{a\to 0^+} (-a)^{-n} = (-1)^n b^n \lim_{a\to 0^+} a^{-n}$$

Betrachten wir nun den Fall $b < 0$. Diesmal ändert sich eine wesentliche Randbedingung, das Vorzeichen von b. Deswegen muss hier auch unbedingt der Cache der berechneten Grenzwerte geleert werden, da einige der gespeicherten Ergebnisse nur unter der Annahme $b > 0$ gültig sind.

```
In[9]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = -1; Protect[Sign];

        Unprotect[Power]; √(b^2) = -b; Protect[Power];

        MrvLimitClearCache[];
```

```
In[10]:= MrvLimit[f, a → 0, Direction → -1]
MrvLimit :: UnknownSign : Cannot determine sign : n (-∞)
```

Anscheinend müssen wir in diesem Fall auch das Vorzeichen von n beachten. Wir überprüfen zunächst $b < 0, n > 0$:

```
In[11]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = -1; Sign[n] = 1; Protect[Sign];

         Unprotect[Power]; Sqrt[b^2] = -b; Protect[Power];

         MrvLimitClearCache[];
```

```
In[12]:= MrvLimit[f, a → 0, Direction → -1, Debug → 1]
```

$$\text{Series expansion : } (-\omega)^{-n}\,\omega^{n}\left(\frac{(b^2)^{n/2}\,d^{-n}\,\text{Gamma}\left[-\frac{cd}{b^2}+\frac{e}{b}+n\right]}{\text{Gamma}\left[\frac{-cd+be}{b^2}\right]}+(\ldots)\,\omega+O[\omega]^2\right)$$

```
MrvLimit::SeriesFail : Series failed at
```

$$\frac{\text{Gamma}\left[n+\frac{d}{2\omega}-\frac{-bd+\frac{2e}{\omega}}{2\omega\sqrt{b^2-\frac{4\ll 1\gg\ll 1\gg}{\ll 1\gg}}}\right]\mathbb{e}^{\ll 1\gg}\ \ll 1\gg\ \ll 1\gg\ (\ll 1\gg-\ll 1\gg)^{n/2}}{\text{Gamma}\left[\frac{d}{2\omega}-\frac{-bd+\frac{2e}{\omega}}{2\omega\sqrt{b^2-\frac{4\ll 1\gg\ll 1\gg}{\ll 1\gg}}}\right]}$$

Der Term $(-\omega)^{-n}\omega^n$ bereitet diesmal die Bauchschmerzen. Der Faktor wird kompensiert durch den Faktor $(-a)^n a^{-n}$:

```
In[13]:= MrvLimit[f * (-a)^n * a^(-n), a → 0, Direction → -1]
```

$$\text{Out[13]= } \frac{(b^2)^{n/2}\,d^{-n}\,\text{Gamma}\left[-\frac{cd}{b^2}+\frac{e}{b}+n\right]}{\text{Gamma}\left[\frac{-cd+be}{b^2}\right]}$$

Da $(-a)^{-n}a^n = (-1)^n$ und $(b^2)^{n/2} = \left(\sqrt{b^2}\right)^n = (-b)^n$ ist, ergibt sich im Fall $b < 0, n > 0$ der Grenzwert:

$$\lim_{a\to 0^+} f = (-1)^n(-b)^n d^{-n}\frac{\Gamma(-cdb^{-2}+eb^{-1}+n)}{\Gamma(-cdb^{-2}+eb^{-1}))} = b^n d^{-n}\text{Pochhammer}\left(-cdb^{-2}+eb^{-1},n\right)$$

Nächster Schritt ist $b < 0, n < 0$. Wir versuchen gleich die Berechnung mit dem gleichen Vorfaktor wie eben:

```
In[14]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = -1; Sign[n] = -1; Protect[Sign];

         Unprotect[Power]; Sqrt[b^2] = -b; Protect[Power];

         MrvLimitClearCache[];
```

```
In[15]:= MrvLimit[f * (-a)^n * a^(-n), a → 0, Direction → -1]
```

$$\text{Out[15]= } \frac{(b^2)^{n/2}\,d^{-n}\,\text{Gamma}\left[-\frac{cd}{b^2}+\frac{e}{b}+n\right]}{\text{Gamma}\left[\frac{-cd+be}{b^2}\right]}$$

Das ist das gleiche Ergebnis, wie im Fall $n > 0$. Es bleibt noch $n = 0$ zu berechnen:

```
In[16]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = -1; Sign[n] = .; Protect[Sign];

         Unprotect[Power]; √(b^2) = -b; Protect[Power];

         MrvLimitClearCache[];
```

```
In[17]:= MrvLimit[f /. n → 0, a → 0, Direction → -1]
Out[17]= 1
```

Diesmal ergibt sich ein einfaches Ergebnis ohne weitere Nebenbedingungen. Das gleiche Ergebnis kommt auch heraus, wenn man in die anderen zwei Lösungen für n den Wert $n = 0$ einsetzt.

Schließlich, als letzter Fall, bleibt $b = 0$ zu berechnen:

```
In[18]:= Unprotect[Sign]; Sign[d] = 1; Sign[b] = .; Protect[Sign];

         Unprotect[Power]; √(b^2) = .; Protect[Power];

         MrvLimitClearCache[];
```

```
In[19]:= MrvLimit[f /. b → 0, a → 0, Direction → -1]
```

```
MrvLimit :: SeriesFail : Series failed at
```

$$2^n e^{\text{LogGamma}\left[-1+n+\frac{d}{\omega}\right]-\text{LogGamma}[\,\ll 1\gg\,]\ll 1\gg\ll 1\gg+\text{LogGamma}\left[n+\frac{d}{2\omega}-\frac{e}{2\sqrt{-\ll 1\gg}}\right]}\left(-\frac{1}{\frac{1}{\omega}}\right)^{-n}\left(-\frac{c}{\frac{1}{\omega}}\right)^{n/2}$$

Wagen wir die kühne Spekulation, dass diesmal der kompensierende Faktor $(-a)^n(-ca)^{-n/2}$ ist, ausgehend von der Struktur der Funktion, an der Series scheiterte:

```
In[20]:= MrvLimit[f * (-a)^n (-c a)^(-n/2) /. b → 0, a → 0, Direction → -1]
Out[20]= 1
```

Richtig geraten. Der Grenzwert lautet für $b = 0$ also:

$$\lim_{a\to 0^+} f = \lim_{a\to 0^+} (-a)^{-n}(-ca)^{n/2} = (-1)^n(-c)^{n/2} \lim_{a\to 0^+} a^{-n/2}$$

Es ergibt sich also folgendes Gesamtbild:

$$\begin{aligned}
&\lim_{a\to 0^+} f = (-1)^n b^n \lim_{a\to 0^+} a^{-n} && \text{, wenn } b > 0, d > 0 \text{ ist,}\\
&\lim_{a\to 0^+} f = (-1)^n(-c)^{n/2} \lim_{a\to 0^+} a^{-n/2} && \text{, wenn } b = 0, d > 0 \text{ ist.}\\
&\lim_{a\to 0^+} f = b^n d^{-n}\,\text{Pochhammer}\left(-cdb^{-2}+eb^{-1}, n\right) && \text{, wenn } b < 0, d > 0 \text{ ist,}
\end{aligned}$$

Das Beispiel zeigt eindrucksvoll, dass ein einfaches Vorzeichen einer Konstante dramatische Auswirkungen auf das Ergebnis einer Grenzwertaufgabe haben kann. In diesem Fall ändert sich selbst die Wachstumsklasse vollständig: Für $b > 0$ konstant, für $b \leq 0$ wechselnd 0 oder $+\infty$, je nach dem Vorzeichen von n.

Trotzdem kann man mit etwas Intuition und Experimentierfreude die Aufgabe so umstellen, dass sie für MrvLimit lösbar bleibt, wodurch MrvLimit in den Händen eines geübten Mathematikers auch bei solchen Aufgaben zu einem wertvollen Werkzeug wird.

Literaturverzeichnis

[Gru96] D. Gruntz: On Computing Limits in a Symbolic Manipulation System
Diss. ETH 11432, Swiss Federal Institute of Technology, Zürich (1996)

Dieses Buch diente als Vorlage für diese Arbeit und steht Interessierten im Internet unter http://www.cs.fh-aargau.ch/~gruntz/publications2.html zum Lesen zur Verfügung.

[Sha04] J. Shackell: Symbolic Asymptotics
Springer Verlag (2004)

Dieses recht junge Buch stellt die theoretischen Grundlagen dieser Arbeit umfassend und weiterführend dar.

[Har10] G. H. Hardy: Orders of Infinity
Cambridge Tracts in Mathematics and Mathematical Physics 12 (1910)

Die wegweisende Arbeit von Hardy über asymptotisches Wachstum ist das Fundament des Algorithmus.

[RSSH96] D. Richardson, B. Salvy, J. Schackell, J. v. d. Hoeven:
Asymptotic Expansions of exp-log Functions
Proceedings of the International Symposium on Symbolic and Algebraic Computation (1996)

[Ros83] M. Rosenlicht: Hardy Fields
Journal of Mathematical Analysis and Applications 93 (1983), s. 297

[Ric69] D. Richardson:
Solution of the Indentity Problem for Integral Exponential Functions
Zeitschrift f. math. Logik und Grundlagen der Math. 15 (1969), s.333

[Mac80] A. Macintyre: The Laws of Exponentiation
Model Theory and Arithmetic Proceedings, Paris 1979/80

[Koe93] W. Koepf: Mathematik mit Derive
Vieweg, Braunschweig/Wiesbaden (1993)

Auf dieses Buch geht ein Beispiel aus Kapitel 1.4 zurück.

A. Versicherung des Verfassers

Ich versichere hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Udo Richter