Masterarbeit

Konstruktive Analysis mit exakten reellen Zahlen

angefertigt von Franziskus Wiesnet

Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilans Universität München

20. September 2017

Betreuer: Prof. Dr. Helmut Schwichtenberg

Vorwort

Persönliches Vorwort des Autors

Ein herzliches Willkommen an den Leser dieser Arbeit über die konstruktive Mathematik.
Damit ich auch direkt in dieser Arbeit das Wort ergreifen kann, habe ich mich entschieden, das Vorwort in einen persönlichen und in einen sachlichen Teil zu gliedern. Zunächst sei allen an diesem Werk beteiligten Personen gedankt. Allen voran natürlich meinem Betreuer Prof. Dr. Helmut Schwichtenberg. Es war eine sehr enge und lehrreiche Zusammenarbeit mit ihm und ich hoffe, dass wir auch in Zukunft noch weiter miteinander arbeiten. Weiter möchte ich Quirin Schroll, einem meiner Kommilitonen, danken. Mit ihm zusammen ist die Haskeldatei `Effizient.hs` auf der beiliegenden CD entstanden. Dort wird die Theorie des letzten Abschnitts im dritten Kapitel effizient umgesetzt. Zuletzt noch Danke an alle, die diese Arbeit korrekturgelesen haben. Dazu gehören neben Prof. Schwichtenberg und Qurin Schroll auch Christine Lichtenegger.
Weil ich plane, in Zukunft noch des Öfteren Mathematikbücher zu verfassen, habe ich diese Masterarbeit als Einstieg dafür genutzt. Insbesondere ist sie darauf ausgelegt, dass jeder Interessent der mathematischen Logik sie auch verstehen kann. Mir war wichtig, dass ich mich nicht nur an einen kleinen Personenkreis wende, wie es oft bei Bachelor- und Masterarbeiten der Fall ist, sondern viele mit dieser Arbeit erreiche. Ein Aspekt dieser Arbeit ist auch, die Mathematik als „die Kunst des Lernens“, wie es die Übersetzung aus dem Altgriechischen auch sagt, zu verstehen. Mit einer klaren Strukturierung und einer ansprechenden äußeren Form soll das Lernen aus diesem Buch besonders angenehm sein und dem einen oder anderen vielleicht sogar Spaß machen. Deswegen habe ich auch mit Randbemerkungen und Farben gearbeitet. Für Verbesserungsvorschläge bezüglich der didaktischen Aufbereitung bin ich ebenso dankbar, wie auch für das Melden von inhaltlichen oder sprachlichen Fehlern.
Da das Schreiben und insbesondere das Betrachten des Geschriebenen durchaus auch Spaß und Freude gemacht haben, kann ich mir gut vorstellen, nach dem Einreichen dieser Masterarbeit noch weitere Kapitel hinzuzufügen. Themen für weitere Kapitel wären beispielsweise stetige Funktionen auf den reellen Zahlen, komplexe Zahlen oder ein tieferer Einblick in die interne Umsetzung von Minlog. Zunächst ist eine englische Übersetzung geplant, damit die Einführung in Minlog auch bei internationalen Publikationen verwendet werden kann. Dass die Arbeit zunächst auf Deutsch erscheint, hat den Grund, dass dies meine Muttersprache ist, wodurch es für mich leichter ist, zunächst meine Ideen in deutscher Sprache umzusetzen und dann auf Englisch zu übersetzen. Außerdem sind deutsche Texte über Minlog kaum zu finden, sodass damit auch wieder eine Lücke geschlossen wird.
Nun wünsche ich aber viel Spaß und lehrreiche Stunden mit meiner Arbeit.

Sachliches Vorwort des Autors

Diese Masterarbeit über konstruktive Analysis teilt sich in drei Kapitel. Das erste Kapitel dient dazu, dem Leser eine Einführung in die konstruktive Mathematik zu geben. Dazu führen wir das Kalkül des natürlichen Schließens und die Theorie der berechenbaren Funktionale ein. In dieser Theorie ist es uns möglich Programmextraktion aus Beweisen durchzuführen. Das zweite Kapitel ist eine Einführung in den Beweisassisenten Minlog. Dort wird erklärt, wie sich die Theorie aus dem ersten Kapitel mit Computerhilfe umsetzten lässt. Nach dem Studium dieses Kapitels sollte das Schreiben von Beweisen in Minlog technisch keine Probleme mehr machen. Eine praktische Anwendung der ersten beiden Kapiteln wird im dritten Kapitel gegeben. Dort werden zunächst die konstruktiven reellen Zahlen eingeführt und gezeigt, wie man aus einem Beweis die Algorithmen zur Addition und zur Division von reellen Zahlen in der SD-Darstellung erhält. Die Implementierung der Division in Minlog befindet sich auf der beiliegenden CD. Dort sind auch Haskelldateien `RealCoI.hs`, `CoIDiv.hs` und `Effizient.hs`, in der die extrahierten Terme als Haskell-Datei umgesetzt sind. Die ersten beiden Dateien sind direkt aus den entsprechenden Minlog-Dateien entstanden, während `Effizient.hs` auf Effizienz optimiert und noch etwas verallgemeinert wurde. Das Neue ist, dass nun auch ein Algorithmus für die Division in der SD-Darstellung extrahiert ist.
Zum Verständnis dieser Arbeit sind nur wenige Vorkenntnisse der mathematischen Logik erforderlich. Optimal wäre es, eine Einführungsvorlesung der mathematischen Logik besucht zu haben. Wirklich notwendig ist aber nur, dass man die mathematische Denk- und Argumentationsweise verinnerlicht hat. Dieses Buch ist auch selbst als Einführungs- und Lehrbuch für die konstruktive Mathematik gedacht. Zum Umgang damit sei noch erwähnt, dass wichtige Begriffe blau hervorgehoben sind und, ist eine Zeichenkette in einem Befehl für Minlog rot geschrieben, so muss sie zunächst noch durch eine Zeichenkette ersetzt werden, damit ein für Minlog lesbarer Befehl entsteht. Am Ende der Arbeit befinden sich das Stichwortverzeichnis und das Verzeichnis für alle in diesem Buch genannten Minlog-Befehle. Für interessierte Leser, die sich gerne weiterhin mit der konstruktiven Mathematik beschäftigen wollen, gibt es im Literaturverzeichnis eine Auswahl an Skripten.

Inhaltsverzeichnis

Kapitel 1

Einführung in die Theorie der berechenbaren Funktionale

Motivation 1.0.1. Das Ziel dieses Kapitels wird es sein, die theoretischen Grundlagen für konstruktive Mathematik zu geben. Insbesondere wollen wir am Ende den rechnerischen Gehalt eines Beweises bestimmen können. Die Theorie werden wir in den darauf folgenden Kapiteln dann mit Computerhilfe anwenden. Es wird davon ausgegangen, dass Definitionen aus einer Logikanfängervorlesung schon bekannt sind, wie beispielsweise die Definitionen von Formeln, Termen oder freien Variablen. In der Literatur [7], welche als Vorlage für dieses Kapitel dient, sind auch diese Begriffe noch erklärt.

1.1 Das Kalkül des natürlichen Schließens

Definition 1.1.1. Eine Herleitung im Kalkül des natürlichen Schließens ist rekursiv definiert. Jede Herleitung für eine Formel A ist ein Herleitungsbaum M und besitzt eine Annahmenmenge Γ. Wir definieren nun die Annahmeregel sowie die Regeln für $\rightarrow$ und $\forall$:

Annahmeregel: Ist A eine Formel, so ist

$$u\colon\ A$$

eine Herleitung von A mit Annahmenmenge $\{(u\colon\ A)\}$. Dabei wird die Annahme A mit einer Annahmenvariablen u versehen, um später auf diese Annahme zurückgreifen zu können.

$\rightarrow^+$-Regel: Ist M eine Herleitung einer Formel A mit Annahmenmenge Γ, B eine weitere Formel und u eine Annahmenvariable, so ist

$$\frac{\begin{array}{c}|M\\ A\end{array}}{B \rightarrow A}\ \rightarrow^+_u$$

eine Herleitung von $A \rightarrow B$ mit Annahmenmenge $\Gamma \setminus \{(u\colon B)\}$.

$\rightarrow^-$-Regel: Sind A und B Formeln, M eine Herleitung von $A \rightarrow B$ mit Annahmenmenge Γ sowie N eine Herleitung von A mit Annahmenmenge Δ. Dann ist

$$\dfrac{\begin{matrix}|M & |N\\ A\to B & A\end{matrix}}{B}\ \to^-$$

eine Herleitung von B mit Annahmenmenge $\Gamma\cup\Delta$.

$\forall^+$**-Regel:** Ist M eine Herleitung einer Formel A mit Annahmenmenge Γ und x eine Variable, die in keiner Formel von Γ frei ist. Dann ist

$$\dfrac{\begin{matrix}|M\\ A\end{matrix}}{\forall_x A}\ \forall^+_x$$

eine Herleitung von $\forall_x A$ mit Annahmenmenge Γ.

$\forall^-$**-Regel:** Ist M eine Herleitung von $\forall_x A(x)$ mit Annahmenmenge Γ und r ein Term, so ist

$$\dfrac{\begin{matrix}|M & \\ \forall_x A(x) & r\end{matrix}}{A(r)}\ \forall^-$$

eine Herleitung von $A(r)$ mit Annahmenmenge Γ.

Notation 1.1.2. Wir sagen, A ist aus Γ herleitbar und schreiben $\Gamma\vdash A$, wenn es eine Herleitung von A gibt, deren Annahmenmenge eine Teilmenge von Γ ist. $\varnothing\vdash A$ wird durch $\vdash A$ abgekürzt.
Die Negation einer Formel A wird durch $\neg A := A\to\bot$ definiert . Dabei ist $\bot$ das Falsum und wird zunächst nur als Prädikatenvariable angesehen.

$\to$ soll rechts assoziativ sein, d.h. $A\to B\to C$ ist zu lesen als $A\to(B\to C)$.

Beispiel 1.1.3. Eine Herleitung der Formel $(A\to B\to C)\to((C\to A)\to B)\to A\to C$ wird wie folgt notiert:

$$\dfrac{\dfrac{\dfrac{\dfrac{[u:A\to B\to C]\quad [v:A]}{B\to C}\quad \dfrac{[w:(C\to A)\to B]\quad \dfrac{[v:A]}{C\to A}\to^+_t}{B}}{C}}{A\to C}\to^+_v}{((C\to A)\to B)\to A\to C}\to^+_w}{(A\to B\to C)\to((C\to A)\to B)\to A\to C}\to^+_u$$

Wie man sieht, reicht es, den Herleitungsbaum anzugeben. Wird eine Annahme wieder durch eine $\to^+$-Regel abgebunden, so wird sie eingeklammert. Auf diese Weise kann man sofort die Annahmenmenge der Herleitung ablesen, ohne dass sie explizit angegeben werden muss. Hier sieht man, dass die Annahmenmenge die leere Menge ist.
Ist außerdem klar, welche Regel wie angewendet wurde (zum Beispiel bei der $\to^-$-Regel), kann man die Beschriftung am Rande der Zeile auch weglassen.

Beispiel 1.1.4. Für die Formel $\forall_x(A\to B)\to\forall_x A\to\forall_x B$ haben wir folgende Herleitung:

$$\dfrac{\dfrac{\dfrac{\dfrac{[u:\forall_x(A\to B)]\quad x}{A\to B}\quad \dfrac{[v:\forall_x A]\quad x}{A}}{B}\ \forall^+_x}{\forall_x B}}{\forall_x A\to\forall_x B}\to^+_v}{\forall_x(A\to B)\to\forall_x A\to\forall_x B}\to^+_u$$

Es gilt $x \notin FV(\{\forall_x(A \to B), \forall_x A\})$, deswegen ist die Variablenbedingung bei der Alleinführung $\forall_x^+$ erfüllt.

Definition 1.1.5. Wir definieren nun noch die Regeln für $\wedge, \vee, \exists$.

$\wedge^+$**-Regel:** Ist M eine Herleitung von A mit Annahmenmenge Γ und N eine Herleitung von B mit Annahmenmenge Δ, dann ist

$$\dfrac{\begin{matrix}|M & |N\\ A & B\end{matrix}}{A \wedge B}\,\wedge^+$$

eine Herleitung von $A \wedge B$ mit Annahmenmenge $\Gamma \cup \Delta$.

$\wedge^-$**-Regel:** Sei M eine Herleitung von $A \wedge B$ mit Annahmenmenge Γ und N eine Herleitung von C mit Annahmenmenge Δ sowie u, v zwei Annahmenvariablen, dann ist

$$\dfrac{\begin{matrix}|M & |N\\ A \wedge B & C\end{matrix}}{C}\,\wedge^-_{u,v}$$

eine Herleitung von C mit Annahmenmenge $\Gamma \cup (\Delta \setminus \{(u:A),(v:B)\})$.

$\vee^+$**-Regeln:** Es sei M eine Herleitung von A mit Annahmenmenge Γ und B eine weitere Formel, dann ist

$$\dfrac{\begin{matrix}|M\\ A\end{matrix}}{A \vee B}\,\vee^+_0$$

eine Herleitung von $A \vee B$ mit Annahmenmenge Γ und

$$\dfrac{\begin{matrix}|M\\ A\end{matrix}}{B \vee A}\,\vee^+_1$$

ist eine Herleitung von $B \vee A$ mit Annahmenmenge Γ.

$\vee^-$**-Regel:** Ist M eine Herleitung von $A \vee B$ mit Annahmenmenge Γ, N eine Herleitung von C mit Annahmenmenge Δ und L eine Herleitung von C mit Annahmenmenge Ξ, so ist

$$\dfrac{\begin{matrix}|M & |N & |L\\ A \vee B & C & C\end{matrix}}{C}\,\vee^-$$

eine Herleitung von C mit der Annahmenmenge $\Gamma \cup (\Delta \setminus \{(u:A)\}) \cup (\Xi \setminus \{(v:B)\})$.

$\exists^+$**-Regel:** Ist $A(x)$ eine Formel, r ein Term und M eine Herleitung von $A(r)$ mit Annahmenmenge Γ, so ist

$$\dfrac{\begin{matrix}r & |M\\ & A(r)\end{matrix}}{\exists_x A(x)}\,\exists^+$$

eine Herleitung von $\exists_x A$ mit Annahmenmenge Γ.

$\exists^-$**-Regel:** Ist M eine Herleitung von $\exists_x A$ mit Annahmenmenge Γ, N eine Herleitung von B mit Annahmenmenge Δ und gelten die Variablenbedingungen $x \notin FV(\Delta \setminus \{(u:A)\})$ und $x \notin FV(B)$, dann ist

$$\dfrac{\begin{matrix}|M & |N\\ \exists_x A & B\end{matrix}}{B}\,\exists^-_{x,u}$$

eine Herleitung von B mit Annahmenmenge $\Gamma \cup (\Delta \setminus \{u:A\})$.

Bemerkung 1.1.6. Es ist auch möglich, die Regeln aus Definition 1.1.5 durch Axiome zu ersetzen. Für $\wedge$ beispielsweise wären diese gegeben durch

$$\wedge^+ : A \to B \to A \wedge B$$
$$\wedge^- : A \wedge B \to (A \to B \to C) \to C.$$

Wir werden später in diesem Kapitel induktiv definierte Prädikate einführen und sehen, dass es sich bei $\vee, \wedge$ und $\exists$ um solche handelt.

Definition 1.1.7. Wir definieren den schwachen Existenzquantor $\tilde{\exists}$ sowie die schwache Disjunktion $\tilde{\vee}$ durch $\tilde{\exists}_x A := \neg\forall_x \neg A$ und $A \tilde{\vee} B := \neg(\neg A \wedge \neg B)$.

Bemerkung 1.1.8. Der normale Existenzquantor sowie das normale Oder sind echt stärker als ihre schwachen Pendants, denn es gilt $\vdash A \vee B \to A \tilde{\vee} B$ und $\vdash \exists_x A \to \tilde{\exists}_x A$:

$$\dfrac{\dfrac{\dfrac{[u : A \vee B] \qquad \dfrac{\dfrac{[t : \neg A \wedge \neg B]}{\bot}\wedge^-_{v,w} \qquad \dfrac{[t : \neg A \wedge \neg B]}{\bot}\wedge^-_{v,w}}{\bot}\wedge^-_{r,s}}{\bot}\to^+_t}{A \tilde{\vee} B}\to^+_u}{A \vee B \to A \tilde{\vee} B}$$

$$\dfrac{\dfrac{\dfrac{[u : \exists_x A] \qquad \dfrac{\dfrac{[v : \forall_x \neg A] \qquad x}{\neg A} \qquad [w : A]}{\bot}\exists^-_{x,w}}{\bot}\to^+_v}{\tilde{\exists}_x A}\to^+_u}{\exists_x A \to \tilde{\exists}_x A}$$

Die Variablenbedingung bei $\exists^-_{x,w}$ ist erfüllt, denn $x \notin FV(\forall_x \neg A)$ und $x \notin FV(\bot)$.
Die Umkehrungen, also die Formeln $A \tilde{\vee} B \to A \vee B$ und $\tilde{\exists}_x A \to \exists_x A$, sind im Allgemeinen nicht herleitbar. Dies kann man mit Hilfe von Gegenmodellen zeigen, welche aber nicht Gegenstand dieser Arbeit sind. Bei Interesse sei aber auf Kapitel 1.3 von [7] verwiesen.

Motivation 1.1.9. Im natürlichen Schließen hat das Falsum $\bot$ keine weitere Bedeutung, sondern ist nur eine Prädikatenvariable. Insbesondere gilt das Ex-falso-quodlibet, das heißt $\bot \to A$, nicht für jede Formel A, denn über $\bot$ ist nichts gegeben. Es lässt sich außerdem zeigen, dass für eine Formel A im Allgemeinen $\neg\neg A \to A$ nicht herleitbar und sogar echt stärker als $\bot \to A$ ist. Das bringt uns zur Definition der intuitionistischen und klassischen Logik.

Definition 1.1.10. Die Menge Efq besteht genau aus den Formeln $\forall_{\vec{x}}(\bot \to R\vec{x})$ und die Menge $Stab$ besteht genau aus den Formeln $\forall_{\vec{x}}(\neg\neg R\vec{x} \to R\vec{x})$, wobei $R \neq \bot$ ein n-stelliges Relationssymbol ist für $n \in \mathbb{N}$. Wir schreiben $\Gamma \vdash_i A$ für $\Gamma \cup Efq \vdash A$ und sagen „A ist intuitionistisch aus Γ herleitbar". Ebenso schreiben wir $\Gamma \vdash_c A$ für $\Gamma \cup Stab \vdash A$ und sagen „A ist klassisch aus Γ herleitbar".

Bemerkung 1.1.11. Durch Induktion über den Formelaufbau kann man leicht beweisen, dass $\vdash_i \bot \to A$ für jede Formel A und $\vdash_c \neg\neg A \to A$ für jede Formel A, die nicht $\exists$ oder $\vee$ enthält, gilt.

1.2 Typen, Algebren und Terme

Motivation 1.2.1. In diesem Abschnitt definieren wir die Terme von Gödels T und dessen Erweiterung T^+, aus denen später das extrahierte Programm eines Beweises bestehen soll und mit denen auch das Programm Minlog arbeitet. Dieser Abschnitt ist daher sehr theoretisch. Wir werden aber versuchen, die Theorie durch viele Beispiele mit Leben zu füllen. Bei all den Definitionen ist wichtig, dass man diese erst rein syntaktisch auffassen sollte. Die Bedeutung der definierten Objekte wird sich erst später durch ihre Verwendung ergeben.

Notation 1.2.2. Anstelle von $a_0, \dots, a_{n-1}$ für $n \in \mathbb{N}_0$ schreiben wir oft $(a_i)_{i<n}$ und, wenn wir die Anzahl dieser Objekte implizit lassen wollen, schreiben wir nur $\vec{a}$. Wir verwenden auch häufig die abkürzende Schreibweise $(a_i)_{i<n} \to b$ bzw. $\vec{a} \to b$ für $a_0 \to \cdots \to a_{n-1} \to b$.

Definition 1.2.3. Wir definieren nun rekursiv Typen, Konstruktortypen und Algebren. Dabei seien ξ und $\vec{\alpha}$ verschiedene Typvariablen . Die α_l werden Typparameter genannt.

- Jedes α_l ist ein Typ mit Parameter $\vec{\alpha}$.
- Jede Algebra mit Parameter $\vec{\alpha}$ ist auch ein Typ mit Parameter $\vec{\alpha}$.
- Ist ρ ein Typ ohne Parameter und σ ein Typ mit Parameter $\vec{\alpha}$, dann ist auch $\rho \to \sigma$ ein Typ mit Parameter $\vec{\alpha}$.
- Sind $\vec{\rho}$ Typen mit Parameter $\vec{\alpha}$ und $\vec{\sigma}_0, \dots, \vec{\sigma}_{n-1}$ Typen für $n \in \mathbb{N}_0$ mit Parameter $\vec{\alpha}$, dann ist $\vec{\rho} \to (\vec{\sigma}_i \to \xi)_{i<n} \to \xi$ ein Konstruktortyp mit Parameter $\vec{\alpha}$.
- Sind $\kappa_0, \dots, \kappa_{k-1}$ für $k \in \mathbb{N}^+$ Konstruktortypen mit Parameter $\vec{\alpha}$, dann ist $\mu_\xi(\kappa_0, \dots, \kappa_{k-1})$ eine Algebra mit Parameter $\vec{\alpha}$.

Beispiel 1.2.4. Hier ist eine Liste von wichtigen Algebren, auf die wir uns auch weiter beziehen werden.

Einheitsalgebra : $\mathbb{U} := \mu_\xi(\xi)$.

Boolesche Algebra : $\mathbb{B} := \mu_\xi(\xi, \xi)$

Algebra der natürlichen Zahlen : $\mathbb{N} := \mu_\xi(\xi, \xi \to \xi)$

Algebra der positiven, binären Zahlen : $\mathbb{P} := \mu_\xi(\xi, \xi \to \xi, \xi \to \xi)$

Algebra der Ordinalzahlen : $\mathbb{O} := \mu_\xi(\xi, \xi \to \xi, (\mathbb{N} \to \xi) \to \xi)$

Listen mit Parameter α : $\mathbb{L}(\alpha) := \mu_\xi(\xi, \alpha \to \xi \to \xi)$

Produkttyp zweier Typen α und β : $\alpha \times \beta := \mu_\xi(\alpha \to \beta \to \xi)$

Summentyp zweier Typen α und β : $\alpha + \beta := \mu_\xi(\alpha \to \xi, \beta \to \xi)$

Definition 1.2.5. Wir definieren für jeden Konstruktortyp $\kappa_i(\xi)$ in einer Algebra $\iota = \mu_\xi(\kappa_0(\xi), \ldots, \kappa_{n-1}(\xi))$ ein Konstruktorsymbol C_i vom Typ $\kappa_i(\iota)$.
Will man den Typ eines Konstruktors klar machen, schreibt man diesen hochgestellt dazu: $C_i^{\kappa_i(\iota)}$
Einige Konstruktoren haben auch standardisierte Namen. Zum Beispiel wird der einzige Konstruktor von $\mathbb{U}$ mit **u** bezeichnet und die beiden Konstruktoren von $\mathbb{B}$ mit tt (truth) und ff (falsehood). Die zwei Konstruktoren der natürlichen Zahlen bezeichnet man mit 0 (zero) und S (successor). Die Konstruktoren der positiven Zahlen kann man mit $1^{\mathbb{P}}$, $S_0^{\mathbb{P}\to\mathbb{P}}$ und $S_1^{\mathbb{P}\to\mathbb{P}}$ bezeichnen. Bei der Interpretation von S_0 bzw. S_1 ist jedoch wichtig, dass zum Beispiel $S_0 1$ als die Zahl 10 und nicht als 01 in Binärdarstellung zu verstehen ist. Für den Listentyp bezeichnet $\text{nil}^{\mathbb{L}(\alpha)}$ den Konstruktor der leeren Liste und den zweistelligen Konstruktor bezeichnet man entweder mit $\text{cons}^{\alpha\to\mathbb{L}(\alpha)\to\mathbb{L}(\alpha)}$ oder als Infixnotation mit $\_ :: \_$. Für den Summentyp $\alpha + \beta$ werden die beiden Konstruktoren durch $\text{in}_0^{\alpha\to\alpha+\beta}$ und $\text{in}_1^{\beta\to\alpha+\beta}$ dargestellt. Beim Produkttyp schreiben wir den einzigen Konstruktor als Paar. Das heißt, für $C_0 ab$ schreiben wir $\langle a, b\rangle$.

$1 :: nil$ ist zum Beispiel die Liste von natürlichen Zahlen, die nur $1 := S0$ als Eintrag hat.

Definition 1.2.6. Zu jeder Algebra ι definieren wir den Rekursionsoperator $\mathscr{R}_\iota^\tau$ in einen Typ τ:
Ist $\kappa = \vec{\rho} \to (\vec{\sigma}_i \to \xi)_{i<n} \to \xi$ ein Konstruktortyp, so setzen wir den Schritttyp $\delta := \vec{\rho} \to (\vec{\sigma}_i \to \iota)_{i<n} \to (\vec{\sigma}_i \to \tau)_{i<n} \to \tau$. Ist nun $\iota = \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$, dann ist der Typ von $\mathscr{R}_\iota^\tau$ gegeben durch

$$\iota \to \delta_0 \to \cdots \to \delta_{k-1} \to \tau.$$

Definition 1.2.7. Die Terme von Gödels T sind rekursiv definiert. Jeder dieser Terme besitzt einen Typen. Will man diesen explizit machen, schreibt man ihn hochgestellt hinter dem Term. Außerdem besitzt jeder Term t eine Menge freier Variablen $FV(t)$, welche wir gleichzeitig definieren:

- Jede getypte Variable x^τ für einen Typ τ ist ein Term mit $FV(x) = x$.
- Jeder Konstruktor C_i einer Algebra ist ein Term mit dem Typ wie in Definition 1.2.5 und $FV(C_i) = \varnothing$.
- Der Rekursionsoperator $\mathscr{R}_\iota^\tau$ zu einer Algebra ι ist ein Term mit dem Typ wie in Definition 1.2.6 und $FV(\mathscr{R}_\iota^\tau) = \varnothing$.
- Ist $M^{\rho\to\sigma}$ ein Term und N^ρ ein Term, dann ist $M^{\rho\to\sigma} N^\rho$ ein Term von Typ σ und $FV(MN) = FV(M) \cup FV(N)$.
- Ist M^τ ein Term und x^ρ eine getypte Variable, dann ist $\lambda_{x^\rho} M^\tau$ ein Term vom Typ $\rho \to \tau$ mit $FV(\lambda_x M) = FV(M) \setminus \{x\}$.

Definition 1.2.8. Wir definieren nun die Relation β-Konversion $\rightarrowtail_\beta$ sowie die Relation η-Konversion $\rightarrowtail_\eta$ zwischen Termen vom selben Typ durch: $\lambda_x M(x) N \rightarrowtail_\beta M(N)$ und, ist $x \notin FV(M)$, dann gilt $\lambda_x(Mx) \rightarrowtail_\eta M$.
Mit $\rightarrowtail$ bezeichnen wir die Vereinigung aller Konversionsregeln (auch die, die noch definiert werden).

Definition 1.2.9. Sei $\iota = \mu_\xi(\vec{\kappa})$ eine Algebra und C_i der i-te Konstruktor dieser Algebra. Weiter sei $\kappa_i(\xi) = (\rho_j)_{j<m} \to (\vec{\sigma}_j \to \xi)_{j<n} \to \xi$ und $\vec{L} = (L_j^{\rho_j})_{j<m}$ sowie $\vec{N} = (N_j^{\vec{\sigma}_j \to \iota})_{j<m}$ seien Terme. Dann ist $C_i\vec{L}\vec{N}$ vom Typ ι. Ist nun außerdem noch $\vec{M} = (M_j^{\delta_j})_{j<n}$ mit δ_j wie in Definition 1.2.6, so haben wir die folgende Konversionsregel

$$\mathscr{R}_\iota^\tau(C_i\vec{L}\vec{N})\vec{M} \mapsto_{\mathscr{R}} (M_i\vec{L}\vec{N})(\lambda_{\vec{x}}(\mathscr{R}_\iota^\tau N_j\vec{x}\vec{M}))_{j<n}.$$

Dabei soll $\vec{x}$ genau so viele Komponenten mit entsprechendem Typ haben, dass alle Argumente von N_j aufgefüllt sind und damit $N_j\vec{x}$ vom Typ ι ist.

Beispiel 1.2.10. Wir geben nun für jede Algebra aus Beispiel 1.2.4 den Typ des Rekursionsoperators mit seinen Konversionsregeln an.

Der Rekursionsoperator zur Einheitsalgebra ist sehr einfach. Dieser ist durch den Typ

$$\mathscr{R}_\mathbb{U}^\tau : \mathbb{U} \to \tau \to \tau$$

gegeben und hat die einzige Konversionsregel

$$\mathscr{R}_\mathbb{U}^\tau \mathbf{u} N \mapsto N.$$

Für die boolesche Algebra sieht der Rekursionsoperator ähnlich aus und lässt sich als Fallunterscheidung interpretieren. Der Typ ist gegeben durch

$$\mathscr{R}_\mathbb{B}^\tau : \mathbb{B} \to \tau \to \tau \to \tau$$

und die beiden Konversionsregeln sind

$$\mathscr{R}_\mathbb{B}^\tau \mathrm{tt} MN \mapsto M$$
$$\mathscr{R}_\mathbb{B}^\tau \mathrm{ff} MN \mapsto N.$$

Interessant wird es bei der Algebra der natürlichen Zahlen. Diese hat die Konstruktortypen $\kappa_0 = \xi$ und $\kappa_1 = \xi \to \xi$. Das gibt die Schritttypen $\delta_0 = \tau$ und $\delta_1 = \mathbb{N} \to \tau \to \tau$ und somit als Typ des Rekursionsoperators

$$\mathscr{R}_\mathbb{N}^\tau : \mathbb{N} \to \tau \to (\mathbb{N} \to \tau \to \tau) \to \tau.$$

Die beiden Konversionsregeln sehen dann wie folgt aus:

$$\mathscr{R}_\mathbb{N}^\tau 0 MN \mapsto M$$
$$\mathscr{R}_\mathbb{N}^\tau Sn MN \mapsto Nn\mathscr{R}_\mathbb{N}^\tau nNM$$

Wir erkennen, dass mit dem Rekursionsoperator auf den natürlichen Zahlen Funktionen f rekursiv definiert werden können, wie man es aus den Mathematikvorlesungen im ersten Semester kennt. Dabei ist $M = f(0)$ der Nullfall und $f(Sn) = N(n, f(n))$ der Rekursionsschritt.

Ähnlich wie bei den natürlichen Zahlen sieht der Rekursionsoperator bei den positiven Zahlen aus. Die ersten beiden Konstruktortypen κ_0 und κ_1 sind die gleichen wie bei den natürlichen Zahlen und für den letzten Konstruktortyp gilt $\kappa_1 = \kappa_2$ und somit auch $\delta_1 = \delta_2$. Der Typ des Rekursionsoperators ist

$$\mathscr{R}_\mathbb{P}^\tau : \mathbb{P} \to \tau \to (\mathbb{P} \to \tau \to \tau) \to (\mathbb{P} \to \tau \to \tau) \to \tau$$

und die drei Konversionsregeln sind

$$\begin{aligned}\mathscr{R}^\tau_{\mathbb{P}} 1 LMN &\mapsto L\\ \mathscr{R}^\tau_{\mathbb{P}} S_0 pLMN &\mapsto Mp\mathscr{R}^\tau_{\mathbb{P}} pLMN\\ \mathscr{R}^\tau_{\mathbb{P}} S_1 pLMN &\mapsto Np\mathscr{R}^\tau_{\mathbb{P}} pLMN.\end{aligned}$$

Bei den Ordinalzahlen haben wir zusätzlich zu den Konstruktortypen κ_0, κ_1 von den natürlichen Zahlen noch den Konstruktortyp $\kappa_2 = (\mathbb{N} \to \xi) \to \xi$. Das gibt den Schritttyp $\delta_2 = (\mathbb{N} \to \mathbb{O}) \to (\mathbb{N} \to \tau) \to \tau$. Der Typ des Rekursionsoperators ist damit

$$\mathscr{R}^\tau_{\mathbb{O}} : \mathbb{O} \to \tau \to (\mathbb{O} \to \tau \to \tau) \to ((\mathbb{N} \to \mathbb{O}) \to (\mathbb{N} \to \tau) \to \tau) \to \tau.$$

Die ersten beiden Regeln sind analog wie bei $\mathbb{N}$ und die dritte Konversionsregel ist gegeben durch

$$\mathscr{R}^\tau_{\mathbb{O}} C_2 fLMN \mapsto Nf\lambda_x(\mathscr{R}^\tau_{\mathbb{O}} fxLMN).$$

Beim Listentyp $\mathbb{L} := \mathbb{L}(\alpha)$ haben wir $\kappa_0 = \xi$ und $\kappa_1 = \alpha \to \xi \to \xi$, was die Schritttypen $\delta_0 = \tau$ und $\delta_1 = \alpha \to \mathbb{L} \to \tau \to \tau$ ergibt. Der Typ des Rekursionsoperators ist somit

$$\mathscr{R}^\tau_{\mathbb{L}} : \mathbb{L} \to \tau \to (\alpha \to \mathbb{L} \to \tau \to \tau) \to \tau$$

und die beiden Konversionsregeln sind

$$\begin{aligned}\mathscr{R}^\tau_{\mathbb{L}} \mathsf{nil} MN &\mapsto M\\ \mathscr{R}^\tau_{\mathbb{L}} (a :: w) MN &\mapsto Naw\mathscr{R}^\tau_{\mathbb{L}} wMN.\end{aligned}$$

Für den Produkttyp $\alpha \times \beta$ haben wir den Konstruktortyp $\kappa_0 = \alpha \to \beta \to \xi$ und den zugehörigen Schritttypen $\delta_0 = \alpha \to \beta \to \tau$. Der Rekursionsoperator hat den Typ

$$\mathscr{R}^\tau_{\alpha\times\beta} : \alpha \times \beta \to (\alpha \to \beta \to \tau) \to \tau$$

und die einzige Konversionsregel ist

$$\mathscr{R}^\tau_{\alpha\times\beta} \langle a, b\rangle N \mapsto Nab.$$

Zuletzt gehen wir noch auf den Rekursionsoperator des Summentyps $\alpha + \beta$ ein. Die beiden Schritttypen sind $\delta_0 = \alpha \to \tau$ und $\delta_1 = \beta \to \tau$ und damit ist

$$\mathscr{R}_{\alpha+\beta} : \alpha + \beta \to (\alpha \to \tau) \to (\beta \to \tau) \to \tau.$$

Die beiden Konversionsregeln erinnern etwas an eine Verallgemeinerung der Konversionsregeln des Rekursionsoperators der booleschen Algebra:

$$\begin{aligned}\mathscr{R}^\tau_{\alpha+\beta} \mathsf{in}_0 aMN &\mapsto Ma\\ \mathscr{R}^\tau_{\alpha+\beta} \mathsf{in}_1 bMN &\mapsto Nb\end{aligned}$$

Beispiel 1.2.11. Mit Hilfe des Rekursionsoperators lassen sich viele uns bekannte Funktionen definieren.
Auf den natürlichen Zahlen ist möglicherweise aus den Anfängervorlesungen die rekursive Definition der Addition bekannt. Für natürliche Zahlen n und m definiert man

$$\begin{aligned} m+0 &:= m \\ m+(Sn) &:= S(m+n). \end{aligned}$$

Mit dem Rekursionsoperator definieren wir daher

$$m+n := \mathscr{R}^{\mathbb{N}}_{\mathbb{N}} n m \lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl).$$

Dieses Beispiel wenden wir zum besseren Verständnis noch auf die Zahlen $1 := S0$ und $2 := SS0$ an:

$$\begin{aligned} 1+2 := S0+SS0 &:= \mathscr{R}^{\mathbb{N}}_{\mathbb{N}} SS0S0\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl) \to \lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl)S0\mathscr{R}^{\mathbb{N}}_{\mathbb{N}} S0S0\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl) \\ &\to \lambda_{l^{\mathbb{N}}}(Sl)\mathscr{R}^{\mathbb{N}}_{\mathbb{N}} S0S0\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl) \to \lambda_{l^{\mathbb{N}}}(Sl)\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl)0\mathscr{R}^{\mathbb{N}}_{\mathbb{N}} 0S0\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl) \\ \to \lambda_{l^{\mathbb{N}}}(Sl)\lambda_{l^{\mathbb{N}}}(Sl)\mathscr{R}^{\mathbb{N}}_{\mathbb{N}} 0S0\lambda_{k^{\mathbb{N}},l^{\mathbb{N}}}(Sl) &\to \lambda_{l^{\mathbb{N}}}(Sl)\lambda_{l^{\mathbb{N}}}(Sl)S0 \to \lambda_{l^{\mathbb{N}}}(Sl)SS0 \to SSS0 =: 3 \end{aligned}$$

Dabei ist $\to$ eine Erweiterung von $\mapsto$ und wie in der unteren Definition 1.2.12 gegeben.
Durch den Rekursionsoperator in die boolesche Algebra kann man auch Eigenschaften von Objekten einer Algebra definieren. Nehmen wir dafür als Beispiel die Eigenschaft G, dass eine natürliche Zahl gerade ist. Diese Eigenschaft können wir durch Rekursion so definieren:

$$\begin{aligned} G(0) &:= \mathrm{tt} \\ G(SN) &:= \neg G(N) \end{aligned}$$

Mit $\neg G(N)$ ist das Komplement von $G(N)$ gemeint, wobei $\neg\mathrm{tt} := \mathrm{ff}$ und $\neg\mathrm{ff} := \mathrm{tt}$. Das Komplement ist mit den Rekursionsoperator definiert durch $\neg x := \mathscr{R}^{\mathbb{B}}_{\mathbb{B}} x\mathrm{fftt}$. Daher definieren wir

$$G(N) := \mathscr{R}^{\mathbb{B}}_{\mathbb{N}} N\mathrm{tt}\lambda_{n^{\mathbb{N}},b^{\mathbb{B}}}(\mathscr{R}^{\mathbb{B}}_{\mathbb{B}} b\mathrm{fftt}).$$

Der Leser kann nun durch die Anwendung mehrerer Konversionsschritte bestimmen, ob $3 := SSS0$ gerade ist oder nicht.

Definition 1.2.12. Wir definieren die Relation $\to$ als Erweiterung von $\mapsto$ auf Termen der selben Typen wie folgt:
Gilt $M \mapsto M'$, so auch $M \to M'$.
Gilt $M \to M'$, so auch $NM \to NM'$, $MN \to M'N$ und $\lambda_x M \to \lambda_x M'$.
Ein Term M befindet sich in Normalform, wenn es kein N gibt mit $M \to N$.
Mit $\to^*$ bezeichnen wir den reflexiven und transitiven Abschluss von $\to$ und mit $\doteq$ bezeichnen wir den reflexiven, transitiven und symmetrischen Abschluss von $\to$.

Motivation 1.2.13. Wir haben nun den Rekursionsoperator durch seinen Typ und seine Berechnungsregeln erklärt und gesehen, dass wir durch diesen viele bekannte Funktionen definieren können. Was jedoch auch auffällt ist, dass die Definition von diesen Funktionen häufig sehr umständlich ist. Das gilt insbesondere dann, wenn

man eine Funktion mit mehreren Argumenten definieren will. Betrachten wir beispielsweise Gleichheit von natürlichen Zahlen als Funktion mit Typ $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$. Die Definitionregeln seien dabei

$$\begin{aligned} 0 = 0 &:= \mathrm{tt} \\ Sn = 0 &:= \mathrm{ff} \\ 0 = Sn &:= \mathrm{ff} \\ Sn = Sm &:= n = m. \end{aligned}$$

Der Leser kann hier stoppen und sich selbst überlegen, wie man diese Funktion mit Hilfe des Rekursionsoperators definieren kann. Eine mögliche Definition wäre die folgende:

$$\_ = \_ := \lambda_n \left(\mathscr{R}_{\mathbb{N}}^{\mathbb{N} \to \mathbb{B}} n \left(\lambda_m \mathscr{R}_{\mathbb{N}}^{\mathbb{B}} m \mathrm{tt} \lambda_{k,b} \mathrm{ff} \right) \lambda_{l,f} \lambda_m \left(\mathscr{R}_{\mathbb{N}}^{\mathbb{B}} m \mathrm{ff} \lambda_{k,b} f k \right) \right)$$

Diese Definition wirkt schon sehr unübersichtlich und es gibt gebundene Variablen, die gar nicht verwendet werden, wie b oder l. Zudem stellt sich auch die Frage, wie man so Gleichheit für allgemeine Typen definieren soll. Viel leichter wäre es, $=$ auf den natürlichen Zahlen direkt als Term zu betrachten, der die obigen vier Konversionsregeln erfüllt. Aus diesem Grund führen wir nun Programmkonstanten ein, zu denen auch der bereits definierte Rekursionsoperator gehört. Hierfür brauchen wir noch eine vorbereitende Definition.

Definition 1.2.14. Ein Konstruktormuster ist ein Term und wie folgt rekursiv definiert:
Jede Variable ist ein Konstruktormuster.
Ist $C^{(\rho_i)_{i<n} \to \iota}$ ein Konstruktor zur Algebra ι und sind $\vec{P} = (P_i^{\rho_i})_{i<n}$ Konstruktormuster und gilt $FV(P_i) \cap FV(P_j) = \varnothing$ für alle $i \neq j$, dann ist auch $(C\vec{P})^{\iota}$ ein Konstruktormuster.

Definition 1.2.15. Wir definieren die Erweiterung T^+ von Gödels T durch die Regeln wie in Definition 1.2.7 mit der zusätzlichen Regel, dass auch jedes Programmkonstantensymbol D ein Term ist mit $FV(D) = \varnothing$. Dabei sind Programmkonstanten in folgender Definition erklärt:

In der Literatur wird eine Programmkonstante auch „definierte Konstante“ genannt. Dieser Begriff wirkt aber etwas überladen, deswegen verwenden wir hier den Namen „Programmkonstante“, so wie sie auch in Minlog bezeichnet wird.

Definition 1.2.16. Eine Programmkonstante D ist durch ihr Symbol D, einen Typ $(\rho_i)_{i<m} \to \sigma$ und einer Liste von Berechnungsregeln

$$D\vec{P}_i := M_i \qquad i \in \{1, \ldots, n\}$$

gegeben. Dabei ist $\vec{P}_i = (P_{ij}^{\rho_j})_{j<m}$ für jedes i eine Liste von Konstruktormustern mit m Komponeten. Weiter kommt jede freie Variable in $\vec{P}_i$ höchstens einmal vor. Jedes M_i ist ein Term vom Typ σ und es gilt $FV(M_i) \subseteq FV(\vec{P}_i)$. Außerdem gilt noch folgende Konsistenzbedingung: Sei $\vec{x}$ eine Liste der freien Variablen aller $P_i(\vec{x})$ und gelte $P_i(\vec{s}) \doteq P_j(\vec{t})$ für Terme $\vec{s}, \vec{t}$, dann ist auch $M_i(\vec{s}) \doteq M_j(\vec{t})$.
Jede Berechnungsregel liefert eine Konversionsregel $\rightarrowtail_D$ durch $D\vec{P}_i(\vec{z}) \rightarrowtail_D M_i$.

Beispiel 1.2.17. Der Rekursionsoperator $\mathscr{R}_{\iota}^{\tau}$ für jede Algebra ι und jedem Typ τ ist ein wichtiges Beispiel für eine Programmkonstante.
Eine weitere wichtige Programmkonstante ist der Caseoperator $\mathscr{C}_{\iota}^{\tau}$ zu einer Algebra $\iota = \mu_{\xi}((\rho_{ij}(\xi))_{j<n_i} \to \xi)_{i<k}$ in einen Typ τ. Der Typ des Caseoperators ist

$$\mathscr{C}_{\iota}^{\tau} : \iota \to ((\rho_{ij}(\iota))_{j<n_i} \to \tau)_{i<k} \to \tau$$

und die Berechnungsregeln sind

$$\mathscr{C}_\iota^\tau(C_i(x_j)_{j<n_i})(y_i)_{i<k} := y_i(x_j)_{j<n_i}$$

für jedes $i < k$. Vergleicht man den Caseoperator und den Rekursionsoperator, so fällt auf, dass der Caseoperator eine abgeschwächte Form des Rekursionsoperators ist. Stellen wir dazu die Konversionsregel des Rekursionsoperators aus Definition 1.2.9 mit der des Caseoperators gegenüber:

$$\mathscr{R}_\iota^\tau(C_i\vec{L}\vec{N})(M_i)_{i<k} \mapsto_{\mathscr{R}} (M_i\vec{L}\vec{N})(\lambda_{\vec{x}}(\mathscr{R}_\iota^\tau N_j\vec{x}\vec{M}))_{j<n}$$
$$\mathscr{C}_\iota^\tau(C_i\vec{L}\vec{N})(y_i)_{i<k} \mapsto_{\mathscr{C}} y_i\vec{L}\vec{N}$$

Dabei sind $\vec{L}, \vec{N}$ und $\vec{M}$ wie in Definition 1.2.9. Wir sehen, dass bei den Konversionsregeln des Rekursionsoperators noch ein Term mehr als Argument steht. Ist jedes M_i in diesem Argument konstant, lässt sich der Term auch mit dem Caseoperator beschreiben.

Definition 1.2.18. Eine Algebra $\iota = \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$ heißt finitär, wenn jeder ihrer Konstruktortypen die Form $\kappa_i = \vec{\iota} \to (\xi)_{j<n} \to \xi$ hat, wobei $\vec{\iota}$ eine (möglicherweise leere) Liste von finitären Algebren ist, und κ_0 die Form $\kappa_0 = \vec{\iota} \to \xi$ hat.
Für finitäre Algebren ι können wir nun die entscheidbare Gleichheit als Programmkonstante $=_\iota^{\iota\to\iota\to\mathbb{B}}$ einführen:
Für alle $i \neq j$ haben wir die Berechnungsregeln

$$(C_i\vec{x} =_\iota C_j\vec{y}) := \mathsf{ff}$$

und für jeden Konstruktor C_i mit Typ $\vec{\rho} \to (\iota)_{j<n_i} \to \iota$ haben wir die Regel

$$\left(C_i\vec{x}_1{}^{\vec{\rho}}\vec{x}_2{}^{(\iota)_{i<n}} =_\iota C_i\vec{y}_1{}^{\vec{\rho}}\vec{y}_2{}^{(\iota)_{i<n}}\right) := \left(\vec{x}_1 =_{\vec{\rho}} \vec{y}_1 \ \mathtt{andb}\ \vec{x}_2 =_{(\iota)_{i<n}} \vec{y}_2\right)$$

Dabei ist $\mathtt{andb}^{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$ die boolesche Konjuktion und durch die Berechnungsregeln

$$\begin{aligned}
\mathsf{tt}\ \mathtt{andb}\ y &:= y\\
\mathsf{ff}\ \mathtt{andb}\ y &:= \mathsf{ff}\\
x\ \mathtt{andb}\ \mathsf{tt} &:= x\\
x\ \mathtt{andb}\ \mathsf{ff} &:= \mathsf{ff}.
\end{aligned}$$

gegeben. Außerdem schreiben wir $\vec{x} =_{\vec{\rho}} \vec{y}$ als Abkürzung für

$$x_0 =_{\rho_0} y_0\ \mathtt{andb}\ \ldots\ \mathtt{andb}\ x_{n-1} =_{\rho_{n-1}} y_{n-1}.$$

Das Beispiel für die entscheidbare Gleichheit auf den natürlichen Zahlen ist in Motivation 1.2.13 gegeben.

Motivation 1.2.19. Wir haben in diesem Abschnitt Gödels T und seine Erweiterung T^+ definiert und zwischen den Termen aus T^+ haben wir die Reduktionsrelation $\to$ und dessen Abschlüsse $\to^*$ und $\doteq$ eingeführt. Damit haben wir nun eine Sprache, in der wir Programmextraktion aus Beweisen betreiben können. Was uns noch fehlt sind die Formeln und Prädikate, über die wir etwas beweisen wollen.

1.3 Herleitungsterme

Motivation 1.3.1. Im ersten Abschnitt dieses Kapitels haben wir das Kalkül des natürlichen Schließens eingeführt. Eine Herleitung einer Aussage A ist dort gegeben durch einen Herleitungsbaum. Dieser Herleitungsbaum ist ein zweidimensionales

Konstrukt und kann oft sehr groß werden, sodass es praktisch unmöglich ist, komplexere Beweise als Baum darzustellen. Die Darstellung eines Beweises als Beweisterm wird weniger Platz in Anspruch nehmen und ist für den Computer auch besser geeignet, da es sich um ein eindimensionales Konstrukt handelt. Ein Mensch hat mit Herleitungstermen wahrscheinlich etwas mehr Schwierigkeiten als mit Herleitungsbäumen. Sie sind dennoch für eine effiziente Darstellung von Beweisen wichtig.

Definition 1.3.2. So wie in Definition 1.1.1 definieren wir Herleitungsterme. Diese sind äquivalent zu den Herleitungsbäumen, werden jedoch nicht als zweidimensionales Konstrukt notiert, sodass die Notation kompakter ist. Das, was in Definition 1.1.1 die Annahmenmenge ist, entspricht der Menge der freien Variablen $FV(M)$ eines Herleitungsterms M.

- Jede Annahmenvariable u^A mit Typ A ist ein Herleitungsterm von A mit $FV(u^A) = u^A$.
- Ist M^B ein Herleitungsterm von B, dann ist $\lambda_{u^A} M^B$ ein Herleitungsterm von $A \to B$ mit $FV(\lambda_{u^A} M^B) = FV(M^B) \setminus \{u^A\}$
- Ist $M^{A\to B}$ ein Herleitungsterm von $A \to B$ und N^A ein Herleitungsterm von A, so ist $M^{A\to B} N^A$ ein Herleitungsterm von B und $FV(M^{A\to B} N^A) = FV(M^{A\to B}) \cup FV(N^A)$.
- Ist M^A ein Herleitungsterm von A und x eine Variable, die nicht frei in der Formel einer Variablen von $FV(M^A)$ ist, dann ist $\lambda_x M^A$ ein Herleitungsterm von $\forall_x A$ und $FV(\lambda_x M^A) = FV(M^A)$.
- Ist $M^{\forall_x A(x)}$ ein Herleitungsterm von $\forall_x A(x)$ und r ein Term, dann ist $M^{\forall_x A(x)} r$ ein Herleitungsterm für $A(r)$ und $FV(M^{\forall_x A(x)} r) = FV(M^{\forall_x A(x)})$

Bemerkung 1.3.3. Es ist leicht zu sehen, dass es zwischen den Termen aus Gödels T, wie in Definition 1.2.7, und den Herleitungstermen große Ähnlichkeiten gibt. Es fehlen lediglich die Regeln, die den Rekursionsoperatoren und den Konstruktoren entsprechen. Später werden wir noch induktiv definierte Prädikate einführen, welche uns dann auch diese Regeln liefern werden. Diese Ähnlichkeit wird in der Literatur unter dem Namen *Curry-Howard Korrespondenz* abgehandelt. Wir werden uns hier aber damit nicht aufhalten, da wir die Herleitungsterme nur aufgrund ihrer Kompaktheit verwenden.

1.4 Prädikate und Formeln

Definition 1.4.1. Mit X und $\vec{Y}$ bezeichnen wir verschiedene Prädikatenvariablen. Wir werden nun n-stellige Prädikatenformen, Klauselformen und Formelformen rekursiv definieren. Diese haben Parameter, die wir mit $\vec{Y}$ bezeichnen. Eine Formelform ohne Parameter, also wenn $\vec{Y}$ keine Einträge hat, heißt Formel und analog heißt eine Prädikatenform ohne Parameter einfach Prädikat.

- Ist Y_l eine n-stellige Prädikatenvariable und sind $\vec{r}$ genau n Terme, dann ist $Y_l \vec{r}$ eine Formelform mit Parameter $\vec{Y}$.
- Ist A eine Formel und B eine Formelform mit Parameter $\vec{Y}$ sowie x eine Variable, dann sind $A \to B$ und $\forall_x B$ Formelformen mit Parameter $\vec{Y}$.

- Ist C eine Formelform mit Parameter $\vec{Y}$ und sind $\vec{x}$ genau n Variablen, dann ist $\{\vec{x}|C\}$ eine n-stellige Prädikatenform mit Parameter $\vec{Y}$.
- Ist P eine n-stellige Prädikatenform mit Parameter $\vec{Y}$ und sind $\vec{r}$ genau n Terme, dann ist $P\vec{r}$ eine Formelform mit Parameter $\vec{Y}$.
- Sind $\vec{A}$ Formelformen mit Parameter $\vec{Y}$ und sind $\vec{B}_0,\dots,\vec{B_{n-1}}$ Formeln, dann ist $\forall_{\vec{x}}(\vec{A}\to(\forall_{\vec{y}_i}(\vec{B}_i\to X\vec{s}_i))_{i<n}\to X\vec{t})$ eine Klauselform über X mit Parameter $\vec{Y}$.
- Sind $K_0,\dots,K_{n-1}$ Klauselformen mit $n\geq 1$ über X mit Parameter $\vec{Y}$, dann ist auch $\mu_X(K_0,\dots,K_{n-1})$ eine Prädikatenform mit Parameter $\vec{Y}$.

Bei einer Klauselform $\forall_{\vec{x}}(\vec{A}\to(\forall_{\vec{y}_i}(\vec{B}_i\to X\vec{s}_i)_{i<n}\to X\vec{t}))$ heißen die $\vec{A}$ Parameterprämissen und die $\forall_{\vec{y}_i}(\vec{B}_i\to X\vec{s}_i)$ nennen wir Rekursionsprämissen. Ein Prädikat der Form $I=\mu_X(K_0,\dots,K_{n-1})$ heißt induktiv definiertes Prädikat. Eine Prädikatenform der Gestalt $\{\vec{x}|C\}$ heißt Komprehensionsterm und wir identifizieren $\{\vec{x}|C(\vec{x})\}\vec{r}$ mit $C(\vec{r})$.

Definition 1.4.2. Zu jedem induktiv definierten Prädikat $I=\mu_X(K_0,\dots,K_{k-1})$ definieren wir Einführungs- und Eliminationsaxiome. Dazu sei für $i<k$

$$K_i(X)=\forall_{\vec{x}}(\vec{A}\to(\forall_{\vec{y}_j}(\vec{B}_j\to X\vec{s}_j))_{j<n}\to X\vec{t}).$$

Das dazu korrespondierende Einführungsaxiom ist

$$I_i^+ : K_i(I)=\forall_{\vec{x}}(\vec{A}\to(\forall_{\vec{y}_j}(\vec{B}_j\to I\vec{s}_j))_{j<n}\to I\vec{t}).$$

Das Eliminationsaxiom zu einem Prädikat P ist gegeben durch

$$I^-(P):\forall_{\vec{x}}(I\vec{x}\to(K_i(I,P))_{i<k}\to P\vec{x}),$$

dabei ist

$$K_i(I,P):=\forall_{\vec{x}}(\vec{A}\to(\forall_{\vec{y}_j}(\vec{B}_j\to I\vec{s}_j))_{j<n}\to(\forall_{\vec{y}_j}(\vec{B}_j\to P\vec{s}_j))_{j<n}\to P\vec{t}).$$

Man beachte, dass wir die gebundenen Variablen $\vec{x}$ so wählen müssen, dass es nicht zu einer Kollision mit freien Variablen in P kommt. Gegebenenfalls muss man die gebundenen Variablen erst umbenennen, bevor man ein konkretes P einsetzt.

Beispiel 1.4.3. Wir definieren die Leibnizgleichheit auf Termen von Typ ρ als das Prädikat

$$\mathrm{Eq}(\rho):=\mu_X(\forall_{x^\rho}Xxx).$$

Das gibt uns das einzige Einführungsaxiom $\forall_{x^\rho}\mathrm{Eq}(\rho)xx$. Als Eliminationsaxiom haben wir

$$\forall_{x,y}(\mathrm{Eq}(\rho)xy\to\forall_x Pxx\to Pxy).$$

Man kann leicht überprüfen, dass $\mathrm{Eq}(\rho)$ reflexiv, symmetrisch und transitiv ist. Außerdem gilt die für eine Gleichheit charakterisierende Eigenschaft:

Lemma 1.4.4. Für jede Aussage $A(x)$ gilt $\forall_{x,y}(\mathsf{Eq}xy \to A(x) \to A(y))$.

Beweis. Wir setzen in das Eliminationsaxiom das Prädikat $P = \{x, y | A(x) \to A(y)\}$ ein. Das gibt uns

$$\forall_{x,y}(\mathsf{Eq}xy \to \forall_x(A(x) \to A(x)) \to A(x) \to A(y)).$$

Eq ist eine Kurzschreibweise für $\mathsf{Eq}(\rho)$, wenn ρ klar oder egal ist.

Da $\forall_x(A(x) \to A(x))$ immer gilt, folgt $\forall_{x,y}(\mathsf{Eq}xy \to A(x) \to A(y))$ □

Definition 1.4.5. Die Theorie der berechenbaren Funktionale **TCF** ist eine Formelmenge mit den Termen aus T^+ und den logischen Verknüpfungen und Regeln aus Definition 1.1.1, also nur für $\forall$ und $\to$. Die Prädikatsymbole sind die Prädikate aus Definition 1.4.1, wobei wir jedoch nur die induktiv definierten Prädikate zulassen, deren erste Klausel keine Rekursionsprämissen hat und deren Klauseln nur wieder zugelassene Prädikate verwenden. Die Axiome dieser Theorie sind die entsprechenden Einführungs- und Eliminationsaxiome aus Definition 1.4.2 und die Gleichheitsaxiome bezüglich $\doteq$.

Definition 1.4.6. Wir definieren das Falsum mit Hilfe der Leibnizgleichheit aus Beispiel 1.4.3 durch $\mathsf{F} := \mathsf{Eq}\,\mathrm{ff}^{\mathbb{B}}\mathrm{tt}^{\mathbb{B}}$.

Satz 1.4.7. In der Theorie der berechenbaren Funktionale gilt das Ex-falso-quodlibet für jede Formel A. Das heißt kurzgeschrieben $\mathbf{TCF} \vdash \mathsf{F} \to A$.

Man beachte, dass A keine Prädikatenparameter haben darf.

Beweis. Wir beweisen zunächst, wenn x^ρ, y^ρ Terme vom selben Typ sind, so gilt $\mathbf{TCF} \vdash \mathsf{F} \to \mathsf{Eq}xy$: Nach Einführungsaxiom für Eq gilt $\mathsf{Eq}(\mathscr{C}^\rho_{\mathbb{B}}\mathrm{tt}xy)(\mathscr{C}^\rho_{\mathbb{B}}\mathrm{tt}xy)$. Damit gilt nach Lemma 1.4.4 $\mathsf{Eq}(\mathscr{C}^\rho_{\mathbb{B}}\mathrm{ff}xy)(\mathscr{C}^\rho_{\mathbb{B}}\mathrm{tt}xy)$ also $\mathsf{Eq}xy$.
Nun beweisen wir die Aussage $\mathbf{TCF} \vdash \mathsf{F} \to A$ durch Induktion über A. Für den Fall, dass $A = I\vec{s}$ für ein induktiv definiertes Prädikat I ist, nehmen wir die Klausel K_0 von I. Diese hat nach der Forderung in der Definition von **TCF** keine Rekursionsprämissen und damit die Form $\forall_x(\vec{B} \to I\vec{t})$. Nach Induktionshypothese folgen aus F alle $\vec{B}$ und damit $I\vec{t}$. Zusammen mit Lemma 1.4.4 und der gerade gezeigten Gleichheit aller Terme vom selben Typ erhalten wir also $I\vec{s}$. Ist $A = B \to C$ oder $A = \forall_x C$, gilt nach Induktionsvoraussetzung $\mathbf{TCF} \vdash \mathsf{F} \to C$ und damit sicher $\mathbf{TCF} \vdash \mathsf{F} \to B \to C$ und weil in **TCF** keine Axiome mit freien Variablen sind, folgt auch $\mathsf{F} \to \forall_x C$. □

Notation 1.4.8. Durch die Leibnizgleichheit können wir nun boolesche Terme auch mit Aussagen identifizieren: Ist $s^{\mathbb{B}}$ ein boolescher Term, dann fassen wir ihn auch als Formel $\mathsf{Eq}(s, \mathrm{tt})$ auf.

Beispiel 1.4.9. Wir haben in **TCF** nur die logische Verknüpfung $\to$ und den Quantor $\forall$. Der Grund dafür ist, dass die logischen Verknüpfungen $\wedge$ und $\vee$ sowie der Existenzquantor $\exists$ als induktiv definierte Prädikate eingeführt werden.
Den Existenzquantor definieren wir durch

$$Ex(Y) := \mu_X(\forall_{x^\rho}(Yx \to X)).$$

Das Einführungsaxiom ist dann

$$\exists^+ : \forall_x(A \to \exists_x A),$$

wobei wir $\exists_x A := Ex(\{x^\rho | A\})$ setzen. Das Eliminationsaxiom ist

$$\exists^- : \exists_x A \to \forall_x(A \to P) \to P.$$

Man vergleiche dies mit den Regeln für $\exists$ aus Definition 1.1.5. Es ist leicht zu zeigen, dass diese Regeln jeweils äquivalent zu den Axiomen sind.
Die Konjunktion zweier Aussagen ist gegeben durch

$$And(Y,Z) := \mu_X(Y \to Z \to X).$$

Schreiben wir $A \wedge B := And(\{|A\},\{|B\})$, so haben wir die Axiome

$$\wedge^+ : A \to B \to A \wedge B$$

und

$$\wedge^- : A \wedge B \to (A \to B \to P) \to P.$$

Dies erklärt auch, warum man in Definition 1.1.5 die Regel $\wedge^-$ nicht durch die zwei Regeln oder Axiome $A \wedge B \to A$ und $A \wedge B \to B$ ersetzt. Diese sind, wie man sich überlegen kann, äquivalent zu $\wedge^-$, aber im Sinne der induktiv definierten Prädikaten haben wir uns für diese eine Regel entschieden.
Für die Disjunktion zweier Aussagen haben wir ein induktiv definiertes Prädikat mit zwei Klauseln:

$$Or(Y,Z) := \mu_X(Y \to X, Z \to X),$$

was zu den beiden Einführungsaxiomen

$$\vee_0^+ : A \to A \vee B \qquad \vee_1^+ : B \to A \vee B$$

führt, wobei $A \vee B := Or(\{|A\},\{|B\})$ ist. Das Eliminationsaxiom ist

$$\vee^- : A \vee B \to (A \to P) \to (B \to P) \to P.$$

1.5 Totalität

Motivation 1.5.1. Wir haben nun bereits einige Algebren definiert. Insbesondere die Algebra der natürlichen Zahlen $\mathbb{N} := \mu_\xi(\xi, \xi \to \xi)$. Aus den Anfängervorlesungen ist sicherlich noch bekannt, dass man Aussagen über natürliche Zahlen häufig mit Induktion zeigen kann oder sogar muss. In **TCF** gibt es solch ein Induktionsaxiom nicht für beliebige Terme. Das liegt daran, dass nicht gesagt ist, dass zum Beispiel jedes Element der Algebra $\mathbb{N}$ auch die Form $S \ldots S0$ hat. Etwas allgemeiner formuliert ist ein Term vom Typ einer Algebra nicht unbedingt gleich einem Term, der aus den Konstruktoren dieser Algebra besteht. In dieser Arbeit beschäftigen wir uns nicht mit Modellen zur Theorie der berechenbaren Funktionalen. Es sei hierzu aber auf [7] verwiesen. Dort wird ein Modell diskutiert, in dem nicht jeder Term jene Form hat. Da man jedoch trotzdem Induktion über Objekte einer Algebra machen will, führt man die Totalitätsprädikate ein und trifft dann nur Aussagen über totale Objekte. Wir geben dafür zunächst ein Beispiel an:

Beispiel 1.5.2. Bei den natürlichen Zahlen ist es sehr kanonisch zu fordern, dass genau die Zahlen $0,1,2,\ldots$ total sein sollen; oder formal ausgedrückt: 0 soll total sein, also $\mathsf{T}_\mathbb{N}0$ und ist n total, so soll auch Sn total sein, also $\forall_n(\mathsf{T}_\mathbb{N}n \to \mathsf{T}_\mathbb{N}Sn)$. Damit ist $\mathsf{T}_\mathbb{N} := \mu_X(X0, \forall_n(Xn \to XSn))$ und wir haben als Eliminationsaxiom

$$\mathsf{T}_\mathbb{N}^- : \forall_n(\mathsf{T}_\mathbb{N}n \to P0 \to \forall_n(\mathsf{T}_\mathbb{N}n \to Pn \to PSn) \to Pn).$$

Man erkennt, dass dieses Axiom genau die Form des Induktionsaxioms hat.
Bei den Listen mit Parameter α ist es etwas komplexer. Eine Möglichkeit wäre natürlich nur zu fordern, dass die leere Liste total sein soll, also $\mathbf{T}_{\mathbb{L}(\alpha)}\mathrm{nil}$, und wenn eine Liste total ist, dann soll auch die Liste, die durch Anhängen eines Elements aus der ursprünglichen Liste entsteht, wieder total sein, also $\forall_{x,l}(\mathbf{T}_{\mathbb{L}(\alpha)} l \to \mathbf{T}_{\mathbb{L}(\alpha)} x :: l)$. Hier fordern wir also nicht, dass x ein totales Objekt sein soll. Diese Art der Totalität nennt sich strukturelle Totalität. In unserem Fall ist nur die Struktur der Liste aber nicht ihr Inhalt total. Im Gegensatz dazu könnte man auch noch fordern, dass das Objekt in der Liste total sein soll, was uns dann die (gesamte) Totalität $\mathbf{G}_{\mathbb{L}(\alpha)}$ liefert mit den Klauseln $\mathbf{G}_{\mathbb{L}(\alpha)}\mathrm{nil}$ und $\forall_{x,l}(\mathbf{G}_\alpha x \to \mathbf{G}_{\mathbb{L}(\alpha)} l \to \mathbf{G}_{\mathbb{L}(\alpha)} x :: l)$. Zu bemerken ist, dass α ein beliebiger Typ und nicht unbedingt eine Algebra sein muss. Wir geben daher nun die Totalität und die strukturelle Totalität für jeden Typen an.

Definition 1.5.3. Wir definieren das Totalitätsprädikat $\mathbf{G}_\alpha$ und das strukturelle Totalitätsprädikat $\mathbf{T}_\alpha$ für jeden Typ α rekursiv über den Aufbau des Typen.
Ist $\iota = \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$ eine Algebra mit $\kappa_i = \vec{\rho} \to (\vec{\sigma}_j \to \xi)_{j<n} \to \xi$, dann definieren wir

Die Notation $\mathbf{G}_{\vec{\rho}}\vec{x}$ ist dabei eine abkürzende Schreibweise für $(\mathbf{G}_{\rho_l} x_l)_l$. Die analoge Schreibweise wenden wir auch bei $\mathbf{T}$ an.

$$K_i := \forall_{\vec{x}\vec{y}}(\mathbf{G}_{\vec{\rho}}\vec{x} \to (\forall_{\vec{z}_j}(\mathbf{G}_{\vec{\sigma}_j}\vec{z}_j \to X y_j \vec{z}_j))_{j<n} \to X C_i \vec{x}\vec{y})$$

und

$$\mathbf{G}_\iota := \mu_X(K_0, \ldots K_{k-1}).$$

Für die strukturelle Totalität definieren wir

$$K'_i := \forall_{\vec{x}\vec{y}}((\forall_{\vec{z}_j}(\mathbf{T}_{\vec{\sigma}_j}\vec{z}_j \to X y_j \vec{z}_j))_{j<n} \to X C_i \vec{x}\vec{y})$$

und

$$\mathbf{T}_\iota := \mu_X(K'_0, \ldots K'_{k-1}).$$

Bei einem Pfeiltypen $\rho \to \sigma$ haben wir jeweils nur eine Klausel:

$$\mathbf{G}_{\rho\to\sigma} := \mu_X(\forall_f(\forall_x(\mathbf{G}_\rho x \to \mathbf{G}_\rho(fx)) \to Xf))$$
$$\mathbf{T}_{\rho\to\sigma} := \mu_X(\forall_f(\forall_x(\mathbf{T}_\rho x \to \mathbf{T}_\rho(fx)) \to Xf))$$

Bemerkung 1.5.4. Beweist man nun Aussagen über totale Objekte eines Typs, kann man dies durch Induktion tun, was eben dem Eliminationsaxiom für die Totalität entspricht. Dies wird später bei der Beweisführung mit dem Computer noch eine Rolle spielen. Dort wird von einer Variablen standardmäßig angenommen, dass sie total ist, wie wir noch sehen werden.
Noch zu beachten ist, dass im Allgemeinen weder $\mathbf{G}_\rho \subseteq \mathbf{T}_\rho$ noch $\mathbf{T}_\rho \subseteq \mathbf{G}_\rho$ gilt. Um dies formal zu beweisen, bräuchte man wieder Gegenmodelle. Dass Ersteres nicht gelten muss, sieht man anhand des Listentyps aus Beispiel 1.5.2. Ein Gegenbeispiel für die zweite Aussage ist in einigen Modellen die Programmkonstante $\mathtt{head}^{\mathbb{L}(\mathbb{U})\to\mathbb{U}}$ mit den Berechnungsregeln $\mathtt{head}\ \mathrm{nil} = \mathbf{u}$ und $\mathtt{head}\ x :: l = x$. Diese ist dort total aber nicht strukturell total.
Zu beachten ist noch, dass nicht für jeden Typ das Totalitätsprädikat ein Prädikatensymbol in **TCF** ist. Dies wäre im einzelnen Fall immer zu überprüfen. Für finitäre Algebren ist dies aber beispielsweise immer der Fall.

Motivation 1.5.5. Als eine Anwendung der Totalität, wollen wir nun zeigen, dass die entscheidbare Gleichheit von totalen Termen einer finitären Algebra die Leibnizgleichheit impliziert. Dafür brauchen wir noch zwei Lemmata als Vorbereitung.

Notation 1.5.6. Damit wir in Aussagen und Termen viele Klammern einsparen, verwenden wir die Punktnotation. Ein Punkt in einer Aussage oder einem Term bedeutet, dass die Bindung zwischen der rechten und der linken Seite des Punktes maximal schwach ist. Zum Beispiel sind $\forall_x.A \to B \to C$ als $\forall_x(A \to B \to C)$ und $(A \to \forall_x.B \to C) \to D$ als $(A \to \forall_x(B \to C)) \to D$ zu verstehen.

Lemma 1.5.7. Es bezeichne andb die in Definition 1.2.18 gegebene Programmkonstante auf der booleschen Algebra, dann gilt $\forall_{a^{\mathbb{B}},b^{\mathbb{B}}}.\mathsf{G}_{\mathbb{B}}a \to \mathsf{G}_{\mathbb{B}}b \to a \ \mathtt{andb}\ b \to a \wedge b$ und $\forall_{a^{\mathbb{B}},b^{\mathbb{B}}}.\mathsf{G}_{\mathbb{B}}a \to \mathsf{G}_{\mathbb{B}}b \to \mathsf{G}_{\mathbb{B}}\ (a\ \mathtt{andb}\ b)$.

Beiweis. Die Totalität auf $\mathbb{B}$ hat die beiden Klauseln $X\mathsf{tt}$ und $X\mathsf{ff}$. Das gibt als Eliminationsaxiom

Man erinnere sich, dass ein boolescher Term a durch $Eq(a,\mathsf{tt})$ mit einer Aussage identifiziert wird.

$$\forall_a.\mathsf{G}_{\mathbb{B}}a \to P\mathsf{tt} \to P\mathsf{ff} \to Pa.$$

Wir setzen $P = \{a | Eq(a,\mathsf{tt}) \vee Eq(a,\mathsf{ff})\}$. Dann gilt sicher $P\mathsf{tt}$ und $P\mathsf{ff}$ und wir erhalten aus $\mathsf{G}_{\mathbb{B}}a$ die Aussage $Eq(a,\mathsf{tt}) \vee Eq(a,\mathsf{ff})$ und analog erhalten wir die Aussage $Eq(b,\mathsf{tt}) \vee Eq(b,\mathsf{ff})$. Nun sind nur noch die vier daraus resultierenden Fälle zu überprüfen. Gilt $Eq(a,\mathsf{tt})$ und $Eq(b,\mathsf{tt})$, folgt sofort nach Definition $a \wedge b$, und es folgt $Eq(a\ \mathtt{andb}\ b,\mathsf{tt})$, weil $(\mathsf{tt}\ \mathtt{andb}\ \mathsf{tt}) := \mathsf{tt}$ ist, und damit auch $\mathsf{G}_{\mathbb{B}}(a\ \mathtt{andb}\ b)$ nach dem ersten Einführungsaxiom der Totalität. In allen anderen Fällen gilt $Eq(a\ \mathtt{andb}\ b,\mathsf{ff})$ und es folgt die erste Aussage aus dem Ex-falso-quodlibet, welches in Satz 1.4.7 gezeigt wurde. Die zweite Aussage folgt, nach dem zweiten Einführungsaxiom der Totalität. □

Bemerkung 1.5.8. So wie andb definiert wurde, lassen sich auch die boolesche Disjunktion orb und die boolesche Implikation impb mit den kanonischen Berechnungsregeln definieren. Die analogen Aussagen gelten dann leicht ersichtlich auch für diese Programmkonstanten.

Lemma 1.5.9. Es sei ι eine finitäre Algebra, dann gilt $\forall_{x^\iota,y^\iota}.\mathsf{G}_\iota x \to \mathsf{G}_\iota y \to \mathsf{G}_{\mathbb{B}}(x = y)$.

Beweis. Da $\iota =: \mu_\xi(\kappa_0,\ldots,\kappa_{k-1})$ eine finitäre Algebra ist, haben wir für jedes $i < k$ die Darstellung

$$\kappa_i = \vec{\rho}_i \to (\xi)_{j<n_i} \to \xi.$$

Wir machen nun Induktion über den Aufbau von ι. Das bedeutet, wir können annehmen, dass die Aussage schon für alle finitären Algebren $(\vec{\rho}_i)_{i<k}$ gilt. Das Eliminationsaxiom für die Totalität auf ι ist gegeben durch

$$\forall_x.\mathsf{G}_\iota x \to \left(\forall_{\vec{x}_i\vec{y}_i}.\mathsf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathsf{G}_\iota y_{ij})_{j<n_i} \to (Py_{ij})_{j<n_i} \to PC_i\vec{x}_i\vec{y}_i\right)_{i<k} \to Px.$$

Setzen wir nun für P das Prädikat $\{w | \forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = w)\}$ ein, so erhalten wir

Das Wichtige an diesem und auch dem nächsten Beweis ist eigentlich nur, wie man jeweils das Prädikat bei dem Eliminationsaxiom wählt. Alles andere ist nur Technik.

$$\begin{aligned}\forall_x.\mathsf{G}_\iota x \to (\forall_{\vec{x}_i\vec{y}_i}.\mathsf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathsf{G}_\iota y_{ij})_{j<n_i} \to (\forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = y_{ij}))_{j<n_i} \\ \to \forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = C_i\vec{x}_i\vec{y}_i))_{i<k} \to \forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = x).\end{aligned}$$

Es reicht daher für jedes $i < k$ die Aussage

$$\forall_{\vec{x}_i\vec{y}_i}.\mathsf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathsf{G}_\iota y_{ij})_{j<n_i} \to (\forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}z = y_{ij})_{j<n_i} \to \forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = C_i\vec{x}_i\vec{y}_i)$$

zu zeigen. Dazu sei also $l < k$ sowie $\vec{u}_l$ und $\vec{v}_l$ fixiert, weiter gelte $\mathsf{G}_{\vec{\rho}_l}\vec{u}_l$, $(\mathsf{G}_\iota v_{lj})_{j<n_l}$ und $(\forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}z = v_{lj})_{j<n_l}$. Dann haben wir

$$\forall_z.\mathsf{G}_\iota z \to \mathsf{G}_{\mathbb{B}}(z = C_l\vec{u}_l\vec{v}_l)$$

zu zeigen. Setzen wir dafür im Eliminationsaxiom $P = \{x|\mathbf{G}_{\mathbb{B}}(x = C_l\vec{u}_l\vec{v}_l)\}$. Dadurch erhalten wir

$$\forall_x.\mathbf{G}_\iota x \to (\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (\mathbf{G}_{\mathbb{B}}(y_{ij} = C_l\vec{u}_l\vec{v}_l))_{j<n_i} \\ \to \mathbf{G}_{\mathbb{B}}(C_i\vec{x}_i\vec{y}_i = C_l\vec{u}_l\vec{v}_l))_{i<k} \to \mathbf{G}_{\mathbb{B}}(x = C_l\vec{u}_l\vec{v}_l).$$

und wollen also die Aussage

$$\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (\mathbf{G}_{\mathbb{B}}(y_{ij} = C_l\vec{u}_l\vec{v}_l))_{j<n_i} \to \mathbf{G}_{\mathbb{B}}(C_i\vec{x}_i\vec{y}_i = C_l\vec{u}_l\vec{v}_l)$$

für jedes $i < k$ zeigen. Falls $i \neq l$ ist, ist $(C_i\vec{x}_i\vec{y}_i = C_l\vec{u}_l\vec{v}_l) := \mathrm{ff}$ und damit gilt sicher $\mathbf{G}_{\mathbb{B}}(C_i\vec{x}_i\vec{y}_i = C_l\vec{u}_l\vec{v}_l)$ nach dem zweiten Einführungsaxiom vom $\mathbf{G}_{\mathbb{B}}$. Zu zeigen ist also nur noch die Aussage

$$\forall_{\vec{x}_l\vec{y}_l}.\mathbf{G}_{\vec{\rho}_l}\vec{x}_l \to (\mathbf{G}_\iota y_{lj})_{j<n_l} \to (\mathbf{G}_{\mathbb{B}}(y_{lj} = C_l\vec{u}_l\vec{v}_l))_{j<n_l} \to \mathbf{G}_{\mathbb{B}}(C_l\vec{x}_l\vec{y}_l = C_l\vec{u}_l\vec{v}_l).$$

Seien dazu $\vec{x}_l$, $\vec{y}_l$ fixiert und wir nehmen die Aussagen $\mathbf{G}_{\vec{\rho}_l}\vec{x}_l$, $(\mathbf{G}_\iota y_{lj})_{j<n_l}$ und $(\mathbf{G}_{\mathbb{B}}(y_{lj} = C_l\vec{u}_l\vec{v}_l))_{j<n_l}$ an. Eine Berechnungsregel der entscheidbaren Gleichheit ist

$$(C_l\vec{x}_l\vec{y}_l = C_l\vec{u}_l\vec{v}_l) := (\vec{x}_l = \vec{u}_l \text{ andb } \vec{y}_l = \vec{v}_l).$$

Nach Induktionshypothese gilt $\mathbf{G}_{\mathbb{B}}(\vec{x}_l = \vec{u}_l)$, weil nach Voraussetzung $\mathbf{G}_{\vec{\rho}_l}\vec{x}_l$ und $\mathbf{G}_{\vec{\rho}_l}\vec{u}_l$ gilt. Weiter haben wir, dass die Aussagen $(\forall_z.\mathbf{G}_\iota z \to \mathbf{G}_{\mathbb{B}}(z = v_{lj}))_{j<n_l}$ und $(\mathbf{G}_\iota y_{lj})_{j<n_l}$ gelten. Damit folgen die Aussagen $\mathbf{G}_{\mathbb{B}}(\vec{y}_l = \vec{v}_l)$ und mit Lemma 1.5.7 folgt dann $\mathbf{G}_{\mathbb{B}}(\vec{x}_l = \vec{u}_l \text{ andb } \vec{y}_l = \vec{v}_l)$, was zu zeigen war. □

Satz 1.5.10. Ist ι eine finitäre Algebra, dann gilt

$$\forall_{x^\iota,y^\iota}.\mathbf{G}_\iota x \to \mathbf{G}_\iota y \to x = y \to Eq(x, y).$$

Beweis. Auch hier haben wir für die finitäre Algebra $\iota =: \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$, dass jede Klausel κ_i die Form

$$\kappa_i = \vec{\rho}_i \to (\xi)_{j<n_i} \to \xi$$

hat und wir machen auch hier wieder Induktion über den Aufbau von ι. Das heißt, die Aussage gelte bereits für alle $\vec{\rho}_i$ anstelle von ι. Außerdem ist das Eliminationsaxiom der Totalität auf ι genauso wie im Beweis des vorherigen Lemmas gegeben durch

$$\forall_x.\mathbf{G}_\iota x \to \left(\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (Py_{ij})_{j<n_i} \to PC_i\vec{x}_i\vec{y}_i\right)_{i<k} \to Px.$$

Setzt man nun für P das Prädikat $\{x|\forall_z(\mathbf{G}_\iota z \to x = z \to Eq(x, z))\}$ ein, so erhält man

$$\forall_x.\mathbf{G}_\iota x \to (\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (\forall_z.\mathbf{G}_\iota z \to y_{ij} = z \to Eq(y_{ij}, z))_{j<n_i} \\ \to \forall_z.\mathbf{G}_\iota z \to C_i\vec{x}_i\vec{y}_i = z \to Eq(C_i\vec{x}_i\vec{y}_i, z))_{i<k} \to \forall_z.\mathbf{G}_\iota z \to x = z \to Eq(x, z).$$

Es reicht also, für jedes $i < k$ die Aussage

$$\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (\forall_z.\mathbf{G}_\iota z \to y_{ij} = z \to Eq(y_{ij}, z))_{j<n_i} \\ \to \forall_z.\mathbf{G}_\iota z \to C_i\vec{x}_i\vec{y}_i = z \to Eq(C_i\vec{x}_i\vec{y}_i, z)$$

zu zeigen. Wir nehmen daher $l < k$ und fixieren $\vec{u}_l$ und $\vec{v}_l$. Weiter gelte $\mathbf{G}_{\vec{\rho}_l}\vec{u}_l$, $(\mathbf{G}_\iota v_{lj})_{j<n_l}$ und $(\forall_z.\mathbf{G}_\iota z \to v_{lj} = z \to Eq(v_{lj}, z))_{j<n_l}$. Zu zeigen ist dann die Aussage

$$\forall_z.\mathbf{G}_\iota z \to C_l\vec{u}_l\vec{v}_l = z \to Eq(C_l\vec{u}_l\vec{v}_l, z).$$

Dafür setzen wir $\{z|C_l\vec{u}_l\vec{v}_l = z \to Eq(C_l\vec{u}_l\vec{v}_l, z)\}$ in das Eliminationsaxiom für P ein. Das gibt

$$\forall_x.\mathbf{G}_\iota x \to (\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (C_l\vec{u}_l\vec{v}_l = y_{ij} \to Eq(C_l\vec{u}_l\vec{v}_l, y_{ij}))_{j<n_i}$$
$$\to (C_l\vec{u}_l\vec{v}_l = C_i\vec{x}_i\vec{y}_i) \to Eq(C_l\vec{u}_l\vec{v}_l, C_i\vec{x}_i\vec{y}_i))_{i<k} \to C_l\vec{u}_l\vec{v}_l = z \to Eq(C_l\vec{u}_l\vec{v}_l, z).$$

Wir sind also fertig, wenn wir für jedes $i < k$ die Aussage

$$\forall_{\vec{x}_i\vec{y}_i}.\mathbf{G}_{\vec{\rho}_i}\vec{x}_i \to (\mathbf{G}_\iota y_{ij})_{j<n_i} \to (C_l\vec{u}_l\vec{v}_l = y_{ij} \to Eq(C_l\vec{u}_l\vec{v}_l, y_{ij}))_{j<n_i}$$
$$\to (C_l\vec{u}_l\vec{v}_l = C_i\vec{x}_i\vec{y}_i) \to Eq(C_l\vec{u}_l\vec{v}_l, C_i\vec{x}_i\vec{y}_i)$$

zeigen. Ist $i \neq l$, so haben wir die Berechnungsregel $(C_l\vec{u}_l\vec{v}_l = C_i\vec{x}_i\vec{y}_i) := \mathsf{ff}$ und die Aussage gilt wegen Satz 1.4.7. Es muss also nur noch

$$\forall_{\vec{x}_l\vec{y}_l}.\mathbf{G}_{\vec{\rho}_l}\vec{x}_l \to (\mathbf{G}_\iota y_{lj})_{j<n_l} \to (C_l\vec{u}_l\vec{v}_l = y_{lj} \to Eq(C_l\vec{u}_l\vec{v}_l, y_{lj}))_{j<n_l}$$
$$\to (C_l\vec{u}_l\vec{v}_l = C_l\vec{x}_l\vec{y}_l) \to Eq(C_l\vec{u}_l\vec{v}_l, C_l\vec{x}_l\vec{y}_l)$$

gezeigt werden. Hierzu sei $\vec{x}_l$ und $\vec{y}_l$ gegeben und weiter gelte $\mathbf{G}_{\vec{\rho}_l}\vec{x}_l$, $(\mathbf{G}_\iota y_{lj})_{j<n_l}$, $(C_l\vec{u}_l\vec{v}_l = y_{lj} \to Eq(C_l\vec{u}_l\vec{v}_l, y_{lj}))_{j<n_l}$ und $(C_l\vec{u}_l\vec{v}_l = C_l\vec{x}_l\vec{y}_l)$. Zu zeigen ist nun noch $Eq(C_l\vec{u}_l\vec{v}_l, C_l\vec{x}_l\vec{y}_l)$.
Aus $C_l\vec{u}_l\vec{v}_l = C_l\vec{x}_l\vec{y}_l$ folgt nach Berechnungsregel und Lemma 1.5.7 $\vec{u}_l = \vec{x}_l$ und $\vec{v}_l = \vec{y}_l$. Nach Induktionshypothese und, weil gegeben ist, dass die $\vec{u}_l$ und $\vec{x}_l$ total sind, folgt $Eq(\vec{u}_l, \vec{x}_l)$. Aus den gegebenen Aussagen $(\forall_z.\mathbf{G}_\iota z \to v_{lj} = z \to Eq(v_{lj}, z))_{j<n_l}$ und $(\mathbf{G}_\iota y_{lj})_{j<n_l}$ folgt zusammen mit $\vec{v}_l = \vec{y}_l$ dann $Eq(\vec{v}_l, \vec{y}_l)$ und daraus folgt zusammen mit $Eq(\vec{u}_l, \vec{x}_l)$ die gewünschte Aussage $Eq(C_l\vec{u}_l\vec{v}_l, C_l\vec{x}_l\vec{y}_l)$. □

1.6 Dekorationen

Motivation 1.6.1. In diesem Abschnitt werden wir die Dekorationen von → und ∀ einführen. Dadurch wird an der Formel ersichtlich, welche Annahmen und Variablen rechnerisch in den Beweis eingehen. Dazu sei zunächst auf die Brouwer-Heyting-Kolmogorov Interpretation (BHK-Interpretation) von konstruktiven Beweisen verwiesen: Ein Beweis einer Aussage der Form $A \to B$ ist dabei eine Vorschrift, die aus jedem Beweis von A einen Beweis von B erstellt. Ebenso ist ein Beweis einer Aussage der Form $\forall_x A(x)$ eine Vorschrift, die zu jedem Term t vom entsprechenden Typ einen Beweis von $A(t)$ konstruiert.
Bei dem rechnerischen Gehalt eines Beweises, sollte es so ähnlich aussehen: Der rechnerische Gehalt eines Beweises von $A \to B$ sollte der rechnerische Gehalt eines Beweises von B in Abhängigkeit des rechnerischen Gehalts eines Beweises von A sein. Für den rechnerischen Gehalt einer Aussage $\forall_x A(x)$ erwarten wir eine Vorschrift, die zu jedem Term t mit dem selben Typ wie x, den rechnerischen Gehalt von $A(t)$ liefert.
Im Falle des rechnerischen Gehaltes ist dies aber doch noch etwas anderes als bei Beweisen. Es könnte sein, dass für den rechnerischen Gehalt der Aussage $A \to B$ der rechnerische Gehalt von A gar nicht notwendig ist. Wäre für einen Beweis von $A \to B$ der Beweis von A nicht notwendig, könnte man einfach nur B beweisen. Es kann jedoch sein, dass die Aussage A rechnerisch irrelevant ist, aber ihre Gültigkeit schon gebraucht wird. Das ist zum Beispiel der Fall, wenn A eine Aussage über die Gleichheit zweier Objekte ist, wie wir noch sehen werden. Aus diesem Grund werden wir nun die Dekoration $\to^{nc}$ von → einführen. Die Aussage $A \to^{nc} B$ oder in Worten „A impliziert nicht-rechnerisch B “, bedeutet dann, dass der rechnerische Gehalt von

A für den rechnerischen Gehalt von $A \to B$ irrelevant ist. Analog bedeutet $\forall_x^{nc} A$ oder in Worten „Für alle nicht-rechnerischen x gilt A“, dass x nicht in den rechnerischen Gehalt von $\forall_x A$ eingeht.

Definition 1.6.2. Wir führen die Dekoration der Implikation durch $\to^{nc}$ und die Dekoration des Allquantors durch $\forall^{nc}$ ein. Die dekorierten Operatoren verhalten sich syntaktisch genauso wie die nicht dekorierten. Das heißt, die Regeln für $\forall$ und $\to$ aus Definition 1.4.1 gelten auch für $\forall^{nc}$ und $\to^{nc}$. Außerdem haben sie die gleichen Eliminationsregeln. Für die Einfürungsregeln gilt jedoch eine Verschärfung:

$\to^{nc+}$**-Regel:** Ist M^B eine Herleitung von B, und gilt außerdem, dass $u^A \notin CA(M)$ ist, dann ist $(\lambda_{u^A} M^B)^{A \to^{nc} B}$ eine Herleitung von $A \to^{nc} B$ mit $FV(\lambda_u M) = FV(M) \setminus \{u\}$.

$\forall^{nc+}$**-Regel:** Ist M^A eine Herleitung und ist x eine Variable, die nicht frei in irgendeiner Formel von $FV(M)$ ist und gilt zudem noch $x \notin CV(M)$, dann ist $(\lambda_x M^A)^{\forall_x^{nc} A}$ eine Herleitung von $\forall_x^{nc} A$ mit $FV(\lambda_x M) = FV(M)$.

Dabei bezeichnen $CA(M)$ die Menge der rechnerischen Annahmen von M und $CV(M)$ die Menge der rechnerischen Variablen von M und sind wie folgt definiert:

Definition 1.6.3. Zu einem Beweis M einer Formel A definieren wir die Menge $CV(M)$ der rechnerischen Variablen von M und die Menge $CA(M)$ der rechnerischen Annahmen von M wie folgt: Ist M eine Herleitung einer nicht rechnerischen Formel (was erst im nächsten Abschnitt definiert wird), setzen wir $CV(M) = CA(M) = \varnothing$, ist M die Herleitung einer rechnerischen Formel, so definieren wir rekursiv:

$$
\begin{aligned}
CV\left(c^A\right) &:= \varnothing \quad \text{für ein Axiom } c^A \\
CV\left(u^A\right) &:= \varnothing \\
CV\left(\left(\lambda_{u^A} M^B\right)^{A \to B}\right) := CV\left(\left(\lambda_{u^A} M^B\right)^{A \to^{nc} B}\right) &:= CV(M) \\
CV\left(\left(M^{A \to B} N^A\right)^B\right) &:= CV(M) \cup CV(N) \\
CV\left(\left(M^{A \to^{nc} B} N^A\right)^B\right) &:= CV(M) \\
CV\left(\left(\lambda_x M^A\right)^{\forall_x A}\right) := CV\left(\left(\lambda_x M^A\right)^{\forall_x^{nc} A}\right) &:= CV(M) \setminus \{x\} \\
CV\left(\left(M^{\forall_x A(x)} r\right)^{A(r)}\right) &:= CV(M) \cup FV(r) \\
CV\left(\left(M^{\forall_x^{nc} A(x)} r\right)^{A(r)}\right) &:= CV(M)
\end{aligned}
$$

$$
\begin{aligned}
CA\left(c^A\right) &:= \varnothing \quad \text{für ein Axiom } c^A \\
CA\left(u^A\right) &:= u \\
CA\left(\left(\lambda_{u^A} M^B\right)^{A \to B}\right) := CA\left(\left(\lambda_{u^A} M^B\right)^{A \to^{nc} B}\right) &:= C(M^A) \setminus \{u\} \\
CA\left(\left(M^{A \to B} N^A\right)^B\right) &:= CA(M) \cup CA(N) \\
CA\left(\left(M^{A \to^{nc} B} N^A\right)^B\right) &:= CA(M) \\
CA\left(\left(\lambda_x M^A\right)^{\forall_x A}\right) := CA\left(\left(\lambda_x M^A\right)^{\forall_x^{nc} A}\right) &:= CA(M) \\
CA\left(\left(M^{\forall_x A(x)} r\right)^{A(r)}\right) := CA\left(\left(M^{\forall_x^{nc} A(x)} r\right)^{A(r)}\right) &:= CA(M)
\end{aligned}
$$

Notation 1.6.4. Will man $\to$ klar von $\to^{nc}$ unterscheiden, schreibt man für $\to$ auch $\to^{c}$ und analog schreibt man $\forall^{c}$ für $\forall$. Ist beides möglich, schreiben wir $\to^{c/nc}$ bzw. $\forall^{c/nc}$. Wenn es irrelevant ist, ob man $\to$ oder $\to^{nc}$ verwendet, schreiben wir immer $\to$.

Motivation 1.6.5. Wir haben nun auch dekorierte Versionen des Allquantors und Implikationspfeils. Erinnern wir uns an die vorherigen Abschnitte, in denen wir induktiv definierte Prädikate eingeführt haben, stellt sich nun die Frage, ob es auch sinnvoll ist, die dort verwendeten Allquantoren und Implikationspfeile zu dekorieren. Die Antwort darauf ist natürlich *ja*.

Definition 1.6.6. Zu den Definitionsregeln von Formeln und Prädikaten aus Definition 1.4.1 fügen wir noch folgende Regeln hinzu:

- Ist A eine Formel und B eine Formelform mit Parameter $\vec{Y}$ sowie x eine Variable, dann sind $A \to^{nc} B$ und $\forall_x^{nc} B$ Formelformen mit Parameter $\vec{Y}$.
- Sind $\vec{A}$ Formelformen mit Parameter $\vec{Y}$ und sind $\vec{B}_0, \dots, \vec{B}_{n-1}$ Formeln, dann ist $\forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} X\vec{s}_j\right)\right)_{j<n} \to^{c} X\vec{t}\right)$ eine Klauselform über X mit Parameter $\vec{Y}$.
- Sind $K_0, \dots, K_{k-1}$ Klauselformen, dann ist $\mu_X^{nc}(K_0, \dots, K_{k-1})$ ein Prädikat.

Man beachte, dass jede Rekursionsprämisse rechnerisch eingeht.

Ist $\mu_X(K_0, \dots, K_{k-1})$ ein induktiv definiertes Prädikat mit

$$K_i(X) := \forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} X\vec{s}_j\right)\right)_{j<n} \to^{c} X\vec{t}\right),$$

dann ist das i-te Einführungsaxiom gegeben durch

$$I_i^{+} : K_i(I) = \forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} I\vec{s}_j\right)\right)_{j<n} \to^{c} I\vec{t}\right)$$

und das Eliminationsaxiom ist gegeben durch

$$I^{-}(P) : \forall_{\vec{x}}^{nc}\left(I\vec{x} \to^{c} (K_i(I,P))_{i<k} \to^{c} P\vec{x}\right)$$

wobei

$$K_i(I,P) := \forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} I\vec{s}_j\right)\right)_{j<n} \to^{c} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} P\vec{s}_j\right)\right)_{j<n} \to^{c} P\vec{t}\right).$$

Bei einem nicht rechnerischen induktiv definierten Prädikat, also einem Prädikat der Form $I^{nc} := \mu_X^{nc}(K_0, \dots, K_{k-1})$, sind die Einführungsaxiome und das Eliminationsaxiom genauso wie bei rechnerischen induktiv definierten Prädikaten mit dem einzigen Unterschied, dass beim Eliminationsaxiom nur nicht rechnerische Prädikate für P eingesetzt werden dürfen. Dabei ist ein Prädikat nicht rechnerisch, wenn der Typ des Prädikats der Nulltyp ist nach Definition 1.7.1.

Beispiel 1.6.7. Durch die Dekorationen haben wir nun verschiedene Varianten der in Beispiel 1.4.9 angegebenen induktiv definierten Prädikate. Wir geben hier nun eine

Liste davon an.

$$
\begin{aligned}
Ex^d(Y) &:= \mu_X(\forall_x^c(Yx \to^c X))\\
Ex^l(Y) &:= \mu_X(\forall_x^c(Yx \to^{nc} X))\\
Ex^r(Y) &:= \mu_X(\forall_x^{nc}(Yx \to^c X))\\
Ex^u(Y) &:= \mu_X(\forall_x^{nc}(Yx \to^{nc} X))\\
Ex^{nc}(Y) &:= \mu_X^{nc}(\forall_x(Y \to X))\\
And^d(Y,Z) &:= \mu_X(Y \to^c Z \to^c X)\\
And^l(Y,Z) &:= \mu_X(Y \to^c Z \to^{nc} X)\\
And^r(Y,Z) &:= \mu_X(Y \to^{nc} Z \to^c X)\\
And^u(Y,Z) &:= \mu_X(Y \to^{nc} Z \to^{nc} X)\\
And^{nc}(Y,Z) &:= \mu_X^{nc}(Y \to Z \to X)\\
Or^d(Y,Z) &:= \mu_X(Y \to^c X, Z \to^c X)\\
Or^l(Y,Z) &:= \mu_X(Y \to^c X, Z \to^{nc} X)\\
Or^r(Y,Z) &:= \mu_X(Y \to^{nc} X, Z \to^c X)\\
Or^u(Y,Z) &:= \mu_X(Y \to^{nc} X, Z \to^{nc} X)\\
Or^{nc}(Y,Z) &:= \mu_X^{nc}(Y \to X, Z \to X)
\end{aligned}
$$

Die Abkürzungen, wie sie in Beispiel 1.4.9 eingeführt wurden, werden auch für die dekorierten Varianten verwendet, wobei die Abkürzung noch mit dem entsprechenden Superscript versehen wird. Wir schreiben also zum Beispiel $\exists_x^d A$ für $Ex^d(\{x|A\})$ oder $A \vee^l B$ für $Or^l(\{|A\},\{|B\})$.
Man kann sich leicht überlegen, dass eine Dekoration bei den Klauseln eines nicht-rechnerischen induktiv definierten Prädikats irrelevant ist. Das heißt, man könnte natürlich auch die Klauseln des nicht-rechnerischen Prädikats dekorieren. Die verschiedenen Prädikate, die man dann erhält, wären aber äquivalent. Für die Leibnizgleichheit aus Beispiel 1.4.3 verwenden wir nun immer die nicht-rechnerische Form, also $Eq(\rho) := \mu_X(\forall_x^{nc} Xxx)$.

Definition 1.6.8. Auch die Totalitätsprädikate $\mathbf{T}$ und $\mathbf{G}$ haben dekorierte Versionen. Bei der dekorierten gesamten Totalität $\mathbf{G}$ werden alle Allquantoren zu $\forall^{nc}$ und alle Implikationspfeile bleiben rechnerisch. Vergleichen wir das mit Definition 1.5.3, so ändert sich das dort definierte K_i zu

$$K_i := \forall_{\vec{x}\vec{y}}^{nc}(\mathbf{G}_{\vec{\rho}}\vec{x} \to (\forall_{\vec{z}_j}^{nc}(\mathbf{G}_{\vec{\sigma}_j}\vec{z}_j \to Xy_j\vec{z}_j))_{j<n} \to XC_i\vec{x}\vec{y})$$

und für Pfeiltypen haben wir

$$\mathbf{G}_{\rho\to\sigma} := \mu_X(\forall_f^{nc}(\forall_x^{nc}(\mathbf{G}_\rho x \to \mathbf{G}_\rho(fx)) \to Xf)).$$

Die dekorierte strukturelle Totalität $\mathbf{T}$ hat auch nur rechnerische Implikationen und genau die Variablen, bei denen die Totalität nicht gefordert wird, gehen auch rechnerisch ein. Im Gegensatz zu Definition 1.5.3 ändert sich K_i' zu

$$K_i' := \forall_{\vec{x}}\forall_{\vec{y}}^{nc}((\forall_{\vec{z}_j}^{nc}(\mathbf{T}_{\vec{\sigma}_j}\vec{z}_j \to Xy_j\vec{z}_j))_{j<n} \to XC_i\vec{x}\vec{y})$$

und die strukturelle Totalität von Pfeiltypen ist gegeben durch

$$\mathbf{T}_{\rho\to\sigma} := \mu_X(\forall_f^{nc}(\forall_x^{nc}(\mathbf{T}_\rho x \to \mathbf{T}_\rho(fx)) \to Xf)).$$

Die in Abschnitt 1.5 bewiesenen Aussagen über totale Terme gelten auch für die dekorierte Totalität. Die Beweise werden jeweils genauso geführt wie ohne die Dekorationen. Es ist aber etwas kompliziert und daher eher Sache eines Computers, jeweils die Regeln der dekorierten logischen Symbole zu überprüfen.
Wir bezeichnen mit **G** bzw. **T** nun immer die dekorierten Totalitätsprädikate.

1.7 Typen von Formeln

Definition 1.7.1. Wir definieren nun den Typ $\tau(A)$ einer Formel A. Dieser wird später der Typ des extrahierten Terms sein. Simultan definieren wir auch den Typ eines Prädikats P. Dazu führen wir noch die Sprechweise des Nulltyp $\circ$ ein. Ist eine Formel oder ein Prädikat vom Typ $\circ$, bedeutet dies, dass sie oder es keinen rechnerischen Gehalt hat. Wir setzen noch $\rho \to \circ := \circ$ und $\circ \to \sigma := \sigma$. Weiter sei eine globale injektive Zuordnung von allen rechnerischen Prädikatenvaribalen X zu den Typenvariablen ξ gegeben.

- Ist X eine rechnerische Prädikatenvariable so ist $\tau(X) := \xi$ die X zugeordnete Typenvariable.
- Ist $I = \mu_X(K_0, \ldots, K_{k-1})$ ein induktiv definiertes Prädikat, so ist
$$\tau(I) := \mu_\xi(\tau(K_0), \ldots, \tau(K_{k-1})).$$
- Ist I^{nc} ein nicht rechnerisches induktiv definiertes Prädikat oder eine nichtrechnerische Prädikatenvariable, dann ist $\tau(I^{nc}) = \circ$.
- Ist $P\vec{t}$ eine Formel mit einem Prädikat P und Termen $\vec{t}$, dann ist $\tau(P\vec{t}) := \tau(P)$.
- Ist $\{\vec{x}|A\}$ ein Prädikat mit einer Formel A, dann ist $\tau\left(\{\vec{x}|A\}\right) := \tau(A)$.
- Sind A und B Formeln, dann ist $\tau(A \to^c B) := \tau(A) \to \tau(B)$ und $\tau(A \to^{nc} B) := \tau(B)$.
- Ist A eine Formel und x^ρ eine Variable vom Typ ρ, dann ist $\tau(\forall^c_x A) := \rho \to \tau(A)$ und $\tau(\forall^{nc}_x A) := \tau(A)$

Beispiel 1.7.2. Wir können nun den Typ der Prädikate aus Beispiel 1.6.7 bestimmen und uns heuristisch Gedanken darüber machen, ob dies mit unserer Erwartung zusammenpasst:

$$\tau(Ex^d(Y)) = \mu_\xi(\tau(\forall^c_{x^\rho}(Yx \to^c X)) = \mu_\xi(\rho \to \tau(Y) \to \xi) = \rho \times \tau(Y)$$

Der Typ von $Ex^d(Y)$, ist also der Produkttyp bestehend aus dem Typ der quantifizierten Variablen und den Typ des Parameters Y. Dies deckt sich auch mit der Erwartung, dass wir als rechnerischen Gehalt von $\exists^d_x A(x)$ einen Term t vom Typ der Variablen x erhalten, sodass $A(t)$ gilt, und wir natürlich auch den rechnerischen Gehalt von $A(t)$ bekommen.
Für die anderen Varianten von $Ex(Y)$ haben wir:

$$\begin{aligned}\tau(Ex^l(Y)) &= \mu_\xi(\rho \to \xi)\\ \tau(Ex^r(Y)) &= \mu_\xi(\tau(Y) \to \xi)\\ \tau(Ex^u(Y)) &= \mu_\xi(\xi) = \mathbb{U}\end{aligned}$$

Die ersten beiden Typen lassen sich mit ρ bzw. $\tau(Y)$ identifizieren. Bei $\exists^l_x A(x)$ ist nur der Term t, für den $A(t)$ gilt, rechnerisch relevant, da A bei $\forall^c_x(A \to^{nc} X)$ nicht rechnerisch eingeht. Daher erwarten wir genau nur diesen Term als rechnerischen Gehalt, was auch mit dem Typ übereinstimmt. Bei $\exists^r_x A(x)$ ist es genau andersherum. Hier geht nur A rechnerisch ein, weswegen wir als extrahierten Term genau den extrahierten Term von A erwarten. Bei $\exists^u_x A$ geht nichts rechnerisch ein. Damit passt auch der Einheitstyp intuitiv zu diesem Prädikat.
Bei der Konjunktion verhält es sich ähnlich. Hier haben wir

$$\begin{aligned}\tau(And^d(Y,Z)) &= \mu_\xi(\tau(Y) \to \tau(Z) \to \xi)\\ \tau(And^l(Y,Z)) &= \mu_\xi(\tau(Y) \to \xi)\\ \tau(And^l(Y,Z)) &= \mu_\xi(\tau(Z) \to \xi)\\ \tau(And^u(Y,Z)) &= \mathbb{U}.\end{aligned}$$

Je nachdem, welche Seite rechnerisch eingeht, deren extrahierten Term erwarten wir im extrahierten Term der Konjunktion.
Bei der Disjunktion erwarten wir eine andere Art des extrahierten Terms. Einerseits wollen wir die Information, wann welche Seite der Disjunktion gilt, und andererseits wollen wir gegebenenfalls auch den extrahierten Term der dann geltenden Aussage, wenn diese rechnerisch eingeht. Das passt auch zu den entsprechenden Typen:

$$\begin{aligned}\tau(Or^d(Y,Z)) &= \mu_\xi(\tau(Y) \to \xi, \tau(Z) \to \xi) = \tau(Y) + \tau(Z)\\ \tau(Or^l(Y,Z)) &= \mu_\xi(\tau(Y) \to \xi, \xi)\\ \tau(Or^r(Y,Z)) &= \mu_\xi(\xi, \tau(Z) \to \xi)\\ \tau(Or^u(Y,Z)) &= \mu_\xi(\xi,\xi) = \mathbb{B}\end{aligned}$$

Hier beachte man, dass wir in jedem Fall die Information bekommen, welche Seite der Disjunktion gilt. Damit trägt Or^u immer noch mehr rechnerischen Gehalt als Or^{nc}. Bei Ex^u und And^u ist das nicht der Fall.

1.8 Realisierung und der extrahierte Term

Motivation 1.8.1. Nun haben wir genügend theoretische Vorbereitungen getroffen, sodass wir den extrahierten Term eines Beweises definieren können. Ist A eine Aussage mit Beweis M, dann wird der extrahierte Term $et(M)$ vom Typ $\tau(A)$ sein.

Definition 1.8.2. Um den extrahierten Term $et(M)$ eines Beweises M in **TCF** zu definieren, brauchen wir eine injektive Zuordnung von Annahmenvariablen u^A zu Objektvariablen $x_u^{\tau(A)}$. Außerdem definieren wir rein formal ε als den einzigen Term vom Nulltyp und setzen $\varepsilon t := \varepsilon$ sowie $t\varepsilon := t$. Ist nun M^A eine Herleitung einer nichtrechnerischen Aussage A, dann setzen wir $et(M^A) := \varepsilon$. Die anderen Fälle sind wie

folgt definiert:

$$\begin{aligned}
et\left(u^A\right) &:= x_u^{\tau(A)} \\
et\left(\left(\lambda_{u^A} M^B\right)^{A\to^c B}\right) &:= \lambda_{x_u^{\tau(A)}} et(M) \\
et\left(\left(M^{A\to^c B} N^A\right)^B\right) &:= et(M)et(N) \\
et\left(\left(\lambda_{x^\rho} M^a\right)^{\forall_x^c A}\right) &:= \lambda_{x^\rho} et(M) \\
et\left(\left(M^{\forall_x^c A(x)} r\right)^{A(r)}\right) &:= et(M)r \\
et\left(\left(\lambda_{u^A} M^B\right)^{A\to^{nc} B}\right) &:= et(M) \\
et\left(\left(M^{A\to^{nc} B} N^A\right)^B\right) &:= et(M) \\
et\left(\left(\lambda_{x^\rho} M^a\right)^{\forall_x^{nc} A}\right) &:= et(M) \\
et\left(\left(M^{\forall_x^{nc} A(x)} r\right)^{A(r)}\right) &:= et(M)
\end{aligned}$$

Ist $I = \mu_X(K_0, \ldots, K_{k-1})$ ein induktiv definiertes Prädikat, so definieren wir:

$$\begin{aligned}
et\left(I_i^+\right) &:= C_i \\
et\left(I^-(P)\right) &:= \mathscr{R}_{\tau(I)}^{\tau(P)}
\end{aligned}$$

Dabei gehören die Konstruktoren C_i zur Algebra $\tau(I)$.

Beispiel 1.8.3. Als erstes Beispiel wollen wir Prädikate aus Beispiel 1.7.2 betrachten und sehen, dass der extrahierte Term konsistent mit unserer Erwartung ist. Beginnen wir mit der einfachen Aussage $\exists_x^l Eq(\mathbb{N})(x,1)$. Eine Herleitung dieser Aussage sieht wie folgt aus:

$$\left(\left(Ex^l\left(\{x|Eqx1\}\right)_0^+\right)^{\forall_x^c(Eqx1\to^{nc}\exists_x^l Eqx1))} 1\right)^{Eq11\to^{nc}\exists_x Eqx1} \left(\left(Eq_0^+\right)^{\forall_x^{nc} Eqxx} 1\right)^{Eq11}$$

Wenden wir nun die eben definierte Funktion et darauf an, so erhalten wir:

$$\begin{aligned}
&et\left(\left(Ex^l(\{x|Eq(x,1)\})_0^+ 1\right)\left(Eq_0^+ 1\right)\right) = et\left(Ex^l(\{x|Eq(x,1)\})_0^+ 1\right) = \\
&et\left(Ex^l(\{x|Eq(x,1)\})_0^+\right) 1 = C_0 1
\end{aligned}$$

C_0 ist dabei der einzige Konstruktor der Algebra $\mu_\xi(\mathbb{N}\to\xi)$, welche man mit $\mathbb{N}$ identifizieren kann. Wir erhalten damit 1 als extrahierten Term. Dies stimmt auch mit der Intuition überein, dass man einen Term t erwartet, so dass $Eq(t,1)$ gilt. Man beachte noch, dass Eq keinen echten rechnerischen Gehalt hat, weil $\tau(Eq) = \mathbb{U}$ ist. Hätten wir also $\exists^d$ anstelle von $\exists^r$ verwendet, wäre der extrahierte Term $\langle 1, \mathbf{u}\rangle$, wobei $\mathbf{u}$ der Konstruktor von $\mathbb{U}$ ist, gewesen. Dadurch hätten wir aber keine weitere Information, sondern nur überflüssige Schreibarbeit.
Für das nächste Beispiel gehen wir davon aus, dass wir für die drei Aussagen A, B und C die Herleitungen $M^{A\vee^l B}$, $N^{A\to C}$ und $L^{B\to^{nc} C}$ haben. Dann ist

$$(Or^l(A,B))^- MNL$$

eine Herleitung von C. Diese gibt uns den extrahierten Term

$$et\Big((Or^l(A,B))^- MNL\Big) = \mathscr{R}_\iota^{\tau(C)} et(M)^\iota et(N)^{\tau(A)\to\tau(C)} et(L)^{\tau(C)},$$

dabei ist $\iota = \tau(Or^l(A,B)) = \mu_\xi(\tau(A)\to\xi,\xi)$. Beginnt also $et(M)$ mit dem Konstruktor C_0, dann bedeutet dies, dass wir eine Herleitung von A haben, beginnt $et(M)$ mit C_1 so haben wir eine Herleitung von B vorliegend. Im ersten Fall steht nach C_0 noch der rechnerische Gehalt des Beweises von A.

Motivation 1.8.4. Nachdem wir nun den extrahierten Term einer Aussage definiert und auch an Beispielen gesehen haben, dass dieser das Gewünschte liefert, stellt sich nun die Frage, ob dies immer der Fall ist und was überhaupt das Gewünschte ist. Um zu beantworten, was das Gewünschte ist, definieren wir das Realisierungsprädikat einer Aussage. Wir haben uns schon am Beispiel des Existenzquantors überlegt, dass ein Term t eine Aussage der Form $\exists_x^l A(x)$, dann realisieren sollte, wenn $A(t)$ gilt. Er ist damit im übertragenen Sinne ein Zeuge für die Aussage $\exists_x^l A(x)$.

Definition 1.8.5. Sei A eine Formel und t ein Term vom Typ $\tau(A)$ (oder der Nullterm ε), dann definieren wir $t\ \mathbf{r}\ A$ oder in Worten „t realisiert A“ durch die folgenden Regeln:

Man beachte, dass der Nullterm wieder nur eine Sprechweise ist, um nicht immer eine Fallunterscheidung zwischen einer rechnerischen und einer nicht-rechnerischen Formel machen zu müssen. Das Realisierungsprädikat für nicht-rechnerische Formeln ist eigentlich nullstellig. Ist A nicht-rechnerisch, so bedeutet zum Beispiel $\forall_{x,y}(y\ \mathbf{r}\ A\to B)$ nichts anderes wie $\forall_x(A\to B)$.

Ist A nicht-rechnerisch, dann definieren wir $\varepsilon\ \mathbf{r}\ A := A$. Weiter definieren wir

$$\begin{aligned} t\ \mathbf{r}\ (A\to^c B) &:= \forall_x^{nc}(x\ \mathbf{r}\ A\to^{nc} tx\ \mathbf{r}\ B),\\ t\ \mathbf{r}\ (A\to^{nc} B) &:= \forall_x^{nc}(x\ \mathbf{r}\ A\to^{nc} t\ \mathbf{r}\ B),\\ t\ \mathbf{r}\ \forall_x^c A &:= \forall_x^{nc}(tx\ \mathbf{r}\ A),\\ t\ \mathbf{r}\ \forall_x^{nc} A &:= \forall_x^{nc}(t\ \mathbf{r}\ A). \end{aligned}$$

Für eine Aussage $I\vec{s}$ mit einem rechnerischem induktiv definiertem Prädikat I definieren wir

$$t\ \mathbf{r}\ I\vec{s} := I^{\mathbf{r}} t\vec{s}.$$

Dabei ist $I^{\mathbf{r}}$ das sogenannte Zeugenprädikat von I und in folgender Definition gegeben:

Definition 1.8.6. Es sei $I = \mu_X(K_0,\dots,K_{k-1})$ ein induktiv definiertes Prädikat mit

$$K_i = \forall_{\vec{x}}^{c/nc}(\vec{A}\to^{c/nc}(\forall_{\vec{y}_j}^{c/nc}(\vec{B}_j\to^{c/nc} X\vec{s}_j))_{j<n_i}\to^c X\vec{t})$$

und $\tau(I)=\iota$. Wir definieren zu jedem K_i eine Klausel $K_i^{\mathbf{r}}$ durch

$$K_i^{\mathbf{r}} := \forall_{\vec{x},\vec{u},\vec{f}}\left(\vec{u}\ \mathbf{r}\ \vec{A}\to\left(\forall_{\vec{y}_j,\vec{v}_j}\left(\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to X\left(f_j\dot{\vec{y}}_j\dot{\vec{v}}_j\right)\vec{s}_j\right)\right)_{j<n}\to X\left(C_i\dot{\vec{x}}\dot{\vec{u}}\vec{f}\right)\vec{t}\right).$$

Dabei treten in $\dot{\vec{x}}$ genau die x_j auf, welche in K_i rechnerisch gebunden wurden, und in $\dot{\vec{u}}$ treten genau die u_j auf, bei denen A_j in K_i rechnerisch eingeht. Analog ist auch jedes $\dot{\vec{y}}_j$ und $\dot{\vec{v}}_j$ gegeben. Außerdem bezeichnet C_i den i-ten Konstruktor der Algebra $\iota=\tau(I)$. Das induktiv definierte Prädikat $I^{\mathbf{r}} := \mu_X^{nc}(K_0^{\mathbf{r}},\dots,K_{k-1}^{\mathbf{r}})$ ist dann das Zeugenprädikat von I.

Beispiel 1.8.7. Betrachten wir als Beispiel zunächst die Zeugenprädikate für einige induktiv definierte Prädikate aus Beispiel 1.6.7. Gehen wir zunächst auf den Existenzquantor ein:

Bei $Ex^d(P)$ haben wir $\tau(Ex^d(P)) = \rho \times \tau(P)$ und die einzige Klausel $K_0 = \forall_{x^\rho}(Px \to X)$, das gibt die Zeugenklausel $K_0^{\mathbf{r}} = \forall_{x,u}(u\ \mathbf{r}\ Px \to X\langle x, u\rangle)$. Zu lesen ist dies wie folgt: Wenn wir ein x^ρ und ein $u^{\tau(P)}$ gefunden haben, sodass, u ein Realisierer von Px ist, dann ist das Paar $\langle x, u\rangle$ ein Realisierer von $Ex^d(P)$.
Im Gegensatz zu $Ex^d(P)$ können wir nun einen Blick auf $Ex^u(P)$ werfen. Es ist $\tau(Ex^u(P))) = \mathbb{U}$ und damit ist $Ex^u(P)$ praktisch ohne rechnerischen Gehalt. Für die einzige Klausel $K_0 = \forall^{nc}_{x^\rho}(Px \to^{nc} X)$ haben wir die Zeugenklausel $\forall_{x,u}(u\ \mathbf{r}\ Px \to X\mathbf{u})$. Wir sehen also, dass $(Ex^u)^{\mathbf{r}}(P)$ ein Prädikat ist, das nur den Einheitskonstruktor als Argument haben kann, und damit nur die Information liefert, ob $(Ex^u)^{\mathbf{r}}(P)$ gilt.
Bei $And^l(Y, Z)$ haben wir $\tau(And^l(Y, Z)) = \mu_\xi(\tau(Y) \to \xi)$ und die Klausel $K_0 = Y \to Z \to^{nc} X$ und damit $K_0^{\mathbf{r}} = \forall_{u_1,u_2}(u_1\ \mathbf{r}\ Y \to u_2\ \mathbf{r}\ Z \to XC_0u_1)$. Wir sehen also, dass ein Realisierer von $A \wedge^l B$ im Grunde ein Realisierer von A ist.
Besonders interessant wird es bei den verschiedenen Varianten von $Or(Y, Z)$. Hier eine Liste davon:

$$(Or^d(Y,Z))^{\mathbf{r}} = \mu_X(\forall_u(u\ \mathbf{r}\ Y \to X\mathsf{inl}u), \forall_u(u\ \mathbf{r}\ Z \to X\mathsf{inr}u))$$
$$(Or^l(Y,Z))^{\mathbf{r}} = \mu_X(\forall_u(u\ \mathbf{r}\ Y \to XC_0u), \forall_u(u\ \mathbf{r}\ Z \to XC_1))$$
$$(Or^r(Y,Z))^{\mathbf{r}} = \mu_X(\forall_u(u\ \mathbf{r}\ Y \to XC_0), \forall_u(u\ \mathbf{r}\ Z \to XC_1u))$$
$$(Or^u(Y,Z))^{\mathbf{r}} = \mu_X(\forall_u(u\ \mathbf{r}\ Y \to X\mathsf{tt}), \forall_u(u\ \mathbf{r}\ Z \to X\mathsf{ff}))$$

Bei jedem dieser Prädikate fällt auf, dass die Algebra von $Or^{d/l/r/u}(Y, Z)$ immer genau zwei Konstruktoren hat und gilt XC_0, haben wir bereits die Information, dass wir einem Realisierer für die linke Seite haben, und XC_1 gibt die Information, dass wir einen Realisierer für die rechte Seite der Disjunktion haben. Dies ist unabhängig davon, welche Argumente auf den Konstruktor folgen.

1.9 Korrektheitssatz

Motivation 1.9.1. Wir kommen nun zum Korrektheitssatz für den extrahierten Term. Heuristisch ausgedrückt, sagt dieser, dass der extrahierte Term eines Beweises auch der Realisierer der bewiesenen Formel ist oder kurz gesagt: Alles was beweisbar ist, ist auch realisierbar.
Der Korrektheitssatz ist der wichtigste Satz für die Beweisextraktion.

Satz 1.9.2. Sei M eine Herleitung einer Aussage A unter den Annahmen $(u_i : B_i)_{i<n}$. Weiter enthalte jeder nicht-rechnerische Teil von M keine Regeln für den rechnerischen Implikationspfeil. Dann gilt $\mathbf{TCF} \cup \{x_{u_i}\ \mathbf{r}\ B_i | i < n\} \vdash et(M)\ \mathbf{r}\ A$.

Nicht-rechnerische Teile eine Beweises sind in diesem Fall Teilherleitungen von nicht-rechnerischen Aussagen.

Beweis. Der Beweis geht mit Induktion über M. Wir gehen zunächst die einfachen Fälle der logischen Schlussregeln durch.

$M = u^A$**:** In diesem Fall ist $et(M) = x_u$ und die Aussage $x_u\ \mathbf{r}\ A$ ist eine der Annahmen.

$M = (\lambda_{u^A} N^B)^{A \to^c B}$**:** Dann haben wir $et(M) = \lambda_{x_u} et(N)$, wobei wegen der Annahme über die Herleitung $et(N) \neq \varepsilon$ ist, und wir haben die Aussage $\lambda_{x_u} et(N)\ \mathbf{r}\ (A \to^c B)$ zu zeigen. Diese ist äquivalent zu $\forall^{nc}_x(x\ \mathbf{r}\ A \to^{nc} \lambda_{x_u} et(N)x\ \mathbf{r}\ B)$ und es ist $\lambda_{x_u} et(N)x \doteq et(N)[x_n := x]$. Nach Umbenennen der lokalen Variablen x zu x_u erhalten wir $\forall^{nc}_{x_u}(x_u\ \mathbf{r}\ A \to^{nc} et(N)\ \mathbf{r}\ B)$. Wegen der Induktionshypothese gilt $et(N)\ \mathbf{r}\ B$ unter der zusätzlichen Annahme $x_u\ \mathbf{r}\ A$ und, weil das Realisierungsprädikat nicht rechnerisch ist, folgt die zu zeigende Aussage.

$M = (\lambda_x N^A)^{\forall^c_x A}$**:** Hier ist $et(M) = \lambda_x et(N)$ und die Aussage $\forall^{nc}_x(et(N)\ \mathbf{r}\ A)$ zu beweisen. Die Aussage $et(N)\ \mathbf{r}\ A$ gilt schon nach Induktionshypothese und, weil x nach Variablenbedingung in keiner Annahme frei ist und die Formel $et(N)\ \mathbf{r}\ A$ nicht rechnerisch ist, folgt $\forall^{nc}_x(et(N)\ \mathbf{r}\ A)$.

$M = N^{A\to^c B} L^A$**:** Wir brauchen eine Herleitung von $et(NL)\ \mathbf{r}\ B$. Dabei haben wir mit der Induktionshypothese $et(N)\ \mathbf{r}\ (A \to^c B)$ also $\forall^{nc}_x(x\ \mathbf{r}\ A \to^{nc} et(N)x\ \mathbf{r}\ B)$ und wir erhalten auch $et(L)\ \mathbf{r}\ A$ nach der Induktionshypothese. Das gibt uns genau $et(N)et(L)\ \mathbf{r}\ B$ und wegen $et(N)et(L) = et(NL)$ folgt die Behauptung. Auch hier ist wieder zu beachten, dass nach Annahme $et(N) \neq \varepsilon$ ist.

$M = N^{\forall^c_x A(x)} t$**:** Gefragt ist nun eine Herleitung von $et(Nt)\ \mathbf{r}\ A(t)$. Nach der Induktionshypothese haben wir eine Herleitung von $et(N)\ \mathbf{r}\ \forall^c_x A(x)$, dies ist äquivalent zu $\forall^{nc}_x(et(N)x\ \mathbf{r}\ A(x))$. Setzen wir hier für x den Term t ein, haben wir wegen $et(Nt) = et(N)t$ die Aussage.

$M = (\lambda_{u^A} N^B)^{A\to^{nc} B}$**:** Für diesen Fall ist eine Herleitung von $et(N)\ \mathbf{r}\ (A \to^{nc} B)$ nötig. Mit der Definition ist das äquivalent zu $\forall^{nc}_y(y\ \mathbf{r}\ A \to^{nc} et(N)\ \mathbf{r}\ B)$. Da wir bereits eine Herleitung von $et(N)\ \mathbf{r}\ B$ mit der zusätzlichen Annahme $x_u\ \mathbf{r}\ A$ nach Induktionshypothese haben, folgt damit dieser Fall.

$M = (\lambda_x N^A)^{\forall^{nc}_x A}$**:** Nach Definition ist $et(\lambda_x N) = et(N)$ und wir haben nach Induktionshypothese $et(N)\ \mathbf{r}\ A$ und damit auch $\forall_x(et(N)\ \mathbf{r}\ A)$, was uns $et(N)\ \mathbf{r}\ \forall^{nc}_x A$ und damit $et(\lambda_x N)\ \mathbf{r}\ \forall^{nc}_x A$ liefert. Damit ist dieser Fall gezeigt.

$M = N^{A\to^{nc} B} L$**:** Wir wollen eine Herleitung von $et(N)\ \mathbf{r}\ B$ und haben nach Induktionshypothese eine Herleitung von $et(N)\ \mathbf{r}\ (A \to^{nc} B)$ also $\forall^{nc}_y(y\ \mathbf{r}\ A \to et(N)\ \mathbf{r}\ A)$ sowie von $et(L)\ \mathbf{r}\ A$. Dies gibt uns genau $et(N)\ \mathbf{r}\ B$.

$M = N^{\forall^{nc}_x A(x)} t$**:** Hier brauchen wir eine Herleitung von $et(Nt)\ \mathbf{r}\ A(t)$ und es ist $et(Nt) = et(N)$. Mit der Induktionshypothese haben wir bereits $et(N)\ \mathbf{r}\ \forall^{nc}_x A(x)$ also $\forall^{nc}_x(et(N)\ \mathbf{r}\ A(x))$. Setzen wir t für x ein, haben wir genau $et(N)\ \mathbf{r}\ A(t)$, was gefragt war.

Damit sind wir die logischen Regeln durchgegangen.
Nun ist die Korrektheit für die Axiome zu beweisen. Sei dazu $I = \mu_X(K_0, \ldots, K_{k-1})$ zunächst ein rechnerisches induktiv definiertes Prädikat mit $K_i(X) = \forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_j}(\vec{B}_j \to X\vec{s}_j))_{j<n} \to X\vec{t})$. Wir nehmen der Einfachheit halber an, dass jede Implikation und jeder Allquantor rechnerisch eingeht. Der allgemeine Fall ist nur etwas mehr Schreibarbeit.
Es ist nun für das Einführungsaxiom zu zeigen, dass $C_i\ \mathbf{r}\ K_i(I)$ gilt. Die folgenden Zeilen sind äquivalent zueinander wegen der Definitionen des Realisierungsprädikats und der Tatsache, dass das Realisierungsprädikat nicht rechnerisch ist, weswegen die Dekorationen irrelevant sind:

$$C_i\ \mathbf{r}\ \forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_j}(\vec{B}_j \to I\vec{s}_j))_{j<n} \to I\vec{t})$$
$$\forall_{\vec{x}}(C_i\vec{x}\ \mathbf{r}\ (\vec{A} \to (\forall_{\vec{y}_j}(\vec{B}_j \to I\vec{s}_j))_{j<n} \to I\vec{t}))$$
$$\forall_{\vec{x}}\forall_{\vec{u}}(\vec{u}\ \mathbf{r}\ \vec{A} \to C_i\vec{x}\vec{u}\ \mathbf{r}\ ((\forall_{\vec{y}_j}(\vec{B}_j \to I\vec{s}_j))_{j<n} \to I\vec{t}))$$
$$\forall_{\vec{x}}\forall_{\vec{u}}(\vec{u}\ \mathbf{r}\ \vec{A} \to \forall_{\vec{f}}((\vec{f}\ \mathbf{r}\ \forall_{\vec{y}_j}(\vec{B}_j \to I\vec{s}_j))_{j<n} \to C_i\vec{x}\vec{u}\vec{f}\ \mathbf{r}\ I\vec{t}))$$
$$\forall_{\vec{x}}\forall_{\vec{u}}(\vec{u}\ \mathbf{r}\ \vec{A} \to \forall_{\vec{f}}((\forall_{\vec{y}_j,\vec{v}_j}(\vec{v}_j\ \mathbf{r}\ \vec{B}_j \to f_j\vec{y}_j\vec{v}_j\ \mathbf{r}\ I\vec{s}_j))_{j<n} \to I^{\mathbf{r}}C_i\vec{x}\vec{u}\vec{f}\vec{t}))$$
$$\forall_{\vec{x}}\forall_{\vec{u}}(\vec{u}\ \mathbf{r}\ \vec{A} \to \forall_{\vec{f}}((\forall_{\vec{y}_j,\vec{v}_j}(\vec{v}_j\ \mathbf{r}\ \vec{B}_j \to I^{\mathbf{r}}f_j\vec{y}_j\vec{v}_j\vec{s}_j))_{j<n} \to I^{\mathbf{r}}C_i\vec{x}\vec{u}\vec{f}\vec{t}))$$

Das ist wiederum äquivalent zu

$$\forall_{\vec{x}}\forall_{\vec{u}}\forall_{\vec{f}}(\vec{u}\ \mathbf{r}\ \vec{A}\to(\forall_{\vec{y}_j,\vec{v}_j}(\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to I^{\mathbf{r}}f_j\vec{y}_j\vec{v}_j\vec{s}_j))_{j<n}\to I^{\mathbf{r}}C_i\vec{x}\vec{u}\vec{f}\vec{t}),$$

was das i-te Einführungsaxiom von $I^{\mathbf{r}}$ ist.
Als nächstes zeigen wir, dass $\mathscr{R}\ \mathbf{r}\ I^-$ mit $\mathscr{R}:=\mathscr{R}^{\tau(P)}_{\tau(I)}$ gilt, was nach Definition genau

$$\forall_{\vec{x}}\forall_w(w\ \mathbf{r}\ I\vec{x}\to\forall_{\vec{w}}((w_i\ \mathbf{r}\ K_i(I,P))_{i<k}\to\mathscr{R}w\vec{w}\ \mathbf{r}\ P\vec{x}))$$

ist. Es seien nun $\vec{x}$, w gegeben und es gelte $w\ \mathbf{r}\ I\vec{x}$ also $I^{\mathbf{r}}w\vec{x}$. Weiter seien nun $\vec{w}$ gegeben und es gelte $w_i\ \mathbf{r}\ K_i(I,P)$ für jedes $i<k$. Nach Definition ist das äquivalent zu

$$w_i\ \mathbf{r}\ \forall_{\vec{x}}.\vec{A}\to(\forall_{\vec{y}_j}.\vec{B}_j\to I\vec{s}_j)_{j<n}\to(\forall_{\vec{y}_j}.\vec{B}_j\to P\vec{s})_{j<n}\to P\vec{t}$$

und nach mehrfachem Anwenden der Definitionsregeln des Realisierungsprädikates haben wir

$$\begin{aligned}\forall_{\vec{x},\vec{u},\vec{f},\vec{g}}.\vec{u}\ \mathbf{r}\ \vec{A}\to(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to f_j\vec{y}_j\vec{v}_j\ \mathbf{r}\ I\vec{s}_j)_{j<n}\\ \to(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to g_j\vec{y}_j\vec{v}_j\ \mathbf{r}\ P\vec{s}_j)_{j<n}\to w_i\vec{x}\vec{u}\vec{f}\vec{g}\ \mathbf{r}\ P\vec{t},\end{aligned}\tag{1.1}$$

dabei sollte eigentlich jeder Pfeil und jeder Quantor mit nc dekoriert sein. Da das Realisierungsprädikat nicht rechnerisch ist, spielt dies aber ohnehin keine Rolle. Wir müssen nun unter diesen Prämissen die Aussage $\mathscr{R}w\vec{w}\ \mathbf{r}\ P\vec{x}$ zeigen. Weil wir $I^{\mathbf{r}}w\vec{x}$ gegeben haben, verwenden wir das Eliminationsaxiom für $I^{\mathbf{r}}$. Dieses ist gegeben durch

$$\forall_{\vec{x},w}(Iw\vec{x}\to(K^{\mathbf{r}}_i(I^{\mathbf{r}},Q))_{i<k}\to Qw\vec{x})$$

für jedes nicht rechnerisches Prädikat Q. Da $\mathscr{R}w\vec{w}\ \mathbf{r}\ P\vec{x}$ nicht rechnerisch ist, können wir dies für $Qw\vec{x}$ einsetzen. Es reicht dann also $K^{\mathbf{r}}_i(I^{\mathbf{r}},Q)$ für jedes $i<k$ zu zeigen. Schreiben wir dies mit Definition 1.8.6 um, so haben wir zu zeigen, dass

$$\begin{aligned}\forall_{\vec{x},\vec{u},\vec{f}}.\vec{u}\ \mathbf{r}\ \vec{A}\to(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to I^{\mathbf{r}}f_j\vec{y}_j\vec{v}_j\vec{s}_j)_{j<n}\\ \to(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to Qf_j\vec{y}_j\vec{v}_j\vec{s}_j)_{j<n}\to QC_i\vec{x}\vec{u}\vec{f}\vec{t}\end{aligned}$$

herleitbar ist. Seien daher $\vec{x}$, $\vec{u}$, $\vec{f}$ gegeben und es gelte $\vec{u}\ \mathbf{r}\ \vec{A}$, $(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to I^{\mathbf{r}}f_j\vec{y}_j\vec{v}_j\vec{s}_j)_{j<n}$ sowie $(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j\ \mathbf{r}\ \vec{B}_j\to Qf_j\vec{y}_j\vec{v}_j\vec{s}_j)_{j<n}$. Wir wollen $QC_i\vec{x}\vec{u}\vec{f}\vec{t}$ also $\mathscr{R}C_i\vec{x}\vec{u}\vec{f}\vec{w}\ \mathbf{r}\ P\vec{t}$ zeigen. Mit der Berechnungsregel von $\mathscr{R}$ ist $\mathscr{R}C_i\vec{x}\vec{u}\vec{f}\vec{w}\doteq w_i\vec{x}\vec{u}\vec{f}(\lambda_{\vec{y}_j,\vec{v}_j}\mathscr{R}f_j\vec{y}_j\vec{v}_j\vec{w})_{j<n}$. Wir können daher die Formel 1.1 verwenden. wobei wir für $\vec{g}:=(\lambda_{\vec{y}_j,\vec{v}_j}\mathscr{R}f_j\vec{y}_j\vec{v}_j\vec{w})_{j<n}$ einsetzen. Die Prämissen der Formel 1.1 sind bereits gegeben und damit folgt genau das, was zu zeigen war.
Im Falle der nicht-rechnerischen induktiv definierten Prädikate sind die Axiome auch nicht rechnerisch, da jeweils die Konklusion nicht rechnerisch ist, und somit der Typ der ganzen Formel der Nulltyp ist. Damit ist der extrahierte Term der Nullterm ε, was auch per Definition ein Realisierer ist. Man beachte, dass beim Eliminationsaxiom von nicht-rechnerischen induktiv definierten Prädikaten die Prädikatenvariable auch nicht rechnerisch ist, also den Nulltyp als Typ hat. □

Bemerkung 1.9.3. Wir haben schon in Beispiel 1.7.2 gesehen, dass es induktiv definierte Prädikate gibt, welche als Typ den Einheitstyp haben und daher immer nur den rechnerischen Gehalt **u** geben, welcher keine Information trägt. Das ist bei

induktiv definierten Prädikaten $\mu_X(K_0)$ der Fall, welche nur eine Klausel besitzen, in der an jeder möglichen Stelle nc steht. Das heißt, die Klausel hat die Form $K_0 = \forall^{nc}_{\vec{x}}(\vec{A} \to^{nc} X\vec{t})$. Um den extrahierten Term nicht unnötig in die Länge zu ziehen, kann man alternativ auch den Typ solcher Prädikate als Nulltyp definieren. Damit wird $\mu_X(K_0)$ gleichgestellt zu $\mu^{nc}_x(K_0)$ mit dem einzigen Unterschied, dass es auch erlaubt ist, im Eliminationsaxiom von $\mu_X(K_0)$ rechnerische Prädikate einzusetzen. Das Eliminationsaxiom von $\mu_X(K_0) =: I$ ist

$$I^-(P) : \forall^{nc}_{\vec{x}}\left(I\vec{x} \to^c \forall^{nc}_{\vec{x}}\left(\vec{A} \to^{nc} P\vec{t}\right) \to^c P\vec{t}\right).$$

und wir haben

$$\tau(I^-(P)) = \tau(I\vec{x}) \to \tau\left(\forall^{nc}_{\vec{x}}\left(\vec{A} \to^{nc} P\vec{t}\right)\right) \to \tau(P\vec{t}) = \circ \to \tau(P\vec{t}) \to \tau(P\vec{t}) = \tau(P) \to \tau(P).$$

Definiert man nun für solche Prädikate den extrahierten Term $et(I^-(P)) := \lambda_x x$ als die Identität auf $\tau(P)$, dann gilt der Korrektheitssatz auch für diese Konstruktion, denn nach Definition haben wir folgende Äquivalenzumformungen:

$$\begin{aligned}
&\lambda_x x \ \mathbf{r}\ \forall^{nc}_{\vec{x}}\left(I\vec{x} \to^c \forall^{nc}_{\vec{x}}\left(\vec{A} \to^{nc} P\vec{t}\right) \to^c P\vec{x}\right)\\
&\forall^{nc}_{\vec{x}}\forall^{nc}_{y}\left(y \ \mathbf{r}\ I\vec{x} \to^{nc} \lambda_x x \ \mathbf{r}\ \left(\forall^{nc}_{\vec{x}}\left(\vec{A} \to^{nc} P\vec{t}\right) \to^c P\vec{x}\right)\right)\\
&\forall^{nc}_{\vec{x}}\forall^{nc}_{y}\left(y \ \mathbf{r}\ I\vec{x} \to^{nc} \forall^{nc}_{z}\left(z \ \mathbf{r}\ \forall^{nc}_{\vec{x}}\left(\vec{A} \to^{nc} P\vec{t}\right) \to^{nc} (\lambda_x x)z \ \mathbf{r}\ P\vec{x}\right)\right)\\
&\forall^{nc}_{\vec{x}}\forall^{nc}_{y}\left(y \ \mathbf{r}\ I\vec{x} \to^{nc} \forall^{nc}_{z}\left(\forall^{nc}_{\vec{x}}\forall^{nc}_{\vec{y}}\left(\vec{y} \ \mathbf{r}\ \vec{A} \to^{nc} z \ \mathbf{r}\ P\vec{t}\right) \to^{nc} z \ \mathbf{r}\ P\vec{x}\right)\right)
\end{aligned}$$

Weil $\varepsilon \ \mathbf{r}\ I\vec{x} := I\vec{x}$ gilt, ist dies äquivalent zu

$$\forall^{nc}_{\vec{x}}\forall^{nc}_{z}\left(I\vec{x} \to^{nc} \forall^{nc}_{\vec{x}}\forall^{nc}_{\vec{y}}\left(\vec{y} \ \mathbf{r}\ \vec{A} \to^{nc} z \ \mathbf{r}\ P\vec{t}\right) \to^{nc} z \ \mathbf{r}\ P\vec{x}\right)$$

und nach Induktionsvorraussetzung im Beweis des Korrektheitssatzes, folgen aus $\vec{A}$ die Aussagen $\exists_{\vec{y}}(\vec{y} \ \mathbf{r}\ \vec{A})$. Damit folgt die obige Aussage aus dem Eliminationsaxiom von I, wobei wir das Prädikat $\{\vec{x}|z \ \mathbf{r}\ P\vec{x}\}$ in das Eliminationsaxiom einsetzen.
In Minlog ist auch die Ausnahme für solche Prädikate implementiert. Im Allgemeinen wird der Einheitskonstruktor beim Erstellen des rechnerischen Gehaltes von vielen Programmen wegkompliert.

Kapitel 2

Minlog

In diesem Kapitel möchten wir dem Leser den Umgang mit dem Beweisassistenten Minlog vermitteln. Minlog basiert auf der Theorie der berechenbaren Funktionale, die in Kapitel 1 vorgestellt wurde. Wir werden uns hier auch an den theoretischen Grundlagen aus dem ersten Kapitel orientieren.
Auf der Webseite des Minlog-Systems [5] ist eine Anleitung zum Download und Einrichten von Minlog gegeben. Wir benutzen den Entwicklungszweig (dev branch). Wie man auf diesen Zweig wechseln kann, wird dort auch erklärt. Ist Minlog auf dem Computer installiert, sind im Ordner `doc` des angelegten Verzeichnisses das Minlog-Tutorium `tutor.pdf` und das Referenzmanual `ref.pdf` abgespeichert. Einige Beispiele aus diesem Kapitel sind aus dem Minlog-Tutorium übernommen. Im Referenzmanual sind noch weitere Befehle für Minlog beschrieben.

2.1 Grundlegende Befehle in Minlog

2.1.1 Deklaration von Prädikatenvariablen

Um Variable aller Art, das heißt Typvariable, Termvariable sowie Prädikatenvariable, sinnvoll nutzen zu können, ist es erforderlich, diese zu deklarieren. Für das erste Beispiel benötigen wir gleich Aussagenvariablen, also nullstellige Prädikatenvaribalen, damit wir einen Satz beweisen können. Eine Prädikatenvariable wird mit dem Befehl `add-pvar-name` deklariert. Dieser hat die Form

```
(add-pvar-name NAME (make-arity TYPS)).
```

Für `NAME` setzt man die Liste der Zeichenketten ein, die man deklarieren möchte. Für `TYPS` wird eine Liste von Typen angegeben. Diese Liste gibt die erwarteten Typen der Argumente des Prädikats an. In unserem Fall wollen wir nullstellige Prädikatenvariablen also Aussagenvariablen deklarieren, weswegen wir diese mit A,B und C bezeichnen und die Liste der Argumententypen leer ist. Wir geben daher

```
(add-pvar-name "A" "B" "C" (make-arity))
```

ein. Die Ausgabe des Computers ist dann

```
ok, predicate variable A: (arity) added
ok, predicate variable B: (arity) added
ok, predicate variable C: (arity) added
```

Für Demonstrationszwecke geben wir noch die Deklaration einer Prädikatenvariable P an, die ein Argument vom Typ $\mathbb{N}$ und ein Argument vom Typ $\mathbb{N} \to \mathbb{N}$ hat:

```
(add-pvar-name "P" (make-arity (py "nat") (py "nat=>nat")))
```

Dabei steht nat für den Typ der natürlichen Zahlen. Wir werden diesen erst später formal in Minlog einführen. Der Pfeil → zwischen Typen wird durch => eingegeben und der Befehl

```
(py STRING)
```

gibt an, dass die Zeichenkette STRING als Typ interpretiert werden soll.
Sehr häufig werden wir auch die Befehle (pt STRING), (pv STRING) und (pf STRING) verwenden. Diese geben analog an, dass STRING als Term, Variable bzw. Formel aufgefasst werden soll.

2.1.2 Erster Beweis

Nachdem wir nun Aussagenvariablen A, B und C eingeführt haben, können wir nun den kleinen Satz $(A \to B \to C) \to ((C \to A) \to B) \to A \to C$ beweisen. Einen Herleitungsbaum ist dafür in Beispiel 1.1.3 angegeben. Schreiben wir einen Beweis in Minlog, ist es nicht so, dass wir bei jedem Beweis einen Herleitungsbaum oder gar einen Herleitungsterm eingeben müssen. Dies wäre für komplexere Beweise viel zu mühsam. Die eingegebenen Befehle werden intern zu einem Beweisbaum verarbeitet. Diesen kann man sich dann auch (in der Notation von Minlog) ausgeben lassen, wie wir sehen werden.
Am Anfang eines Beweises teilen wir dem Programm mit, welche Aussage wir beweisen wollen. Dies geschieht mit dem Befehl

```
(set-goal FORMEL).
```

Für FORMEL ist hier die Zielformel in Anführungszeichen einzusetzen. Wir schreiben daher

```
(set-goal "(A -> B -> C) -> ((C -> A) -> B) -> A -> C").
```

Wie man sieht, steht -> für den Implikationspfeil →. Als Ausgabe erhalten wir dann:

```
------------------------------------------------------------------
?_1:(A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

Damit ist nun die Aussage $(A \to B \to C) \to ((C \to A) \to B) \to A \to C$ als Zielformel festgelegt. Die Zielformel steht immer unterhalb der gestrichelten Linie. Im Bereich oberhalb der Linie, auch genannt Kontext, befinden sich die Annahmen, die uns zur Verfügung stehen. Im Moment haben wir keine Annahme zur Verfügung, um die Aussage zu beweisen. Da es sich bei unserer Aussage aber um eine Implikation mit drei Prämissen handelt, liegt es nahe, diese drei Prämissen in den Kontext zu befördern und dann die Konklusion C mit Hilfe dieser Prämissen zu zeigen. Dies tun wir mit dem Befehl

```
(assume NAMES).
```

Anstelle von NAMES schreiben wir eine Liste von Namen, mit denen wir die Annahmen bezeichnen wollen. Unsere Liste wird aus drei Namen bestehen. Wir schreiben zum Beispiel

```
(assume "Annahme1" "Annahme2" "Annahme3")
```

und das Programm gibt uns folgendes zurück:

```
ok, we now have the new goal

  Annahme1:A -> B -> C
  Annahme2:(C -> A) -> B
  Annahme3:A
------------------------------------------------------------------
?_2:C
```

Die drei Annahmen befinden sich nun im Kontext. Hätten wir zwei Namen angegeben, dann wären nur die ersten beiden Prämissen im Kontext, und hätten wir mehr als drei Namen angeben, würden wir eine Fehlermeldung erhalten. Wir haben also nun die Aussage C unter den drei Annahmen oberhalb der Linie zu zeigen. Da `Annahme1` als Konklusion genau C hat, würden wir gerne diese Annahme verwenden. Dies geschieht mit dem Befehl

```
(use ANNAHME),
```

wobei wir für `ANNAHME` den Namen der Annahme einsetzen, die wir verwenden möchten. Diese Annahme muss als Konklusion die Zielformel haben, ansonsten erhalten wir eine Fehlermeldung. Wir geben also

```
(use "Annahme1")
```

ein. Die Annahme mit Namen `Annahme1` hat zwei Prämissen, der Computer fordert nun einen Beweis für jede dieser Prämissen und deswegen erhalten wir zwei neue Zielformeln:

```
ok, ?_2 can be obtained from

  Annahme1:A -> B -> C
  Annahme2:(C -> A) -> B
  Annahme3:A
------------------------------------------------------------------
?_4:B



  Annahme1:A -> B -> C
  Annahme2:(C -> A) -> B
  Annahme3:A
------------------------------------------------------------------
?_3:A
```

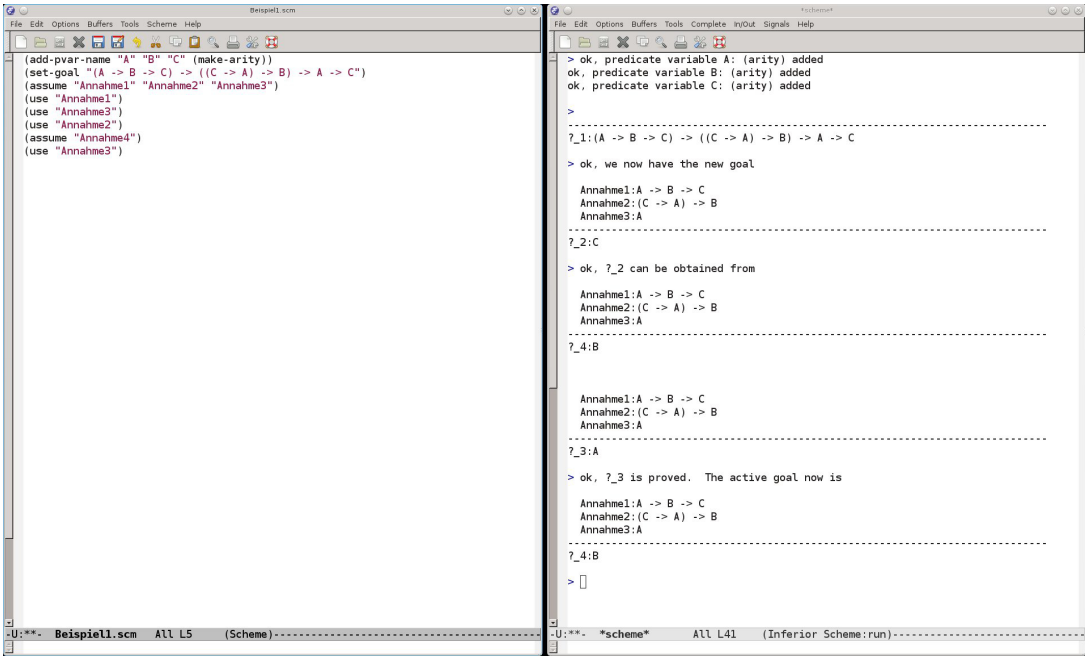
Zunächst müssen wir die untere Zielformel beweisen, diese ist aber genau `Annahme3`, also schreiben wir

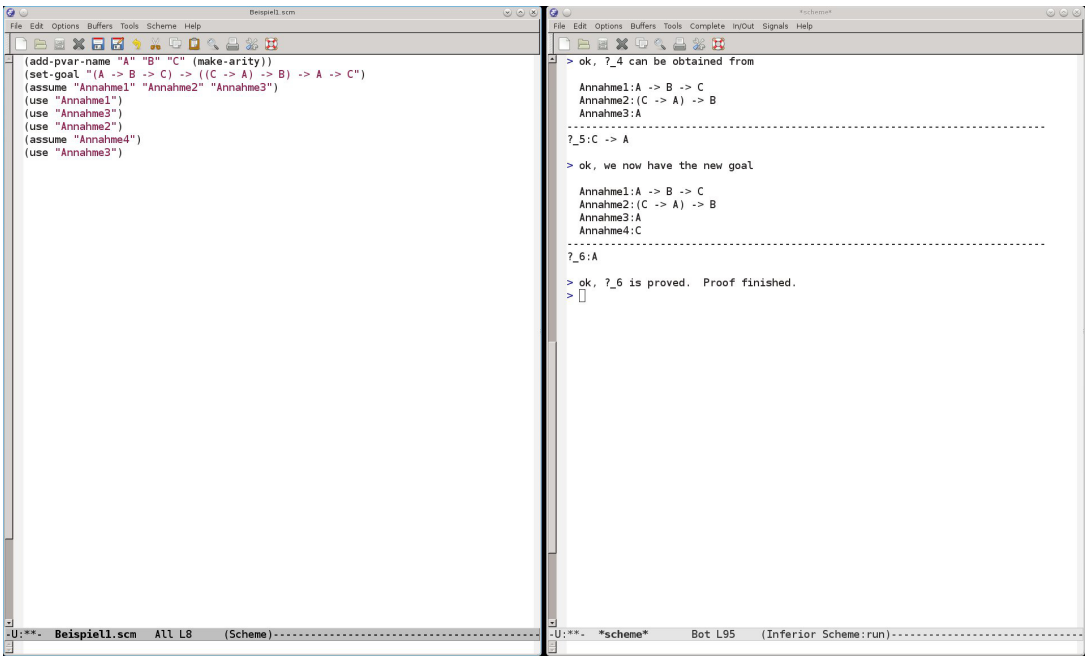
```
(use "Annahme3").
```

Der Computer teilt uns mit, dass die letztere Aussage nun bewiesen ist und wir nur noch die erste Aussage zeigen müssen.

```
ok, ?_3 is proved.  The active goal now is

  Annahme1:A -> B -> C
  Annahme2:(C -> A) -> B
  Annahme3:A
------------------------------------------------------------------------
?_4:B
```

Um diese Aussage zu zeigen, verwenden wir natürlich `Annahme2` mittels Eingabe von `(use "Annahme2")` und haben dann zu zeigen, dass $C \to A$ gilt. Dazu nehmen wir die Aussage C durch `(assume "Annahme4")` in den Kontext und verwenden dann `Annahme3` mit den Befehl `(use "Annahme3")`, um A zu beweisen. Das beendet den Beweis und wir erhalten die Ausgabe: `ok, ?_6 is proved.  Proof finished.` Die folgenden Abbildungen zeigen den Beweis links und die Ausgabe von Minlog rechts nochmal im Ganzen.





2.1.3 Anzeigen von Beweisen

Nachdem wir nun einen Beweis am Computer geführt haben, wollen wir uns diesen Beweis auch anzeigen lassen. Dafür gibt es unterschiedliche Darstellungsmöglichkeiten. Durch den Befehl

```
(display-proof)
```

erhalten wir eine Darstellung, die vergleichbar mit den Beweisbäumen aus Definition 1.1.1 ist. Eine Erweiterung dieses Befehls ist

```
(cdp),
```

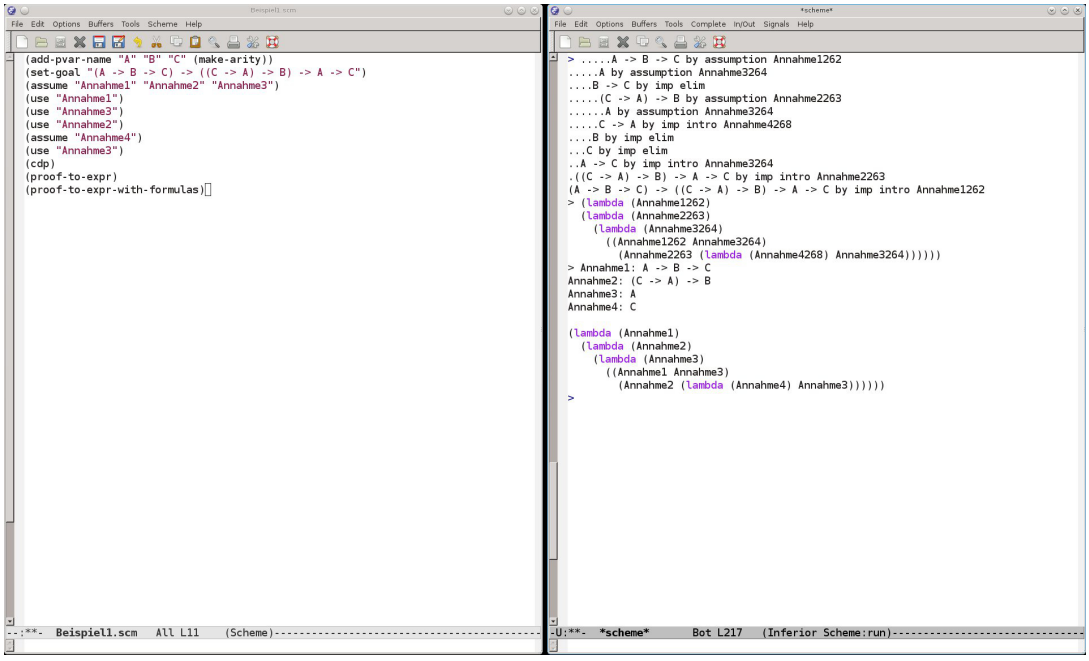
was eine Kurzform für `(check-and-display-proof)` ist. Hier wird der Beweis angezeigt und zusätzlich auf Korrektheit überprüft.
Wollen wir den Beweis als Herleitungsterm ausgeben lassen, so können wir den Befehl

```
(proof-to-expr)
```

oder den Befehl

```
(proof-to-expr-with-formulas)
```

verwenden. In beiden Fällen erhalten wir den Beweis in Form eines Herleitungsterms, wobei im letzten Fall noch der Typ jeder Variablen angegeben wird. In folgender Abbildung sehen wir die Ausgabe des Programms nach den Befehlen `(cdp)`, `(proof-to-expr)` und `(proof-to-expr-with-formulas)`.



Die Ausgabe nach den letzten beiden Befehlen ist dabei leicht zu verstehen. Es handelt sich jeweils um einen Herleitungsterm, wobei für $\lambda_u M$ der Ausdruck `(lambda (u) (M))` und für MN der Ausdruck `(M N)` ausgegeben wird.
Bei `(cdp)` soll die Ausgabe als Herleitungsbaum verstanden werden. Dabei bedeutet die Anzahl der Punkte vor der Formel, auf welcher Höhe des Herleitungsbaumes sie sich befindet. Weiter steht hinter jeder Formel, aus welcher Regel sie abgeleitet wurde.

2.1.4 Abspeichern von Theoremen

In der Regel möchte man bewiesene Aussagen später wieder verwenden, sei es, um sie auf einen Spezialfall zu reduzieren oder, um sie als Lemma in einen größeren Beweis zu integrieren. Der Befehl

```
(save NAME)
```

speichert nach einem vollendeten Beweis die bewiesene Aussage unter dem Namen `NAME` im System ab. In den vorherigen Abschnitten haben wir die Aussage

```
(A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

bewiesen. Diese können wir nach dem Beweis nun mit

```
(save "Theorem1")
```

in Minlog abspeichern. Die Ausgabe ist dann

```
ok, Theorem1 has been added as a new theorem.
ok, program constant cTheoremOne: (alpha412=>alpha413=>alpha411)=>
((alpha411=>alpha412)=>alpha413)=>alpha412=>alpha411
of t-degree 1 and arity 0 added
```

In späteren Beweisen können wir diese Aussage nun verwenden, beispielsweise mit `(use "Theorem1")`.
Durch das Kommando

```
(display-theorems NAME)
```

wird die Aussage und der Name des Theorems `NAME` angezeigt. Man kann auch eine Liste von Namen anstelle von `NAME` angeben, dann werden alle entsprechenden Theoreme angezeigt. Geben wir in unserem Fall `(display-theorems "Theorem1")` ein, so erhalten wir die Ausgabe

```
Theorem1 (A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

Mit dem `pretty-print-` oder kurz pp-Befehl in der Form

```
(pp NAME)
```

wird auch die Formel des Theorems mit Namen `NAME` angezeigt. Hier ist die Ausgabe nur

```
(A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

und wir können für `NAME` nicht mehrere Namen angeben.
Mit pp ist es auch möglich, beliebige im System eingespeicherte Formeln und ebenso Terme anzeigen zu lassen, was wir später noch häufig brauchen werden.

2.1.5 Darstellungseinstellungen

Bevor wir auf weitere Befehle zum Schreiben von Beweisen eingehen, werden in diesem Abschnitt einige nützliche Kommandos gezeigt, mit denen man die Darstellungsweise gegebenenfalls übersichtlicher gestalten kann.
Der erste dafür nützliche Befehl ist

```
(set! COMMENT-FLAG BOOL).
```

Wird der Wert von BOOL durch #f ersetzt, werden keine Kommentare mehr angezeigt. Nur Unregelmäßigkeiten wie beispielsweise Fehlermeldungen und manche Warnungen werden von Minlog ausgegeben. Das Abschalten der Kommentare ist sinnvoll, wenn man eine längere Kette von Befehlen nur einlesen will. Insbesondere spart dies Rechenkapazität und damit Zeit, weil der Computer dann nicht mit dem Ausgeben der Kommentare beschäftigt ist. Setzt man den Wert von BOOL auf #t, so werden die Kommentare wieder normal angezeigt.
Eine Möglichkeit, um während eines Beweises manche Annahmen im Kontext auszublenden, bietet der Befehl

```
(drop STRING).
```

Durch diesen Befehl werden die Annahmen, deren Namen anstelle von STRING stehen, im weiteren Beweis nicht mehr angezeigt, verschwinden dadurch aber nicht. Insbesondere kann man sie weiterhin verwenden. Im letzten Beweis hatten wir beispielsweise die Ausgabe

```
ok, we now have the new goal

  Annahme1:A -> B -> C
  Annahme2:(C -> A) -> B
  Annahme3:A
------------------------------------------------------------------
?_2:C
```

nachdem wir alle Prämissen in den Kontext befördert haben. Geben wir hier nun den Befehl

```
(drop "Annahme1" "Annahme2" "Annahme3")
```

ein, so erhalten wir die Ausgabe

```
ok, we now have the new goal


------------------------------------------------------------------
?_3:C
```

zurück. Den Beweis können wir aber danach genauso wie oben weiterführen.
Mit dem Kommando

```
(display STRING)
```

wird der Text, der an der Stelle von STRING steht, ausgegeben. Dies geschieht auch, wenn (set! COMMENT-FLAG #f) gesetzt wurde. Dadurch kann man auf wichtige Deklarationen, Definitionen, Theoreme und so weiter aufmerksam machen. Im obigen Beispiel wäre es also möglicherweise sinnvoll, nach dem Deklarieren der Prädikatenvariablen mit

```
(display "A, B und C sind nun nullstellige Prädikatenvariablen.")
```

daran zu erinnern, welche Bezeichnungen jetzt vergeben sind. Außerdem lässt sich mit dem Befehl `(newline)` ein Zeilenumbruch erstellen.
Zuletzt sei an dieser Stelle noch der Befehl

```
(undo)
```

erwähnt. Mit ihm kann man den letzten Schritt in einem Beweis rückgängig machen. Minlog gibt dann den Status vor dem letzten Kommando erneut aus.

2.1.6 Einbinden von externen Dateien

In der Bibliothek – genauer gesagt im Ordner `lib` – von Minlog befinden sich sehr hilfreiche vorgefertigte Dateien. Eine der wichtigsten davon ist die Datei `nat.scm`. In ihr wird die Algebra der natürlichen Zahlen eingeführt. Es werden auch einige Funktionen wie zum Beispiel $+$ und $*$ oder auch die boolschwertigen Funktionen $<$ und $\leq$ auf den natürlichen Zahlen definiert und Sätze darüber bewiesen. In den eigenen Beweisen möchte man selbstverständlich diese Vorarbeit auch nutzen. Um dafür nicht die ganze Datei händisch laden zu müssen, gibt es einige Befehle. Mit dem Befehl

```
(libload NAME)
```

wird die Datei mit dem Namen `NAME` aus dem Ordner `lib` geladen. Das bedeutet, ihr ganzer Inhalt wird eingelesen. Hier empfiehlt es sich vor dem Ausführen des Befehls, das Kommando `(set! COMMENT-FLAG #f)` einzuschieben, ansonsten wird während dem Einlesen jeder Befehl kommentiert. Wollen wir also `nat.scm` laden, so schreiben wir

```
(set! COMMENT-FLAG #f)
(libload "nat.scm")
(set! COMMENT-FLAG #t).
```

Man beachte, dass es nicht möglich ist, für `NAME` eine Liste von Dateien anzugeben, die geladen werden sollen. Es kann immer nur eine Datei angegeben werden.
Möchte man Dateien nicht nur aus dem Ordner `lib` laden, gibt es dafür den allgemeinen Befehl

```
(load FILE),
```

wobei man für `FILE` die Adresse der Datei angibt. Zum Beispiel ist die Eingabe von `(load "lib/nat.scm")` gleichwertig wie die Eingabe von `(libload "nat.scm")`. Man sieht also, dass man bei der Adresse der Datei vom Ordner, in dem sich die Minlogdatei befindet, ausgeht.

2.1.7 Beweise in der Prädikatenlogik

Wir wollen nun als Nächstes Aussagen mit einem Allquantor beweisen. Als erstes Beispiel nehmen wir dafür die Formel $\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn$, wobei n eine Variable vom Typ der natürlichen Zahlen ist und P, Q einstellige Prädikatenvariablen sind, die als Argument eine natürliche Zahl erwarten. Hierzu binden wir zunächst die Bibliotheksdatei `nat.scm` ein, so wie oben gezeigt. In dieser Datei wird unter vielen anderen festgelegt, dass die Algebra der natürlichen Zahlen mit `nat` bezeichnet wird, und dass `n` und `m` Variablen vom Typ der natürlichen Zahlen sind. Die Prädikatenvariablen `P` und `Q` definieren wir durch den Befehl

```
(add-pvar-name "P" "Q" (make-arity (py "nat"))).
```

Nun können wir die Zielformel mit `set-goal` setzen. Dabei wird der Allquantor mit `all` bezeichnet und hat als erstes Argument die Variable und als zweites Argument die Aussage, über die quantifiziert wird. Wir schreiben also

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n")
```

und erhalten

```
------------------------------------------------------------------
?_1:all n(P n -> Q n) -> all n P n -> all n Q n
```

als Ausgabe. Zunächst befördern wir wieder die zwei Prämissen in den Kontext mittels

```
(assume "Annahme1" "Annahme2").
```

Nun möchten wir eine Allaussage beweisen. Dazu nehmen wir eine Variable, die noch nicht vergeben ist, in den Kontext und beweisen dann die Formel, die auf diese Variable spezialisiert wurde. Auch das geschieht mittels dem Befehl `assume`. Wobei wir dieses Mal als Argument einen Variablennamen für eine Variable vom Typ der natürlichen Zahlen schreiben müssen. Wir schreiben daher

```
(assume "m")
```

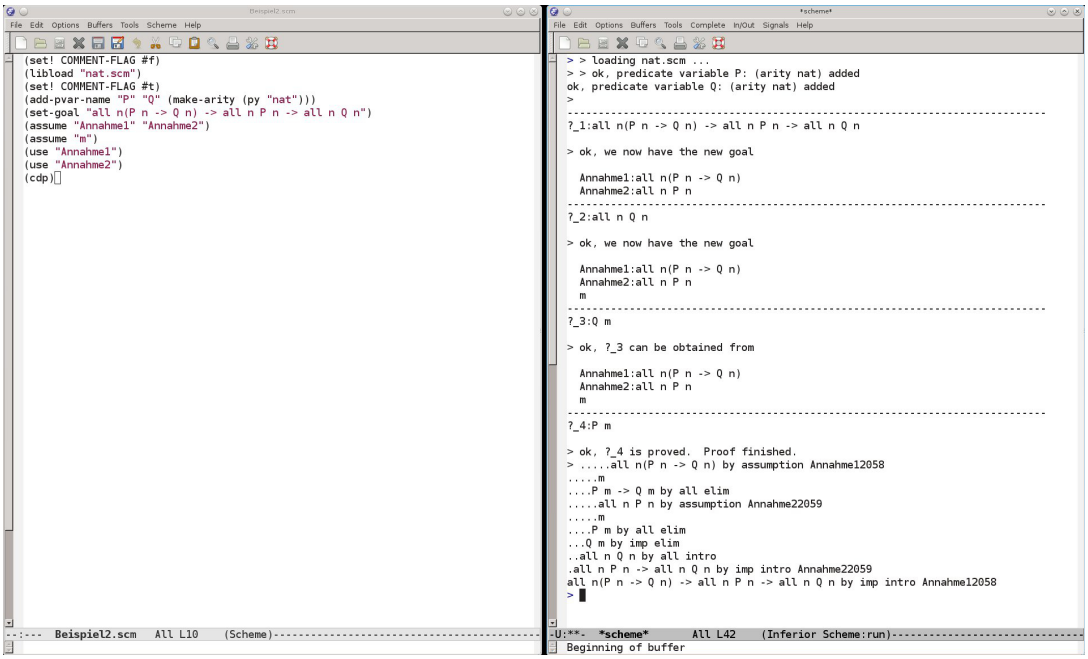
und das Programm gibt

```
ok, we now have the new goal

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_3:Q m
```

zurück. Wir hätten natürlich auch `(assume "n")` schreiben können und dann die Zielformel `Q n` erhalten. Es muss sich aber immer um eine Variable vom Typ `nat` handeln. `(assume "a")` würde beispielsweise eine Fehlermeldung geben, weil a keine Variable vom Typ `nat` ist.
Um nun `Q m` zu beweisen, verwenden wir `Annahme1`, da diese `Q n` als Konklusion hat und über `n` quantifiziert ist. Dies machen wir mit `(use "Annahme1")`. Das Programm erkennt, dass nun für das `n` in `Annahme1` das fixierte `m` eingesetzt wird und möchte einen Beweis für die Prämisse `P m`.

```
 ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_4:P m
```

Diesen geben wir mit (use "Annahme2") ein, was den Beweis auch schon beendet. Wir verwenden nun noch den Befehl (cdp), um uns die Herleitung anzeigen zu lassen.



2.1.8 use-with

Manchmal kann Minlog nicht sofort erkennen, wie es eine Formel, die mit dem use-Befehl verwendet wird, spezialisieren soll, um die Zielformel zu erhalten. Ein Beispiel dafür sieht man am Ende des Beweises von Abschnitt 2.2.4. In solchen Fällen muss man dem Programm direkt sagen, wie es eine Formel zu spezialisieren hat. Dafür gibt es das Kommando

(use-with NAME LISTE).

Für NAME setzt man den Namen der Formel ein, die man spezialisieren möchte. Anschließend folgt anstelle von LISTE eine Liste von Formelnamen und Termen. Dabei wird für jeden Allquantor der entsprechende Spezialisierungsterm mit Hilfe von (pt TERM) angegeben und für jede Prämisse muss der Name einer Formel angegeben werden, der diese Prämisse beweist. Wahlweise kann man anstelle einer Formel auch "?" schreiben, dann muss diese Prämisse noch anschließend bewiesen werden.
Betrachten wir dafür als Beispiel den Beweis aus dem letzten Abschnitt. Die letzten beiden Befehle waren (use "Annahme1") und (use "Annahme2"). Bevor diese Kommandos eingegeben wurden, hatten wir folgende Aufforderung des Systems:

```
ok, we now have the new goal

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------------
?_3:Q m
```

Wir erkennen, dass eine Spezialisierung von Annahme1 auf m zu der Formel P m -> Q m führt. Die Formel P m haben wir noch nicht bewiesen, sollte also unser nächstes Ziel werden. Deswegen können wir anstelle von (use "Annahme1") auch (use-with "Annahme1" (pt "m") "?") schreiben und erhalten dann genau wie vorher die Ausgabe

```
 ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
-----------------------------------------------------------------
?_4:P m
```

zurück. Man beachte eben, dass für jeden Implikationspfeil und für jeden Allquantor genau ein Argument in use-with erwartet wird und dies in genau der entsprechenden Reihenfolge wie auch in der Formel. Die Formel P m können wir nun entweder wieder durch (use "Annahme2") beweisen oder durch das etwas längere (use-with "Annahme2" (pt "m")).
Man erkennt dabei auch, dass use eine automatisierte Form von use-with ist, sodass man use-with nur dann verwenden muss, wenn use vom System nicht richtig erkannt wird.

2.1.9 inst-with

Mit dem Befehl inst-with kann man ähnlich zu dem Befehl use-with eine Aussage spezialisieren. Die spezialisierte Aussage muss bei inst-with nicht gleich der Zielformel sein. Sie wird auch nicht sofort verwendet, sondern im Kontext abgespeichert. Der Befehl hat die Form

```
(inst-with NAME LISTE).
```

Analog zu use-with soll für NAME der Name einer Formel und für LISTE eine Liste von Formelnamen und Termen eingesetzt werden, mit denen die Formel NAME dann spezialisiert wird. Nehmen wir als Beispiel wieder die gleiche Ausgangssituation wie im vorherigen Abschnitt:

```
ok, we now have the new goal

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
-----------------------------------------------------------------
?_2:Q m
```

Wir können hier zunächst Annahme2 auf m spezialisieren, indem wir

```
(inst-with "Annahme2" (pt "m"))
```

eingeben und dann

```
ok, ?_2 can be obtained from

  Annahme1:all n(P n -> Q n)
```

```
  Annahme2:all n P n
  m  3:P m
------------------------------------------------------------------
?_3:Q m
```

erhalten. Wir sehen, dass nun die spezialisierte Formel im Kontext erscheint und mit 3 markiert wurde. Wollen wir später auf diese Annahme zugreifen, wird der Name 3 nicht in Anführungszeichen geschrieben. Da diese Nummerierung einer Formel nichts über die Formel selbst aussagt, möchte man ihr häufig einen anderen Namen geben als nur eine Zahl. Dafür gibt es den erweiterten Befehl

```
(inst-with-to NAME LISTE NAME1).
```

Dieses Kommando verhält sich bei den Argumenten `NAME` und `LISTE` analog wie `inst-with`. Der Unterschied zu `inst-with` ist, dass `NAME1` nun ein Name für die neue Formel im Kontext ist. Wir können damit zum Beispiel `Annahme1` auf unsere Zielformel spezialisieren, indem wir

```
(inst-with-to "Annahme1" (pt "m") 3 "Zielformel")
```

eingeben. Die Ausgabe ist dann wie erwartet

```
ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m  3:P m
  Zielformel:Q m
------------------------------------------------------------------
?_5:Q m
```

und mit `(use "Zielformel")` lässt sich der Beweis beenden.

2.1.10 assert und cut

In Beweisen, insbesondere in längeren Beweisen, werden häufig zunächst erst Zwischenresulate bewiesen, aus denen dann die eigentliche Aussage gefolgert wird. Das wird in informalen Beweisen oft durch Ausdrücke wie „Es reicht zu zeigen, dass . . . gilt. “ oder „Wir beweisen zunächst . . . .“ gemacht. In Minlog gibt es dafür die Befehle

```
(assert FORMEL)
```

und

```
(cut FORMEL).
```

In beiden Fällen teilen wir dem System mit, dass wir die Zielformel `ZIEL` dadurch beweisen möchten, indem wir einen Beweis von `FORMEL` und einen Beweis von `FORMEL -> ZIEL` liefern. Der Unterschied zwischen den beiden Befehlen liegt darin, dass bei `assert` zuerst `FORMEL` und dann `FORMEL -> ZIEL` gezeigt werden muss, während es bei `cut` genau umgekehrt ist.
Als Beispiel verwenden wir wieder den Beweis der Aussage $\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn$ aus dem letzten und vorletzten Abschnitt. Im letzten Abschnitt haben wir nach der Ausgabe

```
 ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_4:P m
```

den Befehl `(use-with "Annahme1" (pt "m") "?")` verwendet. Dabei mussten wir als letztes Argument `"?"` schreiben, da wir die Prämisse `P m` noch nicht gegeben hatten. Mit `assert` oder `cut` können wir uns zunächst diese Prämisse beschaffen, sodass wir diese dann anstelle von `"?"` schreiben können. Wir geben daher `(assert "P m")` ein und erhalten dann folgende Ausgabe:

```
ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_4:P m -> Q m


  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_5:P m
```

Die beiden neuen Ziele `P m -> Q m` und `P m` können wir mit `(use "Annahme2")` sowie `(use "Annahme1")` gleich beweisen.
Wenn wir `(cut "P m")` anstelle von `(assert "P m")` eingeben, erhalten wir

```
ok, ?_3 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_5:P m



  Annahme1:all n(P n -> Q n)
  Annahme2:all n P n
  m
------------------------------------------------------------------
?_4:P m -> Q m
```

zurück. Dies sind dieselben Formeln in vertauschter Reihenfolge.

2.1.11 Beweissuche

Minlog ist in der Lage, auch selbst nach einem Beweis für eine Aussage zu suchen. Dies ist selbstverständlich nur bei Aussagen mit einfachen Beweisen von Erfolg gekrönt. Der Befehl für das Suchen eines Beweises zum gesetzten Ziel lautet

```
(search).
```

Da wir in den letzten Kapiteln einige verschiedene und auch unnötig komplizierte Möglichkeiten angegeben haben, um die Aussage `all n(P n -> Q n) -> all n P n -> all n Q n` in Minlog zu beweisen, geben wir nun die kürzeste Möglichkeit an: Nach der Eingabe von

```
(set-goal "allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n")
(search)
```

erhalten wir direkt die Ausgabe

```
ok, ?_1 is proved by minimal quantifier logic.  Proof finished.
```

Der Beweis ist damit schon fertig. Dieser sehr angenehme Befehl führt leider häufig nicht zum gewünschten Ergebnis. Die Wahrscheinlichkeit, dass ein Beweis gefunden wird, sinkt insbesondere dann, wenn für den Beweis zuvor abgespeicherte Theoreme oder globale Annahmen gebraucht werden, oder, wenn Minlog einen Zeugen für eine Existenzaussage finden müsste.
Manchmal hat man nicht nur ein Ziel sondern mehrere offene Ziele, die alle leicht zu beweisen sind. Für diese Fälle gibt es den Befehl

```
(auto).
```

Dieser Befehl ruft so lange den Befehl `search` auf, bis der Beweis beendet ist oder `search` keinen Beweis mehr für die Zielformel findet. Mit `auto` ist es also möglich, mehrere offene Zielformeln mit einem einzigen Befehl abzuarbeiten.

2.1.12 Cheaten in Minlog

Häufig ist es der Fall, dass Aussagen, die für einen Menschen vollkommen klar sind, sehr lange brauchen, bis man diese formal am Computer bewiesen hat. Hier empfiehlt es sich oft, diese Aussage erst als globale Annahme festzulegen und dann möglicherweise später zu beweisen, um sich auf den wesentlichen Beweis zu konzentrieren. Mit dem Befehl

```
(add-global-assumption NAME FORMEL)
```

wird die Formel `FORMEL` unter dem Namen `NAME` als globale Annahme festgelegt. Diese ist dann im System abgespeichert und kann in jedem Beweis verwendet werden, zum Beispiel durch den `use`-Befehl. Mit dem Befehl

```
(display-global-assumptions NAME)
```

wird die globale Annahme mit Namen `NAME` angezeigt. Anstelle von `NAME` kann man auch eine Liste von Namen globaler Annahmen eingeben, dann werden alle entsprechenden Annahmen aufgelistet. Ist die Liste leer, das heißt, schreiben wir nur `(display-global-assumptions)`, so erscheint jede globale Annahme. Geben wir dies direkt nach dem Start von Minlog ein, erhalten wir die folgende Ausgabe:

```
Stab ((Pvar -> F) -> F) -> Pvar
Efq F -> Pvar
StabLog ((Pvar -> bot) -> bot) -> Pvar
EfqLog bot -> Pvar
```

Wir sehen also, dass das Ex-falso-quodlibet und die Stabilität als globale Annahmen von Beginn an in Minlog eingespeichert sind. Mit dem Kommando

```
(remove-global-assumption LISTE)
```

werden alle globalen Annahmen in der Liste LISTE entfernt. Wollen wir also Minlog ohne globale Annahmen verwenden, so können wir dies durch

```
(remove-global-assumption "Stab" "Efq" "StabLog" "EfqLog")
```

erreichen.
Oft bemerkt man in einem Beweis, dass man ein Teilziel dieses Beweises als globale Annahme setzen möchte. Für solche Fälle ist der Befehl

```
(admit)
```

geeignet. Durch die Eingabe dieses Kommandos wird die derzeitige Zielformel zu den globalen Annahmen hinzugefügt und gilt in diesem Beweis als gezeigt. Wir können auf diese Weise beliebige Aussagen in Minlog zeigen, was den Titel dieses Abschnittes rechtfertigt. Deshalb ist es ratsam, in fertigen Beweisen möglichst selten globale Annahmen oder `admit` zu verwenden.

2.1.13 In Minlog suchen

Beim Schreiben eines Beweises zu einem bestimmten Thema, zum Beispiel über die Addition von natürlichen Zahlen, möchte man häufig wissen, welche Aussagen bereits zur Verfügung stehen. Eine Liste solcher Aussagen kann man durch den Befehl

```
(search-about LISTE)
```

erhalten. Dabei gibt man für LISTE eine Liste von Strings ein und Minlog gibt dann die Liste von genau den Theoremen und die Liste von genau den globalen Annahmen aus, deren Namen alle diese Strings enthalten. Speichert man also Theoreme und globale Annahmen ab, so empfiehlt es sich, einen aussagekräftigen Namen zu verwenden, auch wenn dieser dadurch lang werden könnte.
Haben wir zum Beispiel die Datei `nat.scm` geladen und wollen etwas über die Addition auf den natürlichen Zahlen beweisen, können wir mit

```
(search-about "Nat" "Plus")
```

zunächst sehen, was schon alles über die Addition von natürlichen Zahlen gegeben ist:

Die Liste ist eigentlich viel länger. Anstelle der Punkte stehen natürlich noch viel mehr Theoreme.

```
Theorems with name containing Nat and Plus
NatPlusDouble
all n,m NatDouble n+NatDouble m=NatDouble(n+m)
.
.
.
NatPlus0CompRule
all n^ n^ +0 eqd n^
No global assumptions with name containing Nat and Plus
```

Wir finden also einige Theoreme über die Addition auf natürlichen Zahlen und sehen ebenso, dass es darüber keine globalen Annahmen gibt.

2.2 Algebren und induktiv definierte Prädikate

In diesem Abschnitt soll es darum gehen, wie wir Algebren und induktiv definierte Prädikate, die im ersten Kapitel definiert wurden, in Minlog implementieren können.

2.2.1 Algebren

Wir erinnern uns an Kapitel 1. Dort hatte eine Algebra die Form

$$\mu_\xi(\kappa_0, \ldots, \kappa_{k-1}).$$

Der Befehl, um eine Algebra in Minlog einzuführen, sieht in allgemeiner Form wie folgt aus:

```
(add-algs NAME LISTE)
```

Für `NAME` setzt man dabei den Namen der Algebra in Anführungszeichen ein. Anstelle von `LISTE` steht eine Liste von Paaren

```
'(NAME TYP)
```

mit einer Bezeichnung `NAME` für den Konstruktor vom Typ `TYP`. Man sieht, dass bei der Definition der Algebra bereits eine Bezeichnung für diese und eine Bezeichnung für alle ihre Konstruktoren festgelegt werden muss. Betrachten wir dazu einige Beispiele:
In der Datei `nat.scm` ist die Algebra der natürlichen Zahlen durch

```
(add-algs "nat" '("Zero" "nat") '("Succ" "nat=>nat"))
```

eingeführt. Die Null wird also mit `Zero` und der Nachfolger mit `Succ` bezeichnet. Eine im System eingespeicherte Algebra können wir uns mit dem Befehl

```
(display-alg NAME),
```

wobei statt `NAME` der Name der Algebra in Anführungszeichen eingesetzt wird, anzeigen lassen. Wir erhalten dann eine Liste mit den Konstruktoren der Algebra und ihrer Typen. Bei `(display-alg "nat")` erhalten wir als Ausgabe:

```
nat
Zero: nat
Succ: nat=>nat
```

Die boolesche Algebra ist bereits standardmäßig in Minlog mit dem Namen `boole` eingespeichert. Die beiden Konstruktoren sind dabei `True` und `False`. Das sehen wir auch, wenn wir `(display-alg "boole")` eingeben:

```
boole
True: boole
False: boole
```

Führen wir als weiteres Beispiel die Algebra der Ordinalzahlen ein: Da wir bereits die natürlichen Zahlen `nat` mit Konstruktoren `Zero` und `Succ` eingeführt haben, ist es nicht mehr möglich, die Konstruktoren der Ordinalzahlalgebra mit `Zero` oder `Succ` zu benennen. Wir führen sie daher wie folgt ein:

```
(add-algs "OrdNum" '("ZeroOrd" "OrdNum")
                   '("SuccOrd" "OrdNum=>OrdNum")
                   '("Sub" "(nat=>OrdNum)=>OrdNum"))
```

Es ist auch möglich Algebren mit Parameter einzuführen. In Minlog stehen die Zeichenketten `alpha`, `beta`, `gamma`, `alpha0`, `beta0`, `gamma0` und so weiter für Typvariablen. In der Bibliotheksdatei `list.scm` wird die Listenalgebra definiert durch

```
(add-algs "list"
  '("list" "Nil")
  '("alpha=>list=>list" "Cons")).
```

Der Listentyp hängt vom Typparameter `alpha` ab. Dieser muss auch immer explizit angegeben werden, was wir auch bei der Eingabe von `(display-alg "list")` sehen:

```
list
Nil: list alpha
Cons: alpha=>list alpha=>list alpha
```

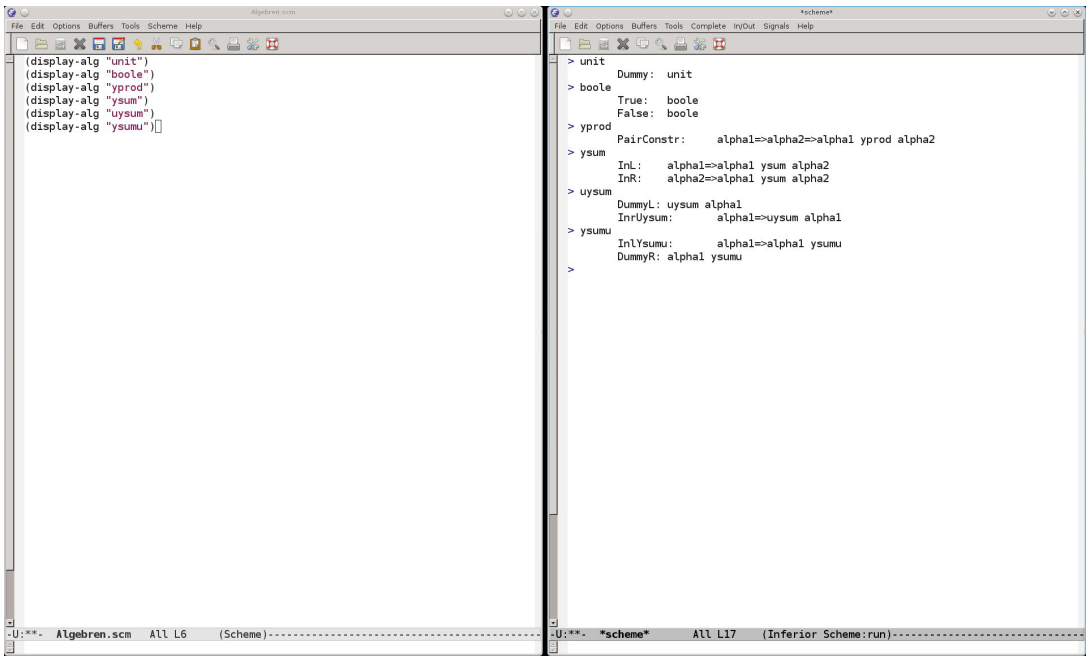
Dort wird der Parameter immer mitangegeben. Man sieht, dass der Parameter hinter dem Namen der Algebra steht. Möchte man lieber die Darstellung `alpha list` haben, so erreicht man dies mit dem Hinzufügen von `'postfix-typeop` nach dem Namen der Algebra. Beim Listentyp sieht dies wie folgt aus:

```
(add-algs "list" 'postfix-typeop
  '("list" "Nil")
  '("alpha=>list=>list" "Cons"))
```

Als Ausgabe bei `(display-alg "list")` haben wir dann

```
list
Nil: alpha list
Cons: alpha=>alpha list=>alpha list
```

In folgender Abbildung sehen wir noch weitere in Minlog vorab eingespeicherte Algebren.



2.2.2 Deklaration von Termvariablen

Wollen wir Variablen mit einem Typ angeben, muss dem System vorher mitgeteilt werden, welche Variable welchen Typ hat. Für jeden Typ mit dem Namen `TYP` ist vorprogrammiert, dass die gleiche Zeichenkette `TYP`, als Namen für eine Variable verwendet werden kann. Befehle wie

```
(set-goal "all nat nat=nat")
```

sind für Minlog also klar verständlich. Auch für Pfeiltypen wie zum Beispiel `nat=>nat` kann der Name auch als Variable aufgefasst werden.
Ist außerdem `v` als Variable vom Typ `TYP` deklariert, so werden auch `v0`, `v1` und so weiter als Variablen mit Typ `TYP` aufgefasst. Statt obigen Befehl können wir also äquivalent auch

```
(set-goal "all nat1000 nat1000=nat1000")
```

schreiben. Da der Name eines Typs sehr lang werden kann, ist es für die Lesbarkeit oft besser, zusätzliche Variablennamen zu deklarieren. Das geschieht durch den Befehl

```
(add-var-name NAMES TYP).
```

Für `NAMES` wird eine Liste von den gewünschten Variablennamen angegeben und anstelle von `TYP` wird der Typ der Variablen geschrieben. Für Letzteres benötigt man den Befehl `(py STRING)`, der dem Programm mitteilt, dass die Zeichenkette `STRING` als Typ zu interpretieren ist. In der Datei `nat.scm` werden `n`, `m` und `l` als Variablennamen für natürliche Zahlen durch den Befehl

```
(add-var-name "n" "m" "l" (py "nat"))
```

deklariert.

2.2.3 Induktiv definierte Prädikate

Induktiv definierte Prädikate werden durch den Befehl

```
(add-ids (list (list NAME (make-arity TYPS) ALGNAME)) LIST)
```

definiert. Anstelle von `NAME` steht die Bezeichnung des induktiv definierten Prädikats. Die Typen der Argumente des Prädikats werden für `TYPS` einsetzt. Die Bezeichnung für die Algebra zu diesem induktiv definierten Prädikat gemäß Definition 1.7.1 setzt man für `ALGNAME` ein. Die Einführungsregeln des Prädikats werden für `LIST` eingesetzt, wobei die einzelnen Regeln in der Form `'(FORMEL NAME)` eingegeben werden. Dabei ist `FORMEL` das jeweilige Einführungsaxiom und `NAME` dessen Bezeichnung.
Als Beispiel möchten wir auf den natürlichen Zahlen das einstellige induktiv definierte Prädikat `EvenI` einführen, welches angibt, dass eine natürlichen Zahl gerade ist. `EvenI` ist dadurch definiert, dass die Zahl 0 gerade ist und, wenn eine natürliche Zahl n gerade ist, so ist es auch $n+2$. In Minlog implementieren wir dies durch:

In Minlog ist die Dezimalschreibweise für natürliche Zahlen schon einprogrammiert, das heißt 0 anstelle von `Zero` oder 1 anstelle von `Succ Zero` versteht das Programm ohne Weiteres.

```
(add-ids
(list (list "EvenI" (make-arity (py "nat")) "algEvenI"))
'("EvenI 0" "InitEvenI")
'("all n(EvenI n -> EvenI(Succ (Succ n)))" "GenEvenI")).
```

Wir können uns nun mit `(display-alg "algEvenI")` die Algebra zu `EvenI` anzeigen lassen und erhalten

```
algEvenI
CInitEvenI: algEvenI
CGenEvenI: algEvenI=>algEvenI
```

als Ausgabe. Was auffällt und auch allgemein gilt, ist, dass der Konstruktor, der zur Einführungsregel mit Namen `Regel` gehört, von Minlog die Bezeichnung `CRegel` erhält.
Ebenso sieht man, dass die Konstruktoren von `nat` genau analog sind. Aus diesem Grund wäre es auch möglich bei der Einführung von `EvenI` anstelle von `algEvenI` auch `nat` zu schreiben. Schreibt man allgemein einen schon belegten Namen einer Algebra anstelle von `ALGNAME`, so wird überprüft, ob die Konstruktoren dieselben sind, und gegebenfalls wird dann diese Algebra und deren Konstruktoren verwendet.
Mit Hilfe des Befehls

```
(display-idpc NAME)
```

können wir uns zum induktiv definierten Prädikat `NAME` die Einführungsregeln und seine Algebra ausgeben lassen. Für das eben eingeführte Prädikat `EvenI` erhalten wir dann nach Eingabe von `(display-idpc "EvenI")` die folgende Ausgabe:

```
EvenI with content of type algEvenI
InitEvenI: EvenI 0
GenEvenI: all n(EvenI n -> EvenI(Succ(Succ n)))
```

Die Einführungsaxiome sind als Theoreme im System abgespeichert und lassen sich beispielsweise mit dem `use`-Befehl verwenden. Eine andere Möglichkeit ist auch der Befehl

```
(intro N).
```

Mit ihm verwendet man das `N`-te Einführungsaxiom des Prädikats in der Zielformel, wobei die Nummerierung mit 0 begonnen wird.

2.2.4 Beweis mit induktiv definierten Prädikaten

Wir definieren als Gegenstück zu dem eben eingeführten Prädikat `EvenI` das Prädikat `OddI`, welches angibt, dass eine natürliche Zahl ungerade ist:

```
(add-ids
(list (list "OddI" (make-arity (py "nat")) "algOddI"))
'("OddI 1" "InitOddI")
'("all n(OddI n -> OddI(Succ(Succ n)))" "GenOddI"))
```

Damit wollen wir nun den Satz beweisen, dass der Nachfolger einer geraden Zahl ungerade ist. Wir geben also

```
(set-goal "all n(EvenI n -> OddI(Succ n))")
```

ein. Mit `(assume "n" "EvenIn")` befördern wir die Variable `n` und die Prämisse `EvenI n` in den Kontext und erhalten

```
n  EvenIn:EvenI n
----------------------------------------------------------------------
?_2:OddI(Succ n)
```

als Ausgabe. Um das neue Ziel zu beweisen, verwenden wir das Eliminationsaxiom von EvenI auf das Prädikat {n|OddI(Succ n)}. Ist NAME der Name einer Formel, die nur aus einem induktiv definierten Prädikat besteht, dann teilen wir durch

```
(elim NAME)
```

dem Programm mit, dass wir das Eliminationsaxiom des induktiv definierten Prädikates verwenden wollen, um die Zielformel zu zeigen. Für den Parameter in dem Eliminationsaxiom wird die Zielformel eingesetzt, wobei genau die freien Variablen abstrahiert werden, die auch in der Formel NAME vorkommen. Aus Definition 1.4.2 wissen wir, dass

$$\forall_n(\texttt{EvenI}\ n \to P0 \to \forall_n(\texttt{EvenI}\ n \to Pn \to P(SSn)) \to Pn)$$

das Eliminationsaxiom mit Parameter P von EvenI ist. Das Programm wird also noch einen Beweis der Prämissen $P0$ und $\forall_n(\texttt{EvenI}\ n \to Pn \to P(SSn))$ fordern, wobei in unserem Fall $Pn := \texttt{OddI}\ Sn$ ist. In der Tat erhalten wir auch nach Eingabe von (elim "EvenIn") die Ausgabe

```
ok, ?_2 can be obtained from

  n  EvenIn:EvenI n
-------------------------------------------------------------------
?_4:all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))



  n  EvenIn:EvenI n
-------------------------------------------------------------------
?_3:OddI 1
```

Um OddI 1 zu beweisen, verwenden wir mit (use "InitOddI") das erste Einführungsaxiom von OddI. Für das Ziel ?_4 nehmen wir mittels

```
(assume "m" "Annahme1" "OddI1")
```

die Variable und beide Annahmen in den Kontext und wollen das neue Ziel

```
ok, we now have the new goal

  n  EvenIn:EvenI n
  m  Annahme1:EvenI m
  OddI1:OddI(Succ m)
-------------------------------------------------------------------
?_5:OddI(Succ(Succ(Succ m)))
```

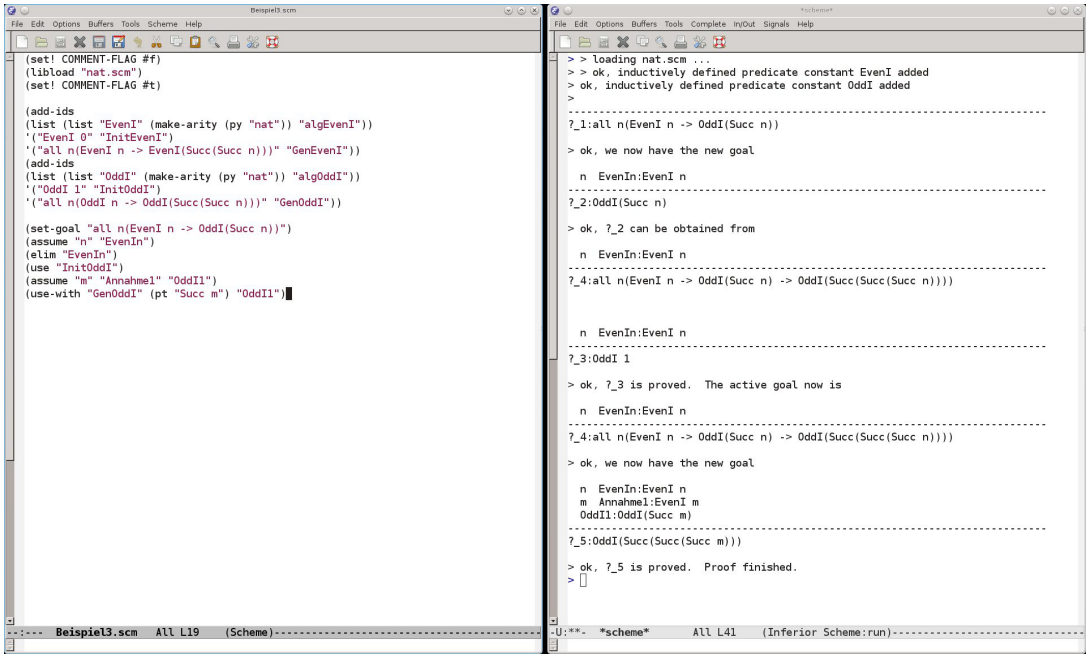
durch das zweite Einführungsaxiom vom OddI beweisen. Dieses ist gegeben durch

```
GenOddI: all n(OddI n -> OddI(Succ(Succ n))).
```

Da Minlog mit einem einfachen use-Befehl jedoch nicht erkennt, wie dieses Axiom angewendet werden soll, müssen wir use-with verwenden und geben daher

```
(use-with "GenOddI" (pt "Succ m") "OddI1")
```

ein. Das beendet den Beweis direkt.



Wollen wir uns den Beweis mit (cdp) anzeigen lassen, sieht dies zwar etwas unübersichtlich aus,

```
......allnc n^3824(EvenI n^3824 -> OddI 1 -> all n(EvenI n ->
      OddI(Succ n) -> OddI(Succ(Succ(Succ n)))) ->
      OddI(Succ n^3824)) by axiom Elim
......n
.....EvenI n -> OddI 1 -> all n(EvenI n -> OddI(Succ n) ->
     OddI(Succ(Succ(Succ n)))) -> OddI(Succ n)
     by allnc elim
.....EvenI n by assumption EvenIn2064
....OddI 1 -> all n(EvenI n -> OddI(Succ n) ->
    OddI(Succ(Succ(Succ n)))) -> OddI(Succ n) by imp elim
....OddI 1 by axiom Intro
...all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))
    -> OddI(Succ n) by imp elim
........all n(OddI n -> OddI(Succ(Succ n))) by axiom Intro
........Succ m
.......OddI(Succ m) -> OddI(Succ(Succ(Succ m))) by all elim
.......OddI(Succ m) by assumption OddI12068
......OddI(Succ(Succ(Succ m))) by imp elim89
.....OddI(Succ m) -> OddI(Succ(Succ(Succ m)))
     by imp intro OddI12068
....EvenI m -> OddI(Succ m) -> OddI(Succ(Succ(Succ m)))
    by imp intro Annahme12067
...all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))
   by all intro
..OddI(Succ n) by imp elim
.EvenI n -> OddI(Succ n) by imp intro EvenIn2064
all n(EvenI n -> OddI(Succ n)) by all intro
```

wir erkennen aber dennoch, dass die erste Formel durch Elimination folgt und die Formeln `OddI 1` sowie `all n(OddI n -> OddI(Succ(Succ n)))` durch Einführung folgen. Das passt auch damit zusammen, dass wir genau einmal das Eliminationsaxiom und zweimal ein Einführungsaxiom verwendet haben.

2.3 Dekorationen in Minlog

Beim Umgang mit den dekorierten logischen Symbolen ist es sehr nützlich, einen Computer als Unterstützung zu verwenden. Denn, wie man in 1.6.2 sieht, muss bei der $\to^{nc}$- bzw. $\forall^{nc}$-Einführungsregel die Menge der rechnerischen Annahmen bzw. die Menge der rechnerischen Variablen bestimmt werden. Mit Computerhilfe lässt sich das leicht erledigen, während es mit Stift und Papier sehr lange dauern kann und damit auch fehleranfällig ist.

2.3.1 Der nicht-rechnerische Allquantor

Wir gehen zunächst auf den nicht-rechnerischen Allquantor ein. In Minlog wird dieser mit `allnc` bezeichnet und hat die gleichen syntaktischen Regeln wie der gewöhnliche Allquantor. Im Abschnitt 2.1.7 haben wir in Minlog die Formel $\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn$ bewiesen. Jetzt möchten wir die dekorierte Form $\forall_n^{nc}(Pn \to Qn) \to \forall_n^{nc} Pn \to \forall_n^{nc} Qn$ dieser Aussage beweisen. Dazu laden wir zunächst wieder `nat.scm` aus der Bibliothek und deklarieren die Prädikate `P` und `Q` genau wie in Abschnitt 2.1.7. Dann geben wir

```
(set-goal "allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n")
```

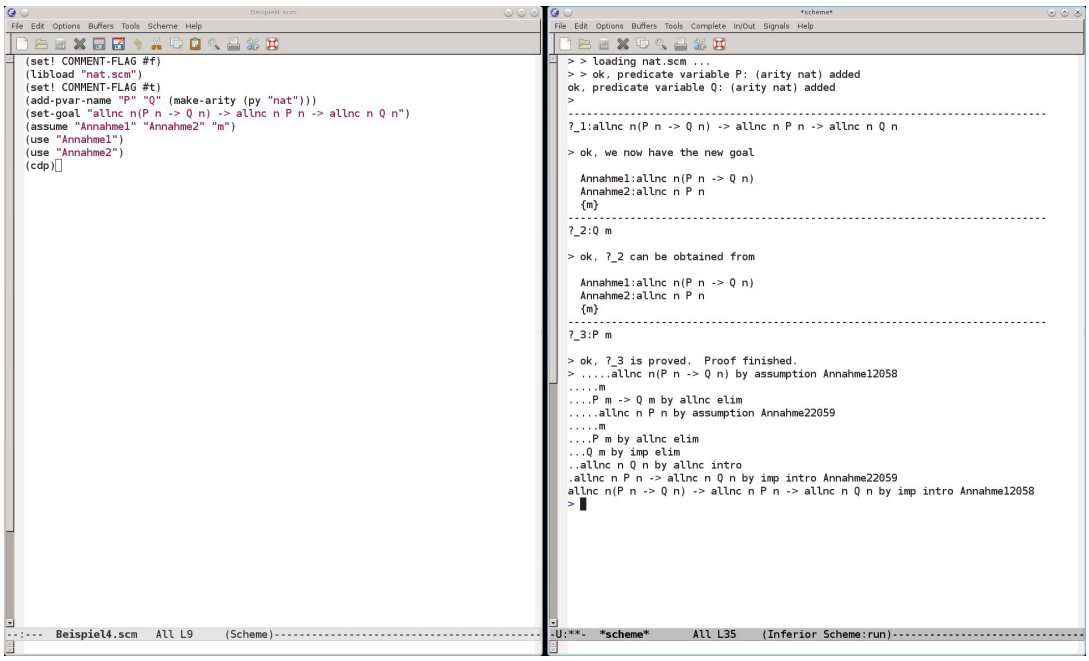
ein, um unsere Zielformel festzulegen. Die Ausgabe des Programms ist dann wie erwartet

```
------------------------------------------------------------------
?_1:allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n
```

Mittels `(assume "Annahme1" "Annahme2" "m")` befördern wir die beiden Prämissen und die Variable `m` in den Kontext. Die Ausgabe des Programms ist fast identisch zu dem Beispiel mit dem gewöhnlichen Allquantor aus Abschnitt 2.1.7.

```
ok, we now have the new goal

  Annahme1:allnc n(P n -> Q n)
  Annahme2:allnc n P n
  {m}
------------------------------------------------------------------
?_2:Q m
```

Es gibt jedoch den Unterschied, dass um die Variable `m` im Kontext geschweifte Klammern erscheinen. Dies teilt uns mit, dass wir `m` nicht rechnerisch verwenden sollen. Das bedeutet, wir dürfen `m` in keinem Term verwenden, der auf einen rechnerischen Allquantor angewendet wird. In unserem Fall sind jedoch die beiden Annahmen im Kontext nicht-rechnerische Allaussagen, sodass wir diese ohne Weiteres auf `m` spezialisieren können. Wir schreiben daher `(use "Annahme1")` gefolgt von `(use "Annahme2")`, was den Beweis beendet. Die folgende Abbildung zeigt noch einmal den gesamten Beweis in Minlog und die Ausgabe nach Eingabe von `(cdp)`.



Wir sehen, dass hier nun die Regeln `allnc elim` und `allnc intro` auftreten. Beim Umgang mit dem nicht-rechnerischen Allquantor in Minlog sollte man beachten, dass Minlog lediglich eine Warnung ausgibt, wenn wir eine nicht-rechnerische Variable rechnerisch verwenden wollen. Den Beweis können wir trotzdem weiterführen. Als Beispiel hierfür machen wir in der oben bewiesenen Aussage den ersten nicht-rechnerischen Allquantor zu einem gewöhnlichen Allquantor und schreiben

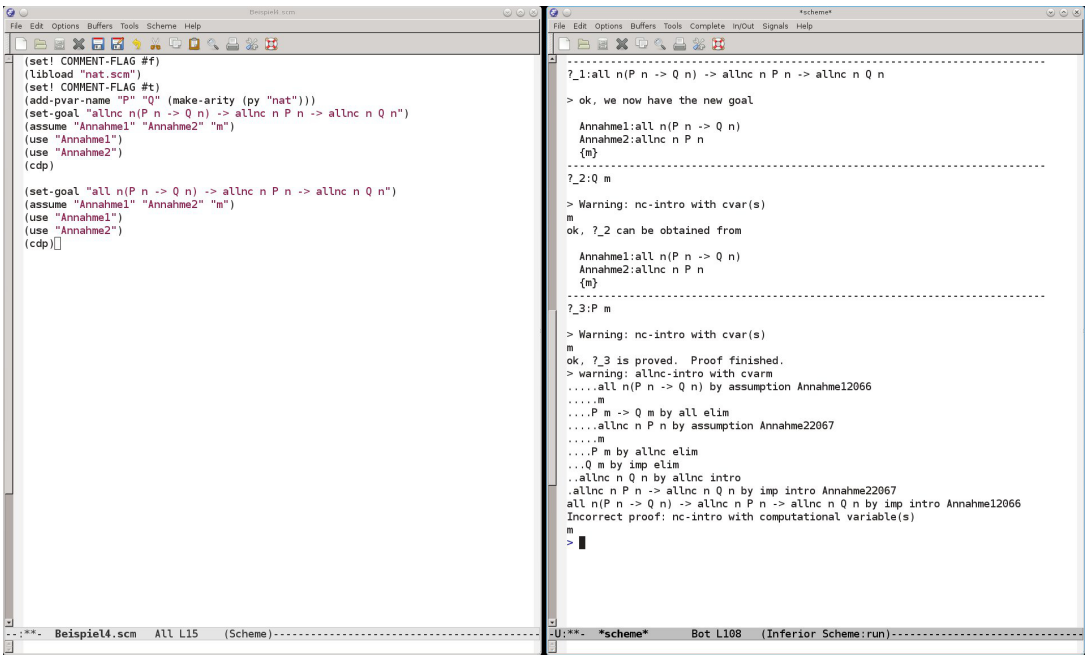
```
(set-goal "all n(P n -> Q n) -> allnc n P n -> allnc n Q n").
```

Auch hier verwenden wir wieder `(assume "Annahme1" "Annahme2" "m")`. Geben wir nun aber `(use "Annahme1")` ein, erhalten wir keine Fehlermeldung. Es erscheint am Anfang nur eine Warnung:

```
Warning: nc-intro with cvar(s)
m
ok, ?_2 can be obtained from

  Annahme1:all n(P n -> Q n)
  Annahme2:allnc n P n
  {m}
-----------------------------------------------------------------------
?_3:P m
```

Dass dies nicht direkt zu einer Fehlermeldung führt, liegt darin begründet, dass an dieser Stelle noch nicht klar ist, ob es sich wirklich um eine falsche Anwendung handelt. So würde man möglicherweise korrekte Beweise vorzeitig abbrechen. Außerdem gibt es von der logischen Struktur aus betrachtet keinen Unterschied zwischen $\forall$ und $\forall^{nc}$. Insbesondere wäre der Beweis ohne Dekorationen natürlich korrekt. Beenden wir nun den „Beweis“ und verwenden `(cdp)`, so erhalten wir auch einen Beweisbaum, bekommen aber mitgeteilt, dass der „Beweis“ inkorrekt ist, weil wir eine nicht-rechnerische Alleinführung mit einer rechnerischen Variablen gemacht haben.

Hier sieht man auch einen Unterschied, zwischen (cdp) und (dp). Gibt man (dp) ein, wird nur der Beweisbaum ausgegeben und nicht gemeldet, dass dieser inkorrekt ist.



2.3.2 Der nicht-rechnerische Implikationspfeil

Dem aufmerksamen Leser wird auffallen, dass in der Zielformel ein Implikationspfeil rechnerisch ist. Das ist auch notwenig, ansonsten kann man die Aussage für allgemeines C nicht beweisen.

Der nicht-rechnerische Implikationspfeil wird in Minlog mit --> bezeichnet und besitzt die gleichen syntaktischen Regeln wie ->. Als Anwendungsbeispiel möchten wir in Minlog zeigen, dass die nicht-rechnerische Implikation transitiv ist. Dazu führen wir mit (add-pvar-name "A" "B" "C" (make-arity)) drei Aussagenvariablen ein und setzen dann mit

```
(set-goal "(A --> B) --> (B --> C) -> A --> C")
```

unser Ziel. Die drei Prämissen bringen wir wieder in den Kontext, indem wir den Befehl (assume "Annahme1" "Annahme2" "Annahme3") eingeben.

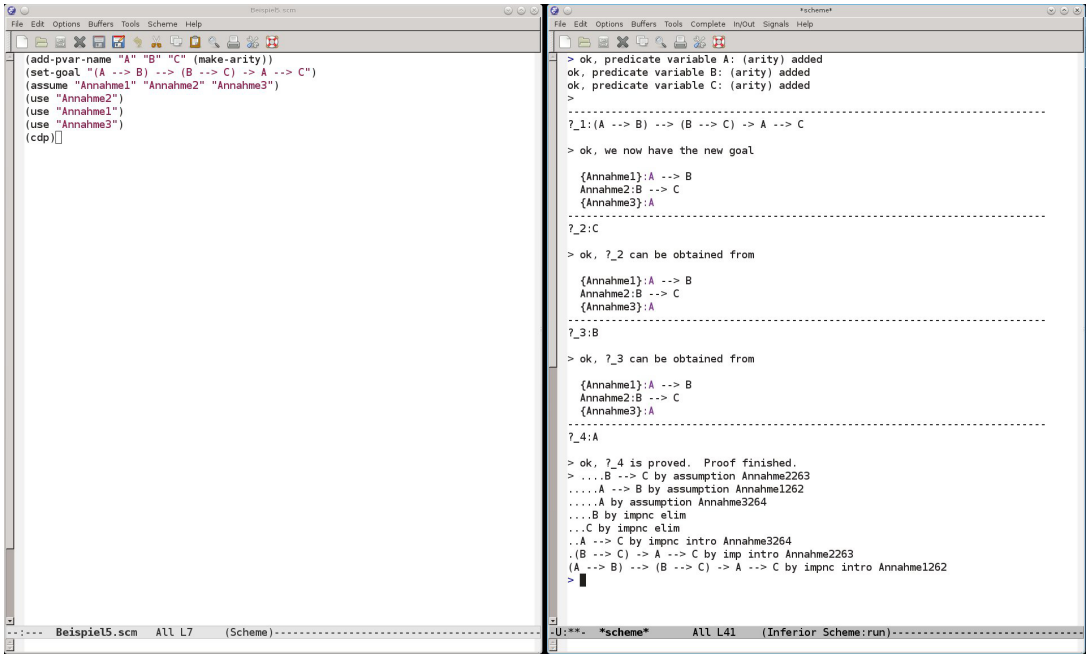
```
ok, we now have the new goal

  {Annahme1}:A --> B
  Annahme2:B --> C
  {Annahme3}:A
----------------------------------------------------------------------
?_2:C
```

Wir sehen, dass nun alle Annahmen, die nicht-rechnerisch eingingen, in geschweiften Klammern stehen. Die Annahmen in geschweiften Klammern dürfen nur in einem nicht-rechnerischen Teil des Beweises verwendet werden. Ob man sich in einem nicht-rechnerischen Teil eines Beweises befindet, wird jedoch vom System selbst nicht angezeigt und muss vom Benutzer überprüft werden. Unser Beweis wird nun damit fortgesetzt, dass wir Annahme2 verwenden. Nach Annahme2 impliziert B nicht-rechnerisch C, deswegen befinden wir uns nach der Eingabe von (use "Annahme2") in einem nicht-rechnerischen Teil des Beweises. Wir dürfen, um B zu zeigen, somit alle Annahmen im Kontext verwenden. Daher können wir mit (use "Annahme1") den Beweis fortsetzen und haben die Ausgabe

```
ok, ?_3 can be obtained from
```

```
  {Annahme1}:A --> B
  Annahme2:B --> C
  {Annahme3}:A
------------------------------------------------------------------------
?_4:A
```

Hier befinden wir uns weiterhin in einem nicht-rechnerischen Teil des Beweises, da wir immer noch das Teilziel `B` zeigen wollen, welches nicht rechnerisch eingeht. Daher beenden wir den Beweis mit `(use "Annahme3")`.
Eine Anwendung von `(cdp)` zeigt uns, dass in dem Beweis die Regeln `impnc elim` und `impnc intro` verwendet wurden, welche genau die Regeln zum nicht-rechnerischen Implikationspfeil sind. Da wir keine Fehlermeldung bei `(cdp)` bekommen, wurde jede Regel korrekt angewandt.



Hätten wir aber jeden Implikationspfeil in der Zielformel nicht-rechnerisch gemacht, würden wir keinen gültigen Beweis erhalten. Denn die Aussage `C` kann rechnerischen Gehalt haben, würde jedoch keine Prämisse rechnerisch eingehen, stellt sich die Frage, woher dieser rechnerische Gehalt kommen soll.
Analog zum nicht-rechnerischen Allquantor würden wir auch hier nur eine Warnung erhalten, falls wir eine Annahme, die nicht-rechnerisch eingehen soll, in einem rechnerischen Teil des Beweises verwenden würden.

2.3.3 Dekorierte Prädikate

Den nicht-rechnerischen Allquantor und Implikationspfeil kann man auch bei der Definition von induktiv definierten Prädikaten verwenden. In Abschnitt 2.2.3 haben wir das Prädikat `EvenI` definiert. Bei der letzten Regel `GenEvenI`, die durch

```
all n(EvenI n -> EvenI(Succ (Succ n)))
```

gegeben ist, macht es beispielsweise Sinn, den Allquantor durch einen nicht-rechnerischen Allquantor zu ersetzen, denn heuristisch betrachtet, ist die Information über `n` auch bereits in `EvenI n` enthalten. Die Regel `GenEvenI` hat dann die Form

```
allnc n(EvenI n -> EvenI(Succ (Succ n))).
```

Im Vergleich zu den induktiv definierten Prädikaten ohne Dekorationen ändert sich nichts Wesentliches.
Auch beim Einführen eines nicht-rechnerischen induktiv definierten Prädikats gibt es wenig Unterschied. Will man ein induktiv definiertes Prädikat als nicht-rechnerisch einführen, so tut man dies dadurch, dass man keinen Namen für die Algebra des Prädikats eingibt.
Um ein schönes Beispiel für nicht-rechnerische induktiv definierte Prädikate angeben zu können, benutzen wir an dieser Stelle die Algebra der Listen. Diese wird in der Bibliotheksdatei `list.scm` eingeführt. Wir wollen das Prädikat `RevI` mit zwei Listen als Argumente definieren, welches angeben soll, dass die erste Liste rückwärts gelesen die zweite Liste ergibt. Dazu laden wir zunächst die Dateien `nat.scm` und `list.scm` ein. Der Listentyp wird mit einer Typvariablen `alpha` definiert und ist in Abschnitt 2.2.1 gegeben. Wir führen zunächst zwei Variablen `xs` und `ys` vom Typ `list alpha` und eine Variable `x` vom Typ `alpha` ein. In der Datei `list.scm` wird für den Konstruktor `Cons alpha` die Abkürzung :: als Infixnotation eingeführt. Für `x :: (Nil alpha)` wurde auch die Abkürzung `x:` eingeführt und mit ++ wird die Verkettung von zwei Listen bezeichnet. Wir führen damit unser induktiv definiertes Prädikat `RevI` wie folgt ein:

```
(add-ids (list (list "RevI" (make-arity (py "list alpha")
                                        (py "list alpha"))))
'("RevI(Nil alpha)(Nil alpha)" "InitRevI")
'("all xs,ys,x(RevI xs ys -> RevI(xs++x:)(x::ys))" "GenRevI"))
```

Als Anwendung davon, beweisen wir nun, dass `RevI` symmetrisch ist. Wir geben also

```
(set-goal "all xs,ys(RevI xs ys -> RevI ys xs)")
```

in Minlog ein. Nachdem wir mit `(assume "xs" "ys" "RevIxsys")` die Prämissen in den Kontext gesetzt haben, ist die Ausgabe des Systems:

```
ok, we now have the new goal

  xs  ys  RevIxsys:RevI xs ys
-------------------------------------------------------------------
?_2:RevI ys xs
```

Die einzige Möglichkeit, diese Aussage zu beweisen, besteht offenbar darin, das Eliminationsaxiom zusammen mit der Annahme `RevIxsys` auf die Zielformel zu verwenden. Da `RevI` ein nicht-rechnerisches induktiv definiertes Prädikat ist, muss die Zielformel beim Verwenden von `elim` nicht-rechnerisch sein. Das ist hier der Fall. Deswegen geben wir `(elim "RevIxsys")` ein und erhalten die Ausgabe

```
ok, ?_2 can be obtained from

  xs  ys  RevIxsys:RevI xs ys
-------------------------------------------------------------------
?_4:all xs,ys,x(RevI xs ys -> RevI ys xs -> RevI(x::ys)(xs++x:))
```

```
  xs  ys  RevIxsys:RevI xs ys
----------------------------------------------------------------
?_3:RevI(Nil alpha)(Nil alpha)
```

Die Zielformel ?_3 zeigen wir durch (use "InitRevI"). Um die andere Zielformel ?_2 zu zeigen, nehmen wir zunächst durch

```
(assume "xs1" "ys1" "x" "RevIxs1ys1" "RevIys1xs1")
```

alles in den Kontext und verwenden nun wieder den Befehl elim; dieses Mal auf RevIys1xs1. Wir geben also (elim "RevIys1xs1") ein und erhalten die Ausgabe

```
ok, ?_5 can be obtained from

  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
----------------------------------------------------------------
?_7:all xs,ys,x0(
     RevI xs ys -> RevI(x::xs)(ys++x:) ->
     RevI(x::xs++x0:)((x0::ys)++x:))



  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
----------------------------------------------------------------
?_6:RevI(x:)((Nil alpha)++x:)
```

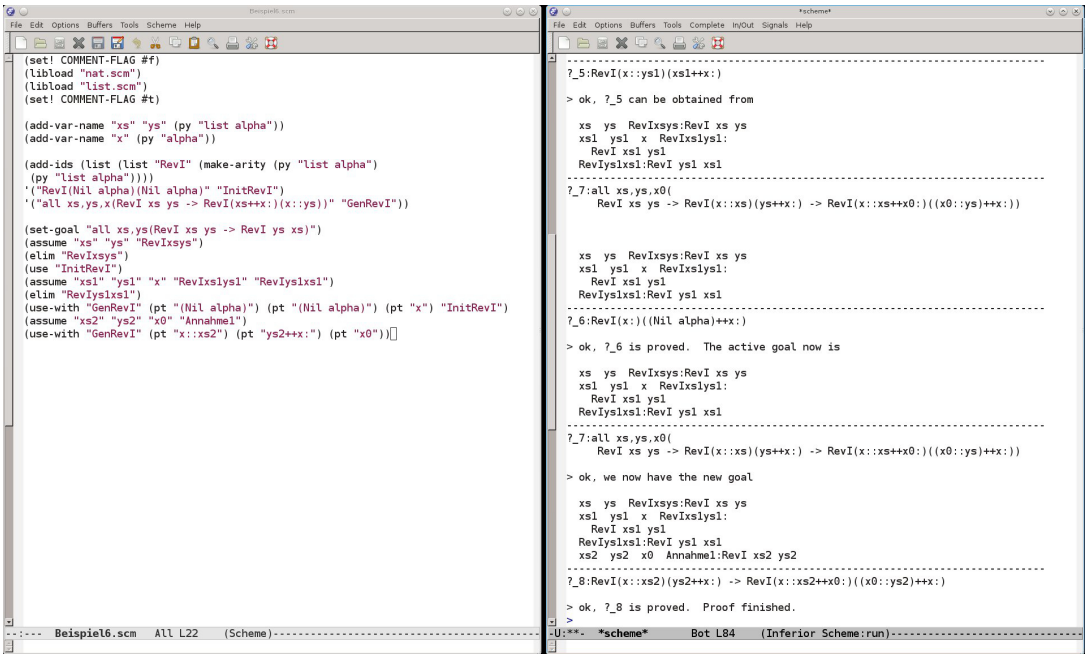
Wir wissen, dass x: und (Nil alpha)++x: die selben Terme in unserer Theorie sind. Das System erkennt dies auch. Nach InitRevI gilt RevI(Nil alpha)(Nil alpha) und mit GenRevI erhalten wir dann RevI x: x:. Minlog erkennt jedoch mit dem use-Befehl nicht sofort, wie GenRevI anzuwenden ist, daher verwenden wir use-with:

```
(use-with "GenRevI" (pt "(Nil alpha)") (pt "(Nil alpha)")
                                       (pt "x") "InitRevI")
```

Zurück bekommen wir dann

```
ok, ?_6 is proved.  The active goal now is

  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
----------------------------------------------------------------
?_7:all xs,ys,x0(
     RevI xs ys -> RevI(x::xs)(ys++x:)
     -> RevI(x::xs++x0:)((x0::ys)++x:))
```

Wir sehen, dass `RevI(x::xs)(ys++x:) -> RevI(x::xs++x0:)((x0::ys)++x:))` die Form des Axioms `GenRevI` hat. Wir nehmen daher alles bis auf die letzte Prämisse mit `(assume "xs2" "ys2" "x0" "Annahme1")` in den Kontext und beenden durch `(use-with "GenRevI" (pt "x::xs2") (pt "ys2++x:") (pt "x0"))` den Beweis.



Lassen wir uns durch `(display-idpc "RevI")` das Prädikat `RevI` anzeigen, steht anstelle der Angabe der Algebra, dass das Prädikat nicht rechnerisch ist:

```
RevI non-computational
InitRevI: RevI(Nil alpha)(Nil alpha)
GenRevI: all xs,ys,x(RevI xs ys -> RevI(xs++x:)(x::ys))
```

2.3.4 Leibnizgleichheit und Simplifizierung

Ein sehr wichtiges Beispiel für ein nicht-rechnerisches induktiv definiertes Prädikat ist die Leibnizgleichheit. Sie ist bereits in Minlog eingespeichert und wird mit `EqD` bezeichnet. Geben wir `(display-idpc "EqD")` ein, sehen wir durch welches Einführungsaxiom sie gegeben ist:

```
EqD non-computational
InitEqD: allnc alpha^ alpha^ eqd alpha^
```

Das ist die dekorierte Form der Leibnizgleichheit, wie sie in Beispiel 1.4.3 eingeführt wurde. Dem System ist auch die Infixnotation `T1 eqd T2` anstelle von `EqD T1 T2` bekannt. Außerdem beachte man, dass, wie in Bemerkung 1.9.3 erwähnt, jedes Prädikat bei der Elimination der Leibnizgleichheit verwendet werden darf.
Nach der Einführung der Leibnizgleichheit ist in Lemma 1.4.4 die für eine Gleichheit charakteristische Eigenschaft $\forall_{x,y}(\mathrm{Eq}xy \to A(x) \to A(y))$ gezeigt. Auch in Minlog ist diese Eigenschaft bekannt, sodass es dafür den Befehl `simp` gibt: Haben wir eine Formel `FORMEL` der Form `T1 eqd T2`, die eine Leibnizgleichheit von zwei Termen `T1` und `T2` ist, oder eine Abstraktion davon, dann wird in einem Beweis durch den Befehl

```
(simp FORMEL)
```

in der derzeitigen Zielformel der Term `T1` durch den Term `T2` ersetzt. Falls `FORMEL` noch Prämissen haben sollte, wird auch ein Beweis von diesen Prämissen verlangt. Formal gesehen verwendet man die Formel $\forall_{x,y}(\mathrm{Eq}xy \to A(x) \to A(y))$ aus Lemma 1.4.4 oder genauer genommen verwendet man die äquivalente Formel $\forall_{x,y}(\mathrm{Eq}xy \to A(y) \to A(x))$, wobei für A die Zielformel, für x der Term `T1` und für y der Term `T2` eingesetzt wird. Die Prämisse $\mathrm{Eq}xy$ ist dann schon gegeben und das System fordert dann einen Beweis von $A(y)$.
Als kleines Beispiel beweisen wir für eine natürliche Zahl n und eine Funktion $f : \mathbb{N} \to \mathbb{N}$ die Aussage $f(n) = n \to f(f(n)) = f(n)$. Wir geben also

```
(set-goal "all f,n(f n eqd n -> f(f n)eqd f n)")
```

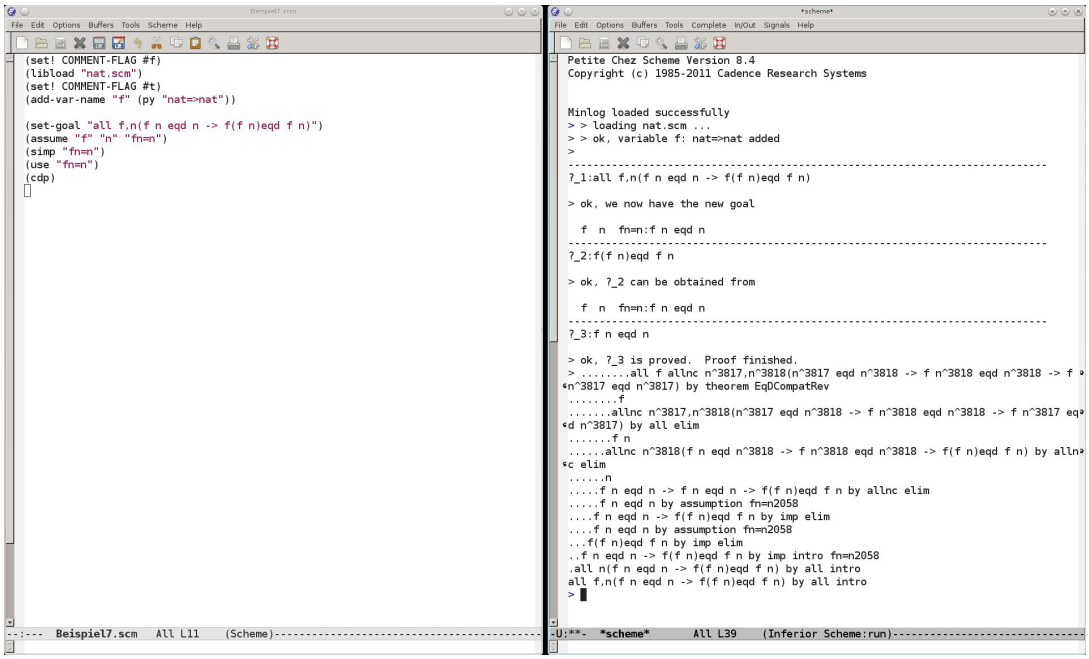
ein und nehmen mit `(assume "f" "n" "fn=n")` die Variablen und die Prämisse in den Kontext. Zu zeigen ist nun Folgendes:

```
ok, we now have the new goal

  f  n  fn=n:f n eqd n
-------------------------------------------------------------------
?_2:f(f n)eqd f n
```

Mit `(simp "fn=n")` wird nun beides Mal `f n` durch `n` ersetzt und wir erhalten

```
ok, ?_2 can be obtained from

  f  n  fn=n:f n eqd n
-------------------------------------------------------------------
?_3:f n eqd n
```

zurück und beenden den Beweis mit `(use "fn=n")`.



Eine Darstellung des Beweises mit `(cdp)` zeigt uns, dass das Theorem `EqDCompatRev` angewandt wurde. Dabei bedeutet `Rev` eben, dass es sich um die Formel $\forall_{x,y}(\mathrm{Eq}xy \to$

$A(y) \to A(x)$) handelt und nicht um $\forall_{x,y}(\mathrm{Eq}xy \to A(x) \to A(y))$. Will man letztere Formel verwenden, so kann man `simp` leicht abwandeln mit

```
(simp "<-" FORMEL).
```

Der Pfeil `<-` steht dafür, dass man die Gleichheit hier in die andere Richtung verwenden möchte. Dies wäre in unserem obigen Beweis auch möglich, ist aber nicht zielführend. Trotzdem geben wir zu Demonstrationszwecken die Ausgabe von Minlog nach `(simp "<-" "fn=n")` anstelle von `(simp "fn=n")` an:

```
ok, ?_2 can be obtained from

  f  n  fn=n:f n eqd n
-----------------------------------------------------------------
?_3:f(f(f n))eqd f(f n)
```

Es wurde jedes `n` in der Zielformel durch `f n` ersetzt.
Es ist nicht nur möglich für `FORMEL` den Namen einer schon bekannten Formel anzugeben. Man kann auch `(pf GLEICHHEIT)` einsetzen, wobei `GLEICHHEIT` wie vorher eine Abstraktion von `T1 eqd T2` sein sollte. Das System fordert dann zusätzlich noch einen Beweis von `GLEICHHEIT`. Ansonsten ist alles andere analog wie vorher.

2.3.5 Beispiele induktiv definierter Prädikate

In diesem Abschnitt wollen wir auf die Disjunktion, die Konjuktion und den Existenzquantor als induktiv definierte Prädikate eingehen.
Beginnen wir mit der Konjunktion. Von ihr gibt es in Minlog vier verschiedene Varianten:

```
AndD with content of type yprod
InitAndD: Pvar1 -> Pvar2 -> Pvar1 andd Pvar2
AndL with content of type identity
InitAndL: Pvar1 -> Pvar2 --> Pvar1 andl Pvar2
AndR with content of type identity
InitAndR: Pvar1 --> Pvar2 -> Pvar1 andr Pvar2
AndNc non-computational
InitAndNc: Pvar1 --> Pvar2 --> Pvar1 andnc Pvar2
```

Hier ist auch wieder die Infixnotation geläufig. Außerdem fällt auf, dass es keine unitäre Konjuktion gibt. Das liegt daran, dass bereits der Typ von einem möglichen `AndU` der Einheitstyp ist. Nach Bemerkung 1.9.3 ist `AndU` damit überflüssig und kann durch `AndNc` ersetzt werden.
Um das Einführungsaxiom der jeweiligen Konjunktion zu verwenden, gibt es auch den Befehl `(split)`. Dieser ist für die obigen Prädikate äquivalent zu `(intro 0)`.
Beim Existenzquantor ist es wie bei der Konjunktion: Auch hier haben wir vier Varianten:

```
ExD with content of type yprod
InitExD: all alpha^((Pvar alpha)alpha^ ->
               exd alpha^0 (Pvar alpha)alpha^0)
ExL with content of type identity
InitExL: all alpha^((Pvar alpha)alpha^ -->
               exl alpha^0 (Pvar alpha)alpha^0)
```

```
ExR with content of type identity
InitExR: allnc alpha^((Pvar alpha)alpha^ ->
               exr alpha^0 (Pvar alpha)alpha^0)
ExNc non-computational
InitExNc: allnc alpha^((Pvar alpha)alpha^ -->
               exnc alpha^0 (Pvar alpha)alpha^0)
```

Weil das Eliminationsaxiom für einen Existenzquantor etwas kompliziert anzuwenden ist, gibt es dafür den Befehl

```
(by-assume ANNAHME VAR NAME).
```

Mit diesem Befehl wird die neue Variable `VAR` eingeführt, die die Existenzaussage `ANNAHME` $=: \exists_x A(x)$ erfüllt. Es wird dann $A($`VAR`$)$ im Kontext unter `NAME` abgespeichert.
Beweisen wir als Beispiel die Aussage $\forall_n(P(n) \to Q(n)) \to \exists_n P(n) \to \exists_n Q(n)$. Wir geben also

```
(set-goal "all n (P n -> Q n) -> exd n  P n -> exd n Q n")
(assume "Annahme1" "Annahme2")
```

ein und erhalten die Ausgabe

```
ok, we now have the new goal

  Annahme1:all n(P n -> Q n)
  Annahme2:exd n P n
----------------------------------------------------------------------
?_2:exd n Q n
```

Hier verwenden wir mit `(by-assume "Annahme2" "n" "Annahme2Inst")` den Befehl `by-assume`. Minlog gibt uns

```
ok, we now have the new goal

  Annahme1:all n(P n -> Q n)
  n  Annahme2Inst:P n
----------------------------------------------------------------------
?_5:exd n Q n
```

zurück. Man sieht, dass `Annahme2` nicht mehr aufgeführt wird. Sie existiert aber dennoch und man könnte sie immer noch verwenden. Um nun die Aussage `exd n Q n` zu beweisen, verwenden wir das Einführungsaxiom auf den Term `n` durch `(intro 0 (pt "n"))`. Die Ausgabe lautet

```
ok, ?_5 can be obtained from

  Annahme1:all n(P n -> Q n)
  n  Annahme2Inst:P n
----------------------------------------------------------------------
?_6:Q n
```

und mit (use "Annahme1") gefolgt von (use "Annahme2Inst") beenden wir den Beweis.
Im Gegensatz zum Ex und And haben wir bei der Disjunktion fünf Varianten, weil der Typ von OrU nicht der Einheitstyp sondern die boolesche Algebra ist.

```
OrD with content of type ysum
InlOrD: Pvar1 -> Pvar1 ord Pvar2
InrOrD: Pvar2 -> Pvar1 ord Pvar2
OrL with content of type ysumu
InlOrL: Pvar1 -> Pvar1 orl Pvar2
InrOrL: Pvar2 --> Pvar1 orl Pvar2
OrR with content of type uysum
InlOrR: Pvar1 --> Pvar1 orr Pvar2
InrOrR: Pvar2 -> Pvar1 orr Pvar2
OrU with content of type boole
InlOrU: Pvar1 --> Pvar1 oru Pvar2
InrOrU: Pvar2 --> Pvar1 oru Pvar2
OrNc non-computational
InlOrNc: Pvar1 -> Pvar1 ornc Pvar2
InrOrNc: Pvar2 -> Pvar1 ornc Pvar2
```

Für die Disjunktion gibt es keine besonderen Befehle. Die Einführungsaxiome und das Eliminationsaxiom lassen sich auch jeweils leicht anwenden.
Von diesen drei induktiv definierten Prädikaten gibt es jeweils noch „interaktive“ Varianten andi, exi und ori. Dabei handelt es sich nicht um neue induktiv definierte Prädikate, sondern das System sucht jeweils die passende Variante aus. Wollen wir beispielsweise die Aussage

$$(A \to B \wedge C) \to (A \to B) \wedge (A \to C)$$

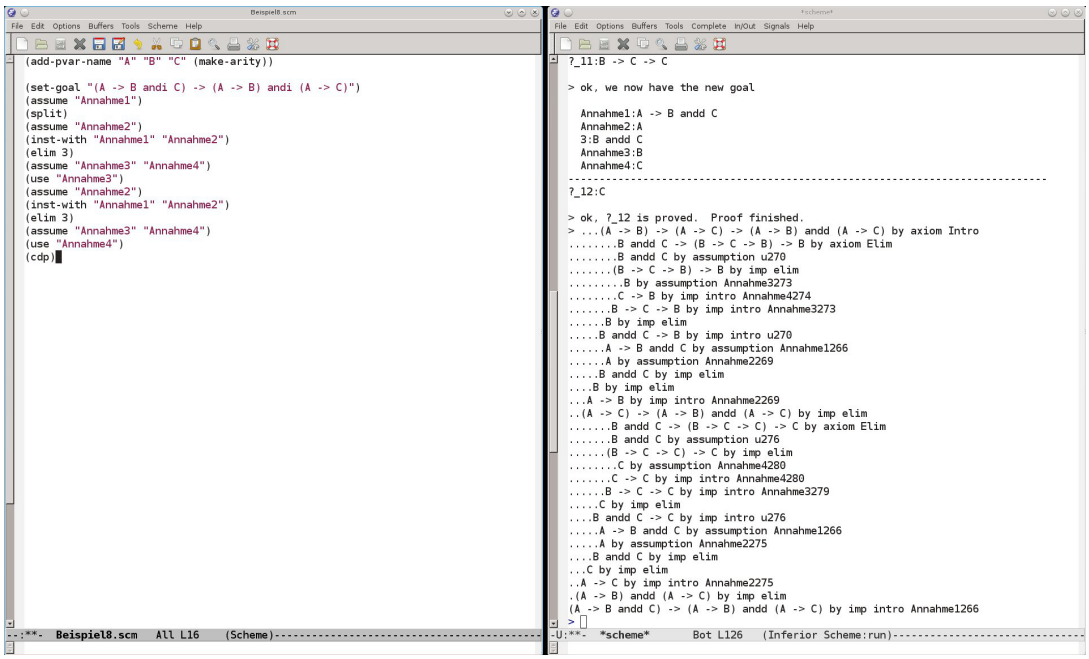
beweisen, so können wir das mit

```
(set-goal "(A -> B andi C) -> (A -> B) andi (A -> C)")
```

tun. Minlog stellt fest, dass B und C rechnerischen Gehalt haben können, sodass andi durch andd ersetzt wird und wir die Ausgabe

```
----------------------------------------------------------------------
?_1:(A -> B andd C) -> (A -> B) andd (A -> C)
```

erhalten. Hier nehmen wir die Prämisse mit (assume "Annahme1") in den Kontext und verwenden mit (split) das Einführungsaxiom von AndD, das heißt, unser Ziel ist es, A -> B und A -> C zu zeigen, was wir jeweils durch Elimination von B andd C tun. Der Rest des Beweises ist nichts Besonderes mehr und wird in der folgenden Abbildung gezeigt.



2.4 Terme in Minlog

2.4.1 define-Befehl

Durch den `define`-Befehl lassen sich Ausdrücke abkürzen. Dabei muss es sich nicht nur um Terme handeln. Mit `define` kann man auch Formeln, Typen oder einfach nur Zeichenketten einspeichern. Das Kommando hat dabei die Form

```
(define STRING AUSDRUCK).
```

Für `STRING` setzt man eine beliebige Zeichenkette ein, und definiert dadurch, dass eine spätere Verwendung dieser Zeichenkette durch `AUSDRUCK` ersetzt werden soll. Wir werden den `define`-Befehl besonders dafür verwenden, um Terme abzukürzen. Wollen wir zum Beispiel die natürliche Zahl 4 definieren, können wir dies durch

```
(define vier (pt "Succ(Succ(Succ(Succ 0)))"))
```

tun und uns diesen Term dann mit `(pp vier)` anzeigen lassen. Als Ausgabe bekommen wir hier aber `4` und nicht `Succ(Succ(Succ(Succ 0)))`, weil im System bereits die Dezimaldarstellung von natürlichen Zahlen implementiert ist. Die extrahierten Terme werden später noch viel länger werden, sodass `define` hier sehr nützlich werden wird.
Wollen wir aber zum Beispiel bestimmte Formeln oder Typen öfters verwenden, kann es auch sinnvoll sein, diese mit `define` festzulegen. Beispiele dafür sind Befehle wie

```
(define Ziel (pf "(A -> B -> C) -> ((C -> A) -> B) -> A -> C"))
```

oder

```
(define Folgen (py "nat=>alpha")).
```

Man beachte noch, dass sich definierte Zeichenketten einfach überschreiben lassen und Minlog auch keine Warnung ausgibt.

2.4.2 Programmkonstanten

Programmkonstanten im Sinne von Definition 1.2.16 sind ein wichtiger Bestandteil der Terme in Minlog. Will man in Minlog eine Programmkonstante einführen, so wird diese zunächst mit dem Befehl

```
(add-program-constant NAME TYP)
```

deklariert. Hier werden noch keine Berechnungsregeln hinzugefügt. Es wird nur die Bezeichnung `NAME` der Programmkonstante und ihr Typ `TYP` eingeführt. Unser Beispiel soll hier die Addition auf den natürlichen Zahlen sein. Diese ist in der Datei `nat.scm` eingeführt durch

```
(add-program-constant "NatPlus" (py "nat=>nat=>nat")).
```

Nachdem nun die Syntax für die Addition eingeführt ist, können wir die Berechnungsregeln hinzufügen. Das geht mit dem Befehl

```
(add-computation-rule TERM1 TERM2).
```

Wir erinnern uns, dass in Definition 1.2.16 eine Berechnungsregel gegeben war durch den syntaktischen Ausdruck

$$D\vec{P} := M,$$

wobei $\vec{P}$ ein Liste von Konstruktormustern und M ein Term ist, in dem höchstens die Variablen frei sind, die auch in $\vec{P}$ frei sind. Für `TERM1` setzen wir $D\vec{P}$ ein und für `TERM2` setzen wir M ein. Für die Addition haben wir die beiden Berechnungsregeln

```
(add-computation-rule (pt "NatPlus n Zero") (pt "n"))
(add-computation-rule (pt "NatPlus n (Succ m)")
                      (pt "Succ (NatPlus n m)")).
```

Es gibt dafür auch eine abkürzende Schreibweise. Der Befehl

```
(add-computation-rules
 "NatPlus n Zero" "n"
 "NatPlus n (Succ m)" "Succ(NatPlus n m)")
```

bewirkt dasselbe wie die beiden Befehle oben, ist nur etwas kürzer, da man `pt` weglassen kann und in jede Zeile eine neue Berechnungsregel schreibt, ohne noch einmal `add-computation-rule` einzugeben.
Mit

```
(display-pconst LISTE)
```

kann man sich die Programmkonstanten in der Liste `LISTE` anzeigen lassen. Wir erhalten bei Eingabe von `(display-pconst "NatPlus")` die Ausgabe

```
NatPlus
  comprules
0 n+0 n
1 n+Succ m Succ(n+m)
```

zurück.
Wollen wir eine Programmkonstante mit Namen `NAME` wieder entfernen, so können wir dies mit

```
(remove-program-constant NAME)
```

tun. Damit wird der Name für die Programmkonstante wieder freigegeben und alle Berechnungsregeln über sie gelöscht. Hier sollte man jedoch vorsichtig sein, denn die Theoreme über die entfernte Programmkonstante werden nicht entfernt. Führt man also eine Programmkonstante mit dem gleichen Namen wieder ein, so nimmt das Programm an, dass die älteren Sätze auch für die neue Programmkonstante gelten. Das muss aber mitnichten so sein, denn die neue Programmkonstante kann ganz andere Berechnungsregeln haben.

2.4.3 Beispiele von Programmkonstanten

Die booleschen Junktoren `andb`, `orb` und `impb`, die wir in Bemerkung 1.5.8 bereits angesprochen haben, sind einfache, aber häufig verwendete Programmkonstanten. Sie heißen im System `AndConst`, `OrConst` und `ImpConst`, werden aber als Infix wie in der Bemerkung notiert. Als Berechnungsregeln haben wir die folgenden:

```
AndConst
  comprules
0 True andb boole^ boole^
1 boole^ andb True boole^
2 False andb boole^ False
3 boole^ andb False False
OrConst
  comprules
0 True orb boole^ True
1 boole^ orb True True
2 False orb boole^ boole^
3 boole^ orb False boole^
ImpConst
  comprules
0 False impb boole^ True
1 True impb boole^ boole^
2 boole^ impb True True
```

Eine wichtige Programmkonstante ist natürlich auch der Rekursionsoperator. In Minlog wird dieser mit

```
(Rec ALGEBRA=>TYP)
```

bezeichnet. Genau genommen ist das der Rekursionsoperator der Algebra `ALGEBRA` in den Typen `TYP`. Ein Beispiel eines Terms mit dem Rekursionsoperator werden wir gleich im nächsten Abschnitt sehen. Zunächst betrachten wir an dieser Stelle den Typ des Rekursionsoperators. Wollen wir allgemein den Typ eines Terms `TERM` haben, erhalten wir diesen durch

```
(term-to-type TERM).
```

Um also den Typ des Rekursionsoperators von den natürlichen Zahlen in einen Typ `alpha` anzeigen zu lassen, geben wir

```
(pp (term-to-type (pt "(Rec nat=>alpha)")))
```

ein und erhalten die erwartete Ausgabe:

```
nat=>alpha=>(nat=>alpha=>alpha)=>alpha
```

Das letzte Beispiel hier ist die entscheidbare Gleichheit für Objekte einer finitären Algebra, wie sie in Definition 1.2.18 gegeben ist. Die entscheidbare Gleichheit wird von Minlog automatisch bei der Definition einer finitären Algebra implementiert und mit = bezeichnet. Die Gleichheit ist dabei so fest in Minlog implementiert, dass sich die Berechnungsregeln nicht explizit anzeigen lassen.

2.4.4 Abstraktion und Anwendung

Wollen wir in Minlog eine oder mehrere Variablen durch λ-Abstraktion in einem Term binden, so geschieht dies, indem man die entsprechenden Variablen in eckigen Klammern und durch Kommata getrennt vor den Term setzt. Um in Minlog einen Term N auf einen Term M anzuwenden, wird N nach M geschrieben. Wir können nun wie in Beispiel 1.2.11 die Addition zweier natürlicher Zahlen definieren. Dabei machen wir daraus einen Term vom Typ $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$:

```
(define Plus (pt "[n,m](Rec nat=>nat) n m ([n0,n1]Succ n1)"))
```

Wollen wir nun auf einen bereits definierten Term `TERM1` einen anderen Term `TERM2` anwenden, gibt es dafür den Befehl

```
(make-term-in-app-form TERM2 TERM1).
```

Wenden wir beispielsweise den Term `1` auf den Term `Plus` an, geht das durch

```
(make-term-in-app-form Plus (pt "1")).
```

Wenn wir uns nun die Normalisierung, welche im übernächsten Abschnitt genauer behandelt wird, dieses Terms durch

```
(pp (nt (make-term-in-app-form Plus (pt "1"))))
```

anzeigen lassen, so erhalten wir die Ausgabe `Succ`. Dies war zu erwarten, da Addition mit `1` genau den Nachfolger ergibt.
Als Gegenstück zur Anwendung kann man aus einem Term `TERM` eine Variable `VAR` durch den Befehl

```
(make-term-in-abst-form VAR TERM)
```

abstrahieren. Geben wir also so etwas wie `(pp (make-term-in-abst-form (pv "n") (pt "n+1")))` in Minlog ein, erhalten wir als Ausgabe `[n]n+1`.

2.4.5 Boolesche Terme als Aussagen

Wir hatten bereits in Notation 1.4.8, dass boolesche Terme mit Hilfe der Leibnizgleichheit als Aussage interpretiert werden können. In Minlog ist das genauso der Fall. So versteht Minlog ohne Weiteres Befehl wie zum Beispiel

```
(set-goal "all boole1, boole2 (boole1 andb boole2 -> boole1)").
```

Die Identifizierung eines booleschen Terms mit einer Aussage geschieht durch die beiden Theoreme `EqDTrueToAtom` und `AtomToEqDTrue`, die in Minlog schon nach dem Start eingespeichert sind:

```
AtomToEqDTrue all boole^(boole^ -> boole^ eqd True)
EqDTrueToAtom all boole^(boole^ eqd True -> boole^)
```

Wir werden später diskutieren, ob die obige Beispielaussage bewiesen werden kann oder nicht. Dem Leser steht natürlich frei, sich dies jetzt zu überlegen.
Arbeitet man in Minlog mit booleschen Variablen, so wird man häufig das Theorem

```
Truth T
```

verwenden müssen, welches mit der Identifizierung von booleschen Termen als Aussagen einfach `T eqd T` besagt. Damit lassen sich boolesche Terme, welche auf `T` normalisieren, beweisen.

2.4.6 Normalisierung

In der Theorie **TCF** gelten Terme als gleich, wenn ihre Zeichenketten bezüglich des reflexiven, transitiven und symmetrischen Abschlusses der Konversionsrelation gleich sind. In Minlog ist diese Gleichheit auch implimentiert. Deswegen gehen wir hier nun auf die Konversionsschritte in Minlog ein.
Programmkonstanten werden über ihre Berechnungsregeln definiert. Im System werden diese automatisch identifiziert. So erkennt Minlog beispielsweise sofort, dass `n+1` und `Succ n` für jede natürlich Zahl `n` leibnizgleich sind. Die Berechnungsregeln sind auch als Theoreme abgespeichert. Dabei ist die `X`-te Berechnungsregel der Programmkonstante `NAME` unter `NAMEXCompRule` abgespeichert. Deswegen werden die Berechnungsregeln bei `display-pconst` durchnummeriert. Mit dem Befehl `search-about` aus Abschnitt 2.1.13 werden diese Theoreme nicht angezeigt. Möchte man aber auch nach solchen Theoremen suchen, so kann man `search-about` um `'all` erweitern, dann gibt Minlog auch diese Sätze aus. Wir können damit explizit nach Berechnungsregeln suchen. Wollen wir zum Beispiel alle Berechnungsregeln für die booleschen Operatoren angezeigt bekommen, erhalten wir diese durch

```
(search-about 'all "CompRule" "Const").
```

Die Ausgabe ist dann die folgende Liste:

```
Theorems with name containing CompRule and Const
NegConst1CompRule
negb False eqd True
NegConst0CompRule
negb True eqd False
OrConst3CompRule
all boole^ (boole^ orb False)eqd boole^
OrConst2CompRule
all boole^ (False orb boole^)eqd boole^
OrConst1CompRule
all boole^ (boole^ orb True)eqd True
OrConst0CompRule
all boole^ (True orb boole^)eqd True
ImpConst2CompRule
all boole^ (boole^ impb True)eqd True
ImpConst1CompRule
all boole^ (True impb boole^)eqd boole^
```

```
ImpConst0CompRule
all boole^ (False impb boole^)eqd True
AndConst3CompRule
all boole^ (boole^ andb False)eqd False
AndConst2CompRule
all boole^ (False andb boole^)eqd False
AndConst1CompRule
all boole^ (boole^ andb True)eqd boole^
AndConst0CompRule
all boole^ (True andb boole^)eqd boole^
No global assumptions with name containing CompRule and Const
```

Die Theoreme zu den Berechungsregeln lassen sich dann sehr gut mit dem simp-Befehl verwenden.
Oft will man aber nicht nur eine Berechunungsregel anwenden, sondern den Term ganz auf Normalform gemäß Definition 1.2.12 bringen, falls diese existiert. Mit

```
(nt TERM)
```

wird versucht, den Term TERM durch iterative Anwendung von den im System eingespeicherten Berechnungsregeln auf Normalform zu bringen. Erinnern wir uns dazu an die Definition der Addition mittels des Rekursionsoperators aus Abschnitt 2.4.4. Diesen können wir nun auf zwei natürliche Zahlen anwenden und normalisieren lassen. Dazu geben wir folgende Befehle in Minlog ein:

```
(define 7+2 (pt "([n,m](Rec nat=>nat) n m ([n0,n1]Succ n1))7 2"))
(pp (nt 7+2))
```

und die Ausgabe ist tatsächlich 9. Bei der Normalisierung ist jedoch immer Vorsicht geboten. Möchte man zum Beispiel Terme normalisieren, die keine Normalform haben, so gerät das System in eine Endlosschleife, welche manuell unterbrochen werden muss.
Auch in einem Beweis kann man mit Hilfe von

```
(ng NAME)
```

die Prämisse mit Namen NAME normalisieren. Schreibt man anstelle von NAME den Ausdruck #t, so werden die Terme in der Zielformel normalisiert. Lässt man das Argument vollständig weg und gibt nur (ng) ein, werden alle Terme normalisiert.
Intern testet das System bei der Eingabe von ng und nt alle ihm bekannten Ersetzungsregeln und wendet diese, wenn möglich, an. Das wird so lange iteriert, bis keine Ersetzungsregel mehr angewendet werden kann. Selbstverständlich muss dieser Algorithmus nicht terminieren, weswegen man dies immer bei der Verwendung von ng oder nt beachten muss.
Möchte man lediglich die β- und η-Konversionsregeln aus Definition 1.2.8 auf einem Term TERM anwenden, so kann man dies mit

```
(term-to-beta-eta-nf TERM)
```

tun. Bei der Eingabe von (pp (term-to-beta-eta-nf (pt "([n]n+1)2"))) wird beispielsweise der Term 2+1 zurückgegeben.
Es ist auch möglich Ersetzungsregeln hinzuzufügen, die vom System nicht automatisch angelegt werden. Das geht mittels des Befehls

```
(add-rewrite-rule TERM1 TERM2).
```

Es wird dabei die globale Annahme, dass TERM1 und TERM2 gleich sind, abgespeichert. Das passiert jedoch nicht, wenn add-rewrite-rule benutzt wird, direkt nachdem bewiesen wurde, dass die beiden Terme gleich sind. In diesem Fall wird die Aussage als Theorem abgespeichert. In beiden Fällen wird dann die entsprechende Ersetzungsregel hinzugefügt, welche auch bei ng und nt berücksichtigt wird. Gibt es zu einer Programmkonstante auch Ersetzungsregeln, werden diese durch display-pconst auch angezeigt. Gibt man (display-pconst "NatPlus") nach dem gesamten Einlesen von nat.scm ein, erhält man

```
NatPlus
  comprules
0 n+0 n
1 n+Succ m Succ(n+m)
  rewrules
0 0+n n
1 Succ n+m Succ(n+m)
2 n+(m+l) n+m+l
```

In dieser Datei werden also auch Ersetzungsregeln über die natürlichen Zahlen bewiesen. Analog zu den Berechnungsregeln, hat die X-te Ersetzungsregel einer Programmkonstante NAME den Namen NAMEXRewRule.

2.4.7 Der extrahierte Term

Nachdem eine Formel in Minlog bewiesen wurde, kann man aus dem Beweis den rechnerischen Gehalt durch den Befehl

```
(proof-to-extracted-term)
```

extrahieren. Als Beispiel wollen wir nun beweisen, dass es zu jeder geraden natürlichen Zahl n eine natürliche Zahl m gibt mit $m+m=n$. Die Addition ist bereits in nat.scm definiert. Das Prädikat EvenI auf nat, welches aussagt, dass eine natürliche Zahl gerade ist, ist in Abschnitt 2.2.3 eingeführt worden. Hier verwenden wir folgende dekorierte Variante:

```
(add-ids
(list (list "EvenI" (make-arity (py "nat")) "algEvenI"))
'("EvenI 0" "InitEvenI")
'("allnc n(EvenI n -> EvenI(Succ (Succ n)))" "GenEvenI")).
```

Um die obige Aussage zu beweisen, tippen wir

```
(set-goal "allnc n(EvenI n -> exl m m+m=n)")
```

ein. Man beachte, dass wir exl verwenden, weil in der Gleichheit m+m=n kein rechnerischer Gehalt steckt. Außerdem werden wir n nicht rechnerisch verwenden, weshalb auch dieses mit einem nc-Allquantor gebunden ist. Für den Beweis nehmen wir mit (assume "n" "EvenIn") die Variable n und die Prämisse in den Kontext und verwenden anschließend mittels (elim "EvenIn") das Eliminationsaxiom von EvenI. Wir erhalten dann

```
ok, ?_2 can be obtained from

  {n}  EvenIn:EvenI n
-------------------------------------------------------------------
?_4:allnc n(EvenI n -> exl m m+m=n -> exl m m+m=Succ(Succ n))



  {n}  EvenIn:EvenI n
-------------------------------------------------------------------
?_3:exl m m+m=0
```

als Ausgabe des Programms. Wir haben also für jede Einführungsregel von `EvenI` die entsprechende Aussage zu zeigen. Die Aussage `exl m m+m=0` zeigen wir, indem wir das Einführungsaxiom von `ExL` mit dem Term `0` verwenden. Also geben wir `(intro 0 (pt "0"))` ein. Mit `(use "Truth")` können wir die entstehende Aussage `0+0=0` dann beweisen. Um nun die etwas längere Aussage `?_4` zu beweisen, nehmen wir mit dem Befehl `(assume "n1" "Evenn1" "IH")` die Prämissen in den Kontext. Aus der Induktionshypothese `IH` erhalten wir durch den Befehl `(by-assume "IH" "m" "IHInst")` ein `m` und die Aussage `m+m=n1`. Die nun zu zeigende Aussage `exl m m+m=Succ(Succ n1)` wird nach der Eingabe von `(intro 0 (pt "m+1"))` zu `m+1+(m+1)=Succ(Succ n1)`. Mit `(ng)` normalisieren wir die Aussagen und erhalten die Induktionshypothese, sodass `(use "IHInst")` den Beweis beendet.
Nun schauen wir uns den extrahierten Term des Beweises an. Hierzu geben wir

```
(define eterm (proof-to-extracted-term))
```

und anschließend `(pp eterm)` ein. Die Ausgabe von Minlog ist so jedoch noch etwas unleserlich:

```
[algEvenI3833]
 (Rec algEvenI=>nat)algEvenI3833(([n^3834]n^3834)0)
 ([algEvenI3839,n3837]
   ([n3835,(nat=>nat)_3836](nat=>nat)_3836 n3835)n3837
   ([m]([n^3838]n^3838)(m+1)))
```

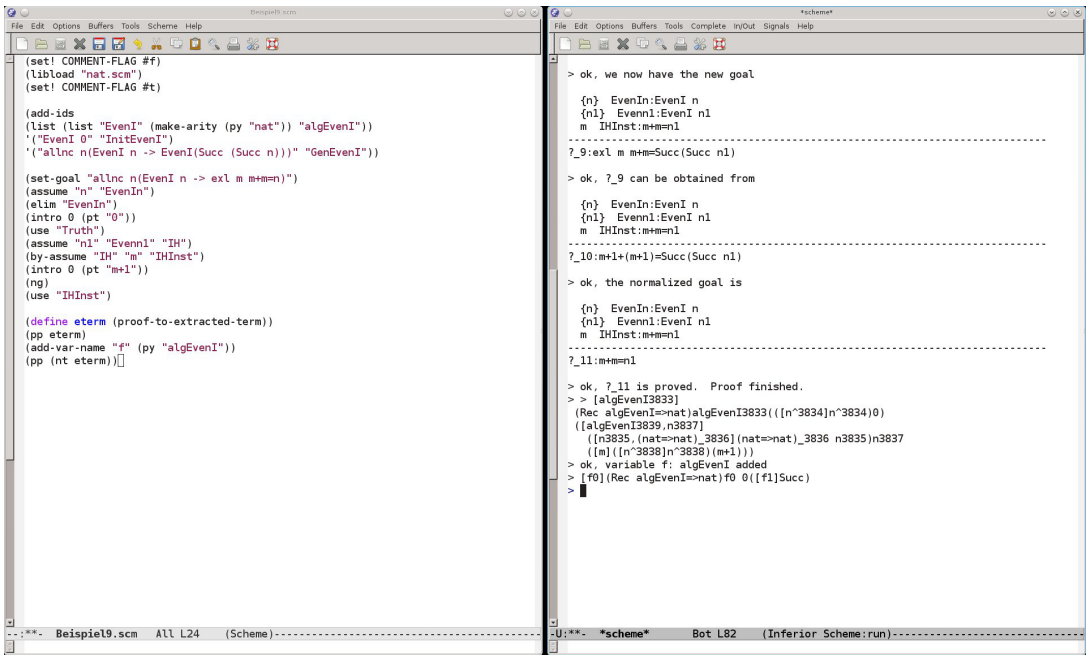
Die normalisierte Form ist deutlich besser lesbar. Es empfiehlt sich allgemein, jeden extrahierten Term zu normalisieren. Wir geben also `(pp (nt eterm))` ein und erhalten die Ausgabe:

```
[algEvenI0](Rec algEvenI=>nat)algEvenI0 0([algEvenI1]Succ)
```

An diesem Term ist möglicherweise noch etwas störend, dass wir keine Variablen aus der Algebra `algEvenI` deklariert haben, weswegen so eine Variable zum Beispiel mit `algEvenI0` bezeichnet wird. Wir führen daher noch die Bezeichnung `f` für Variablen vom Typ `algEvenI` ein und normalisieren den extrahierten Term erneut.

```
[f0](Rec algEvenI=>nat)f0 0([f1]Succ)
```

Das Auftreten des Rekursionsoperators passt genau damit zusammen, dass wir einmal das Eliminationsaxiom von `EvenI` verwendet haben. Der Typ des extrahierten Terms `algEvenI=>nat` ist auch sinnvoll, denn für einen Zeugen, dass `n` gerade ist, erhalten wir eine natürliche Zahl `m`, sodass `m+m=n` ist.



Als kleine Nebenbemerkung sei noch erwähnt, dass, wie der Leser vielleicht bereits gesehen hat, die Konstruktortypen von `algEvenI` die gleichen sind wie bei `nat`. Es ist daher naheliegend, dass diese beiden Typen „isomorph“ sind. Der obige extrahierte Term wäre sogar ein Isomorphismus von `algEvenI` nach `nat`. Wir werden hier jedoch nicht weiter auf Isomorphismen von Typen eingehen. Es ist dennoch interessant, dass wir dadurch den Zeugen für `EvenI n` als die Hälfte von `n` verstehen können, wenn wir `algEvenI` und `nat` identifizieren.

2.5 Totalität in Minlog

In diesem Abschnitt werden wir die Totalitätsprädikate in Minlog betrachten, welche in Definition 1.5.3 eingeführt wurden. Es wird hauptsächlich um die gesamte Totalität gehen. Auf die anderen Varianten werden wir nur einen kurzen Blick werfen.

2.5.1 Einführung des Totalitätsprädikats

Zu jeder Algebra `ALG`, welche in Minlog definiert ist, können wir mittels

```
(add-totality ALG)
```

das entsprechende Totalitätsprädikat als induktiv definiertes Prädikat hinzufügen. Es wird dann ein neues Prädikat mit dem Namen `TotalALG` angelegt. Dabei ist zu beachten, dass der erste Buchstabe von `ALG` in `TotalALG` groß geschrieben wird. Das neue Prädikat kann dann wie jedes andere induktiv definierte Prädikat behandelt werden.
In `list.scm` wird zu der Listenalgebra auch ihre Totalität eingeführt. Wir können mit `(display-idpc "TotalList")` die Einführungsaxiome anzeigen lassen:

```
 TotalList with content of type list
TotalListNil: TotalList(Nil alpha)
TotalListCons: allnc alpha^(
 Total alpha^ ->
```

```
 allnc (list alpha)^0(
  TotalList(list alpha)^0 -> TotalList(alpha^ ::(list alpha)^0)))
```

Wir sehen, dass eine Prämisse von `TotalListCons` die Totalität von `alpha` ist. Es handelt sich also um die gesamte Totalität. In Definition 1.5.3 wurde diese mit **G** bezeichnet.
Es ist auch möglich, das so genannte relative Totalitätsprädikat hinzuzufügen. Dafür gibt es das Kommando

```
(add-rtotality ALG).
```

Die relative Totalität ist eine Verallgemeinerung der gesamten und strukturellen Totalität. Dies sehen wir, wenn wir uns die Einführungsaxiome von der relativen Totalität von `list` mit `(display-idpc "RTotalList")` anzeigen lassen.

```
 RTotalList with content of type list
RTotalListNil: (RTotalList [...])(Nil alpha)
RTotalListCons: allnc alpha^(
 (Pvar alpha)_346 alpha^ ->
 allnc (list alpha)^0(
  (RTotalList [...])(list alpha)^0 ->
  (RTotalList [...])(alpha^ ::(list alpha)^0)))
```

Zur besseren Lesbarkeit haben wir anstelle von

```
(RTotalList (cterm (alpha^1) (Pvar alpha)_346 alpha^1))
```

in der obigen Ausgabe den Ausdruck `(RTotalList [...])` geschrieben. Wir sehen, dass das Prädikat `RTotalList` neben dem Typ `alpha` auch von einer Prädikatenvariablen `(Pvar alpha)_346` auf dem Typ `alpha` abhängt. Die Struktur der Einführungsaxiome von `RTotalList` ist fast analog zu denen von `TotalList`. Der einzige Unterschied besteht darin, dass bei der relativen Totalität `(Pvar alpha)_346 alpha^` in den Prämisse des zweiten Einführungsaxioms steht, während bei der gesamten Totalität dort `Total alpha^` gefordert wird.
Die relative Totalität ist eine Verallgemeinerung der gesamten Totalität, bei der anstelle von den Totalitätsprädikaten anderer Typen jeweils eine Prädikatenvariable steht.
Auch die strukturelle Totalität erhält man aus der relativen Totalität, indem man für diese Prädikatenvaribalen das immer wahre Prädikat einsetzt.

Das immer wahre Prädikat $\mathcal{T}$ ist gegeben durch das Einführungsaxiom $\forall_x \mathcal{T} x$.

2.5.2 Implizite Darstellung der Totalität

Dem Leser wird bei dem genauen Betrachten mancher Ausgaben von Minlog aufgefallen sein, dass Variablennamen häufig am Ende mit dem Zeichen ^ versehen sind. Nun können wir dieses Geheimnis lüften. Wird über eine Variable quantifiziert, fügt Minlog implizit die Prämisse hinzu, dass diese Variable total ist. So stehen die Ausdrücke $\forall_x A$ und $\forall_x^{nc} A$ in Minlog für $\forall_{\widehat{x}}^{nc}(\mathbf{G}\widehat{x} \to A)$ und $\forall_{\widehat{x}}^{nc}(\mathbf{G}\widehat{x} \to^{nc} A)$. Das Zeichen ^ hinter einer Variablen zeigt dem Programm, dass von dieser Variablen nicht noch zusätzlich gefordert wird, dass sie total ist. Diese Abkürzung wird in Minlog als Theorem realisiert. Durch die folgenden beiden Theoreme wird die Abkürzung eingeführt

```
AllTotalIntro
 allnc alpha^(Total alpha^ -> (Pvar alpha)alpha^) ->
 all alpha (Pvar alpha)alpha
AllncTotalIntro
 allnc alpha^(Total alpha^ --> (Pvar alpha)alpha^) ->
 allnc alpha (Pvar alpha)alpha
```

und durch die beiden Theoreme

```
AllTotalElim
 all alpha (Pvar alpha)alpha ->
 allnc alpha^(Total alpha^ -> (Pvar alpha)alpha^)
AllncTotalElim
 allnc alpha (Pvar alpha)alpha ->
 allnc alpha^(Total alpha^ --> (Pvar alpha)alpha^)
```

lässt sich die Abkürzung entfernen.
Analoge Theoreme gibt es auch für die verschiedenen Formen des Existenzquantors. Für die Einführung haben wir die Theoreme

```
ExNcTotalIntro
 exnc alpha^(Total alpha^ andnc (Pvar alpha)alpha^) ->
 exnc alpha (Pvar alpha)alpha
ExRTotalIntro
 exr alpha^(TotalMR alpha^ alpha^ andr (Pvar alpha)alpha^) ->
 exr alpha (Pvar alpha)alpha
ExLTotalIntro
 exr alpha^(Total alpha^ andl (Pvar alpha)^' alpha^) ->
 exl alpha (Pvar alpha)^' alpha
ExDTotalIntro
 exr alpha^(Total alpha^ andd (Pvar alpha)alpha^) ->
 exd alpha (Pvar alpha)alpha
```

und für die Elimination haben wir

```
ExNcTotalElim
 exnc alpha (Pvar alpha)alpha ->
 exnc alpha^(Total alpha^ andnc  (Pvar alpha)alpha^)
ExRTotalElim
 exr alpha (Pvar alpha)alpha ->
 exr alpha^(TotalMR alpha^ alpha^ andr (Pvar alpha)alpha^)
ExLTotalElim
 exl alpha (Pvar alpha)^ alpha ->
 exr alpha^(Total alpha^ andl (Pvar alpha)^ alpha^)
ExDTotalElim
 exd alpha (Pvar alpha)alpha ->
 exr alpha^(Total alpha^ andd (Pvar alpha)alpha^)
```

All diese Theoreme können natürlich nur angewandt werden, wenn das Totalitätsprädikat für den entsprechenden Typ bereits eingeführt wurde. Aber auch Variablen, deren Typen noch keine Totalitätsprädikate besitzen, können mit ^ versehen werden. Solange das Totalitätsprädikat nicht eingeführt ist, gibt es zwar keinen semantischen Unterschied zwischen x und $\widehat{x}$, jedoch kann auch dann beispielsweise eine Aussage der Form $\forall_x A$ nicht auf $\widehat{x}$ spezialisiert werden. Das System unterscheidet also immer zwischen x und $\widehat{x}$,auch wenn ihm die entsprechende Totalität nicht bekannt ist.

2.5.3 Totalität von Programmkonstanten

Will man beweisen, dass eine Programmkonstante PROGCONST total ist, gibt es dafür den Befehl

```
(set-totality-goal PROGCONST).
```

Der Computer erzeugt dann selbstständig als Zielformel die Totalität dieser Programmkonstanten.
In der Datei nat.scm wird als Beispiel die Totalität der Addition auf den natürlichen Zahlen also der Programmkonstante NatPlus beweisen. Hierzu geben wir

```
(set-totality-goal "NatPlus")
```

ein. Minlog gibt uns dann detailliert das Ziel

```
------------------------------------------------------------------
?_1:allnc n^(TotalNat n^ ->
             allnc n^0(TotalNat n^0 -> TotalNat(n^ +n^0)))
```

zurück. Um dieses zu beweisen, nehmen wir mit (assume "n^" "Tn" "m^" "Tm") alle Prämissen und Variablen in den Kontext und beweisen TotalNat(n^ +n^0) durch das Eliminationsaxiom für Tm. Wir geben daher (elim "Tm") ein. Das liefert uns die beiden neuen Ziele

```
ok, ?_2 can be obtained from

  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
------------------------------------------------------------------
?_4:allnc n^0(TotalNat n^0 ->
              TotalNat(n^ +n^0) -> TotalNat(n^ +Succ n^0))



  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
------------------------------------------------------------------
?_3:TotalNat(n^ +0)
```

Nach Normalisierung durch (ng #t) ist das erste Ziel genau die Prämisse Tn, sodass wir dieses Ziel mit (use "Tn") beweisen können. In der zweiten Zielformel, nehmen wir zunächst wieder mit (assume "l^" "Tl" "IH") die Prämissen in den Kontext und haben dann

```
ok, we now have the new goal

  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
  {l^}  Tl:TotalNat l^
  IH:TotalNat(n^ +l^)
------------------------------------------------------------------
?_6:TotalNat(n^ +Succ l^)
```

als Ziel. Normalisieren wir die Zielformel wieder mit `(ng #t)`, bekommen wir die Zielformel `?_7:TotalNat(Succ(n^ +1^))`. Hier können wir die Totalität der Nachfolgerfunktion verwenden, was genau das zweite Einführungsaxiom `TotalNatSucc` von `TotalNat` ist. Wir geben also das Kommando `(use "TotalNatSucc")` ein und müssen nur noch die Totalität des Arguments zeigen, also `TotalNat(n^ +1^)`. Das ist die Induktionshypothese. Mit `(use "IH")` beenden wir den Beweis.
Wenn ein Totalitätsbeweis für eine Programmkonstante `PROGCONST` beendet wurde, kann man die Totalität durch

```
(save-totality)
```

speichern. Minlog fügt dann die Totalität von `PROGCONST` als Theorem mit dem Namen `PROGCONSTTotal` hinzu.

2.5.4 Totale boolesche Terme

Wie in Lemma 1.5.7 und Bemerkung 1.5.8 besprochen, sind für totale boolesche Terme a und b die logischen Junktoren $\to, \vee$ bzw. $\wedge$ äquivalent zu den boolesche Junktoren `impb`, `orb` bzw. `andb`. Insbesondere für `andb` ist dies Minlog auch bekannt. So kann man den Befehl `(split)` auch verwenden, um eine Aussage der Form `a andb b` zu zeigen, wobei `a` und `b` total sein müssen. Andersherum kann man `a` oder `b` direkt mit der Prämisse `a andb b` zeigen.
In Abschnitt 2.4.5 haben wir die Formel

```
all boole1, boole2 (boole1 andb boole2 -> boole1)
```

angegeben. Nun wissen wir, dass die Aussage leicht durch die beiden Befehle

```
(assume "boole1" "boole2" "Annahme")
(use "Annahme")
```

bewiesen werden kann. Man beachte, dass es hierbei wirklich notwendig ist, dass die Variablen total sind. Wenn sich der Leser bei Abschnitt 2.4.5 überlegt hat, dass dies im allgemeinen Fall nicht bewiesen werden kann, ist das vollkommen richtig. An jener Stelle war die implizite Angabe der Totalität auch noch nicht bekannt.
Um noch eine Anwendung von `(split)` zeigen zu können, gehen wir kurz auf die Aussage

```
all boole1, boole2 (boole1 andb boole2 -> boole2 andb boole1)
```

ein. Diese wird in Minlog gezeigt durch

```
(assume "boole1" "boole2" "Annahme")
(split)
(use "Annahme")
(use "Annahme").
```

Man beachte, dass diese und die vorherige Aussage auch richtig sind, wenn nur eine der beiden Variablen total ist. Jedoch müssen die Aussagen dann auf andere Weise gezeigt werden, denn `(split)` und `(use "Annahme")` führen nur zu einem sinnvollen Beweis, wenn jeweils beide Terme total sind. Bemerkenswert ist hierbei noch, dass wir durch `(cdp)` einen sehr langen formalen Beweis aus mehr als 80 Schlussregeln erhalten. Man sieht also, wie viel Arbeit uns durch das Programm abgenommen wird.

Für finitäre Algebren ist uns die entscheidbare Gleichheit = aus Definition 1.2.18 bekannt. In Satz 1.5.10 wurde bewiesen, dass für totale Objekte aus der entscheidbaren Gleichheit auch Leibnizgleichheit folgt. In Minlog ist dies nicht automatisch implementiert, sodass diese Aussage für jede Algebra einzeln bewiesen werden muss. In den Bibliotheksdateien sind jedoch einige Aussagen dieser Form schon bewiesen. So werden beispielsweise in `nat.scm` das Theorem `NatEqToEqD` und in `list.scm` die Theoreme `ListBooleEqToEqD`, `ListNatEqToEqD` und `ListListNatEqToEqD` bewiesen.
Haben wir zu einer finitären Algebra `ALG` die Aussage `ALGEqToEqD` als Theorem abgespeichert, erkennt dies das Programm und wir können als Argumente für den Befehl `simp` nicht nur Formeln der Form `T1 eqd T2` sondern auch der Form `T1=T2` verwenden, wenn `T1` und `T2` totale Terme mit dem Typ `ALG` sind.

2.5.5 Induktion

Wir haben oben die Theoreme, welche die implizite Darstellung der Totalität einführen bzw. eliminieren, angegeben. Möchte man nun aber die Totalität einer Variablen verwenden, ist dies auf diese Weise sehr kompliziert: Man müsste zuerst beispielsweise $\forall_x A$ zu $\forall^{nc}_{\widehat{x}}(\mathbf{G}\widehat{x} \to A(\widehat{x}))$ umschreiben, dann die Variable $\widehat{x}$ und die Prämisse $\mathbf{G}\widehat{x}$ in den Kontext befördern und dann das Eliminationsaxiom von $\mathbf{G}\widehat{x}$ auf $\{\widehat{x}|A(\widehat{x})\}$ anwenden. Um wirklich von der abkürzenden Schreibweise zu profitieren, gibt es den Befehl

```
(ind).
```

Dieser Befehl lässt sich anwenden, wenn die Zielformel die Form $\forall_x A(x)$ hat. Minlog wendet dann das Eliminationsaxiom der Totalität von x auf das Prädikat $\{x|A(x)\}$ an und setzt die einzelnen Prämissen als neue Ziele.
Eine erweiterte Form dieses Befehls ist der Befehl

```
(ind TERM)
```

für einen totalen Term `TERM`. Diesen Befehl können wir für jede Zielformel `A(TERM)` anwenden. Hier wird das Eliminationsaxiom der Totalität von `TERM` auf $\{x|A(x)\}$ angewendet.
Als Anwendungsbeispiel wollen wir in Anlehnung an die Beispiele aus Abschnitt 2.2.3 und 2.2.4 zeigen, dass jede totale natürliche Zahl gerade oder ungerade ist. Hierfür laden wir die Datei `nat.scm` aus der Bibliothek und führen die induktiv definierten Prädikate `EvenI` und `OddI` aus den Abschnitten 2.2.3 und 2.2.4 ein. Wir setzen nun mit

```
(set-goal "all n (EvenI n ord OddI n)").
```

die gewünschte Zielformel. Die verwendete Disjunktion ist dabei natürlich `OrD`, da `EvenI n` und `OddI n` rechnerischen Gehalt haben. Wir verwenden sofort Induktion über die Totalität von `n` und geben daher `(ind)` ein. Minlog gibt uns dann

```
ok, ?_1 can be obtained from

  n3930
-------------------------------------------------------------------
?_3:all n(EvenI n ord OddI n -> EvenI(Succ n) ord OddI(Succ n))
```

```
  n3930
------------------------------------------------------------------
?_2:EvenI 0 ord OddI 0
```

zurück. Die untere Aussage gilt wegen `EvenI 0`. Wir geben daher zweimal `(intro 0)` ein und die Aussage ist gezeigt. Für den Beweis der oberen Aussage nehmen wir mit `(assume "n" "IH")` die Variable und die Annahme in den Kontext und verwenden mit `(elim "IH")` das Eliminationsaxiom für `OrD`. Das Programm fordert dann einen Beweis von `EvenI(Succ n) ord OddI(Succ n)` einmal unter der Annahme `EvenI n` und einmal unter der Annahme `OddI n`:

```
ok, ?_5 can be obtained from

  n3965  n  IH:EvenI n ord OddI n
------------------------------------------------------------------
?_7:OddI n -> EvenI(Succ n) ord OddI(Succ n)
```

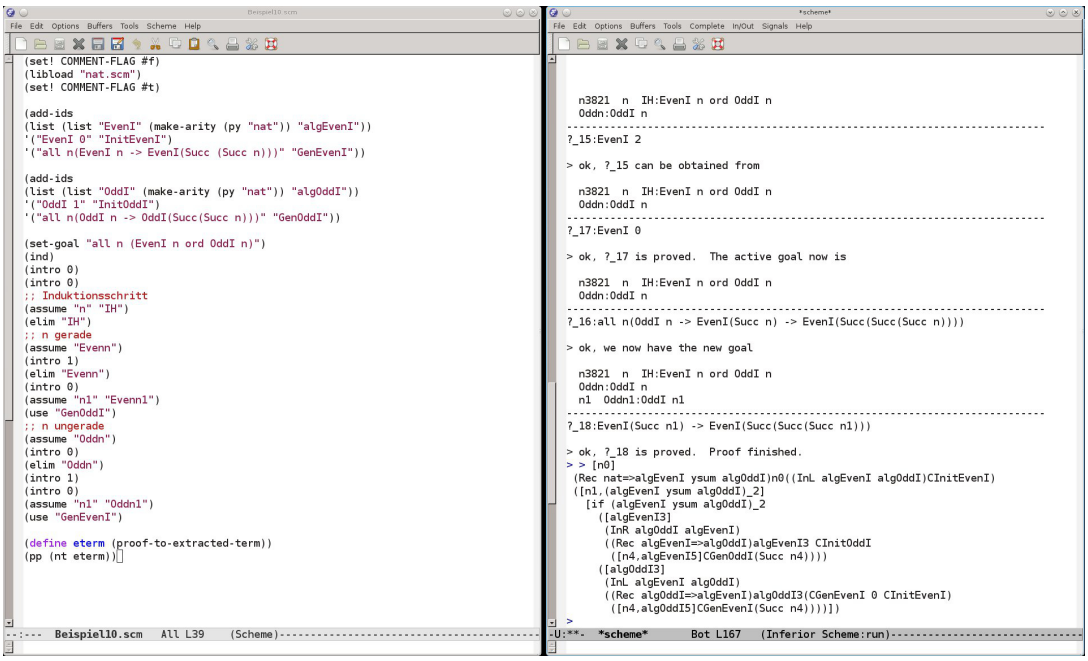
```
  n3965  n  IH:EvenI n ord OddI n
------------------------------------------------------------------
?_6:EvenI n -> EvenI(Succ n) ord OddI(Succ n)
```

Gilt `EvenI n`, zeigen wir `OddI(Succ n)` und geben daher `(assume "Evenn")` und anschließend `(intro 1)` ein. Die Aussage `OddI(Succ n)` beweisen wir dann durch `(elim "Evenn")` aus dem Eliminationsaxiom von `EvenI n`. Zu beweisen ist dann

```
ok, ?_9 can be obtained from

  n3984  n  IH:EvenI n ord OddI n
  Evenn:EvenI n
------------------------------------------------------------------
?_11:all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))
```

```
  n3984  n  IH:EvenI n ord OddI n
  Evenn:EvenI n
------------------------------------------------------------------
?_10:OddI 1
```

Die untere Aussage ist das erste Einführungsaxiom von `OddI`, weswegen diese mit `(intro 0)` direkt bewiesen ist. Die obere Aussage folgt direkt aus dem zweiten Einführungsaxiom `GenOddI` von `OddI`. Wir brauchen nur `(assume "n1" "Evenn1")` gefolgt von `(use "GenOddI")` eingeben. Damit ist der Fall `EvenI n` abgeschlossen. Der Fall `OddI n` ist analog beweisbar. Die gesamten Befehle und den extrahierten Term sehen wir in folgender Grafik:



Der erste Rekursionsoperator im extrahierten Term kommt aus dem Eliminationsaxiom der Totalität. Die anderen Rekursionsoperatoren (`Rec algEvenI=>algOddI`) bzw. (`Rec algOddI=>algEvenI`) haben wir durch die Elimination von `EvenI n` bzw. `OddI n` mittels (`elim "Evenn"`) bzw. (`elim "Oddn"`) erhalten. Die Elimination von `EvenI n ord OddI n` mittels (`elim "IH"`) bringt uns einen Rekursionsoperator über eine Typsumme. Im extrahierten Term tritt dieser in Form des `if` auf. In Minlog wird damit der Caseoperator aus Beispiel 1.2.17 bezeichnet. Für eine Typsumme sind Case- und Rekursionsoperator isomorph, sodass Minlog hier auf den einfacheren Caseoperator zurückgreift. Wann der Caseoperator sonst noch in einem extrahierten Term auftreten kann, werden wir gleich im nächsten Abschnitt diskutieren.

2.5.6 Fallunterscheidung

Häufig ist es der Fall, dass in einem Induktionsbeweis die Induktionshypothese nicht gebraucht wird oder es sogar gar keine Induktionshypothese gibt. Das ist beispielsweise bei der Induktion über totale boolesche Variablen der Fall. Das Eliminationsaxiom der booleschen Totalität ist gegeben durch $\forall_a.\mathbf{G}_{\mathbb{B}}a \to P\mathrm{tt} \to P\mathrm{ff} \to Pa$. Wie wir sehen, handelt es sich hierbei nur um eine Fallunterscheidung, ob $a = \mathrm{tt}$ oder $a = \mathrm{ff}$ ist. Aber auch bei den natürlichen Zahlen gibt es Aussagen, die wir schon beweisen können, wenn wir Fallunterscheidung zwischen Null und Nachfolger machen. Nehmen wir zum Beispiel den modifizierten Vorgänger auf den natürlichen Zahlen. Diese Programmkonstante `Pred` ist gegeben durch die Berechnungsregeln `Pred` $0 := 0$ und `Pred` $Sn := n$. Wollen wir nun zeigen, dass `Pred` total ist, also $\forall_n(\mathbf{G}_{\mathbb{N}}n \to \mathbf{G}_{\mathbb{N}}\texttt{Pred}\ n)$, so können wir dies natürlich machen, indem wir das Eliminationsaxiom von $\mathbf{G}_{\mathbb{N}}n$ verwenden. In Minlog geben wir dann Folgendes ein:

```
(set-totality-goal "Pred")
(use "AllTotalElim")
(ind)
(intro 0)
(use "AllTotalIntro")
```

```
(assume "n^" "Tn^" "Spam")
(use "Tn^")
```

Nach dem Verwenden von `(use "AllTotalIntro")` haben wir die Ausgabe

```
ok, ?_4 can be obtained from

  n4099
-------------------------------------------------------------------
?_5:allnc n^(TotalNat n^ -> TotalNat(Pred n^)
             -> TotalNat(Pred(Succ n^)))
```

wobei die Induktionshypothese `TotalNat(Pred n^)` nicht verwendet werden muss. Betrachen wir den extrahierten Term, so hat dieser die Form

```
[n0][if n0 0 ([n1]n1)]
```

Es ist kein Rekursionsoperator im extrahierten Term, obwohl wir diesem bei der Verwendung von `(ind)` erhalten. Wir würden daher

```
[n0](Rec nat=>nat) n0 0 ([n1,n2]n1)
```

erwarten. Da wird jedoch keine Induktionshypothese verwendet haben, wird auch die gebundene Variable `n2` im obigen Term nicht verwendet. Der Rekursionsoperator kann in diesem Fall durch den Caseoperator ersetzt werden. Das geschieht in Minlog automatisch. Der Caseoperator wird in Minlog durch `[if ...]` darstellt. Dabei wird `...` durch die entsprechenden Argumente des Caseoperators gemäß Beispiel 1.2.17 ersetzt.
Will man im Beweis lediglich Fallunterscheindung über eine totale Variable machen, gibt es dafür den Befehl

```
(cases),
```

welcher analog zum Befehl `(ind)` angewendet wird. Ebenso gibt es auch für diesen Befehl die Erweiterung

```
(cases TERM),
```

um Fallunterscheidung über einen totalen Term `TERM` zu machen. Ein Beweis für die Totalität von `Pred` kann in Minlog dann fast analog durch

```
(set-totality-goal "Pred")
(use "AllTotalElim")
(cases)
(intro 0)
(use "AllTotalIntro")
(assume "n^" "Tn^")
(use "Tn^")
```

geführt werden. Der Unterschied liegt dann insbesondere darin, dass wir nach dem Befehl `(use "AllTotalIntro")` die Ausgabe

```
ok, ?_4 can be obtained from

  n3874
-------------------------------------------------------------------
?_5:allnc n^(TotalNat n^ -> TotalNat(Pred(Succ n^)))
```

erhalten, in welcher die Induktionshypothese `TotalNat(Pred n^)` nicht in den Prämissen steht.
Die hauptsächliche Anwendung des Befehls `cases` liegt bei Variablen, deren Typ eine Algebra mit Konstruktortypen der Form $\kappa(\xi) = \vec{\sigma} \to \xi$ ist. Das gilt zum Beispiel für die boolesche Algebra oder die Typsumme. Aber auch für das Typprodukt ist der Befehl `cases` sehr hilfreich, obwohl es nur einen Fall gibt. Als Lehrbeispiel führen wir die Programmkonstante `sort` ein, die ein Paar von natürlichen Zahlen so umordnet, dass die kleinere Zahl links steht. In Minlog machen wir das mit

```
(add-program-constant "sort"
    (py"(nat yprod nat)=>(nat yprod nat)"))
(add-computation-rules
"sort (n pair m)" "[if (m<n) (m pair n) (n pair m)]")
```

Hier haben wir bereits den Caseoperator in der Definition verwendet. Wir sehen, wie dieser für boolesche Terme zu interpretieren ist. In dem Fall `m<n = Truth`, reduziert der Ausdruck auf `m pair n` und ist `m<n = False`, wird er zu `n pair m`. Wir deklarieren `x` als eine Variable vom Typ `nat yprod nat` und setzen dann unsere Zielformel durch

```
(set-goal "all x lft(sort x) <= rht(sort x)").
```

Hier verwenden wir auch gleich den Befehl `(cases)`, denn Minlog zeigt dann alle möglichen Fälle auf, wie `x` aufgebaut sein kann. Hier gibt es nur den einen Fall, dass `x = n pair m` für natürliche Zahlen `n` und `m` ist. Die Ausgabe ist dann

```
 ok, ?_1 can be obtained from

  x3844
-------------------------------------------------------------------
?_2:all n,n0 lft(sort(n pair n0))<=rht(sort(n pair n0))
```

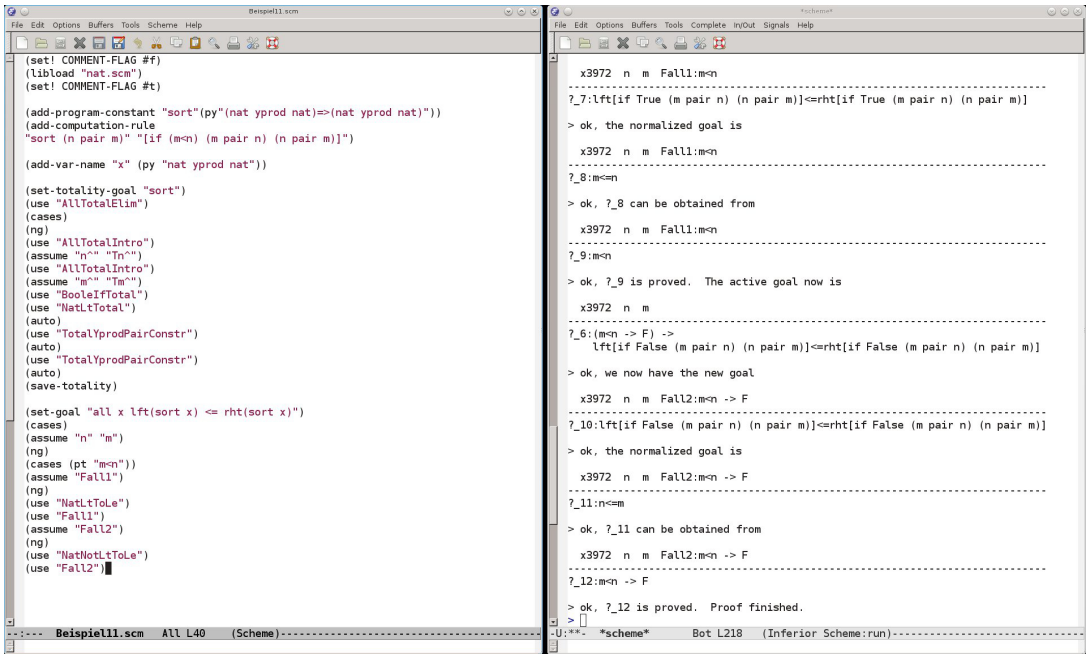
und wir haben erreicht, dass `x` zerlegt wurde. Nun nehmen wir die Variablen `n` und `m` mit `(assume "n" "m")` in den Kontext und normalisieren die Zielformel durch `(ng)`. Da `sort` durch Fallunterscheidung nach `m<n` definiert ist, liegt es nahe, den Beweis auch über Fallunterscheidung nach `m<n` zu führen. Wir geben daher nun `(cases (pt "m<n"))` ein. Als Ausgabe erhalten wir dann zwei mögliche Fälle:

```
ok, ?_4 can be obtained from

  x3851  n  m
-------------------------------------------------------------------
?_6:(m<n -> F) ->
    lft[if False (m pair n) (n pair m)]
    <=rht[if False (m pair n) (n pair m)]



  x3851  n  m
-------------------------------------------------------------------
?_5:m<n -> lft[if True (m pair n) (n pair m)]
         <=rht[if True (m pair n) (n pair m)]
```

Die Fallannahme ist jeweis als Prämisse gegeben und der Term `m<n` wurde schon entsprechend ersetzt. Im ersten Fall nehmen wir durch `(assume "Fall1")` die Fallannahme in den Kontext. Wenn wir nun die Zielformel normalisieren, erhalten wir

```
ok, the normalized goal is

  x3870  n  m  Fall1:m<n
------------------------------------------------------------------
?_8:m<=n
```

Das können wir aus der Fallannahme und dem Theorem `NatLtToLe` beweisen. Wir geben also `(use "NatLtToLe")` gefolgt von `(use "Fall1")` ein, womit der erste Fall bewiesen ist. Der Beweis im zweiten Fall geht analog, außer dass wir das Theorem `NatNotLtToLe` verwenden müssen, um aus der Fallannahme die Aussage `n<=m` zu erhalten. In der folgenden Grafik ist neben dem hier geführten Beweis auch der Beweis zur Totalität von `sort` angegeben.



Kapitel 3

Reelle Zahlen

3.1 Definition von reellen Zahlen

Motivation 3.1.1. Wie üblich sollen auch im konstruktiven Sinne die reellen Zahlen die Vervollständigung der rationalen Zahlen bilden. Dass man dafür reelle Zahlen als Cauchyfolgen von rationalen Zahlen definiert, ist dem Leser womöglich schon bekannt. Im konstruktiven Fall reicht eine Folge alleine jedoch nicht aus. Die Aussage, dass es sich um eine Cauchyfolge handelt, braucht auch einen Zeugen, den Cauchymodulus. Wir werden daher am Anfang dieses Abschnittes reelle Zahlen als ein Paar bestehend aus einer Cauchyfolge und einem Cauchymodulus definieren. Weiter werden wir auf den reellen Zahlen eine Gleichheit und die arithmetischen Funktionen definieren und sehen, dass die reellen Zahlen eine Ringstruktur bezüglich der Addition und der Multiplikation haben. Am Ende dieses Abschnittes werden wir die Ordnungsrelationen $<$ und $\leq$ auf den reellen Zahlen definieren und dazu wichtige Eigenschaften beweisen. Eine davon wird im Vergleichbarkeitssatz gegeben werden. Um reelle Zahlen mit Folgen von rationalen Zahlen zu definieren, brauchen wir zunächst natürlich die rationalen Zahlen, womit sich der folgende Teilabschnitt beschäftigt.

3.1.1 Positive, ganze und rationale Zahlen

Definition 3.1.2. Wie in Beispiel 1.2.4 ist die Algebra der positiven Zahlen gegeben durch

$$\mathbb{P} := \mu_\xi(\xi, \xi \to \xi, \xi \to \xi).$$

Der nullstellige Konstruktor wird mit 1 bezeichnet. Die beiden einstelligen Konstruktoren haben die Bezeichnung S_0 und S_1. Ist p eine positive Zahl, schreibt man auch $p0$ bzw. $p1$ anstelle von $S_0 p$ bzw. $S_1 p$.
Aus den positiven Zahlen definieren wir die Algebra der ganzen Zahlen durch

$$\mathbb{Z} := \mu_\xi(\xi, \mathbb{P} \to \xi, \mathbb{P} \to \xi).$$

Den nullstelligen Konstruktor bezeichen wir mit 0 und die beiden einstelligen Konstruktoren mit $+$ bzw. $-$. Anstelle von $+p$ schreiben wir auch einfach nur p.

Bemerkung 3.1.3. Die positiven Zahlen lassen sich als die Binärdarstellung der natürlichen Zahlen ohne 0 interpretieren. In Minlog gibt es dafür die Programmkonstante `PosToNat`:

```
PosToNat
  comprules
0 PosToNat 1 Succ Zero
1 PosToNat(SZero p) NatDouble(PosToNat p)
2 PosToNat(SOne p) Succ(PosToNat(SZero p))
```

Auf dieser Interpretation beruhend, lassen sich die arithmetischen Operationen wie die Addition, Multiplikation oder das Potenzieren auf den positiven Zahlen definieren. Die Programmkonstante `PosToNat` ist bezüglich dieser Operationen für totale Terme ein Homomorphismus. Das heißt, dass zum Beispiel $\texttt{PosToNat}\,p + \texttt{PosToNat}\,q = \texttt{PosToNat}(p+q)$ für totale positive Zahlen p und q gilt.
Analoges gilt auch für die boolschwertigen Operationen $\leq$ und $<$. Für die genauen Definitionen verweisen wir auf die Datei `pos.scm` in der Bibliothek von Minlog.

Bemerkung 3.1.4. Wir sehen, dass die Algebra der ganzen Zahlen mit den schon bekannten ganzen Zahlen aus den Anfängervorlesungen übereinstimmt. Insbesondere lassen sich die natürlichen Zahlen in die ganzen Zahlen einbetten. In Minlog geschieht dies durch die Programmkonstane `NatToInt`.
Im Gegensatz zu den „herkömmlichen" ganzen Zahlen, werden die hier definierten ganzen Zahlen nicht als Paare von natürlichen Zahlen definiert. Stattdessen werden die ganzen Zahlen per Definition in die positiven ganzen Zahlen, die negativen ganzen Zahlen und die 0 eingeteilt. Die Berechnungsregeln für die arithmetischen Operationen sind dadurch etwas komplexer als man erwarten könnte. Dazu sei auf die Datei `int.scm` in der Minlogbibliothek verwiesen. Auch hier reicht uns ein intuitives Verständnis.

Definition 3.1.5. Die Algebra der rationalen Zahlen ist gegeben durch

$$\mathbb{Q} := \mu_\xi(\mathbb{Z} \to \mathbb{P} \to \xi) = \mathbb{Z} \times \mathbb{P}.$$

Wir schreiben anstelle von $C_0 kp$ die übliche Schreibweise $\frac{k}{p}$.

Motivation 3.1.6. Liegt eine Mengenlehre zugrunde, werden rationale Zahlen als Äquivalenzklassen auf der Menge $\mathbb{Z} \times \mathbb{P}$ definiert. In der Theorie **TCF** sind Mengen keine Objekte. Wir müssen also direkt mit der Äquivalenzrelation auf den rationalen Zahlen arbeiten.

Definition 3.1.7. Auf den rationalen Zahlen definieren wir die rationale Gleichheit $\simeq$ durch die Berechnungsregel

$$\left(\frac{k}{p} \simeq \frac{l}{q}\right) := \left(kq = lp\right).$$

Wir schreiben auch häufig einfach $=$ für $\simeq$. Zwischen den rationalen Zahlen gibt es zwar die entscheidbare Gleichheit, welche auch mit $=$ bezeichnet wird. Diese werden wir aber nicht verwenden, ohne es explizit anzugeben. Bei Minlog wird in der Bibliotheksdatei `rat.scm` die rationale Gleichheit mit == bezeichnet.

Bemerkung 3.1.8. Für die Definition der arithmetischen und booleschen Operatoren in $\mathbb{Q}$ verweisen wir auf die Datei `rat.scm`. Die Operatoren sind mit der rationalen Gleichheit kompatibel, was auch in der Datei bewiesen ist.
An dieser Stelle sei auch der Befehl

```
(simprat FORMEL)
```

erwähnt. Dieser ist analog zum Befehl `simp` aus Abschnitt 2.3.4 mit dem Unterschied, dass anstelle von `FORMEL` eine rationale Gleichheit eingesetzt wird. Natürlich müssen die entsprechenden Funktionen bzw. Prädikate in der Zielformel mit der rationalen Gleichheit kompatibel sein. Wie bei `simp` kann man mit Hinzufügen von `"<-"` die Gleichheit auch in die andere Richtung nutzen.

3.1.2 Reelle Zahlen als Cauchyfolge

Definition 3.1.9. Eine reelle Zahl ist ein Paar $\langle as, M\rangle$, wobei $as : \mathbb{N} \to \mathbb{Q}$ eine Folge rationaler Zahlen und $M : \mathbb{P} \to \mathbb{N}$ eine Funktion von den positiven Zahlen in die natürlichen Zahlen ist. Dabei muss weiter gelten, dass M monoton steigend ist und as eine Cauchyfolge mit Modulus M ist. In Formeln bedeutet das

$$\forall_{p,q}(p \le q \to M(p) \le M(q))$$

und

$$\forall_p \forall_{n,m \ge M(p)} |as(n) - as(m)| \le 2^{-p}.$$

Die Eigenschaft, eine reelle Zahl zu sein, ist damit ein einstelliges Prädikat auf dem Typ $(\mathbb{N} \to \mathbb{Q}) \times (\mathbb{P} \to \mathbb{N})$. Wir schreiben $x \in \mathbb{R}$, wenn wir explizit aussagen wollen, dass x eine reelle Zahl ist. Verwenden wir die Variablen x, y und z in Formeln, so sind diese für uns schon implizit reelle Zahlen. Das heißt, Ausdrücke wie $\forall_x A$ und $\exists_x A$ sind zu verstehen als $\forall_x^{nc}(x \in \mathbb{R} \to A)$ und $\exists_x^r(x \in \mathbb{R} \wedge A)$.

Bemerkung 3.1.10. In Minlog werden reelle Zahlen in der Bibliotheksdatei `rea.scm` eingeführt. Dort wird die Eigenschaft, eine reelle Zahl zu sein, als induktiv definiertes Prädikat realisiert:

```
Real non-computational
RealIntro: all x(Cauchy(x seq)(x mod) -> Mon(x mod) -> Real x)
```

Für $x = \langle as, M\rangle$ ist dabei `x seq = as` und `x mod = M`. `Cauchy` und `Mon` sind dabei wieder induktiv definierte Prädikate und jeweils wie die Formeln in obiger Definition zu verstehen. Die Eigenschaft, eine reelle Zahl zu sein, muss in Minlog natürlich immer explizit mitangegeben werden.
In fast jedem Beweis mit reellen Zahlen in Minlog muss oft gezeigt werden, dass ein Objekt eine reelle Zahl ist. Das ist meistens unmittelbar aus den Annahmen ersichtlich. Zum Beispiel kann dieses Objekt in einer reellen Gleichung oder Ungleichung vorkommen oder das Ergebnis von arithmetischen Operationen mit reellen Zahlen sein. Dies werden wir in den folgenden Kapiteln noch formal diskutieren. In Minlog muss jedoch jedes Mal explizit bewiesen werden, dass eine Zahl reell ist. Um dies abzukürzen, gibt es dort das Kommando

```
(realproof).
```

Es handelt sich dabei um einen spezialisierten `search`-Befehl wie in Abschnitt 2.1.11. Minlog geht mit diesem Befehl genau die Theoremen durch, welche als Konklusion haben, dass ein Objekt eine reelle Zahl ist.

Notation 3.1.11. Eine rationale Zahl a werden wir auch als reelle Zahl auffassen, indem wir a mit dem Paar $\langle \lambda_n a, \lambda_p 0\rangle$ aus der konstanten Cauchyfolge und dem Nullmodulus identifizieren.

3.1.3 Gleichheit von reellen Zahlen

Definition 3.1.12. Die reelle Gleichheit ist wie folgt definiert: Wir sagen, zwei reelle Zahlen $\langle as, M\rangle, \langle bs, N\rangle$ sind gleich und schreiben $\langle as, M\rangle \cong \langle bs, N\rangle$, wenn

$$\forall_p |as(M(p+1)) - bs(N(p+1))| \leq 2^{-p}$$

Bei der reellen Gleichheit kann es auch keine Verwechslung mit der berechenbaren Gleichheit geben, weil diese für reelle Zahlen nicht definiert ist.

gilt. Auch hier schreiben wir häufig $=$ anstelle von $\cong$. In der Datei `rea.scm` wird die reelle Gleichheit mit `===` bezeichnet.

Motivation 3.1.13. Dass die Gleichheit für reelle Zahlen reflexiv und symmetrisch ist, sieht man direkt anhand der Definition. Um zu zeigen, dass sie transitiv ist, brauchen wir die Charakterisierung der Gleichheit aus dem folgenden Lemma. Diese hat den Vorteil, dass sie ohne die Verwendung des Cauchymodulus formuliert ist. Deswegen werden wir die Charakterisierung noch häufig verwenden.

Lemma 3.1.14. Zwei reelle Zahlen $x = \langle as, M\rangle$ und $y = \langle bs, N\rangle$ sind genau dann gleich, wenn

$$\forall_p \exists_n \forall_{m\geq n} |as(m) - bs(m)| \leq 2^{-p}$$

gilt.

Beweis. Es gelte zunächst $x \cong y$. Zu einem fixierten p setzen wir $n = \max\{M(p+2), N(p+2)\}$. Dann gilt für $m \geq n$:

$$\begin{aligned}
&|as(m) - bs(m)| \leq \\
&|as(m) - as(M(p+2))| + |as(M(p+2)) - bs(M(p+2))| + |bs(M(p+2)) - bs(m)| \\
&\leq 2^{-p-2} + 2^{-p-1} + 2^{-p-2} = 2^{-p}
\end{aligned}$$

Es gelte nun $\forall_p \exists_n \forall_{m\geq n} |as(m) - bs(m)| \leq 2^{-p}$ und unser Ziel ist $x \cong y$. Dazu sei p gegeben. Wir wollen $|as(M(p+1)) - bs(N(p+1))| \leq 2^{-p}$ zeigen. Für jedes beliebige q erhalten wir aus der gegeben Aussage ein n, sodass $\forall_{m\geq n} |as(m) - bs(m)| \leq 2^{-q}$ gilt. Es folgt für ein $m \geq \max\{n, M(p+1), N(p+1)\}$ dann:

$$\begin{aligned}
&|as(M(p+1)) - bs(N(p+1))| \leq \\
&|as(M(p+1)) - as(m)| + |as(m) - bs(m)| + |bs(m) - bs(M(p+1))| \leq \\
&2^{-p-1} + 2^{-q} + 2^{-p-1}
\end{aligned}$$

Weil das für jede positive Zahl q gilt, haben wir $|as(M(p+1)) - bs(N(p+1))| \leq 2^{-p}$ und damit $x \cong y$. □

Korollar 3.1.15. Aus Lemma 3.1.14 folgt direkt, dass zwei reelle Zahlen bereits gleich sind, wenn ihre Cauchyfolgen für alle bis auf endlich viele Werte übereinstimmen.

Satz 3.1.16. Die Gleichheit $\cong$ auf den reellen Zahlen ist eine Äquivalenzrelation.

Beweis. Dass $\cong$ reflexiv und symmetrisch ist, folgt direkt aus der Definition. Für die Transitivität seien die reellen Zahlen x, y und z mit ihren Cauchyfolgen as, bs und cs gegeben und es gelte $x \cong y$ sowie $y \cong z$. Wir verwenden die Charakterisierung aus Lemma 3.1.14. Sei also p gegeben. Wegen $x \cong y$ und $y \cong z$ erhalten wir n_1 und n_2 mit $\forall_{m\geq n_1} |as(n) - bs(n)| \leq 2^{-p-1}$ und $\forall_{m\geq n_2} |bs(n) - cs(n)| \leq 2^{-p-1}$. Mit der Dreiecksungleichung folgt damit für alle $m \geq \max\{n_1, n_2\}$: $|as(m) - cs(m)| \leq 2^{-p}$. Also haben wir $x \cong z$. □

Bemerkung 3.1.17. In den folgenden Abschnitten gehen wir unter anderem auf die Kompatibilität der reellen Gleichheit mit den dort definierten Operationen und Prädikaten ein. Um die Kompatibilität mit der reellen Gleichheit in Minlog effizient zu nutzen, gibt es wie bei der rationalen Gleichheit auch für die reelle Gleichheit den Befehl

```
(simpreal FORMEL).
```

Für `FORMEL` wird hier eine reelle Gleichheit erwartet und es muss für Minlog ersichtlich sein, dass die Objekte in der Gleichheit reelle Zahlen sind. Auch `simpreal` kann mit `"<-"` modifiziert werden.

3.1.4 Nicht-negative und positive reelle Zahlen

Motivation 3.1.18. Wir wollen die Relationen $\leq$ und $<$ auf den reellen Zahlen einführen. Nachdem wir im nächsten Teilabschnitt die arithmetischen Funktionen auf den reellen Zahlen definiert haben, werden wir $x \leq y$ bzw. $x < y$ dadurch definieren, dass $y - x$ nicht-negativ bzw. positiv ist. Deswegen definieren wir zunächst in diesem Teilabschnitt, was es heißt, dass eine reelle Zahl nicht-negativ oder positiv ist.

Definition 3.1.19. Eine reelle Zahl $x := \langle as, M\rangle$ heißt nicht-negativ , wenn

$$\forall_p -2^{-p} \leq as(M(p))$$

gilt. Wir schreiben dann $x \in \mathbb{R}_0^+$.
Wir nennen die reelle Zahl x p-positiv und schreiben $x \in_p \mathbb{R}^+$, wenn

$$2^{-p} \leq as(M(p+1))$$

gilt. Gibt es ein p, sodass x p-positiv ist, sagen wir nur „x ist positiv“ und schreiben $x \in \mathbb{R}^+$.

Bemerkung 3.1.20. In der Aussage, dass eine reelle Zahl positiv ist, haben wir einen rechnerischen Gehalt. Übernimmt man die Notation aus der obigen Definition, ist das der Zeuge p. Im Gegensatz dazu ist die Aussage, dass eine reelle Zahl nicht-negativ ist, ohne rechnerischen Gehalt, da es sich um eine allquantifizierte Ungleichung rationaler Zahlen handelt. Auch die p-Positivität ist für ein konkretes p nicht rechnerisch. In Minlog ist die Eigenschaft, nicht-negativ zu sein, durch ein induktiv definiertes Prädikat realisiert:

```
RealNNeg non-computational
RealNNegIntro: all x(Real x -> all p 0<=x seq(x mod p)+(1#2**p)
                      -> RealNNeg x)
```

Die Eigenschaft, p-positiv zu sein, wird dort durch eine Programmkonstante gegeben:

```
RealPos
  comprules
0 RealPos(RealConstr as M)p (1#2**p)<=as(M(p+1))
```

Motivation 3.1.21. Wir wollen nun zeigen, dass Nicht-Negativität und Positivität mit der reellen Gleichheit kompatibel sind. Ähnlich wie bei der Gleichheit ist dazu eine Charakterisierung, welche unabhängig vom Cauchymodulus ist, nützlich. Wir werden diese zunächst für die Nicht-Negativität und dann für die Positivität geben. Zu bemerken sei noch, dass die p-Positivität für festes p nicht kompatibel mit der reellen Gleichheit ist, wie man sich leicht überlegen kann.

Lemma 3.1.22. Eine reelle Zahl $x = \langle as, M\rangle$ ist genau dann nicht-negativ, wenn

$$\forall_p \exists_n \forall_{m\geq n} -2^{-p} \leq as(m)$$

gilt.

Beweis. Sei zuerst x nicht-negativ. Zu einem gegeben p setzten wir $n = M(p+1)$. Dann folgt für jedes $m \geq n$

$$\begin{aligned} -2^{-p} &\leq -2^{-p-1} + as(M(p+1)) = -2^{-p-1} + (as(M(p+1)) - as(m)) + as(m) \\ &\leq -2^{-p-1} + 2^{-p-1} + as(m) = as(m). \end{aligned}$$

Gelte nun $\forall_p \exists_n \forall_{m\geq n} -2^{-p} \leq as(m)$ und sei p gegeben. Unser Ziel ist $-2^{-p} \leq as(M(p))$. Für ein beliebiges q haben wir nach Voraussetzung ein n mit $\forall_{m\geq n} -2^{-q} \leq as(m)$. Es folgt für ein $m \geq \max\{n, M(p)\}$ dann:

$$\begin{aligned} -2^{-p} - 2^{-q} &\leq -2^{-p} + as(m) = -2^{-p} + (as(m) - as(M(p))) + as(M(p)) \\ &\leq -2^{-p} + 2^{-p} + as(M(p)) = as(M(p)). \end{aligned}$$

Weil das für alle q gilt, folgt $-2^{-p} \leq as(M(p))$ und somit $x \in \mathbb{R}_0^+$. □

Korollar 3.1.23. Anhand dieses Lemmas sieht man, dass eine reelle Zahl schon nicht-negativ ist, wenn alle bis auf endlich viele Werte ihrer Cauchyfolge nicht-negativ sind.

Lemma 3.1.24. Nicht-Negativität ist verträglich mit der reellen Gleichheit. Das heißt, sind x, y reelle Zahlen und gilt $x = y$ und $x \in \mathbb{R}_0^+$, dann folgt $y \in \mathbb{R}_0^+$.

Beweis. Seien $x = \langle as, M\rangle$ und $y = \langle bs, N\rangle$ gegeben und es gelte $x = y$ und $x \in \mathbb{R}_0^+$. Um $y \in \mathbb{R}_0^+$ zu zeigen, verwenden wir Lemma 3.1.22. Das heißt, zu einem gegebenen p beweisen wir die Aussage $\exists_n \forall_{m\geq n} -2^{-p} \leq bs(m)$. Mit Lemma 3.1.14 erhalten wir aus $x = y$ die Aussage

$$\forall_p \exists_n \forall_{m\geq n} |as(m) - bs(m)| \leq 2^{-p}$$

und aus Lemma 3.1.22 erhalten wir aus $x \in \mathbb{R}_0^+$ die Aussage

$$\forall_p \exists_n \forall_{m\geq n} -2^{-p} \leq as(m).$$

Spezialisieren wir die beiden gegeben Aussagen auf $p+1$, erhalten wir n_0 und n_1, sodass für jedes $m \geq \max\{n_0, n_1\}$ die Ungleichungen $|as(m) - bs(m)| \leq 2^{-p-1}$ und $-2^{-p-1} \leq as(m)$ gelten. Setzten wir daher $n := \max\{n_0, n_1\}$, dann haben wir für jedes $m \geq n$

$$\begin{aligned} -2^{-p} = -2^{-p-1} - 2^{-p-1} &\leq |as(m) - bs(m)| + as(m) \\ &\leq bs(m) - as(m) + as(m) = bs(m). \end{aligned}$$

Also folgt $y \in \mathbb{R}_0^+$. □

Lemma 3.1.25. Sei $x = \langle as, M\rangle$ eine reelle Zahl. Dann folgt aus $x \in_p \mathbb{R}^+$ die Aussage $\forall_{n\geq M(p+1)} 2^{-p-1} \leq as(n)$.
Gilt in der anderen Richtung $\exists_n \forall_{m\geq n} 2^{-q} \leq as(m)$, so folgt $x \in_{q+1} \mathbb{R}^+$.
Insbesondere ist eine reelle Zahl x genau dann positiv, wenn es eine positive Zahl p gibt, sodass alle bis auf endlich viele Werte ihrer Cauchyfolge größer oder gleich 2^{-p} sind.

Beweis. Es gelte $x \in_p \mathbb{R}^+$. Nach Definition heißt das $2^{-p} \leq as(M(p+1))$. Für $n \geq M(p+1)$ erhalten wir dadurch

$$\begin{aligned} 2^{-p-1} &= -2^{-p-1} + 2^{-p} \leq -2^{-p-1} + as(M(p+1)) \\ &\leq -2^{-p-1} + |as(M(p+1)) - as(n)| + as(n) \leq as(n). \end{aligned}$$

Es sei nun n gegeben, sodass $\forall_{m \geq n} 2^{-q} \leq as(m)$ ist, dann haben wir für $m \geq \max\{as(M(q+2)), n\}$

$$\begin{aligned} 2^{-q-1} &\leq -2^{-q-2} + 2^{-q} \leq -2^{-q-2} + as(m) \\ &\leq (as(M(q+2)) - as(m)) + as(m) = as(M(q+2)). \end{aligned}$$

Somit folgt $x \in_{q+1} \mathbb{R}^+$. □

Lemma 3.1.26. Positivität ist verträglich mit der reellen Gleichheit. Genauer gesagt: Sind x, y reelle Zahlen und gilt $x = y$ und $x \in_p \mathbb{R}^+$, dann folgt $y \in_{p+2} \mathbb{R}^+$.

Beweis. Es sei $x := \langle as, m\rangle$ and $y := \langle bs, N\rangle$ gegeben und es gelte $x = y$ und $x \in_p \mathbb{R}^+$. Aus Lemma 3.1.25 erhalten wir $2^{-p-1} \leq as(M(p+3))$, weil $M(p+3) \geq M(p+1)$ ist, und aus $x = y$ bekommen wir $-2^{-p-2} \leq bs(N(p+3)) - as(M(p+3))$. Zusammen haben wir

$$2^{-p-2} \leq 2^{-p-1} - 2^{-p-2} \leq as(M(p+3)) + bs(N(p+3)) - as(M(p+3) = bs(N(p+3))$$

und somit $y \in_{p+2} \mathbb{R}^+$. □

3.1.5 Arithmetische Funktionen auf den reellen Zahlen

Definition 3.1.27. Es seien zwei reelle Zahlen $x := \langle as, M\rangle$ und $y := \langle bs, N\rangle$ gegeben. Wir definieren ihre Summe $x + y := \langle cs, L\rangle$ durch $cs(n) := as(n) + bs(n)$ für alle n und $L(p) := \max\{M(p+1), N(p+1)\}$ für alle p.
Weiter definieren wir das Negative von x durch $-x := \langle \lambda_n - as(n), M\rangle$ und den Betrag von x durch $|x| := \langle \lambda_n |as(n)|, M\rangle$. Außerdem schreiben wir $x - y$ für $x + (-y)$.

Lemma 3.1.28. Sind x, y zwei reelle Zahlen, so sind auch $x + y$, $-x$ und $|x|$ reelle Zahlen.

Beweis. Dass der Modulus jeweils monoton ist, ist offensichtlich und, dass es sich um Cauchyfolgen handelt, folgt für die Summe und den Betrag aus der Dreiecksungleichung für rationale Zahlen und für das Negative direkt aus der Definition. □

Motivation 3.1.29. Wir wollen nun auch die Multiplikation von reellen Zahlen und das Inverse einer reellen Zahl definieren. Hierzu brauchen wir noch die archimedische Eigenschaft der reellen Zahlen, welche im nächsten Lemma formuliert wird:

Lemma 3.1.30. Für jede reelle Zahl $x := \langle as, M\rangle$ gibt es eine positive Zahl p_x, sodass $|as(n)| \leq 2^{p_x}$ für alle n gilt.

Beweis. Wir wählen p_x, so groß, dass für jedes $m \leq M(1)$ gilt, dass $|as(m)| + 2^{-1} \leq 2^{p_x}$ ist. Ist nun n gegeben, gilt entweder $n \leq M(1)$, dann haben wir nach Wahl von p_x bereits $|as(n)| + 2^{-1} \leq 2^{p_x}$. Ansonsten gilt $n \geq M(1)$ und wir haben $|as(n)| \leq |as(m) - as(M(1))| + |as(M(1))| \leq 2^{-1} + |as(M(1))| \leq 2^{p_x}$. □

Definition 3.1.31. Es seien zwei reelle Zahlen $x := \langle as, M\rangle$ und $y := \langle bs, N\rangle$ gegeben. Nach Lemma 3.1.30 gibt es positive Zahlen p_x und p_y, sodass $\forall_n |as(n)| \leq 2^{p_x}$ und $\forall_n |bs(n)| \leq 2^{p_y}$ gilt. Damit definieren wir nun das Produkt $xy := \langle cs, L\rangle$ durch $cs(n) = as(n)bs(n)$ für alle n und $L(p) := \max\{M(p+1+p_y), N(p+1+p_x)\}$ für alle p.
Gibt es außerdem eine positive Zahl q mit $y \in_q \mathbb{R}^+$, so definieren wir das Inverse $\frac{1}{y} := \langle ds, K\rangle$ von y durch

$$ds(n) := \begin{cases} \frac{1}{bs(n)} & \text{wenn } bs(n) \neq 0 \\ 0 & \text{wenn } bs(n) = 0 \end{cases}$$

für alle n und $K(p) := M(2q+2+p)$. Anstelle von $x\frac{1}{y}$ schreiben wir auch $\frac{x}{y}$.

Bemerkung 3.1.32. Die obere Schranke aus Lemma 3.1.30 hängt besonders von den vorderen Gliedern der Cauchyfolge der reellen Zahl ab. Wie man leicht sieht, ist diese obere Schranke daher nicht kompatibel mit der reellen Gleichheit, da eine Änderung von endlich vielen Werten der Cauchyfolge die obere Schranke beeinflussen kann, aber reelle Gleichheit erhalten bleibt.
Bei der Definition der Multiplikation hängt der Cauchymodulus auch von der Wahl der oberen Schranken ab. Die obere Schranke haben wir nicht explizit angegeben. Da aber die Cauchyfolge unabhängig von dieser Wahl ist, haben wir damit das Produkt von reellen Zahlen zumindest bis auf reelle Gleichheit wohldefiniert. In der Datei `rea.scm` ist die obere Schranke genau gegeben, sodass dort das Produkt reeller Zahlen vollständig wohldefiniert ist. Für uns reicht aber eine Definition bis auf reelle Gleichheit aus.
Bei dem Inversen einer reellen Zahl ist es ähnlich. Hier hängt die Definition von dem Zeugen der Positivität ab und ist damit auch nur bis auf reelle Gleichheit wohldefiniert. In Minlog muss der Zeuge bei der Programmkonstante `RealInv` immer zusätzlich angegeben werden:

```
RealInv
  comprules
0 RealInv(RealConstr as M)q
  RealConstr([n]1/as n)([p]M(2*PosS q+p)max M(PosS q))
```

Auch hier reicht für uns eine Definition bis auf reelle Gleichheit aus.

Lemma 3.1.33. Sind x, y reelle Zahlen, so ist auch xy eine reelle Zahl. Gilt außerdem $y \in_q \mathbb{R}^+$ für eine positive Zahl q, so ist auch $\frac{1}{y}$ eine reelle Zahl.

Beweis. Seien $x := \langle as, M\rangle$ und $y := \langle bs, N\rangle$ gegeben und die Notation ansonsten auch wie in obiger Definition. Wir zeigen zunächst, dass xy eine reelle Zahl ist. Der Modulus L von xy ist gegeben durch $L(p) := \max\{M(p+1+p_y), N(p+1+p_x)\}$ für jedes p. Man erkennt damit leicht, dass L monoton steigend ist. Bleibt noch, die Cauchyeigenschaft zu zeigen. Seien dafür p und $n, m \geq L(p)$ gegeben. Wir haben dann

$$\begin{aligned} |as(n)bs(n) - as(m)bs(m)| = & \\ & |as(n)bs(n) - as(n)bs(m) + as(n)bs(m) - as(m)bs(m)| \\ \leq & |as(n)||bs(n) - bs(m)| + |bs(m)||as(n) - as(m)| \\ \leq & 2^{p_x}|bs(n) - bs(m)| + 2^{p_y}|as(n) - as(m)| \\ \leq & 2^{p_x}2^{-p-1-p_x} + 2^{p_y}2^{-p-1-p_y} = 2^{-p} \end{aligned}$$

und somit ist die Cauchyeigenschaft gezeigt.
Wir zeigen nun, dass $\frac{1}{y}$ eine reelle Zahl ist, wenn y q-positiv ist. Der zu $\frac{1}{y}$ gehörige Modulus K ist gegeben durch $K(p) := M(2q+2+p)$ für jedes p. Dieser ist damit offensichtlich monoton steigend. Außerdem wissen wir nach Lemma 3.1.25, dass $2^{-q-1} \leq |as(n)|$ für jedes $n \geq M(q+1)$ ist, also insbesondere auch für jedes $n \geq M(2q+2+p) = K(p)$. Bezeichnet ds die Cauchyfolge von $\frac{1}{y}$, folgt somit für $n, m \geq K(p)$

$$\begin{aligned} |ds(n) - ds(m)| &= \left| \frac{1}{as(n)} - \frac{1}{as(m)} \right| = \frac{|as(m) - as(n)|}{|as(n)as(m)|} \\ &\leq \frac{|as(m) - as(n)|}{2^{-q-1}2^{-q-1}} \leq 2^{2+2q}2^{-2q-2-p} = 2^{-p}. \end{aligned}$$

Das zeigt die Cauchyeigenschaft. □

Lemma 3.1.34. Die reellen Zahlen bilden zusammen mit der Addition und der Multiplikation einen kommutativen Ring. Dabei ist das Nullelement die reelle Zahl 0, das additiv Inverse einer reellen Zahl ist ihr Negatives und das Einselement ist gegeben durch die reelle Zahl 1.
Die Gleichheit in diesem Ring ist die reelle Gleichheit.

Beweis. Jedes Axiom des Ringes folgt direkt aus Lemma 3.1.14, indem man jede zu beweisende Gleichheit auf die rationale Gleichheit der Werte der Cauchyfolgen reduziert. □

Lemma 3.1.35. Die arithmetischen Funktionen aus Definition 3.1.27 und Definition 3.1.31 sind kompatibel mit der reellen Gleichheit.

Beweis. Die Beweise folgen direkt aus der Charakterisierung der reellen Gleichheit von Lemma 3.1.14 und einigen Abschätzungen. Als Beispiel zeigen wir, dass die Multiplikation mit der reellen Gleichheit kompatibel ist:
Weil die Multiplikation wegen Lemma 3.1.34 kommutativ ist, reicht es, für reelle Zahlen x, y und z die Aussage $x = y \to xz = yz$ zu zeigen. Wir bezeichnen dafür die Cauchyfolgen von x, y und z mit as, bs und cs und verwenden Lemma 3.1.14. Das heißt, zu einem gegeben p beweisen wir

$$\exists_n \forall_{m \geq n} |as(m)cs(m) - bs(m)cs(m)| \leq 2^{-p}.$$

Aus Lemma 3.1.30 erhalten wir ein p_z, sodass $\forall_n |cs(n)| \leq 2^{p_z}$ ist. Wegen $x = y$ haben wir wieder mit Lemma 3.1.25 ein n, sodass $\forall_{m \geq n} |as(m) - bs(m)| \leq 2^{-p-p_z}$ gilt. Damit folgt für jedes $m \geq n$

$$|as(m)cs(m) - bs(m)cs(m)| = |as(m) - bs(m)||cs(m)| \leq 2^{-p-p_z}2^{p_z} = 2^{-p}.$$

Somit haben wir $xz = yz$ gezeigt. □

Lemma 3.1.36. Im Ring der reellen Zahlen ist eine Element x genau dann invertierbar, wenn sein Betrag positiv ist. Das Inverse ist in diesem Fall $\frac{1}{x}$.

Beweis. Es sei zunächst $x \in_p \mathbb{R}^+$ für ein p. Dann sind die Cauchyfolgen der reellen Zahlen $x\frac{1}{x}$ und 1 für alle bis auf endlich viele Werte gleich. Mit Korollar 3.1.15 folgt dann $x\frac{1}{x} = 1$.
Für die andere Richtung der Äquivalenz sei angenommen, dass eine reelle Zahl y existiert, sodass $xy = 1$ gilt. Mit as und bs bezeichnen wir die Cauchyfolgen von x und y. Wir wählen nach Lemma 3.1.30 p_y so, dass $\forall_n |bs(n)| \leq 2^{p_y}$ gilt. Aus $xy = 1$

erhalten wir mit Lemma 3.1.14 ein n, sodass $\forall_{m\geq n}|as(m)bs(m)-1|\leq\frac{1}{2}$ gilt. Damit ist $as(m)bs(m)\geq\frac{1}{2}$ für jedes $m\geq n$ und damit auch $|as(m)bs(m)|\geq\frac{1}{2}$. Es folgt $bs(m)\neq 0$ und deswegen $|as(m)|\geq\frac{1}{2|bs(m)|}\geq 2^{-p_y-1}$. Lemma 3.1.25 liefert uns, dass $|x|$ positiv ist. □

3.1.6 Vergleichbarkeit reeller Zahlen

Definition 3.1.37. Für reelle Zahlen x und y sagen wir „x ist kleiner oder gleich y" und schreiben $x\leq y$, wenn $y-x$ nicht-negativ ist.
Für eine positive Zahl q sagen wir „x ist q-kleiner als y" und schreiben $x<_q y$, wenn $y-x$ q-positiv ist. Wenn ein q existiert mit $x<_q y$, sagen wir auch einfach „x ist kleiner y" und schreiben $x<y$.

Korollar 3.1.38. Seien $x:=\langle as,M\rangle$ und $y:=\langle bs,N\rangle$ reelle Zahlen.
Nach Lemma 3.1.22 ist $x\leq y$ äquivalent zu der Aussage

$$\forall_p\exists_n\forall_{m\geq n}as(m)\leq bs(m)+2^{-p}.$$

Nach Lemma 3.1.25 folgt aus $x<_p y$ die Aussage

$$\forall_{n\geq\max\{M(p+2),N(p+2)\}}as(n)+2^{-p-1}\leq bs(n)$$

und aus $\exists_n\forall_{m\geq n}as(m)+2^{-p}\leq bs(m)$ folgt $x<_{p+1}y$ auch nach Lemma 3.1.25. Insgesamt gilt also, dass $x<y$ äquivalent zu $\exists_{p,n}\forall_{m\geq n}as(m)+2^{p}\leq bs(m)$ ist.

Lemma 3.1.39. Jede reelle Zahl lässt sich beliebig genau durch rationale Zahlen approximieren. In einer Formel heißt das

$$\forall_{x,p}\exists_a|a-x|\leq 2^{-p}.$$

Beweis. Es sei $x:=\langle as,M\rangle$ und p gegeben. Wir zeigen $|as(M(p))-x|\leq 2^{-p}$. Das ist per Definition äquivalent zu $-2^{-q}\leq|as(M(p))-as(M(q+2))|-2^{-p}$ also $|as(M(p))-as(M(q+2))|\leq 2^{-p}+2^{-q}$ für jedes q. Wir machen daher folgende Abschätzungen:

$$\begin{aligned}&|as(M(p))-as(M(q+2))|\\&\quad\leq|as(M(p))-as(M(p+q+2))|+|as(M(p+q+2))-as(M(q+2))|\\&\quad\leq 2^{-p}+2^{-q-2}\leq 2^{-p}+2^{-q}.\end{aligned}$$

□

Lemma 3.1.40. Die Relation $\leq$ auf den reellen Zahlen ist reflexiv, antisymmetrisch und transitiv. Die Relation $<$ auf den reellen Zahlen ist irreflexiv und transitiv.

Beweis. Es seien reelle Zahlen $x:=\langle as,M\rangle$, $y:=\langle bs,N\rangle$ und $z:=\langle cs,L\rangle$ gegeben.
Dass $\leq$ reflexiv ist, folgt sofort aus der Definition.
Für die Antisymmetrie gelte $x\leq y$ und $y\leq x$. Aus Korollar 3.1.38 folgt dann, $\forall_p\exists_n\forall_m as(m)-bs(m)\leq 2^{-p}$ und $\forall_p\exists_n\forall_m bs(m)-as(m)\leq 2^{-p}$. Zusammen liefert dies $\forall_p\exists_n\forall_m|as(m)-bs(m)|\leq 2^{-p}$, was nach Lemma 3.1.14 genau $x=y$ impliziert.
Nun gelte $x\leq y$ und damit $\forall_p\exists_n\forall_{m\geq n}as(m)\leq bs(m)+2^{-p}$ und es gelte $y\leq z$ also $\forall_p\exists_n\forall_{m\geq n}bs(m)\leq cs(m)+2^{-p}$. Wir zeigen $x\leq z$ und damit Trasitivität von $\leq$. Dazu sei p gegeben und unser Ziel ist $\exists_n\forall_{m\geq n}as(m)\leq cs(m)+2^{-p}$. Dies erhalten wir sofort, wenn wir in die beiden gegeben Aussagen $p+1$ einsetzten und so n_0 und n_1 erhalten und dann $n=\max\{n_0,n_1\}$ setzten.

Nun kommen wir zu $<$. Wir zeigen zuerst Irreflexivität, das heißt $\forall_x(x < x \to \mathbf{F})$. Die Formel $x < x$ ist äquivalent zu $\exists_{p,n}\forall_{m\geq n} as(n)+2^{p-} \leq as(n)$. Das liefert $\exists_{p,n}\forall_{m\geq n} 1 \leq 0$, was nach Definition von $\leq$ auf den natürlichen Zahlen äquivalent zu $\mathbf{F}$ ist.
Es gelte nun $x <_p y$ und $y <_q z$ und es sei P der Cauchymodulus von $y - x$ und Q der Cauchymodulus von $z - y$. Mit Lemma 3.1.25 gilt dann $\forall_{n\geq P(p+1)} 2^{-p-1} \leq bs(n) - as(n)$ und $\forall_{n\geq Q(q+1)} 2^{-q-1} \leq cs(n) - bs(n)$. Das liefert für alle $n \geq \max\{P(p+1), Q(q+1)\}$ dann $2^{-p-1} + 2^{-q-1} \leq bs(n) - as(n) + cs(n) - bs(n)$ und damit $2^{-p-1} \leq cs(n) - as(n)$. Lemma 3.1.25 impliziert $x <_{p+2} y$. □

Lemma 3.1.41. Die Relationen $\leq$ und $<$ sind beide verträglich bezüglich der Addition und bezüglich nicht-negativer bzw. positiver Multiplikation.Formal heißt dies, für reelle Zahlen x, y und z gelten folgende Aussagen:

$$x \leq y \to x + z \leq y + z \qquad x < y \to x + z < y + z$$
$$x \leq y \to 0 \leq z \to xz \leq yz \qquad x < y \to 0 < z \to xz < yz.$$

Beweis. Die Cauchyfolgen von x, y und z nennen wir wieder as, bs und cs und wir verwenden die Aussagen aus Korollar 3.1.38.
Es gelte zunächst $x \leq y$. Damit folgt $\forall_p\exists_n\forall_{m\geq n} as(m) \leq bs(m) + 2^{-p}$. Das liefert sofort $\forall_p\exists_n\forall_{m\geq n} as(m) + cs(n) \leq bs(m) + cs(n) + 2^{-p}$ und damit $x + z \leq y + z$. Ist nun $0 \leq z$, dann haben wir noch zusätzlich $\forall_p\exists_n\forall_{m\geq n} 0 \leq cs(m) + 2^{-p}$. Wir wählen gemäß Lemma 3.1.30 ein p_{y-x} und ein p_z mit $\forall_n |bs(n) - as(n)| \leq 2^{p_{y-x}}$ und $\forall_n |cs(n)| \leq 2^{p_z}$. Sei nun eine positive Zahl r gegeben. Unser Ziel ist es, $xz \leq yz$ zu zeigen. Aus $x \leq y$ erhalten wir ein n_0, sodass $\forall_{m\geq n_0} as(m) \leq bs(n) + 2^{-r-p_z-2}$ gilt, und aus $0 \leq z$ bekommen wir ein n_1, sodass $\forall_{m\geq n_1} 0 \leq cs(m) + 2^{-r-p_{y-x}-2}$ gilt. Für $m \geq \max\{n_0, n_1\}$ haben wir somit $as(m)(cs(m) + 2^{-r-p_{y-x}-2}) \leq (bs(m) + 2^{-r-p_z-2})(cs(m) + 2^{-r-p_{y-x}-2})$ und wir können folgende Abschätzungen machen:

$$\begin{aligned} &as(m)cs(m) - bs(m)cs(m) \\ &\qquad \leq (bs(m) - as(m))2^{-r-p_{y-x}-2} + cs(m)2^{-r-p_z-2} + 2^{-r-p_z-2-r-p_{y-x}-2} \\ &\qquad \leq 2^{-r-2} + 2^{-r-2} + 2^{-r-p_z-2-r-p_{y-x}-2} \leq 2^{-r} \end{aligned}$$

Das bedeutet $xz \leq yz$.
Es gelte nun $x <_p y$ also $2^{-p-1} \leq bs(n) - as(n)$ für jedes $n \geq \max\{M(p+1), N(p+1)\}$ und somit auch $2^{-p-1} \leq (bs(n) - cs(n)) - (as(n) - cs(n))$ woraus $x + z <_{p+2} y + z$ folgt. Gilt nun zusätzlich $0 <_q z$, dann folgt $0 \leq cs(n) - 2^{-q-1}$ für $n \geq L(q+1)$. Dies gibt uns für $n \geq \max\{M(p+1), N(p+1), L(p+1)\}$ dann $2^{-p-1}(cs(n) - 2^{-q-1}) \leq (bs(n) - as(n))(cs(n) - 2^{-q-1})$, was zu folgenden Abschätzungen führt:

$$\begin{aligned} as(n)cs(n) &\leq bs(n)cs(n) + 2^{-p-1-q-1} - cs(n)2^{-p-1} - (as(n) - bs(n))2^{-q-1} \\ &\leq bs(n)cs(n) + 2^{-p-q-2} - 2^{-q-1-p-1} - 2^{-p-1-q-1} = bs(n)cs(n) - 2^{-p-q-2} \end{aligned}$$

Damit haben wir $xz <_{p+q+3} yz$. □

Motivation 3.1.42. Aus Lemma 3.1.40 wissen wir bereits, dass $\leq$ eine Ordnungsrelation und $<$ eine strenge Ordnungsrelation auf den reellen Zahlen ist. Werden die reellen Zahlen und die Ordnungsrelationen im klassischen Sinne eingeführt, handelt es sich dabei sogar um totale Ordnungsrelationen. Das heißt für zwei reelle Zahlen x und y gilt immer $y \leq x$ oder $x \leq y$ bzw. $y < x$, $x = y$ oder $x < y$. Im konstruktiven Sinne ist das nicht der Fall. Insbesondere gilt nicht einmal $\forall_x(x = 0 \vee x \neq 0)$. Um dies einzusehen, nehme man für jede beliebige Turingmaschine T die reelle Zahl

$x = \langle as, M\rangle$, wobei M = `PosToNat` und $as(n) = 2^{-n}$, falls T nicht in n Schritten terminiert, und $as(n) = 2^{-k}$, falls T in $k \leq n$ Schritten terminiert und k minimal mit dieser Eigenschaft ist. Man sieht leicht, dass x eine reelle Zahl ist und $x = 0$ genau dann, wenn T nicht terminiert und $x \neq 0$ genau dann, wenn T terminiert. Aus der Aussage $\forall_x(x = 0 \vee x \neq 0)$ würde also das Halteproblem folgen, welches nicht lösbar ist. Somit können wir auch nicht für alle reellen Zahlen entscheiden, ob sie gleich 0 sind oder nicht.
Wir können aber eine reelle Zahl mit einem echten Intervall vergleichen. Diese Aussage haben wir im folgenden Vergleichbarkeitssatz:

Satz 3.1.43. Es seien zwei reelle Zahlen x und y mit $x < y$ gegeben. Ist z eine weitere reelle Zahl, dann gilt $z \leq y$ oder $x \leq z$.

Beweis. Wir definieren $x := \langle as, M\rangle$, $y := \langle bs, N\rangle$ und $z := \langle cs, L\rangle$ und es gelte $x <_p y$. Nach Definition haben wir damit $2^{-p} \leq bs(n) - as(n)$ für $n := \max\{M(p+2), N(p+2)\}$. Es sei $m := \max\{n, L(p+2)\}$. Wir unterscheiden zwei Fälle:
Gilt $cs(m) \leq \frac{as(n)+bs(n)}{2}$, zeigen wir $z \leq y$, indem wir die stärkere Aussage $cs(l) \leq bs(l)$ für alle $l \geq m$ beweisen. Sei also $l \geq m$ gegeben, dann gilt wegen der Definition von m und der Fallannahme:

$$\begin{aligned} cs(l) &\leq cs(m) + 2^{-p-2} \leq \frac{as(n) - bs(n)}{2} + \frac{bs(n) - as(n)}{4} = bs(n) - \frac{bs(n) - as(n)}{4} \\ &\leq bs(n) - 2^{-p-2} \leq bs(l) \end{aligned}$$

Für rationale Zahlen a, b gilt $a \not\leq b \rightarrow b < a$. Das wird auch in `rat.scm` bewiesen. Bei reellen Zahlen gilt dies im Allgemeinen nicht.

Gilt nun $cs(m) \not\leq \frac{as(n)+bs(n)}{2}$, folgt $cs(m) > \frac{as(n)+bs(n)}{2}$. Wir zeigen in diesem Fall $x \leq z$, indem wir analog zum ersten Fall $as(l) \leq cs(l)$ für alle $l \geq m$ beweisen. Aus analogen Gründen wie oben folgt für $l \geq m$

$$\begin{aligned} as(l) &\leq as(m) + 2^{-p-2} \leq as(n) + \frac{bs(n) - as(n)}{4} \leq \frac{as(n) + bs(n)}{2} - \frac{bs(n) - as(n)}{4} \\ &\leq cs(m) - 2^{-p-2} \leq cs(l). \end{aligned}$$

In jedem Fall folgt $x \leq z$ oder $z \leq y$. □

3.2 Darstellung reeller Zahlen und Coinduktion

Motivation 3.2.1. Ziel dieses Abschnittes ist es, einerseits die Theorie **TCF** um coinduktiv definierte Prädikate und den Corekursionsoperator zu erweitern. Da Coinduktion ein etwas kompliziertes Thema ist, wollen wir dieses andererseits gleich mit einem Beispiel unterlegen, in welchem wir Coinduktion verwenden, um reelle Zahlen als Ströme von $-1, 0, 1$ darzustellen. Besonders im nächsten Abschnitt werden wir Coinduktion nutzen, um Rechenoperationen auf den Strömen durchzuführen.

3.2.1 Endliche Binärdarstellung mit Vorzeichen

Motivation 3.2.2. Reelle Zahlen als Cauchyfolge mit Modulus anzugeben, ist für die Praxis nicht sehr hilfreich. In Anwendung ist es in der Regel nur möglich, rationale Intervalle anzugeben, in denen sich die reelle Zahl befindet. Diese Intervalle werden häufig dadurch festgelegt, indem man die ersten Stellen der Dezimal- oder Binärdarstellung einer reellen Zahl angibt. Beginnt die Binärdarstellung einer reellen Zahl beispielsweise mit 0,1 , so heißt dies, dass diese reelle Zahl im Intervall $\left[\frac{1}{2}, 1\right]$ liegt.
In dieser Arbeit werden wir reelle Zahlen nicht in der Binärdarstellung betrachten,

sondern wir erweitern die Binärdarstellung um die Ziffer $\overline{1} := -1$. Dadurch erhalten wir die Binärdarstellung mit Vorzeichen oder auf englisch „signed digit code“ für reelle Zahlen. Die Binärdarstellung mit Vorzeichen kürzen wir im Folgenden mit SD-Darstellung ab.
Wir definieren zunächst das Prädikat $\mathbf{I}$ auf den reellen Zahlen. Dabei sagt $\mathbf{I}x$ aus, dass x eine endliche SD-Darstellung hat. Dabei werden wir nur reelle Zahlen im Intervall $[-1,1]$ betrachten. Eine allgemeine reelle Zahl x lässt sich darstellen durch $x = k + x'$, wobei x' im Intervall $[-1,1]$ liegt und k eine ganze Zahl ist.

Definition 3.2.3. Es bedeute $\mathbf{Sd}\ d$, dass $d = \overline{1}$, $d = 0$ oder $d = 1$ ist. Mit diesem Prädikat definieren wir das induktiv definierte Prädikat $\mathbf{I}$ auf den reellen Zahlen durch

$$\mathbf{I} := \mu_X\left(\forall_x^{nc}(x = 0 \to^{nc} Xx), \forall_{d,x',x}^{nc}\left(\mathbf{Sd}\ d \to Xx' \to |x| \le 1 \to^{nc} x = \frac{d + x'}{2} \to^{nc} Xx\right)\right).$$

Man beachte, dass es sich bei = um die reelle Gleichheit handelt.

Bemerkung 3.2.4. Wir haben $\tau(\mathbf{I}) = \mu_\xi(\xi, \tau(\mathbf{Sd}) \to \xi \to \xi)$. Das ist der Listentyp mit Einträgen in $\tau(\mathbf{Sd})$. $\mathbf{Sd}$ setzen wir auch als induktiv definiertes Prädikat durch $\mathbf{Sd} := \mu_X(X\overline{1}, X0, X1)$ um. Dann ist $\tau(\mathbf{Sd}) = \mu_\xi(\xi, \xi, \xi)$. Bezeichnen wir nun die drei Konstruktoren ebenfalls jeweils mit $\overline{1}, 0, 1$, dann ist ein Realisierer $\mathbf{I}x$ eine endliche Liste $d_n, \dots, d_1$ mit Einträgen $\overline{1}, 0, 1$ und wir können x als $x = \sum_{i=1}^{n} \frac{d_i}{2^i}$ verstehen. In Folgendem werden wir aus diesem Grund $\mathbf{Sd} = \tau(\mathbf{Sd})$ identifizieren. In Minlog muss diese Identifikation durch Programmkonstanten gemacht werden, was die Terme etwas aufbläht.

3.2.2 Coinduktiv definierte Prädikate

Motivation 3.2.5. Ein induktiv definiertes Prädikat $I = \mu_X (K_i(X))_{i<k}$ mit den Klauseln $K_i = \forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} X\vec{s}_j\right)\right)_{j<n_i} \to X\vec{t}\right)$ kann man wegen dem Eliminationsaxiom als kleinsten Fixpunkt der Klauseln verstehen:
Nach Definition 1.6.6 hat das Eliminationsaxiom die Form

$$I^-(P) : \forall_{\vec{x}}^{nc}\left(I\vec{x} \to^{c} (K_i(I,P))_{i<k} \to^{c} P\vec{x}\right)$$

mit

$$K_i(I,P) :=$$
$$\forall_{\vec{x}}^{c/nc}\left(\vec{A} \to^{c/nc} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} I\vec{s}_j\right)\right)_{j<n} \to^{c} \left(\forall_{\vec{y}_j}^{c/nc}\left(\vec{B}_j \to^{c/nc} P\vec{s}_j\right)\right)_{j<n} \to^{c} P\vec{t}\right).$$

Wie man leicht sieht, ist $K_i(I,P)$ eine schwächere Aussage als $K_i(P)$. Erfüllt also insbesondere ein Prädikat P alle Klauseln $K_i(P)$, dann gilt $\forall_{\vec{x}}^{nc}(I\vec{x} \to P\vec{x})$ oder kurz geschrieben $I \subseteq P$. Weil aber schon ausreicht, dass P alle $K_i(I,P)$ erfüllt, kann man das Eliminationsaxiom als verschärftes Axiom vom kleinsten Fixpunkt verstehen.
Betrachtet man induktiv definierte Prädikate als kleinste Fixpunkte ihrer Klauseln, stellt sich nun unweigerlich die Frage, ob es auch Prädikate gibt, welche man als größte Fixpunkte von Klauseln verstehen kann. Dies bringt uns zu den coinduktiv definierten Prädikaten. Man sieht jedoch, dass es keinen Sinn macht, ein neues Prädikat einzuführen, welches genau der größte Fixpunkt der Klauseln ist, denn der größte Fixpunkt von den Klauseln, welche die obige Form haben, wäre das immerwahre Prädikat $\mu_X(\forall_{\vec{x}} X\vec{x})$. Wir ändern die Klauseln daher noch etwas ab.

Definition 3.2.6. Haben wir ein induktiv definiertes Prädikat $I := \mu_X^{c/nc}(K_i(X))_{i<k}$ mit

$$K_i(X) = \forall_{\vec{x}_i}^{c/nc}(\vec{A}_i \to^{c/nc} (\forall_{\vec{y}_{ij}}^{c/nc}(\vec{B}_{ij} \to^{c/nc} X\vec{s}_{ij}))_{j<n_i} \to X\vec{t}_i).$$

≡ steht hier für die Leibnizgleichheit.

Dann ist die Konjunktion aller $K_i(X)$ äquivalent zu

$$\forall_{\vec{x}}^{nc}.\bigvee_{i<k}\left(\exists_{\vec{x}_i}^{d/r}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i \wedge \left(\forall_{\vec{y}_{ij}}^{c/nc}\left(\vec{B}_{ij} \to X\vec{s}_{ij}\right)\right)_{j<n_i}\right) \to X\vec{x}.$$

Dabei seien die einzelnen Einträgen von $\vec{A}_i$ und $\left(\forall_{\vec{y}_{ij}}^{c/nc}\left(\vec{B}_{ij} \to X\vec{s}_{ij}\right)\right)_{j<n_i}$ in der Formel oben jeweils durch ein $\wedge$ verbunden. Die Dekorationen der Konjunktionen seien dabei so gewählt, dass sie den Dekorationen der Implikationspfeilen entsprechen und die Dekorationen der Existenzquantoren sollen den Dekorationen der Allquantoren entsprechen. Wir definieren die Coinduktionsklausel durch die Dualisierung der obigen Aussage:

$${}^{co}K(X) := \forall_{\vec{x}}^{nc}.X\vec{x} \to \bigvee_{i<k}\left(\exists_{\vec{x}_i}^{d/r}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i \wedge \left(\forall_{\vec{y}_{ji}}^{c/nc}\left(\vec{B}_{ij} \to X\vec{s}_{ij}\right)\right)_{j<n_i}\right)$$

Definition 3.2.7. Wir haben hier die Schreibweise $\bigvee_{i<k} C_i$ verwendet. Intuitiv würde man dies verstehen als $C_0 \vee \cdots \vee C_{k-1}$. Formal gesehen handelt es sich um eine Verallgemeinerung der Disjunktion als induktiv definiertes Prädikat. Für $k \geq 1$ ist dieses gegeben durch

$$\bigvee_{i<k} C_i := \mu_X\left(C_i \to^{c/nc} X\right)_{i<k}.$$

Die Dekorationen der Implikationspfeile sind dabei vorher festzulegen. In unserem Fall sind alle Implikationen immer rechnerisch.
Für die Konjunktion gibt es auch die Verallgemeinerung $\bigwedge_{j<n} C_i$. Für $n \geq 0$ definieren wir hier

$$\bigwedge_{j<n} C_j := \mu_X\left(\left(C_j\right)_{j<n} \to^{c/nc} X\right).$$

Auch hier sind die Dekorationen der Implikationspfeile vorher festzulegen. Wir werden die Dekorationen häufig implizit lassen.
Der Typ der allgemeinen Disjunktion ist der allgemeine Summentyp

$$\sum_{i<k} \tau_i := \mu_\xi(\tau_i \to \xi)_{i<k}.$$

Die Konstruktoren bezeichnen wir mit in_0 bis in_{k-1}. Außerdem definieren wir zu Termen $t_0, \ldots, t_{k-1}$ mit den Typen $\tau_0 \to \rho, \ldots, \tau_{k-1} \to \rho$ die Programmkonstante $[t_0, \ldots, t_{k-1}] : \sum_{i<k} \tau_i \to \rho$ durch $[t_0, \ldots, t_{k-1}]\mathsf{in}_i x := t_i x$.
Als Typ der allgemeinen Konjunktion erhalten wir den allgemeinen Produkttyp

$$\prod_{j<n} \tau_j := \mu_\xi\left(\left(\tau_j\right)_{j<n} \to \xi\right).$$

Den einzigen Konstruktor schreiben wir auch in der Tupelnotation $\langle t_0, \ldots, t_{n-1}\rangle := C_0 t_0 \ldots t_{n-1}$. Für jedes $i < n$ definieren wir die Projektion $\pi_i : \prod_{j<n} \tau_j \to \tau_i$ auf die i-te Komponente als Programmkonstante durch die Berechnungsregel $\pi_i\langle t_0, \ldots, t_{n-1}\rangle :=$

t_i.
Wir werden auch häufig das Produkt zwischen zwei Produkttypen als einen Produkttypen sehen. Das heißt, wir identifizieren beispielsweise

$$\prod_{j<n} \tau_j \times \prod_{j<m} \tau_{n+j} = \prod_{j<m+n} \tau_j$$

und damit auch $\langle\langle t_0,\dots,t_{n-1}\rangle,\langle t_n,\dots,t_{n+m-1}\rangle\rangle = \langle t_0,\dots,t_{m+n-1}\rangle$. Diese Identifikation ist für uns sehr nützlich, da wir meistens nur ein verschachteltes Produkt auf jeweils zwei Typen haben. In Minlog gibt es diese Identifizierung nicht, sodass man beispielsweise häufig eine Verschachtelung der Projektoren π_0 (in Minlog `clft`) und π_1 (in Minlog `crht`) im extrahierten Term eines Beweises findet.
Außerdem identifizieren wir $\prod_{j<1}\tau_j$ und $\sum_{i<1}\tau_i$ mit τ_0 selbst.

Beispiel 3.2.8. Nehmen wir das induktiv definierte Prädikat aus dem vorherigen Abschnitt:

$$\mathsf{I} := \mu_X\left(\forall^{nc}_x(x=0\to^{nc} Xx), \forall^{nc}_{d,x',x}\left(\mathsf{Sd}\ d\to Xx'\to|x|\le 1\to^{nc} x=\frac{d+x'}{2}\to^{nc} Xx\right)\right).$$

Die Coinduktionklausel dieses induktiv definierten Prädikats ist äquivalent zu

$$\forall^{nc}_x.Xx\to x=0\vee\exists^r_{d,x'}.\mathsf{Sd}\ d\wedge^d Xx'\wedge^l|x|\le 1\wedge^l x=\frac{d+x'}{2}$$

und weil aus $x=0$ auch die rechte Seite der obigen Disjunktion folgt, haben wir noch kürzer

$$\forall^{nc}_x.Xx\to\exists^r_{d,x'}.\mathsf{Sd}\ d\wedge^d Xx'\wedge^l|x|\le 1\wedge^l x=\frac{d+x'}{2}.$$

Definition 3.2.9. Zu jedem induktiv definierten Prädikat I aus Definition 1.4.1 fügen wir der Theorie **TCF** das coinduktiv definierte Prädikat ${}^{co}I$ als Prädikatensymbol hinzu. Dabei muss das zugrunde liegende induktive definierte Prädikat I selbst nicht unbedingt ein Prädikatensymbol in der Theorie **TCF** gemäß Definition 1.4.5 sein.
Ist I wie in Definition 3.2.6 gegeben, so ist das Eliminationsaxiom von ${}^{co}I$ gegeben durch die Coinduktionsklausel: ${}^{co}I^- := {}^{co}K({}^{co}I)$.
Das Einführungsaxiom von ${}^{co}I$ kann man in Anlehnung an Motivation 3.2.5 als das verschärfte Axiom vom größten Fixpunkt verstehen. Diese ist gegeben durch

$$\begin{aligned}{}^{co}I^+(X) :\forall^{nc}_{\vec x}.X\vec x\to\\ &\forall^{nc}_{\vec x}\left(X\vec x\to\bigvee_{i<k}\exists_{\vec x_i}.\vec t_i\equiv\vec x\wedge\vec A_i\wedge\left(\forall^{c/nc}_{\vec y_{ji}}\left(\vec B_{ij}\to({}^{co}I\cup X)\vec s_{ij}\right)\right)_{j<n_i}\right)\\ &\to{}^{co}I\vec x.\end{aligned}$$

Dabei setzen wir $P\cup Q := \{\vec x|P\vec x\vee^d Q\vec x\}$ für zwei gleichstellige Prädikate P und Q. Diese Axiome fügen wir auch der Theorie **TCF** hinzu.
Wir definieren außerdem den Typen des coinduktiv definierten Prädikats gleich dem Typen des dazugehörigen induktiv definierten Prädikats:

$$\tau({}^{co}I) := \tau(I)$$

Ist also insbesondere das induktiv definierte Prädikat nicht-rechnerisch, so ist auch das dazugehörige coinduktiv definierte Prädikat nicht-rechnerisch. Im Vergleich zum Eliminationsaxiom eines nicht-rechnerischen induktiv definierten Prädikats gibt es bei nicht-rechnerischen coinduktiv definierten Prädikaten keine Einschränkung an die Prädikatenvariable des Einführungsaxioms.

Bemerkung 3.2.10. Das Ex-falso-quodlibet aus Satz 1.4.7 gilt auch noch mit coinduktiv definierten Prädikaten. Der Beweis des Satzes muss nur um den Fall eines coinduktiv definierten Prädikates erweitert werden:
Dazu setzt man in das Einführungsaxiom $X = \{\vec{x}|\mathbf{F}\}$ ein. Es ist dann

$$\forall^{nc}_{\vec{x}}.\mathbf{F} \to \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i \wedge \left(\forall^{c/nc}_{\vec{y}_{ji}} \left(\vec{B}_{ij} \to {}^{co}I\vec{s}_{ij} \vee \mathbf{F}\right)\right)_{j<n_i}$$

zu beweisen. Nach Induktionsvoraussetzung ist sogar jede Formel der Disjunktion aus dem Falsum herleitbar. Hier brauchen wir also im Gegensatz zu den induktiv definierten Prädikaten keine Klausel ohne Rekursionsprämissen. Dies ist auch der Grund, warum wir ohne Einschränkung alle coinduktiv definierten Prädikate in **TCF** aufgenommen haben.

Lemma 3.2.11. Ist I ein induktiv definiertes Prädikat in **TCF** und ${}^{co}I$ das zugehörige coinduktiv definierte Prädikat, dann gilt $I \subseteq {}^{co}I$. Das heißt $\forall_{\vec{x}}(I\vec{x} \to {}^{co}I\vec{x})$.

Beweis. Sei I wie in Definition 3.2.6 und 3.2.9. Setzen wir in das Einführungsaxiom von ${}^{co}I$ das Prädikat I ein, dann reicht es zu zeigen, dass

$$\forall^{nc}_{\vec{x}}.I\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i \wedge \left(\forall^{c/nc}_{\vec{y}_{ji}} \left(\vec{B}_{ij} \to ({}^{co}I \cup I)\vec{s}_{ij}\right)\right)_{j<n_i}$$

gilt. Verwenden wir das Eliminationsaxiom von I auf das Prädikat

$$P\vec{x} := \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i \wedge \left(\forall^{c/nc}_{\vec{y}_{ji}} \left(\vec{B}_{ij} \to (I \cup {}^{co}I)\vec{s}_{ij}\right)\right)_{j<n_i},$$

gibt und das

$$\begin{aligned}&\forall^{nc}_{\vec{x}}.I\vec{x} \to \\ &\quad \left(\forall^{c/nc}_{\vec{x}_i}.\vec{A}_i \to^{c/nc} \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to^{c/nc} I\vec{s}_{ij}\right)_{j<n_i} \to \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to^{c/nc} P\vec{s}_{ij}\right)_{j<n_i} \to P\vec{t}_i\right)_{i<k} \\ &\quad \to P\vec{x}.\end{aligned}$$

Es reicht also, für jedes $i < k$ die Prämisse

$$\forall^{c/nc}_{\vec{x}_i}.\vec{A}_i \to^{c/nc} \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to^{c/nc} I\vec{s}_{ij}\right)_{j<n_i} \to \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to^{c/nc} P\vec{s}_{ij}\right)_{j<n_i} \to P\vec{t}_i$$

zu zeigen. Diese Aussage folgt direkt aus

$$\begin{aligned}\forall^{c/nc}_{\vec{x}_i}.\vec{A}_i \to^{c/nc} &\left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to^{c/nc} I\vec{s}_{ij}\right)_{j<n_i} \to \\ &\vec{t}_i \equiv \vec{t}_i \wedge \vec{A}_i \wedge \left(\forall^{c/nc}_{\vec{y}_{ji}} \left(\vec{B}_{ij} \to (I \cup {}^{co}I)\vec{s}_{ij}\right)\right)_{j<n_i}\end{aligned}$$

und dies gilt, weil $\vec{t}_i \equiv \vec{t}_i$ vom Einführungsaxiom der Leibnizgleichheit kommt und die anderen Aussagen der Konjunktion genau aus der entsprechenden Prämisse folgen. □

3.2.3 Binärdarstellung mit Vorzeichen und Coinduktion in Minlog

Motivation 3.2.12. Wir führen nun Beispiel 3.2.8 fort. Dort haben wir unter anderem auch gesehen, dass die erste Klausel von **I** für das coinduktiv definierte Prädikat

irrelevant ist. Wir definieren daher ${}^{co}\mathbf{I}$ als das coinduktiv definierte Prädikat zu dem induktiv definierten Prädikat

$$\mu_X\left(\forall^{nc}_{d,x',x}\left(\mathbf{Sd}\ d \to Xx' \to |x| \le 1 \to^{nc} x = \frac{d+x'}{2} \to^{nc} Xx\right)\right)$$

Das Eliminationsaxiom von ${}^{co}\mathbf{I}$ ist dann formal gegeben durch

$$\forall^{nc}_x.{}^{co}\mathbf{I}x \to \exists^r_{d,x',y}.y \equiv x \wedge \mathbf{Sd}\ d \wedge^d {}^{co}\mathbf{I}x' \wedge^l |y| \le 1 \wedge^l y = \frac{d+x'}{2}.$$

Der Einfachheit halber werden wir im Folgenden immer die äquivalente Formulierung

$$\forall^{nc}_x.{}^{co}\mathbf{I}x \to \exists^r_{d,x'}.\mathbf{Sd}\ d \wedge^d {}^{co}\mathbf{I}x' \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}$$

verwenden. Man kann durch Induktion leicht zeigen, dass eine reelle Zahl in $\mathbf{I}$ ein Vielfaches von 2^{-p} für eine positive Zahl p ist. Deswegen ist $\mathbf{I}$ ungeeignet um reelle Zahlen exakt darzustellen. Bei ${}^{co}\mathbf{I}$ sieht es besser aus. Interpretieren wir ${}^{co}\mathbf{I}x$ als „x hat eine SD-Darstellung“, so haben wir folgende Interpretation des Eliminationsaxioms: Hat x eine SD-Darstellung, so gibt es $d \in \{\overline{1}, 0, 1\}$ und ein x' mit SD-Darstellung und $x = \frac{d+x'}{2}$. Verwendet man das Einführungsaxiom iterativ, so erhält man immer mehr Anfangsziffern der SD-Darstellung von x. Dieser Vorgang wird wegen der obigen Formulierung des Eliminationsaxioms bei keinem x in ${}^{co}\mathbf{I}$ abbrechen, sodass man eine unendliche SD-Darstellung von x erwarten kann.
Der folgende Satz bestätigt, dass wir mit Hilfe von ${}^{co}\mathbf{I}$ reelle Zahlen exakt darstellen können. Für den Beweis des Satzes werden wir folgende äquivalente Form des Einführungsaxioms von ${}^{co}\mathbf{I}$ verwenden:

$$\forall^{nc}_x.Xx \to \forall^{nc}_x\left(Xx \to \exists^r_{d,x'}\left(\mathbf{Sd}\ d \wedge^d ({}^{co}\mathbf{I} \cup X)x' \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}\right)\right) \to {}^{co}\mathbf{I}x.$$

Satz 3.2.13. Jede reelle Zahl zwischen -1 und 1 liegt in ${}^{co}\mathbf{I}$.

Beweis. Wir wenden das Einführungsaxiom von ${}^{co}\mathbf{I}$ auf das Prädikat

$$\mathbf{P}x := \exists^l_y(x = y \wedge^{nc} -1 \le y \le 1)$$

an. Es ist dann die mittlere Prämisse zu zeigen:

$$\forall^{nc}_x.\exists^l_y(x = y \wedge^{nc} -1 \le y \le 1) \to \exists^r_{d,x'}\left(\mathbf{Sd}\ d \wedge^d ({}^{co}\mathbf{I}x' \vee \mathbf{P}x') \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}\right)$$

Sei also x und y gegeben und gelte $x = y$ und $-1 \le y \le 1$. $|x| \le 1$ gilt dann wegen $-1 \le x \le 1$ unabhängig von der Wahl von d und x' immer und werden wir daher nicht weiter beachten. Wir wenden den Vergleichbarkeitssatz 3.1.43 auf $-\frac{1}{2} < 0$ und y an. Das liefert zwei Fälle:
Ist $y \le 0$, so haben wir $-1 \le 2x+1 \le 1$ und setzten daher $d = -1$ und $x' = 2x+1$. Es gilt dann $\mathbf{Sd}\ d$, $|x'| \le 1$ und $x = \frac{d+x'}{2}$.
Ist $-\frac{1}{2} \le y$, verwenden wir noch einmal den Vergleichbarkeitssatz, dieses Mal auf $0 < \frac{1}{2}$ und y.
Gilt $0 \le y$, folgt $-1 \le 2x-1 \le 1$. Analog zum ersten Fall setzten wir $d = 1$ und $x' = 2x-1$, womit alle gewünschten Aussagen erfüllt sind.
Im Fall $y \le \frac{1}{2}$ haben wir insgesamt $-\frac{1}{2} \le x \le \frac{1}{2}$ und somit $-1 \le 2x \le 1$. Wir setzten daher $d = 0$ und $x' = 2x$. Dann ist $\mathbf{Sd}\ d$, $|x'| \le 1$ und $x = \frac{d+x'}{2}$. □

Bemerkung 3.2.14. In dem obigen Beweis ist wichtig, dass wir $\mathbf{P}x := \exists^l_y(x = y \wedge^{nc} -1 \le y \le 1)$ und nicht $\mathbf{P}x := -1 \le x \le 1$ definiert haben. Der Unterschied besteht darin, dass im Einführungsaxiom die Variable x nicht-recherisch gebunden ist. Daher dürfen wir dieses x nicht auf den Vergleichbarkeitssatz anwenden. Fordern wir aber die Existenz einer rechnerischen Variable y, welche gleich x ist, dürfen wir y und damit indirekt auch x rechnerisch verwenden.

Beispiel 3.2.15. Anhand der Definition von $^{co}\mathbf{I}$ möchten wir erklären, wie sich coinduktiv definierte Prädikate in Minlog einbauen lassen. In der Datei `sdmult.scm` wird zunächst das zugrunde liegende induktiv definierte Prädikat `I` eingeführt, wobei dessen Algebra vorher noch extra mit `(add-algs "str" '("C" "sd=>str=>str"))` definiert wird.

`str` steht als Abkürzung für „Stream“. Die Terme der Algebra sollen als Ströme von -1, 0 und 1 verstanden werden.

```
(add-ids
 (list (list "I" (make-arity (py "rea")) "str"))
 '("allnc d,x,y(Sd d -> Real x -> abs x<<=1 -> I x ->
                y===(1#2)*(x+d) -> I y)"
   "GenI"))
```

Um nun zu einem induktiv definierten Prädikat in Minlog das zugehörige coinduktiv definierte Prädikat anzulegen, gibt es den Befehl

```
(add-co NAME LISTE).
```

Für `NAME` wird dabei der Name des induktiv definierten Prädikats eingesetzt. Anstelle von `LISTE` kann man für jedes Argument des induktiv definierten Prädikats ein Gleichheitsprädikat angeben. In Definitionen 3.2.6 und 3.2.9 haben wir immer die Leibnizgleichheit verwendet. Gibt man andere Gleichheitsprädikate wie beispielsweise die reelle Gleichheit für `LISTE` an, so werden diese anstelle von der Leibnizgleichheit eingesetzt. In `sdmult.scm` wird $^{co}\mathbf{I}$ mittels

```
(add-co "I" (list "RealEq"))
```

eingeführt. Durch dieses Kommando wird das induktiv definierte Prädikate `CoI` angelegt und dazu auch sein Einführungsaxiom als Theorem unter dem Namen `CoIClause` abgespeichert. Dieses steht nun für jeden weiteren Beweis zur Verfügung. Mit `pp` betrachten wir dieses Theorem:

```
allnc x(
 CoI x ->
 exr d,x0,y(
  Sd d andd Real x0 andr
  abs x0<<=1 andr CoI x0 andl y===(1#2)*(x0+d) andnc x===y))
```

Es fällt auf, dass am Ende der Term `x===y` steht. Hätten wir nur `(add-co "I")` eingegeben, stünde dort die Leibnizgleichheit `x eqd y`. Man kann sich auch leicht überlegen, dass beide Varianten äquivalent sind.

Beispiel 3.2.16. Als eine Anwendung der Coinduktion in Minlog, ist der obige Satz in der Datei `RealCoI.scm` in Minlog implementiert. Wir gehen hier nur kurz auf den Anfang des Beweises ein, um zu zeigen, wie sich in Minlog Beweise mit coinduktiv definierten Prädikaten durchführen lassen:
Das Ziel wird festgelegt mit

```
(set-goal "allnc x( exl y(x===y andnc IntN 1 <<= y andnc y<<=1)
                    -> CoI x)")
```

und nach `(assume "x10")` haben wir die Ausgabe

```
 ok, we now have the new goal

  {x10}
----------------------------------------------------------------------
?_2:exl y(x10===y andnc IntN 1<<=y andnc y<<=1) -> CoI x10
```

Hier können wir nun das Einführungsaxiom von `CoI` verwenden. Ähnlich wie bei der Induktion geht dies mit dem Befehl

```
(coind).
```

Minlog erkennt, dass wir das Einführungsaxiom von `CoI` auf das Prädikat in der Prämisse anwenden wollen. Vergleichen wir das mit dem Einführungsaxiom aus Definition 3.2.9, fordert Minlog nun einen Beweis der zweiten Prämisse des Einführungsaxioms. Die Ausgabe ist daher

```
ok, ?_2 can be obtained from

  {x10}  1:exl y(x10===y andnc IntN 1<<=y andnc y<<=1)
----------------------------------------------------------------------
?_3:allnc x(
     exl y(x===y andnc IntN 1<<=y andnc y<<=1) ->
     exr d,x0,y(
      Sd d andd
      Real x0 andr
      abs x0<<=1 andr
      (CoI x0 ord exl y0(x0===y0 andnc IntN 1<<=y0 andnc y0<<=1))
       andl y===(1#2)*(x0+d) andnc x===y))
```

Der Grund, warum wir den Namen `x10` für die Variable gewählt haben, ist, dass wir diese im folgenden Beweis nicht mehr verwenden werden. Die zu zeigende Prämisse des Einführungsaxioms hat `x10` nicht als freie Variable.
Möchte man Minlog explizit mitteilen, auf welches Prädikat das Einführungsaxiom angewendet werden soll, kann man den Befehl

```
(coind NAME)
```

nutzen. Für `NAME` wird der Name einer Formel im Kontext eingesetzt. Um diesen Befehl anzuwenden, darf die Zielformel nur die Form ${}^{co}I\vec{x}$ haben. Geben wir beispielsweise nach dem Setzen der Zielformel den Befehl

```
(assume "x10" "|x|<=1")
```

ein, haben wir die Aussage

```
ok, we now have the new goal

  {x10}  |x|<=1:exl y(x10===y andnc IntN 1<<=y andnc y<<=1)
----------------------------------------------------------------------
?_2:CoI x10
```

und wir können mit

```
(coind "|x|<=1")
```

wie nach der vorherigen Eingabe von `(coind)` fortfahren.

Korollar 3.2.17. Nach Lemma 3.1.39 gibt es zu jeder reellen Zahl eine rationale Zahl, sodass $a-\frac{1}{2} \le x \le a+\frac{1}{2}$ gilt. Weil a eine rationale Zahl ist, gibt es eine ganze Zahl k mit $k-\frac{1}{2} \le a \le k+\frac{1}{2}$. Zusammen haben wir dann $k-1 \le a-\frac{1}{2} \le x \le a+\frac{1}{2} \le k+1$. Nach dem vorherigen Satz 3.2.13 gilt dann ${}^{co}\mathbf{I}(x-k)$. Wir können damit jede reelle Zahl x darstellen als $x = k+y$, wobei k eine ganze Zahl ist und ${}^{co}\mathbf{I}\,y$ gilt.

3.2.4 Programmextraktion für coinduktiv definierte Prädikate

Motivation 3.2.18. Wir haben **TCF** um coinduktiv definierte Prädikate erweitert. Im ersten Kapitel wurde **TCF** insbesondere aus dem Grund eingeführt, damit wir aus einem Beweis einen Term extrahieren können. In Satz 1.9.2 haben wir auch bewiesen, dass der extrahierte Term tatsächlich ein Realisierer der bewiesenen Aussage ist. Die Aussagen und Definitionen möchten wir nun auch auf **TCF** mit coinduktiv definierten Prädikaten erweitern.

Definition 3.2.19. Wir erweitern die Realisierbarkeitsrelation für rechnerische coinduktiv definierte Prädikate ${}^{co}I$ durch die Regel

$$t \;\mathbf{r}\; {}^{co}I\vec{s} := {}^{co}(I^{\mathbf{r}})\,t\vec{s}.$$

Dabei ist $I^{\mathbf{r}}$ das Zeugenprädikat von I gemäß Definition 1.8.6. Wir schreiben auch einfach ${}^{co}I^{\mathbf{r}}$ für ${}^{co}(I^{\mathbf{r}})$.

Beispiel 3.2.20. Für ein rechnerisches induktiv definiertes Prädikat $I = \mu_X(\forall_{\vec{x}_i}.\vec{A}_i \to (\forall_{\vec{y}_i}.\vec{B}_{ij} \to X\vec{s}_{ij}) \to X\vec{t}_i)_{i<k}$ sind nach Definition 1.8.6 die Einführungsaxiome von $I^{\mathbf{r}}$ für jedes $i<k$ gegeben durch

$$\forall_{\vec{x}_i,\vec{u}_i,\vec{f}_i}.\vec{u}_u \;\mathbf{r}\; \vec{A}_i \to \left(\forall_{\vec{y}_{ij},\vec{v}_{ij}}.\vec{v}_{ij} \;\mathbf{r}\; \vec{B}_{ij} \to I^{\mathbf{r}}(f_{ij}\vec{y}_{ij}\vec{v}_{ij})\vec{s}_{ij}\right)_{j<n_i} \to I^{\mathbf{r}}(C_i\vec{x}_i\vec{u}_i\vec{f}_i)\vec{t}_i.$$

Mit der Definition der Realisierbarkeit formulieren wir die Parameterprämissen äquivalent um und haben dann folgende Form des Einführungsaxioms:

$$\forall_{\vec{x}_i,\vec{u}_i,\vec{f}_i}.\vec{u}_u \;\mathbf{r}\; \vec{A}_i \to \vec{f}_i \;\mathbf{r}\; \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to I\vec{s}_{ij}\right)_{j<n_i} \to I^{\mathbf{r}}(C_i\vec{x}_i\vec{u}_i\vec{f}_i)\vec{t}_i$$

Kürzen wir $\vec{A}_i(X) := \vec{A}_i, \left(\forall_{\vec{y}_{ij}}.\vec{B}_{ij} \to X\vec{s}_{ij}\right)_{j<n_i}$ ab, so haben wir als Einführungsaxiom

$$\forall_{\vec{x}_i,\vec{u}_i}.\vec{u}_i \;\mathbf{r}\; \vec{A}_i(I) \to I^{\mathbf{r}}(C_i\vec{x}_i\vec{u}_i)\vec{t}_i$$

für jedes $i<k$. Aus dieser Form können wir leicht das Eliminationsaxiom von ${}^{co}I^{\mathbf{r}}$ bestimmen, indem wir die Konjunktion aller Einführungsaxiome dualisieren:

$$\forall^{nc}_{\vec{x},z}.{}^{co}I^{\mathbf{r}}z\vec{x} \to \bigvee_{i<k}\left(\exists_{\vec{x}_i,\vec{u}_i}.C_i\vec{x}_i\vec{u}_i \equiv z \wedge \vec{t}_i \equiv \vec{x} \wedge \vec{u}_i \;\mathbf{r}\; \vec{A}_i\left({}^{co}I\right)\right).$$

Definition 3.2.21. Für eine Algebra $\iota := \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$ mit $\kappa_i := (\rho_{ij}(\xi))_{j<n_i} \to \xi$ definieren wir den Destruktor $\mathscr{D}$ als Programmkonstante mit dem Typ

$$\mathscr{D} : \iota \to \sum_{i<k} \prod_{j<n_i} \rho_{ij}(\iota).$$

Zu welcher Algebra $\mathscr{D}$ gehört, werden wir nicht immer extra als Index angeben. Dies ist implizit an dem auf $\mathscr{D}$ folgenden Term ersichtlich.

Die Berechnungsregeln sind gegeben durch

$$\mathscr{D} C_i(x_j)_{j<n_i} := \mathsf{in}_i\langle x_0, \ldots, x_{n_i-1}\rangle$$

für jedes $i < k$.

Bemerkung 3.2.22. Der Destruktor zu einer Algebra `ALG` wird in Minlog mit `(Destr ALG)` bezeichnet. Bei der Einführung des coinduktiven Prädikats mittels `add-co` wird gleichzeitig der Destruktor als extrahierter Term für das Eliminationsaxiom angelegt. Wir werden in dem folgenden Lemma sehen, dass dies auch gerechtfertigt ist:

Lemma 3.2.23. Sei I ein rechnerisches induktiv definiertes Prädikat und ${}^{co}I$ das entsprechende coinduktiv definiertes Prädikat, dann ist der Destruktor $\mathscr{D}$ von $\tau(I)$ ein Realisierer des Eliminationsaxioms ${}^{co}I^-$. In einer Formel ausgedrückt heißt das

$$\mathscr{D} \;\mathbf{r}\; {}^{co}I^-.$$

Beweis. Wir nehmen $I := \mu_X(K_i(X))_{i<k}$ und beschränken uns darauf, dass jede Variable und jede Prämisse rechnerisch eingeht. Der allgemeine Fall ist nur mehr Schreibarbeit. Es sei also $K_i(X) = \forall_{\vec{x}}(\vec{A}_i(X) \to X\vec{t}_i)$. Wir wollen nun

$$\mathscr{D} \;\mathbf{r}\; \forall^{nc}_{\vec{x}}.{}^{co}I\vec{x} \to \bigvee_{i<k} \left(\exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\right)$$

zeigen. Das ist äquivalent zu

$$\forall^{nc}_{\vec{x},z}.z \;\mathbf{r}\; {}^{co}I\vec{x} \to \mathscr{D}z \;\mathbf{r}\; \bigvee_{i<k} \left(\exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\right).$$

Diese Aussage beweisen wir nun. Dazu sei $\vec{x}$ und z gegeben und es gelte $z \;\mathbf{r}\; {}^{co}I\vec{x}$. Nach Beispiel 3.2.20 ist das Einführungsaxiom von ${}^{co}I^{\mathbf{r}}$ äquivalent zu

$$\forall^{nc}_{\vec{x},z}.{}^{co}I^{\mathbf{r}}z\vec{x} \to \bigvee_{i<k} \left(\exists_{\vec{x}_i}.C_i\vec{x}_i\vec{u}_i \equiv z \wedge \vec{t}_i \equiv \vec{x} \wedge \vec{u}_i \;\mathbf{r}\; \vec{A}_i\left({}^{co}I\right)\right).$$

Für die gegebenen $\vec{x}$ und z erhalten wir daraus wegen $z \;\mathbf{r}\; {}^{co}I\vec{x}$ Terme $\vec{x}_i$ und die Aussage

$$C_i\vec{x}_i\vec{u}_i \equiv z \wedge \vec{t}_i \equiv \vec{x} \wedge \vec{u}_i \;\mathbf{r}\; \vec{A}_i\left({}^{co}I\right)$$

für ein $i < k$. Damit folgt also $\mathscr{D}z \equiv \mathscr{D}(C_i\vec{x}_i\vec{u}_i) \equiv \mathsf{in}_i\langle\vec{x}_i, \vec{u}_i\rangle$. Wir haben daher, dass folgende Zeilen äquivalent sind:

$$\begin{aligned}
\mathscr{D}z \;\mathbf{r}\; & \bigvee_{i<k} \left(\exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\right)\\
\mathsf{in}_i\langle\vec{x}_i, \vec{u}_i\rangle \;\mathbf{r}\; & \bigvee_{i<k} \left(\exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\right)\\
\langle\vec{x}_i, \vec{u}_i\rangle \;\mathbf{r}\; & \left(\exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\right)\\
\langle\vec{u}_i\rangle \;\mathbf{r}\; & \vec{t}_i \equiv \vec{x} \wedge \vec{A}_i\left({}^{co}I\right)\\
& \vec{t}_i \equiv \vec{x} \wedge \vec{u}_i \;\mathbf{r}\; \vec{A}_i\left({}^{co}I\right)
\end{aligned}$$

Die letzte Zeile ist genau der zweite Teil der obigen Konjunktion und damit gegeben. Das beendet den Beweis. □

Definition 3.2.24. Zu einer Algebra $\iota := \mu_\xi\left((\rho_{ij})_{j<m_i} \to (\vec{\sigma}_{ij} \to \xi)_{j<n_i} \to \xi\right)_{i<k}$ sei der Corekursionsoperator von einem Typ τ nach ι die Programmkonstante

$$^{co}\mathscr{R}^\tau_\iota : \tau \to \left(\tau \to \sum_{i<k} \prod_{j<m_i} \rho_{ij} \times \prod_{j<n_i} (\vec{\sigma}_{ij} \to \iota + \tau)\right) \to \iota$$

gegeben durch die Berechnungsregeln

$$^{co}\mathscr{R}^\tau_\iota NM := [L_0, \dots, L_{k-1}]MN$$

Zur Vereinfachung schreiben wir für eine Variable x vom Produkttyp $\tau_1 \times \cdots \times \tau_n$ den Term $\lambda_{x_1,\dots,x_n} A(x_1, \dots, x_n)$ anstelle von $\lambda_x A(\pi_1 x, \dots, \pi_n x)$.

Wobei wir L_i definieren als

$$L_i := \lambda_{\vec{x}_i} \lambda_{(f_{ij})_{j<n_i}} . C_i \vec{x}_i \left(\lambda_{\vec{z}_{ij}} \left([\mathrm{id}, \lambda_y \left(^{co}\mathscr{R}^\tau_\iota y M\right)] \left(f_{ij} \vec{z}_{ij}\right)\right)\right)_{j<n_i}.$$

In der Liste $\vec{z}_{ij}$ sollen genau so viele Variablen sein, dass $f_{ij}\vec{z}_{ij}$ den Typ $\iota + \tau$ hat. Mit id wird die Identität $\lambda_x x$ auf ι bezeichnet.

Bemerkung 3.2.25. In Minlog wird der Corekursionsoperator $^{co}\mathscr{R}^{\mathtt{TYP}}_{\mathtt{ALG}}$ von `TYP` nach `ALG` mit `(CoRec TYP=>ALG)` bezeichnet. Weil die Berechnungsregeln des Corekursionsoperators im Allgemeinen nicht konvergieren, werden diese bei Normalisierung mit zum Beispiel `nt` nicht beachtet. Wie man die Berechungsregel des Corekursionsoperators anwenden kann, sehen wir im nächsten Beispiel.

Beispiel 3.2.26. Der Corekursionsoperator eignet sich, um unendliche Objekte zu definieren. Als Beispiel definieren wir mit Hilfe des Corekursionsoperators zu jeder natürlichen Zahl n die unendlich aufsteigende Liste $[n, n+1, n+2, \dots]$. Weil wir also eine Funktion von $\mathbb{N}$ nach $\mathbb{L}(\mathbb{N})$ definieren, brauchen wir den Corekursionsoperator $\mathscr{R}^{\mathbb{N}}_{\mathbb{L}(\mathbb{N})}$. Dieser hat den Typ

$$\mathscr{R}^{\mathbb{N}}_{\mathbb{L}(\mathbb{N})} : \mathbb{N} \to (\mathbb{N} \to \mathbb{U} + \mathbb{N} \times (\mathbb{L}(\mathbb{N}) + \mathbb{N})) \to \mathbb{L}(\mathbb{N}).$$

Die Idee ist es nun, corekursiv eine Funktion f durch $f(n) := n :: f(n+1)$ zu definieren. Das machen wir formal so:

$$f := \lambda_n \left(\mathscr{R}^{\mathbb{N}}_{\mathbb{L}(\mathbb{N})} n \lambda_m (\mathrm{in}_1 \langle m, \mathrm{in}_1 Sm \rangle)\right)$$

Der Leser kann sich davon überzeugen, dass tatsächlich $fn \doteq n :: (fSn)$ gilt.
Um das Ganze in Minlog anzuwenden, geben wir

```
(define f (pt"[n](CoRec nat=>list nat)n
              ([m]Inr(m pair(InR nat (list nat))(Succ m)))"))
```

ein und benutzen den Befehl

```
(undelay-delayed-corec TERM N).
```

Damit wird bei jedem Corekursionsoperator im Term `TERM` seine Berechnungsregel `N`-mal angewendet. Als Beispiel geben wir

```
(pp (nt (undelay-delayed-corec f 3)))
```

ein und erhalten als Ausgabe

```
[n0]
 n0::
 Succ n0::
 Succ(Succ n0)::
 (CoRec nat=>list nat)(Succ(Succ(Succ n0)))
 ([n1]Inr(n1 pair(InR nat (list nat))(Succ n1)))
```

Hier war es notwendig, dass wir den Term `(undelay- delayed-corec f 3)` noch normalisiert haben, da die Konversionsegeln des Corekursionsoperators den Term sehr aufblähen.

Lemma 3.2.27. Sei I ein rechnerisches induktiv definiertes Prädikat mit dem dazugehörigen coinduktiv definierten Prädikat ${}^{co}I$. Dann ist für ein Prädikat P der Corekursionsoperator ${}^{co}\mathscr{R}^{\tau(P)}_{\tau(I)}$ ein Realisierer des Einführungsaxioms ${}^{co}I^{+}(P)$. Das heißt

$$ {}^{co}\mathscr{R}^{\tau(P)}_{\tau(I)} \;\mathbf{r}\; {}^{co}I^{+}(P). $$

Beweis. Wir definieren ${}^{co}\mathscr{R} := {}^{co}\mathscr{R}^{\tau(P)}_{\tau(I)}$ und zur Vereinfachung nehmen wir wieder an, dass alle Allquantoren und alle Implikationspfeile rechnerisch sind. Es sei daher

$$ I := \mu_X \left(\forall_{\vec{x}_i}.\vec{A}_i \to (\forall_{\vec{y}_i}.\vec{B}_{ij} \to X\vec{s}_{ij})_{j<n_i} \to X\vec{t}_i\right)_{i<k} $$

und wir kürzen zunächst $\vec{A}_i(X) := \vec{A}_i, (\forall_{\vec{y}_i}.\vec{B}_{ij} \to X\vec{s}_{ij})_{j<n_i}$ ab.
Die zu beweisende Aussage ist damit

$$ {}^{co}\mathscr{R} \;\mathbf{r}\; \forall^{nc}_{\vec{x}}.P\vec{x} \to \forall^{nc}_{\vec{x}}\left(P\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i({}^{co}I \cup P)\right) \to {}^{co}I\vec{x}, $$

was äquivalent ist zu

$$ \forall^{nc}_{\vec{x},y,z}.y \;\mathbf{r}\; P\vec{x} \to z \;\mathbf{r}\; \forall^{nc}_{\vec{x}}\left(P\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i({}^{co}I \cup P)\right) \to {}^{co}\mathscr{R}yz \;\mathbf{r}\; {}^{co}I\vec{x}. $$

Seien nun $\vec{x}$, y und z gegeben und es gelte $y \;\mathbf{r}\; P\vec{x}$ sowie die zweite Prämisse, welche wir noch äquivalent umformen:

$$ \begin{aligned} & z \;\mathbf{r}\; \forall^{nc}_{\vec{x}}\left(P\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i({}^{co}I \cup P)\right) \\ & \forall^{nc}_{\vec{x},y}.y \;\mathbf{r}\; P\vec{x} \to zy \;\mathbf{r}\; \bigvee_{i<k} \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i({}^{co}I \cup P) \\ & \forall^{nc}_{\vec{x},y}.y \;\mathbf{r}\; P\vec{x} \to \bigvee_{i<k} \exists_{w}.zy \equiv \mathsf{in}_i\, w \wedge w \;\mathbf{r}\; \exists_{\vec{x}_i}.\vec{t}_i \equiv \vec{x} \wedge \vec{A}_i({}^{co}I \cup P) \end{aligned} $$

Dass diese Aussagen äquivalent sind, müsste mal formal gesehen durch das Eliminationsaxiom des Zeugenprädikats der Disjunktion bzw. des Existenzquantors zeigen. Da der Beweis schon lang genug ist, verzichten wir auf eine genaue Ausführung. Der Leser kann sich dies als Übungsaufgabe selbst überlegen.

Schreiben wir nun wieder die Abkürzung $\vec{A}_i(P \cup {}^{co}I)$ aus, so können wir dies weiter äquivalent umformen zu:

$$ \begin{aligned} \forall^{nc}_{\vec{x},y}.y \;\mathbf{r}\; P\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i,\vec{u}_i,\vec{f}_j}.zy \equiv \mathsf{in}_i\langle\vec{x}_i,\vec{u}_i,\vec{f}_i\rangle \wedge \vec{t}_i \equiv \vec{x} \wedge \\ \vec{u}_i \;\mathbf{r}\; \vec{A}_i \wedge \forall^{nc}_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij} \;\mathbf{r}\; \vec{B}_{ij} \to f_{ij}\vec{y}_{ij}\vec{z}_{ij} \;\mathbf{r}\; (P\vec{s}_{ij} \vee {}^{co}I\vec{s}_{ij}) \end{aligned} \tag{3.1} $$

Zu zeigen ist nun ${}^{co}\mathscr{R}yz$ **r** ${}^{co}I\vec{x}$ also ${}^{co}I^{\mathbf{r}}({}^{co}\mathscr{R}yz)\vec{x}$. Dafür verwenden wir das Einführungsaxiom von ${}^{co}I^{\mathbf{r}}$. Dieses sieht mit Parameter Q wie folgt aus:

$$\begin{aligned}&\forall_{\vec{x},w}.Qw\vec{x}\to\\&\quad\forall_{\vec{x},w}\Big(Qw\vec{x}\to\bigvee_{i<k}\exists_{\vec{x}_i,\vec{u}_i,\vec{f}_i}.C_i\vec{x}_i\vec{u}_i\vec{f}_i\equiv w\wedge\vec{t}_i\equiv\vec{x}\wedge\\&\qquad\vec{u}_i\ \mathbf{r}\ \vec{A}_i\wedge\Big(\forall_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij}\ \mathbf{r}\ \vec{B}_{ij}\to{}^{co}I^{\mathbf{r}}(f_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\vee Q(f_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\Big)_{j<n_i}\Big)\\&\to{}^{co}I^{\mathbf{r}}w\vec{x}\end{aligned}$$

Wir setzen $Qw\vec{x}:=\exists_u({}^{co}\mathscr{R}uz\equiv w\wedge u\ \mathbf{r}\ P\vec{x})$ in das Einführungsaxiom ein. Spezialisieren wir dieses Einführungsaxiom nun auf unsere gegebenen Terme $\vec{x}$ und ${}^{co}\mathscr{R}yz$, dann erhalten wir die erste Prämisse $\exists_u({}^{co}\mathscr{R}uz\equiv{}^{co}\mathscr{R}yz\wedge u\ \mathbf{r}\ P\vec{x})$ direkt aus der bereits gegeben Aussage y **r** $P\vec{x}$. Für unser Ziel ${}^{co}I^{\mathbf{r}}(\mathscr{R}yz)\vec{x}$ müssen wir noch die zweite Prämisse

$$\begin{aligned}&\forall_{\vec{x},w}.\exists_u\big({}^{co}\mathscr{R}uz\equiv w\wedge u\ \mathbf{r}\ P\vec{x}\big)\to\bigvee_{i<k}\exists_{\vec{x}_i,\vec{u}_i,\vec{f}_i}.C_i\vec{x}_i\vec{u}_i\vec{f}_i\equiv w\wedge\vec{t}_i\equiv\vec{x}\wedge\vec{u}_i\ \mathbf{r}\ \vec{A}_i\wedge\\&\Big(\forall_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij}\ \mathbf{r}\ \vec{B}_{ij}\to{}^{co}I^{\mathbf{r}}(f_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\vee\exists_u\big({}^{co}\mathscr{R}uz\equiv f_{ij}\vec{y}_{ij}\vec{z}_{ij}\wedge u\ \mathbf{r}\ P\vec{s}_{ij}\big)\Big)_{j<n_i}\end{aligned}$$

zeigen. Seien dazu nun $\vec{w}$, w und u gegeben und es gelte ${}^{co}\mathscr{R}uz\equiv w$ und u **r** $p\vec{x}$. Wir setzten in 3.1 nun $\vec{w}$ und u ein und erhalten dadruch ein $i<k$ und Terme $\vec{x}_i,\vec{u}_i,\vec{f}_i$, sodass

$$\begin{aligned}&zu\equiv\mathsf{in}_i\langle\vec{x}_i,\vec{u}_i,\vec{f}_i\rangle\wedge\vec{t}_i\equiv\vec{w}\wedge\vec{u}_i\ \mathbf{r}\ \vec{A}_i\wedge\\&\forall^{nc}_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij}\ \mathbf{r}\ \vec{B}_{ij}\to f_{ij}\vec{y}_{ij}\vec{z}_{ij}\ \mathbf{r}\ ({}^{co}I\vec{s}_{ij}\vee P\vec{s}_{ij})\end{aligned}\tag{3.2}$$

gilt. Um den Beweis zu beenden, reicht es also

$$\begin{aligned}&\exists_{\vec{g}_i}.C_i\vec{x}_i\vec{u}_i\vec{g}_i\equiv w\wedge\vec{t}_i\equiv\vec{w}\wedge\vec{u}_i\ \mathbf{r}\ \vec{A}_i\wedge\\&\Big(\forall_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij}\ \mathbf{r}\ \vec{B}_{ij}\to{}^{co}I^{\mathbf{r}}(g_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\vee\exists_p({}^{co}\mathscr{R}pz\equiv g_{ij}\vec{y}_{ij}\vec{z}_{ij}\wedge p\ \mathbf{r}\ P\vec{s}_{ij})\Big)_{j<n_i}\end{aligned}$$

zu zeigen, dabei folgen die Aussagen $\vec{t}_i\equiv\vec{w}$ und $\vec{u}_i$ **r** $\vec{A}_i$ unabhänig von $\vec{g}_i$ direkt aus der oberen Formel 3.2. Wir müssen daher noch $\vec{g}_i$ angeben und

$$C_i\vec{x}_i\vec{u}_i\vec{g}_i\equiv w$$

sowie für jedes $j<n_i$ die Formel

$$\forall_{\vec{y}_{ij},\vec{z}_{ij}}.\vec{z}_{ij}\ \mathbf{r}\ \vec{B}_{ij}\to{}^{co}I^{\mathbf{r}}(g_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\vee\exists_p({}^{co}\mathscr{R}pz\equiv g_{ij}\vec{y}_{ij}\vec{z}_{ij}\wedge p\ \mathbf{r}\ P\vec{s}_{ij})$$

beweisen. Mit der Berechnungsregel und der Notation der L_i aus Definition 3.2.24 und den gegebenen Leibnizgleichheiten haben wir

$$\begin{aligned}w\equiv{}^{co}\mathscr{R}uz\equiv[L_0,\dots,L_{k-1}]zu\equiv[L_0,\dots,L_{k-1}]\mathsf{in}_i\langle\vec{x}_i,\vec{u}_i,\vec{f}_i\rangle\equiv L_i\langle\vec{x}_i,\vec{u}_i,\vec{f}_i\rangle\equiv\\C_i\vec{x}_i\vec{u}_i\Big(\lambda_{\vec{y}_{ij},\vec{z}_{ij}}\big([\mathsf{id},\lambda_v({}^{co}\mathscr{R}vz)]\,(f_{ij}\vec{y}_{ij}\vec{z}_{ij})\big)\Big)_{j<n_i}.\end{aligned}$$

Wir setzen also $\vec{g}_i:=\Big(\lambda_{\vec{y}_{ij},\vec{z}_{ij}}\big([\mathsf{id},\lambda_v({}^{co}\mathscr{R}vz)]\,(f_{ij}\vec{y}_{ij}\vec{z}_{ij})\big)\Big)_{j<n_i}$. Dann ist die gewünschte Gleichheit $C_i\vec{x}_i\vec{u}_i\vec{g}_i\equiv w$ schon bewiesen. Für die anderen Formeln seien nun $j<n_i$, die Variablen $\vec{y}_{ij}$, $\vec{z}_{ij}$ und die Aussagen $\vec{z}_{ij}$ **r** $\vec{B}_{ij}$ gegeben. Unser Ziel ist

$${}^{co}I^{\mathbf{r}}(g_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}\vee\exists_p({}^{co}\mathscr{R}pz\equiv g_{ij}\vec{y}_{ij}\vec{z}_{ij}\wedge p\ \mathbf{r}\ P\vec{s}_{ij}).$$

Aus 3.2 erhalten wir $f_{ij}\vec{y}_{ij}\vec{z}_{ij}$ **r** $({}^{co}I\vec{s}_{ij} \vee P\vec{s}_{ij})$, was äquivalent ist zu $\exists_p(\mathsf{in}_0 p \equiv f_{ij}\vec{y}_{ij}\vec{z}_{ij} \wedge p$ **r** ${}^{co}I\vec{s}_{ij}) \vee \exists_q(\mathsf{in}_1 q \equiv f_{ij}\vec{y}_{ij}\vec{z}_{ij} \wedge q$ **r** $P\vec{s}_{ij})$. Im ersten Fall sei p gegeben und es gelte $\mathsf{in}_0 p \equiv f_{ij}\vec{y}_{ij}\vec{z}_{ij}$ sowie p **r** ${}^{co}I\vec{s}_{ij}$, dann ist

$$g_{ij}\vec{y}_{ij}\vec{z}_{ij} \equiv [\mathsf{id}, \lambda_v({}^{co}\mathscr{R}vz)](f_{ij}\vec{y}_{ij}\vec{z}_{ij}) \equiv [\mathsf{id}, \lambda_v({}^{co}\mathscr{R}vz)]\mathsf{in}_0 p \equiv p$$

und wegen p **r** ${}^{co}I\vec{s}_{ij}$ ist damit ${}^{co}I^{\mathbf{r}}(g_{ij}\vec{y}_{ij}\vec{z}_{ij})\vec{s}_{ij}$ gezeigt. Im zweiten Fall sei q mit $\mathsf{in}_1 q \equiv f_{ij}\vec{y}_{ij}\vec{z}_{ij}$ und q **r** $P\vec{s}_{ij}$ gegeben. Wir haben dann

$$g_{ij}\vec{y}_{ij}\vec{z}_{ij} \equiv [\mathsf{id}, \lambda_v({}^{co}\mathscr{R}vz)](f_{ij}\vec{y}_{ij}\vec{z}_{ij}) \equiv [\mathsf{id}, \lambda_v({}^{co}\mathscr{R}vz)]\mathsf{in}_1 q \equiv {}^{co}\mathscr{R}qz$$

und somit ist auch in diesem Fall die Zielformel gezeigt. □

Definition 3.2.28. Wir erweitern die Definition des extrahierten Terms auf Beweise mit den Axiomen der coinduktiv definierten Prädikate ${}^{co}I$ durch die Regeln

$$et({}^{co}I^{-}) := \mathscr{D}$$
$$et({}^{co}I^{+}(P)) := {}^{co}\mathscr{R}^{\tau(P)}_{\tau(I)},$$

wenn das zugrunde liegende induktiv definierte Prädikat rechnerisch ist.

Satz 3.2.29. Der Korrektheitssatz gilt auch in der Theorie **TCF** mit den Axiomen der coinduktiv definierten Prädikate. Das bedeutet:
Ist M eine Herleitung einer Aussage A unter den Annahmen $(u_i : B_i)_{i<n}$ ohne eine Regel für den rechnerischen Implikationspfeil in einem nicht-rechnerischen Teil von M, dann gilt $\mathbf{TCF} \cup \{x_{u_i}$ **r** $B_i | i < n\} \vdash et(M)$ **r** A, wobei hier in **TCF** nun auch die Axiome ${}^{co}I^{-}$ und ${}^{co}I^{+}(P)$ für jedes induktiv definierte Prädikat I und jedes Prädikat P mit den gleichen Argumententypen wie I in **TCF** sind.

Beweis. Der Beweis folgt direkt aus dem Beweis des Korrektheitssatzes 1.9.2 ohne die Axiome der coinduktiv definierten Prädikate und aus den Lemmata 3.2.23 und 3.2.27. □

3.3 Arithmetische Funktionen in der SD-Darstellung

Motivation 3.3.1. In diesem Abschnitt wollen wir die Addition und besonders die Division von reellen Zahlen in der SD-Darstellung betrachten. Es soll auf konstruktive Weise aus einer SD-Darstellung von x und einer SD-Darstellung y eine SD-Darstellung von $\frac{x+y}{2}$ und unter gewissen Voraussetzungen eine SD-Darstellung von $\frac{x}{y}$ erzeugt werden. Wir betrachten hier das arithmetische Mittel $\frac{x+y}{2}$ und nicht die eigentliche Summe $x+y$, um das Intervall $[-1,1]$ nicht zu verlassen. Das Mittel wurde bereits unter anderem in der Literatur [8] ausführlich diskutiert. Es befindet sich eine Implementierung in `examples/analysis/sdmult.scm` des Minlogverzeichnisses. Wie der Name vermuten lässt, ist dort auch die Multiplikation implementiert. Bei Interesse an der Multiplikation verweisen wir zusätzlich noch auf [10]. Die Division von reellen Zahlen in der SD-Darstellung ist in der Literatur noch nicht so weit vertreten. Insbesondere gab es bis zu dieser Arbeit noch keine Implementierung in Minlog. Die Überlegungen zu der Division stammen aus der Dissertation [3] von Alberto Ciaffaglione. Dabei war Definition 4.3 aus dieser Arbeit für die Umsetzung der Division sehr hilfreich. Das arithmetische Mittel zweier Zahlen in der SD-Darstellung werden wir für die Division brauchen, sodass wir zunächst darauf eingehen. Außerdem ist das arithmetische Mittel auch didaktisch wertvoll, weil der extrahierte Term

noch hinreichend kurz ist, sodass auch ein Mensch damit gut umgehen kann und wir dadurch eine Anwendung des Corekursionsoperators sehen. Folgendes Lemma werden wir noch häufig implizit verwenden:

Lemma 3.3.2. Sind x und y zwei gleiche reelle Zahlen, dann folgt aus ${}^{co}\mathbf{I}x$ auch ${}^{co}\mathbf{I}y$. In einer Formel ausgedrückt heißt das:

$$\forall^{nc}_{x,y}.{}^{co}\mathbf{I}x \to x = y \to {}^{co}\mathbf{I}y$$

Beweis. Wir verwenden auf das Einführungsaxiom von ${}^{co}\mathbf{I}$ für das Prädikat $\mathbf{P}y := \exists^r_x({}^{co}\mathbf{I}x \wedge^l x = y)$. Um die gewünschte Aussage zu zeigen, reicht es, einen Beweis für die mittlere Prämisse

$$\forall^{nc}_y.\exists^r_x({}^{co}\mathbf{I}x \wedge^l x = y) \to \exists^r_{d,y'}\left(\mathbf{Sd}\ d \wedge^d ({}^{co}\mathbf{I}y' \vee \mathbf{P}y') \wedge^l |y| \le 1 \wedge^l y = \frac{d+y'}{2}\right)$$

anzugeben. Es sei also x und y mit ${}^{co}\mathbf{I}x$ und $x = y$ gegeben. Aus ${}^{co}\mathbf{I}x$ erhalten wir e und x' mit $\mathbf{Sd}\ e$, ${}^{co}\mathbf{I}x'$ und $x = \frac{e+x'}{2}$. Setzen wir in der Existenzaussage $d = e$ und $y' = x'$ ein, so haben wir $\mathbf{Sd}\ d$, ${}^{co}\mathbf{I}y'$ und $y = \frac{d+y'}{2}$ nach Voraussetzung und $|y| \le 1$ folgt wegen $|x| \le 1$, weil ${}^{co}\mathbf{I}x$ gilt. □

Bemerkung 3.3.3. In der Datei `sdmult.scm` heißt dieses Lemma `CoICompat`. Als rechnerischen Gehalt gibt Minlog den Term

$$\texttt{cCoICompat} := \lambda_u.{}^{co}\mathscr{R}^{\mathbf{Sd}\times\mathtt{str}}_{\mathtt{str}}(\mathscr{D}u)\lambda_s\langle\pi_0 s, \mathsf{in}_0(\pi_1 s)\rangle$$

aus. Machen wir einen Konversionsschritt des Corekursionsoperators, erhalten wir nach Normalisierung

$$\lambda_u(\pi_0(\mathscr{D}u) :: \pi_1(\mathscr{D}u)).$$

Für Terme der Form $d :: u$ ist dies die Identitätsfunktion. Weil wir den extrahierten Term ausschließlich auf solche Terme anwenden möchten, werden wir `cCoICompat` häufig einfach weglassen.

3.3.1 Das arithmetische Mittel

Motivation 3.3.4. Wir beweisen nun die Aussage, dass es zu einer SD-Darstellung von zwei reellen Zahlen x und y auch eine SD-Darstellung von deren Durchschnitt $\frac{x+y}{2}$ gibt. Dazu beweisen wir die Formel

$$\forall^{nc}_{x,y}\left({}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to {}^{co}\mathbf{I}\frac{x+y}{2}\right).$$

Da am Ende der Formel das coinduktiv definierte Prädikat ${}^{co}\mathbf{I}$ steht, werden wir das Einführungsaxiom von ${}^{co}\mathbf{I}$ benötigen. Das Eliminationsaxiom werden wir ebenso brauchen, weil auch in beiden Prämissen ${}^{co}\mathbf{I}$ vorkommt. Um den Beweis übersichtlich zu halten, wird dieser über das Prädikat

$$\mathbf{Q} := \left\{ z \,\middle|\, \exists^r_{x,y,i}.{}^{co}\mathbf{I}x \wedge^d {}^{co}\mathbf{I}y \wedge^d \mathbf{Sd}_2\, i \wedge^l z = \frac{x+y+i}{4} \right\}$$

in zwei Abschnitte geteilt. Dabei ist $\mathbf{Sd}_2 := \{-2,-1,0,1,2\}$. $\mathbf{Sd}_2$ wird analog zu $\mathbf{Sd}$ als induktiv definiertes Prädikat realisiert und wir identifizieren wieder $\tau(\mathbf{Sd}_2) = \mathbf{Sd}_2$. Im ersten Teil wird bewiesen, dass $\mathbf{Q}\frac{x+y}{2}$ für jedes x und y mit ${}^{co}\mathbf{I}x$ und ${}^{co}\mathbf{I}y$ gilt. Dafür wird das Eliminationsaxiom von ${}^{co}\mathbf{I}$ benötigt. Im zweiten Teil zeigen wir $\mathbf{P} \subseteq {}^{co}\mathbf{I}$, wofür wir das Einführungsaxiom verwenden.

Lemma 3.3.5.

$$\forall^{nc}_{x,y}.{}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to \exists^{r}_{i,x',y'}.\mathbf{Sd}_2 i \wedge^{d\,co}\mathbf{I}x' \wedge^{d\,co}\mathbf{I}y' \wedge^{l} \frac{x+y}{2} = \frac{x'+y'+i}{4}$$

Beweis. Aus ${}^{co}\mathbf{I}x$ und ${}^{co}\mathbf{I}y$ erhalten wir x', d, y' und e mit ${}^{co}\mathbf{I}x'$, ${}^{co}\mathbf{I}y'$, $\mathbf{Sd}\ d$ und $\mathbf{Sd}\ e$ sowie $x = \frac{x'+d}{2}$ und $y = \frac{y'+e}{2}$. Setzen wir nun $i = d + e$ so haben wir

$$\frac{x+y}{2} = \frac{\frac{x'+d}{2} + \frac{y'+e}{2}}{2} = \frac{x'+y'+i}{4}.$$

□

Bemerkung 3.3.6. Betrachten wir den Beweis auf einer rechnerischen Ebene, sehen wir, dass wir nur die erste Stelle der SD-Darstellung von x und y benötigen. In der Datei `sdmult.scm` wird dieser Satz mit `CoIAvToAvc` bezeichnet. Minlog liefert den extrahierten Term

```
[u,u0]
 cIntPlusSdToSdtwo clft DesYprod u clft DesYprod u0 pair
 crht DesYprod u pair crht DesYprod u0
```

Der Destruktor wird jeweils nur einfach auf `u` und `u0` angewendet. Wegen der Typumwandlung von `Sd` zu `Sdtwo` ist der erste Teil `cIntPlusSdToSdtwo clft DesYprod u clft DesYprod u0` sehr lange. Lassen wir die Typkonversionen implizit, hat der extrahierte Term die folgende Gestalt:

$$\lambda_{u,v}\langle \pi_0(\mathscr{D}u) + \pi_0(\mathscr{D}v), \pi_1(\mathscr{D}u), \pi_1(\mathscr{D}v)\rangle$$

Zu einer gegeben SD-Darstellung erhalten wir ein Tripel aus der Summe der beiden ersten Ziffern und der beiden SD-Darstellungen ohne die erste Ziffer.

Lemma 3.3.7.

$$\forall^{nc}_{i,x,y}.\mathbf{Sd}_2 i \to {}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to$$
$$\exists^{r}_{j,d,x',y'}.\mathbf{Sd}_2 j \wedge^{d}\mathbf{Sd}\ d \wedge^{d\,co}\mathbf{I}x' \wedge^{d\,co}\mathbf{I}y' \wedge^{l} \frac{x+y+i}{4} = \frac{\frac{x'+y'+j}{4}+d}{2}$$

Beweis. Es seien i, x und y gegeben und es gelte $\mathbf{Sd}_2 i$, ${}^{co}\mathbf{I}x$ und ${}^{co}\mathbf{I}y$. Wegen ${}^{co}\mathbf{I}x$ und ${}^{co}\mathbf{I}y$ erhalten wir d, e, x' und y' mit $\mathbf{Sd}\ d$, $\mathbf{Sd}\ e$, ${}^{co}\mathbf{I}x'$ und ${}^{co}\mathbf{I}y'$ sowie $x = \frac{x'+d}{2}$ und $y = \frac{y'+e}{2}$. Wir haben dann

$$\frac{x+y+i}{4} = \frac{x'+y'+d+e+2i}{8}.$$

Weil $|d+e+2i| \le 6$ ist, gibt es j und k mit $\mathbf{Sd}_2 j$ und $\mathbf{Sd}\ k$, sodass $d+e+2i = j+4k$ ist. Damit folgt

$$\frac{x+y+i}{4} = \frac{x'+y'+j+4k}{8} = \frac{\frac{x'+y'+j}{4}+k}{2}.$$

□

Bemerkung 3.3.8. Dieses Lemma besagt, dass das Prädikat **Q** eben genau die Coinduktionsklausel erfüllt. In `sdmult.scm` heißt dieses Lemma `CoIAvcSatCoICl`. Mit ihm werden wir den Satz $\forall^{nc}_{x,y}.{}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to {}^{co}\mathbf{I}\frac{x+y}{2}$ beweisen. Zunächst wollen wir aber noch den rechnerischen Gehalt dieses Lemmas betrachten. Die Ausgabe von Minlog ist:

```
[t,u,u0]
IntToSdtwo
(J(SdToInt clft DesYprod u+
   SdToInt clft DesYprod u0+SdtwoToInt t*2)) pair
IntToSd
(K(SdToInt clft DesYprod u+
   SdToInt clft DesYprod u0+SdtwoToInt t*2)) pair
crht DesYprod u pair crht DesYprod u0
```

Es fällt auf, dass die Funktionen `J` und `K` auftreten. Diese entsprechen jenen j und k aus obigen Beweis, welche so gewählt waren, dass $d+e+2i = j+4k$ und $\mathbf{Sd}_2\, j$ und $\mathbf{Sd}\ k$ gilt. Die Funktionen `J` und `K` sind in Minlog als Programmkonstanten umgesetzt und haben genau die Eigenschaft, dass $z = \mathtt{J}(z) + 4\mathtt{K}(z)$ für jede ganze Zahl z zwischen -6 und 6 gilt. `J` und `K` werden in der Datei `sdJK.scm` eingeführt und die drei Eigenschaften aus dem Beweis, die wir brauchen, werden dort auch bewiesen. Mit dieser Funktion geben wir nun den extrahierten Term in einer etwas kürzeren Form an. Wir lassen wieder die Typkonversionen weg:

$$\lambda_{t,u,v}\langle \mathtt{J}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t), \mathtt{K}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t), \pi_1(\mathscr{D}u), \pi_1(\mathscr{D}v)\rangle$$

Bezeichnen wir diese Funktion mit `f`, so haben wir für i, d, e, u und v:

$$\mathtt{f}(i, d :: u, e :: v) = \langle \mathtt{J}(d+e+2i), \mathtt{K}(d+e+2i), u, v\rangle$$

Wir erkennen, dass auch hier nur jeweils die erste Stelle der SD-Darstellung benötigt wird.

Satz 3.3.9.

$$\forall^{nc}_{x,y}.{}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to {}^{co}\mathbf{I}\frac{x+y}{2}$$

Beweis. Das Einführungsaxiom von ${}^{co}\mathbf{I}$ auf das Prädikat **Q** liefert uns nach Motivation 3.2.12 die Aussage

$$\forall^{nc}_x.\mathbf{Q}x \to \forall^{nc}_x\left(\mathbf{Q}x \to \exists^r_{d,x'}.\mathbf{Sd}\ d \wedge^d ({}^{co}\mathbf{I} \cup \mathbf{Q})x' \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}\right) \to {}^{co}\mathbf{I}x.$$

Wir möchten nun die zweite Prämisse beweisen. Dazu beweisen wir sogar die stärkere Form

$$\forall^{nc}_x\left(\mathbf{Q}x \to \exists^r_{d,x'}\left(\mathbf{Sd}\ d \wedge^d \mathbf{Q}x' \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}\right)\right).$$

Fügen wir dafür die Definition von **Q** ein:

$$\begin{aligned}
&\forall^{nc}_x.\exists_{i,x_0,y_0}\left({}^{co}\mathbf{I}x_0 \wedge^d {}^{co}\mathbf{I}y_0 \wedge^d \mathbf{Sd}_2\, i \wedge^l x = \frac{x_0+y_0+i}{4}\right) \to \\
&\quad \exists^r_{d,x'}.\mathbf{Sd}\ d \wedge^d \exists_{i,x_0,y_0}\left({}^{co}\mathbf{I}x_0 \wedge^d {}^{co}\mathbf{I}y_0 \wedge^d \mathbf{Sd}_2\, i \wedge^l x' = \frac{x_0+y_0+i}{4}\right) \\
&\quad \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}.
\end{aligned}$$

Seien nun x, i, x_0 und y_0 gegeben und es gelte ${}^{co}\mathbf{I}x_0$, ${}^{co}\mathbf{I}y_0$, $\mathbf{Sd}_2 i$ und $x = \frac{x_0+y_0+i}{4}$. Es reicht dann zu zeigen, dass

$$\exists^r_{d,x'}.\mathbf{Sd}\ d \wedge^d \exists_{i,x_0,y_0}\left({}^{co}\mathbf{I}x_0 \wedge^{d\ co}\mathbf{I}y_0 \wedge^d \mathbf{Sd}_2 i \wedge^l x' = \frac{x_0+y_0+i}{4}\right) \wedge^l$$
$$|x| \le 1 \wedge^l \frac{x_0+y_0+i}{4} = \frac{d+x'}{2}$$

gilt. Aus Lemma 3.3.7 erhalten wir j, d, x', y' mit

$$\mathbf{Sd}_2 j \wedge^d \mathbf{Sd}\ d \wedge^{d\ co}\mathbf{I}x' \wedge^{d\ co}\mathbf{I}y' \wedge^l \frac{x_0+y_0+i}{4} = \frac{\frac{x'+y'+j}{4}+d}{2}.$$

Um den Beweis zu beenden, zeigen wir deswegen

$$\mathbf{Sd}\ d \wedge^{d\ co}\mathbf{I}x' \wedge^{d\ co}\mathbf{I}y' \wedge^d \mathbf{Sd}_2 j \wedge^l \frac{x'+y'+j}{4} = \frac{x'+y'+j}{4} \wedge^l$$
$$|x| \le 1 \wedge^l \frac{x_0+y_0+i}{4} = \frac{d+\frac{x'+y'+j}{4}}{2}.$$

Die ersten vier Aussagen und die letzte Aussage der Konjunktion sind bereits gegeben, die fünfte Aussage gilt wegen der Reflexivität der reellen Gleichheit und die Aussage $|x| \le 1$ folgt wegen $x = \frac{x_0+y_0+i}{4}$ und ${}^{co}\mathbf{I}x_0$, ${}^{co}\mathbf{I}y_0$, $\mathbf{Sd}_2 i$ und somit $|x_0| \le 1$, $|y_0| \le 1$ und $|i| \le 2$. □

Bemerkung 3.3.10. Um den rechnerischen Gehalt dieses Satzes zu bestimmen, bedienen wir uns wieder der Implementation in Minlog. Der Satz ist in `sdmult.scm` unter `CoIAverage` gespeichert. Als extrahierten Term liefert Minlog

```
[u,u0]
 [let tuv
   (IntToSdtwo(SdToInt clft DesYprod u+SdToInt clft DesYprod u0)
   pair
   crht DesYprod u pair crht DesYprod u0)
   ((CoRec sdtwo yprod str yprod str=>str)tuv
   ([tuv0]
     [let tsuv
       (IntToSdtwo
       (J
        (SdToInt clft DesYprod clft crht tuv0+
         SdToInt clft DesYprod crht crht tuv0+
         SdtwoToInt clft tuv0*2))pair
       IntToSd
       (K
        (SdToInt clft DesYprod clft crht tuv0+
         SdToInt clft DesYprod crht crht tuv0+
         SdtwoToInt clft tuv0*2))pair
       crht DesYprod clft crht tuv0 pair
       crht DesYprod crht crht tuv0)
       (clft crht tsuv pair
       (InR (sdtwo yprod str yprod str) str)
            (clft tsuv pair crht crht tsuv))]))]
```

Dieser ist kurz genug, um ihn zu verstehen. Zuerst fällt ins Auge, dass zweimal das Wort `let` in dem Term vorkommt. Das liegt an einem kleinen Trick, der bei der Formalisierung des Beweises verwendet wurde: In Minlog ist das Theorem `Id` eingespeichert. Dieses hat die Aussage `Pvar -> Pvar` und besitzt als extrahierten Term die Identität auf dem Typ von `Pvar`. Das Theorem selbst ist nicht animiert. Tritt im extrahierten Term ein Betaredex der Form $(\lambda_x M(x))N$ auf, würde Minlog dies zu $M(N)$ normalisieren. Durch Verwenden von `Id` kann man diesen Betaredex zu $\texttt{cId}(\lambda_x M(x))N$ umwandeln, was links geklammert also wie $(\texttt{cId}\lambda_x M(x))N$ vom Programm gelesen und somit auch nicht normalisiert wird. Dargestellt wird dies dann als $[\texttt{let}\ x\ N\ M(x)]$. Ein Term der Form `[let x s t(x)]` kann also einfach als `t(s)` verstanden werden. Der Vorteil der ersten Schreibweise ist, dass in diesem Term der Term `s` nur einmal vorkommt und sonst durch die Variable `x` ersetzt wird. Ist `s` ein langer Term, kann eine Verwendung von `let` gut zur Lesbarkeit des gesamten Terms beitragen. So ist es auch hier der Fall. Der extrahierte Term wäre ohne `let` deutlich länger. Der Leser kann dies gerne überprüfen, indem er vor dem Normalisieren des extrahierten Terms den Befehl `(animate "Id")` eingibt.
Wie erwartet tritt in dem extrahierten Term genau einmal der Corekursionsoperator `(CoRec sdtwo yprod str yprod str=>str)` also ${}^{co}\mathscr{R}^{\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}}_{\texttt{str}}$ auf. Nach Definition 3.2.24 ist der Typ dieses Corekursionsoperators

$$\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}\to(\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}\to(\mathbf{Sd}\times(\texttt{str}+\mathbf{Sd}_2\times\texttt{str}\times\texttt{str})))\to\texttt{str}.$$

Mit dieser Vorbereitung fällt es etwas leichter, den extrahierten Term in eine annehmbare Form zu bringen:

$$\lambda_{u,v}.{}^{co}\mathscr{R}^{\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}}_{\texttt{str}}\ \langle\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v),\pi_1(\mathscr{D}u),\pi_1(\mathscr{D}v)\rangle\ \lambda_{t_0,u_0,v_0}$$
$$\langle\mathtt{K}(\pi_0(\mathscr{D}u_0)+\pi_0(\mathscr{D}v_0)+2t_0),\mathsf{in}_1\langle\mathtt{J}(\pi_0(\mathscr{D}u_0)+\pi_0(\mathscr{D}v_0)+2t_0),\pi_1(\mathscr{D}u_0),\pi_1(\mathscr{D}v_0)\rangle\rangle$$

Beispiel 3.3.11. Wenden wir den Term aus der obigen Bemerkung nun auf $u:=d_0::d_1::\tilde{u}$ und $v:=e_0::e_1::\tilde{v}$ an, erhalten wir

$${}^{co}\mathscr{R}^{\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}}_{\texttt{str}}\ \langle d_0+e_0,d_1::\tilde{u},e_1::\tilde{v}\rangle\ \lambda_{t,u,v}$$
$$\langle\mathtt{K}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t),\mathsf{in}_1\langle\mathtt{J}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t),\pi_1(\mathscr{D}u),\pi_1(\mathscr{D}v)\rangle\rangle.$$

Darauf verwenden wir nun die Konversionsregel des Corekursionsoperators aus Definition 3.2.24. In unserem Fall ist $k=1$,

$$M=$$
$$\lambda_{t,u,v}\langle\mathtt{K}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t),\mathsf{in}_1\langle\mathtt{J}(\pi_0(\mathscr{D}u)+\pi_0(\mathscr{D}v)+2t),\pi_1(\mathscr{D}u),\pi_1(\mathscr{D}v)\rangle\rangle$$

und

$$L_0=\lambda_x\lambda_f.x::\Big([\mathsf{id},\lambda_y{}^{co}\mathscr{R}^{\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}}_{\texttt{str}}yM]f\Big),$$

wobei x vom Typ $\mathbf{Sd}$ und f vom Typ $\texttt{str}+\mathbf{Sd}_2\times\texttt{str}\times\texttt{str}$ sind. Wenden wir also auf den Term nun die Konversionsregel des Corekursionsoperators an, bekommen wir

$$L_0M\langle d_0+e_0,d_1::\tilde{u},e_1::\tilde{v}\rangle\ .$$

Nach β-Konversionen erhalten wir dann

$$L_0\langle\mathtt{K}(d_1+e_1+2(d_0+e_0)),\mathsf{in}_1\langle\mathtt{J}(d_1+e_1+2(d_0+e_0)),\tilde{u},\tilde{v}\rangle\rangle.$$

Setzen wir nun den Term für L_0 ein und machen noch Konversionen, haben wir

$$\mathtt{K}(d_1+e_1+2(d_0+e_0)) :: \left({}^{co}\mathscr{R}^{\mathbf{Sd}_2\times\mathtt{str}\times\mathtt{str}}_{\mathtt{str}}\langle \mathtt{J}(d_1+e_1+2(d_0+e_0)), \tilde{u}, \tilde{v}\rangle M\right).$$

Man sieht daran, dass man nur die ersten beiden Ziffer in der SD-Darstellung der reellen Zahlen x und y kennen muss, um die erste Ziffer in der SD-Darstellung von $\frac{x+y}{2}$ zu berechnen. Da in M der Destruktor jeweils immer nur einmal auf u und v angewendet wird, kann man sogar allgemeiner feststellen: Um die ersten n Ziffern der SD-Darstellung von $\frac{x+y}{2}$ zu bestimmen, sind die ersten $n+1$ Ziffern der SD-Darstellungen von x und y nötig.

Korollar 3.3.12. Mit dem eben bewiesenen Satz 3.3.9 können wir auf den reellen Zahlen in der Darstellung aus Korollar 3.2.17 sogar die gewöhnliche Addition durchführen. Stellen wir zwei reelle Zahlen x, y als $x = k + x'$ und $y = l + y'$ dar, wobei k, l ganze Zahlen sind und ${}^{co}\mathbf{I}x'$ und ${}^{co}\mathbf{I}y'$ gilt, so folgt $x+y = k+l+2\frac{x'+y'}{2}$. Nach Satz 3.3.9 gilt ${}^{co}\mathbf{I}\frac{x'+y'}{2}$. Also gibt es d, z mit $\mathbf{Sd}\ d$, ${}^{co}\mathbf{I}z$ und $\frac{x'+y'}{2} = \frac{d+z}{2}$. Das liefert uns dann $x+y = k+l+2\frac{x'+y'}{2} = (k+l+d)+z$. Damit hat $x+y$ wieder die gewünschte Darstellung.

3.3.2 Die Division

Motivation 3.3.13. Wir werden nun das arithmetische Mittel verwenden, um einen Divisionsalgorithmus von zwei reellen Zahlen in der SD-Darstellung zu erhalten. Natürlich müssen wir den Definitionsbereich etwas einschränken, damit wir den Wertebereich zwischen -1 und 1 nicht verlassen. Um eine SD-Darstellung von $\frac{x}{y}$ zu bekommen, muss $|x| \leq |y|$ sein. Außerdem wissen wir aus Lemma 3.1.36, dass $|y|$ positiv sein muss, damit wir y überhaupt invertieren können. Wir beschränken uns dafür erst auf den Fall, dass $y \geq \frac{1}{4}$ gilt, und überlegen uns später, wie wir dies auf $|y| \geq 2^{-p}$ für beliebiges p verallgemeinern können. Das Ziel ist es daher, den Satz

$$\forall^{nc}_{x,y}\left({}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to |x| \leq y \to \frac{1}{4} \leq y \to {}^{co}\mathbf{I}\frac{x}{y}\right)$$

zu beweisen. Dafür werden wir wieder die Coinduktionsaxiome verwenden.
Die Hauptidee des Beweises besteht in folgenden drei Darstellungen von $\frac{x}{y}$:

$$\frac{x}{y} = \frac{1+\frac{x_1}{y}}{2} = \frac{0+\frac{x_0}{y}}{2} = \frac{-1+\frac{x_{-1}}{y}}{2}$$

Dabei sind $x_1 = 4\frac{x+\frac{-y}{2}}{2}$, $x_0 = 2x$ und $x_{-1} = 4\frac{x+\frac{y}{2}}{2}$. Je nachdem, welche Information über x gegeben ist, kann man eine dieser Darstellungen von $\frac{x}{y}$ verwenden, um die erste Ziffer von $\frac{x}{y}$ zu erhalten. So haben wir eine corekursive Definition der Division. Bevor wir das Einführungsaxiom von ${}^{co}\mathbf{I}$ verwenden können, müssen wir zeigen, dass x_1, x_0 und x_{-1} wieder in ${}^{co}\mathbf{I}$ liegen. Dazu brauchen wir natürlich noch ein paar Lemmata als Vorbereitung.

Lemma 3.3.14. Die folgenden beiden Aussagen gelten:

$$\forall^{nc}_{x}.{}^{co}\mathbf{I}x \to x \leq 0 \to {}^{co}\mathbf{I}(x+1)$$
$$\forall^{nc}_{x}.{}^{co}\mathbf{I}x \to 0 \leq x \to {}^{co}\mathbf{I}(x-1)$$

Beweis. Weil beide Aussagen analog bewiesen werden, beschränken wir uns darauf, die erste Aussage

$$\forall_x^{nc}.{}^{co}\mathbf{I}x \to x \le 0 \to {}^{co}\mathbf{I}(1+x)$$

zu zeigen. Dafür wenden wir das Einführungsaxiom von ${}^{co}\mathbf{I}$ auf das Prädikat $Px := \exists_y^r({}^{co}\mathbf{I}y \wedge^l y \le 0 \wedge x = y+1)$ an. Das gibt uns

$$\forall_x^{nc}.\exists_y^r({}^{co}\mathbf{I}y \wedge^l y \le 0 \wedge^l x = y+1) \to$$
$$\forall_x^{nc}\left(\exists_y^r({}^{co}\mathbf{I}y \wedge^l y \le 0 \wedge^l x = y+1) \to \exists_{d,x'}^r\left(\mathsf{Sd}\ d \wedge^d ({}^{co}\mathbf{I} \cup P)x' \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}\right)\right)$$
$$\to {}^{co}\mathbf{I}x.$$

Für unsere gewünschte Aussage reicht es, die zweite Prämisse zu zeigen. Sei daher x und y gegeben und es gelte ${}^{co}\mathbf{I}y$, $y \le 0$ und $x = y+1$. Unser Ziel ist es, eine Ziffer d und eine reelle Zahl x' zu finden, sodass

$$\left({}^{co}\mathbf{I}x' \vee \exists_{y'}({}^{co}\mathbf{I}y' \wedge^l y' \le 0 \wedge^l x' = y'+1)\right) \wedge^l |x| \le 1 \wedge^l x = \frac{d+x'}{2}$$

gilt. Aus ${}^{co}\mathbf{I}y$ folgt $|y| \le 1$ und zusammen mit $y \le 0$ und $x = y+1$ folgt $|x| \le 1$ unabhängig von der Wahl von x' und d. Es reicht also x' und d anzugeben, sodass

$$\left({}^{co}\mathbf{I}x' \vee \exists_{y'}({}^{co}\mathbf{I}y' \wedge^l y' \le 0 \wedge^l x' = y'+1)\right) \wedge^l x = \frac{f+x'}{2}$$

gilt. Wegen ${}^{co}\mathbf{I}y$ erhalten wir eine Ziffer e und ein z mit ${}^{co}\mathbf{I}z$ und $y = \frac{e+z}{2}$. Wir machen nun Fallunterscheidung nach e:
Ist $e = 1$, so folgt $0 \ge y = \frac{1+z}{2} = \frac{1}{2} + \frac{z}{2} \ge \frac{1}{2} - \frac{1}{2} = 0$ und somit $x = 1 + y = 1$. Wir setzen in diesem Fall $d = 1$ und $x' = 1$. Dann gilt ${}^{co}\mathbf{I}x'$ und $x = \frac{d+x'}{2}$.
Ist $e = 0$, so setzen wir $d = 1$ und $x' = z+1$. ${}^{co}\mathbf{I}z$ ist dann bereits gegeben. Weiter haben wir $z = 2y - e = 2y \le 0$ und $x' = z+1$ nach Definition von x'. Damit ist die rechte Seite der obigen Disjunktion gezeigt. Zusätzlich gilt $x = y+1 = \frac{z}{2}+1 = \frac{1+(z+1)}{2} = \frac{1+x'}{2}$. Womit wir im Fall $e = 0$ fertig sind.
Zuletzt betrachten wir den Fall $e = -1$. Dort setzen wir $d = 1$ und $x' = z$. Es ist die Aussage ${}^{co}\mathbf{I}z$ schon gegeben und es gilt $\frac{d+x'}{2} = \frac{1+z}{2} = \frac{-1+z}{2} + 1 = y + 1 = x$. Dies beendet den Beweis. □

Bemerkung 3.3.15. Unter `CoINegToCoIPlusOne` und `CoIPosToCoIMinusOne` sind die beiden Aussagen des Lemmas als Theoreme in der Datei `CoIQuad.scm` auf der beiliegenden CD implementiert. Als rechnerischen Gehalt der ersten Aussage, welche wir auch im Beweis betrachtet haben, gibt Minlog

```
[u0]
 (CoRec str=>str)u0
 ([u1]
   [if (DesYprod u1)
     ([s2,u3]
      [if s2
        (SdR pair(InL str str)cCoIOne)
        (SdR pair(InR str str)u3)
        (SdR pair(InL str str)u3)])])
```

aus. Dabei ist cCoIOne der rechnerische Gehalt, des Beweises der Aussage ${}^{co}\mathbf{I}1$. Dieser ist im Grunde die unendliche Liste bestehend aus 1. Wir schreiben daher einfach $\vec{1}$. Der extrahierte Term sieht mit der Notation aus dem ersten Kapitel dann wie folgt aus:

$$\lambda_u.{}^{co}\mathscr{R}^{\mathtt{str}}_{\mathtt{str}}u\left(\lambda_u.\mathscr{C}_{\mathbf{Sd}\times\mathtt{str}}(\mathscr{D}u)\left(\lambda_{d,u'}.\mathscr{C}_{\mathbf{Sd}}d\langle 1,\mathsf{in}_0\vec{1}\rangle\langle 1,\mathsf{in}_1 u'\rangle\langle 1,\mathsf{in}_0 u'\rangle\right)\right)$$

Wir erwarten, dass dieser zu einer SD-Darstellung einer nicht-positiven reellen Zahl x eine SD-Darstellung von $x+1$ liefert.
Aus dem Term kann man Folgendes ablesen: Haben wir eine SD-Darstellung von x, dann betrachten wir die erste Ziffer d. Im extrahierten Term wird durch den zweiten Caseoperator Fallunterscheidung nach dieser Ziffer gemacht. Ist $d=1$, folgt schon, dass x nicht-negativ ist. Damit muss $x=0$ sein und eine SD-Darstellung von $x+1$ ist dann $\vec{1}$. Hier brauchen wir keine Corekursion, weswegen für diesen Fall in_0 im extrahierten Term erscheint. Gleiches haben wir auch, wenn $d=-1$ ist. Hier kann man leicht $x+1$ bestimmen, indem man die erste Ziffer von -1 auf 1 abändert und den Rest der SD-Darstellung gleich lässt. Das macht sich im extrahierten Term durch $\langle 1,\mathsf{in}_0 u'\rangle$ bemerkbar. Nur im Fall $d=0$ haben wir eine echte Corekursion. Denn hier ändern wir die erste Ziffer von 0 zu 1, haben damit aber nur $\frac{1}{2}$ addiert. Deswegen führen wir das Verfahren nun mit dem Rest der SD-Darstellung fort. Für diesen Fall steht das Paar $\langle 1,\mathsf{in}_1 u'\rangle$ im extrahierten Term.
Mit dieser Überlegung können wir den rechnerischen Gehalt auch durch eine Programmkonstante $\mathtt{f}:\mathtt{str}\to\mathtt{str}$ mit den Berechnungsregeln

$$\begin{aligned}\mathtt{f}(1::u)&:=[1,\dots]\\ \mathtt{f}(0::u)&:=1::\mathtt{f}(u)\\ \mathtt{f}(\overline{1}::u)&:=1::u\end{aligned}$$

beschreiben.
Diese Ideen für die erste Aussage des Lemmas kann man auch analog für die zweite Aussage anwenden. Auf diese Weise erhält man für die zweite Aussage eine ähnliche Programmkonstante $\mathtt{g}:\mathtt{str}\to\mathtt{str}$ mit den Berechnungsregeln

$$\begin{aligned}\mathtt{g}(1::u)&:=\overline{1}::u\\ \mathtt{g}(0::u)&:=\overline{1}::\mathtt{g}(u)\\ \mathtt{g}(\overline{1}::u)&:=[\overline{1},\dots].\end{aligned}$$

Lemma 3.3.16.

$$\forall^{nc}_x\left({}^{co}\mathbf{I}x\to|x|\le\frac{1}{2}\to{}^{co}\mathbf{I}(2x)\right)$$

Beweis. Sei x mit ${}^{co}\mathbf{I}x$ und $|x|\le\frac{1}{2}$ gegeben. Wegen ${}^{co}\mathbf{I}x$ gibt es d und x' mit ${}^{co}\mathbf{I}x'$, $\mathbf{Sd}\ d$ und $x=\frac{d+x'}{2}$. Wir machen Fallunterscheidung nach d.
Ist $d=0$, so folgt die Aussage sofort, denn wir haben dann

$$2x=2\frac{0+x'}{2}=x'$$

und ${}^{co}\mathbf{I}x'$ ist bereits gegeben.
Für $d=1$ ist $x=\frac{1+x'}{2}$ und $2x=1+x'$. Weil $|x|\le\frac{1}{2}$ ist, folgt $x'\le 0$. Mit der ersten Aussage von Lemma 3.3.14 sind wir in diesem Fall fertig.
Im Fall $d=-1$ ist $2x=-1+x'$ und $0\le x'$. Hier verwenden wir die zweite Aussage aus Lemma 3.3.14. □

Bemerkung 3.3.17. Minlog gibt uns folgenden extrahierten Term

```
[u0]
 [if (DesYprod u0)
   ([s1,u2]
    [if s1
      (cCoICompat(cCoINegToCoIPlusOne u2))
      (cCoICompat u2)
      (cCoICompat(cCoIPosToCoIMinusOne u2))])]
```

Dabei ist cCoICompat der extrahierte Term aus Lemma 3.3.2. Wie wir in der Bemerkung darunter festgestellt haben, können wir diesen Term einfach weglassen. Mit ähnlichen Überlegungen, wie in Bemerkung 3.3.15, sehen wir, dass der extrahierte Term auch als eine Programmkonstante $\mathtt{D} : \mathtt{str} \to \mathtt{str}$ mit den Berechnungsregeln

$$\mathtt{D}(1 :: u) := \mathtt{f}(u)$$
$$\mathtt{D}(0 :: u) := u$$
$$\mathtt{D}(\overline{1} :: u) := \mathtt{g}(u)$$

umgesetzt werden kann. Die Notation von f und g ist aus Bemerkung 3.3.15 übernommen.

Lemma 3.3.18. Gilt ${}^{co}\mathbf{I}x$ für eine reelle Zahl x, so folgt auch ${}^{co}\mathbf{I}\frac{x}{2}$ und ${}^{co}\mathbf{I}(-x)$.

Beweis. Die beiden Aussagen folgen direkt, aus dem Einführungsaxiom von ${}^{co}\mathbf{I}$. Für die erste Aussage wendet man das Axiom auf das Prädikat $\left\{x \middle| \exists^r_y.{}^{co}\mathbf{I}y \wedge^l x = \frac{y}{2}\right\}$ an und für die zweite Aussage auf das Prädikat $\left\{x \middle| \exists^r_y.{}^{co}\mathbf{I}y \wedge^l x = -y\right\}$. □

Bemerkung 3.3.19. Der rechnerischen Gehalt beider Aussagen ist in diesem Fall leicht durch zwei Programmkonstanten $\mathtt{h}, \mathtt{n} : \mathtt{str} \to \mathtt{str}$ gegeben. Die Halbierungsfunktion h ist eine Shiftfunktion um eine Ziffer nach rechts: $\mathtt{h}(u) := 0 :: u$ und die Negationsfunktion n negiert die einzelnen Ziffern und hat daher die Berechnungsregel $\mathtt{n}(d :: u) := (-d) :: \mathtt{n}(u)$.

Satz 3.3.20.

$$\forall^{nc}_{x,y}.{}^{co}\mathbf{I}x \to {}^{co}\mathbf{I}y \to |x| \le y \to \frac{1}{4} \le y \to {}^{co}\mathbf{I}\frac{x}{y}$$

Beweis. Wir verwenden das Einführungsaxiom von ${}^{co}\mathbf{I}$ auf das Prädikat

$$P := \left\{z \middle| \exists^r_{x.y}.{}^{co}\mathbf{I}x \wedge^d {}^{co}\mathbf{I}y \wedge^d |x| \le y \wedge^l \frac{1}{4} \le y \wedge^l z = \frac{x}{y}\right\}.$$

So erhalten wir die Aussage

$$\begin{aligned}
&\forall^{nc}_z.\exists^r_{x.y}\left({}^{co}\mathbf{I}x \wedge^d {}^{co}\mathbf{I}y \wedge^d |x| \le y \wedge^l \frac{1}{4} \le y \wedge^l z = \frac{x}{y}\right) \to \\
&\forall^{nc}_z\left(\exists^r_{x.y}\left({}^{co}\mathbf{I}x \wedge^d {}^{co}\mathbf{I}y \wedge^d |x| \le y \wedge^l \frac{1}{4} \le y \wedge^l z = \frac{x}{y}\right)\right. \\
&\qquad \left.\to \exists^r_{d,z'}\left(\mathsf{Sd}\ d \wedge^d ({}^{co}\mathbf{I} \cup P)z' \wedge^l |z| \le 1 \wedge^l z = \frac{d+z'}{2}\right)\right) \\
&\to {}^{co}\mathbf{I}z.
\end{aligned}$$

Es reicht wieder, die zweite Prämisse

$$\forall^{nc}_z.\exists^r_{x.y}\left({}^{co}\mathbf{I}x \wedge^d {}^{co}\mathbf{I}y \wedge^d |x| \le y \wedge^l \frac{1}{4} \le y \wedge^l z = \frac{x}{y}\right)$$
$$\to \exists^r_{d,z'}\left(\mathbf{Sd}\ d \wedge^d ({}^{co}\mathbf{I} \cup P)z' \wedge^l |z| \le 1 \wedge^l z = \frac{d+z'}{2}\right)$$

zu zeigen. Seien daher x, y und z gegeben und es gelte ${}^{co}\mathbf{I}x$, ${}^{co}\mathbf{I}y$, $|x| \le y$, $\frac{1}{4} \le y$ und $z = \frac{x}{y}$. Um den Beweis zu beenden, zeigen wir sogar die stärkere Aussage

$$\exists^r_{d,z'}.\mathbf{Sd}\ d \wedge^d Pz' \wedge^l |z| \le 1 \wedge^l z = \frac{d+z'}{2}.$$

Die Aussage $|z| \le 1$ gilt wegen $|x| \le y$ unabhängig von d und z', sodass wir diese nicht weiter beachten. Durch dreifaches Anwenden von ${}^{co}\mathbf{I}^-$ auf x, erhalten wir d_1, d_2 und d_3 in $\mathbf{Sd}$ und ein $\tilde{x}$ mit ${}^{co}\mathbf{I}\tilde{x}$, sodass $x = \frac{4d_1+2d_2+d_3+\tilde{x}}{8}$ oder kurz $x = d_1d_2d_3\tilde{x}$ gilt. Wir unterschieden nun drei Fälle:
Ist $x = 1d_2d_3\tilde{x}$, $x = 01d_3\tilde{x}$ oder $x = 001\tilde{x}$, folgt daraus $0 \le x$. Wir setzen $d = 1$ und $z' = \frac{x'}{y}$ mit $x' := 4\frac{x+\frac{-y}{2}}{2}$. $\mathbf{Sd}\ d$ ist damit erfüllt. Nach Lemma 3.3.18 gilt ${}^{co}\mathbf{I}\frac{-y}{2}$ und nach Satz 3.3.9 folgt damit auch ${}^{co}\mathbf{I}\frac{x+\frac{-y}{2}}{2}$. Weil $0 \le x$ ist, haben wir $\left|\frac{x+\frac{-y}{2}}{2}\right| = \frac{|2x-y|}{4} \le \frac{|2y-y|}{4} = \frac{y}{4} \le \frac{1}{4}$. Damit können wir zweimal Lemma 3.3.16 verwenden und erhalten ${}^{co}\mathbf{I}x'$. Außerdem folgt aus dieser Ungleichungskette auch $|x'| \le y$ und somit gilt Pz'. Dass $\frac{d+z'}{2} = z$ gilt, lässt sich leicht nachrechnen.
Im Fall $x = \overline{1}d_2d_3\tilde{x}$, $x = 0\overline{1}d_3\tilde{x}$ oder $x = 00\overline{1}\tilde{x}$ folgt $x \le 0$ und wir setzen $d = -1$ und $z' = \frac{x'}{y}$ mit $x' := 4\frac{x+\frac{y}{2}}{2}$. Die Aussagen folgen dann analog zum ersten Fall.
Bleibt noch der Fall $x = 000\tilde{x}$. Hier gilt dann $|x| \le \frac{1}{8}$ und wir setzen $d = 0$ und $z' = \frac{x'}{y}$ mit $x' := 2x$. $\mathbf{Sd}\ d$ ist damit erfüllt. Weiter gilt $|x'| \le \frac{1}{4} \le y$ und mit Lemma 3.3.16 auch ${}^{co}\mathbf{I}x'$. Das liefert Pz'. Die Aussage $z = \frac{d+z'}{2}$ folgt auch sofort durch Nachrechnen. □

Bemerkung 3.3.21. Betrachten wir nun den extrahierten Term von Minlog

```
[u0,u1]
 (CoRec str=>str)u0
 ([u2]
   [if (cCoITripleClosure u2)
     ([s3,(sd yprod sd yprod str)_4]
      [if (sd yprod sd yprod str)_4
        ([s5,su6]
         [if su6
           ([s7,u8]
            [if s7
              [if (SdR pair cCoIDivSatCoIClAuxOne u2 u1)
               ([s3,u4]s3 pair(InR str str)u4)]
              [if s5
               [if (SdR pair cCoIDivSatCoIClAuxOne u2 u1)
                ([s3,u4]s3 pair(InR str str)u4)]
               [if s3
                [if (SdR pair cCoIDivSatCoIClAuxOne u2 u1)
                 ([s3,u4]s3 pair(InR str str)u4)]
```

```
                [if (SdM pair cCoIToCoIDouble u2)
                 ([s3,u4]s3 pair(InR str str)u4)]
                [if (SdL pair cCoIDivSatCoIClAuxFour u2 u1)
                 ([s3,u4]s3 pair(InR str str)u4)]]
               [if (SdL pair cCoIDivSatCoIClAuxFour u2 u1)
                ([s3,u4]s3 pair(InR str str)u4)]]
              [if (SdL pair cCoIDivSatCoIClAuxFour u2 u1)
               ([s3,u4]s3 pair(InR str str)u4)]])])])])
```

Der Term cCoITripleClosure ist der rechnerische Gehalt des Lemmas

$$\forall^{nc}_{x}.{}^{co}\mathbf{I}x \to \exists^{r}_{d_1,d_2,d_3,\tilde{x}}.{}^{co}\mathbf{I}\tilde{x} \wedge^{d} \mathbf{Sd}\ d_1 \wedge^{d} \mathbf{Sd}\ d_2 \wedge^{d} \mathbf{Sd}\ d_3 \wedge^{l} x = \frac{4d_1 + 2d_2 + d_3 + \tilde{x}}{8}.$$

Dieses wird einfach durch dreifache Anwendung von ${}^{co}\mathbf{I}^{-}$ bewiesen. Der rechnerische Gehalt ist daher

```
[u0]
 [if (DesYprod u0)
   ([s1,u2]
    [if (DesYprod u2)
      ([s3,u4][if (DesYprod u4)
                ([s5,u6]s5 pair s3 pair s1 pair u6)])])]
```

Im Grunde erhalten wir durch diesen aus einem Strom $d_1 :: d_2 :: d_3 :: u$ das Tupel $\langle d_1, d_2, d_3, u\rangle$.
Dieses Tupel wird im extrahierten Term unseres Divisionssatzes dann durch die ersten drei Caseoperatoren zerlegt. Es folgt dann eine Fallunterscheidung in sieben Fällen. In unserem Beweis waren das die Fälle $x = 1d_2d_3\tilde{x}$, $x = 01d_3\tilde{x}$, $x = 000\tilde{x}$, $x = 001\tilde{x}$, $x = \overline{1}d_2d_3\tilde{x}$, $x = 0\overline{1}d_3\tilde{x}$ und $x = 00\overline{1}\tilde{x}$, welche wir in drei Gruppen unterteilt haben. In den ersten drei Fällen haben wir $d = 1$ und $x' = 4\frac{x+\frac{-y}{2}}{2}$ gesetzt. Im extrahierten Term steht dafür jeweils (SdR pair cCoIDivSatCoIClAuxOne u2 u1). Dabei ist cCoIDivSatCoIClAuxOne der extrahierte Term für die Aussage, dass ${}^{co}\mathbf{I}x'$ gilt. Dieser ist durch die Programmkonstante AuxR : str → str → str mit der Berechnungsregel

$$\mathtt{AuxR}\ u\ v := \mathtt{D}(\mathtt{D}(\mathtt{Av}\ u\ (\mathtt{h}(\mathtt{n}v))))$$

gegeben. Dabei sind D, h und n aus den vorherigen Bemerkungen 3.3.17 und 3.3.19 und Av ist der rechnerische Gehalt von Satz 3.3.9.
Die letzten drei Fälle wurden ähnlich abgehandelt. Hier haben wir $d = \overline{1}$ und $x' = 4\frac{x+\frac{y}{2}}{2}$ gesetzt. Im extrahierten Term entspricht dies dem Auftreten von (SdL pair cCoIDivSatCoIClAuxFour u2 u1), wobei cCoIDivSatCoIClAuxFour wieder der rechnerische Gehalt davon ist, dass ${}^{co}\mathbf{I}x'$ in diesem Fall gilt. Diesen können wir durch die Programmkonstante AuxL : str → str → str mit der Berechnungsregel

$$\mathtt{AuxL}\ u\ v := \mathtt{D}(\mathtt{D}(\mathtt{Av}\ u\ (\mathtt{h}v)))$$

angeben.
Im vierten Fall, also wenn $x = 000\tilde{x}$ ist, haben wir $d = 0$ und $x' = 2x$ gesetzt. Dafür haben wir (SdM pair cCoIToCoIDouble u2) im extrahierten Term. Dabei ist

cCoIToCoIDouble der rechnerische Gehalt aus Lemma 3.3.16 also die Verdopplungsfunktion D. Wir können daher nun den extrahierten Term Div des obigen Satzes auch wie folgt angeben:

$$\texttt{Div}\ u\ v := \begin{cases} 1 :: (\texttt{Div}\ (\texttt{AuxR}\ u\ v)\ v) & \text{falls } u = 1\tilde{u} \vee u = 01\tilde{u} \vee u = 001\tilde{u} \\ 0 :: (\texttt{Div}\ (\texttt{D}u)\ v) & \text{falls } u = 000\tilde{u} \\ \overline{1} :: (\texttt{Div}\ (\texttt{AuxL}\ u\ v)\ v) & \text{falls } u = \overline{1}\tilde{u} \vee u = 0\overline{1}\tilde{u} \vee u = 00\overline{1}\tilde{u} \end{cases}$$

Bemerkung 3.3.22. Anhand der obigen Darstellung des extrahierten Terms überlegen wir uns nun noch heuristisch, wie viel Ziffern man von x und y kennen muss, um die ersten n Ziffern von $\frac{x}{y}$ zu bestimmen. Dazu schreiben wir obige Regel von Div in folgende Form:

$$\texttt{Div}\ u\ v = \texttt{d}(u) :: (\texttt{Div}\ \texttt{G}(u,v)\ v)$$

Dabei ist $\texttt{d}(u) \in \{\overline{1}, 0, 1\}$ von den ersten drei Ziffern abhängig und für $\texttt{G}(u,v)$ setzt man je nach Fall entweder AuxR u v, Du oder AuxL u v ein.
Wie man anhand der Bemerkungen 3.3.15, 3.3.17 und 3.3.19 sieht, benötigt man, um die ersten n Einträge von $\texttt{D}(u)$ zu bestimmen, die ersten $n+1$ Einträge von u. Um die ersten n Einträge von $\texttt{n}(u)$ zu bestimmen, benötigt man die ersten n Einträge von u. Um die ersten n Einträge von $\texttt{h}(u)$ zu bestimmen, benötigt man die ersten $n-1$ Einträge von u.
Aus Beispiel 3.3.11 wissen wir, dass man die ersten n Einträge von Av u v aus den ersten $n+1$ Einträgen von u und v bestimmen kann.
Insgesamt braucht man für die ersten n Einträge von AuxR u v und AuxL u v die ersten $n+3$ Einträge von u und die ersten $n+2$ Einträge von v. Für G gilt damit, dass man die ersten n Einräge von G(u,v) aus den ersten höchstens $n+3$ Einträgen von u und den ersten höchstens $n+2$ Einträgen von v bestimmen kann. Verwenden wir die obige Darstellung von Div mehrmals hintereinander, so haben wir:

$$\texttt{Div}\ u\ v = \texttt{d}(u) :: \texttt{d}(\texttt{G}(u,v)) :: \texttt{d}(\texttt{G}(\texttt{G}(u,v),v)) :: \texttt{d}(\texttt{G}(\texttt{G}(\texttt{G}(u,v),v),v))\ldots$$

Wollen wir also den n-te Eintrag von Div u v, so hängt dieser von den ersten drei Einträgen von $\texttt{G}^{n-1}(u,v)$ ab und diese erhalten wir aus den ersten höchstens $3n$ Einträgen von u und den höchstens ersten $2n+1$ Einträgen von v.

Korollar 3.3.23. Den Divisionsalgorithmus können wir nun auf $\frac{x}{y}$ verallgemeinern für $x = k + x'$ und $y = l + y'$, wobei k und l ganze Zahlen sind und ${}^{co}\mathbf{I}x'$ und ${}^{co}\mathbf{I}y'$ gilt. Dabei stellen wir uns die ganzen Zahlen auch in ihrer SD-Darstellung vor. Damit können wir x bzw y als Gleitkommazahl $d_n d_{n-1} \ldots d_0, d_{-1} d_{-2} \ldots$ darstellen. Für eine ganze Zahl i ist eine Multiplikation von 2^i mit x bzw. y dann einfach nur ein Shift um i Stellen nach links. Bezeichnet j die Anzahl der Ziffern von l, dann haben wir $\frac{x}{y} = \frac{2^j x}{2^j y}$. Wir können also annehmen, dass $l = 0$ und damit ${}^{co}\mathbf{I}y$ gilt.
Um Satz 3.3.20 anzuwenden, müssen wir $\frac{1}{4} \leq y$ erreichen. Dazu führen wir folgende Schleife aus, in der wir nacheinander die einzelnen Ziffern von y auslesen:
Ist die erste Ziffer von y eine $\overline{1}$, wissen wir $y \leq 0$. Daher multiplizieren wir $x = k + x'$ und y mit -1 und beginnen diese Schleife mit den neu erhaltenen Ziffern von vorne. Wegen Lemma 3.3.18 bleibt durch die Veränderung ${}^{co}\mathbf{I}x'$ und ${}^{co}\mathbf{I}y$ erhalten und der Wert von $\frac{x}{y}$ ändert sich auch nicht. Wir haben damit $0 \leq y$ erreicht.
Ist die erste Ziffer von y eine 0 oder sind $1\overline{1}$ die ersten beiden Ziffern von y , so wissen wir $|y| \leq \frac{1}{2}$. Wir multiplizieren dann x und y mit 2. Wegen Lemma 3.3.16

Auf der beiliegenden CD befindet sich eine kommentierte Implementierung von dem Algorithmus in Haskell. Diese ist in Zusammenarbeit mit Quirin Schroll entstanden, dem an dieser Stelle dafür herzlich gedankt sei.

bleibt ${}^{co}\mathbf{I}\, y$ erhalten und die Darstellung von x ändert sich ohnehin nicht, da wir hier shiften können. Auch der Wert von $\frac{x}{y}$ wird nicht geändert.
In jedem anderen Fall, also wenn 11 oder 10 die ersten beiden Ziffern von y sind, gilt $\frac{1}{4} \leq y$. Mit diesem Wissen beenden wir die Schleife und fahren fort.
Es ist klar, dass diese Schleife nicht immer terminiert. Da aber die Division beispielsweise für $y = 0$ auch nicht definiert ist, ist das nicht weiter verwunderlich.
Haben wir nun $\frac{1}{4} \leq y$ und ${}^{co}\mathbf{I}\, y$ erreicht, schreiben wir

$$\frac{x}{y} = \frac{k + x'}{y} = 4k\frac{\frac{1}{4}}{y} + 4\frac{\frac{x'}{4}}{y}$$

Die beiden hier vorkommenden Divisionen können nun mit unserem Divisionsalgorithmus von Satz 3.3.20 durchgeführt werden. Schreiben wir nun k wieder in SD-Darstellung: $k = \sum_{i=0}^{m} d_i 2^i$, dann haben wir

$$\frac{x}{y} = \sum_{i=0}^{m} 2^{i+2} d_i \frac{\frac{1}{4}}{y} + 4\frac{\frac{x'}{4}}{y}$$

Wegen $d_i \in \{\overline{1}, 0, 1\}$ gilt sogar ${}^{co}\mathbf{I}\left(d_i \frac{\frac{1}{4}}{y}\right)$. Wie wir mit Potenzen von 2 multiplizieren, indem wir das Komma shiften, haben wir uns weiter oben schon überlegt und einen Additionsalgorithmus haben wir nach Korollar 3.3.12. Damit hat auch $\frac{x}{y}$ die Darstellung $\frac{x}{y} = i + z$ mit einer ganzen Zahl i und ${}^{co}\mathbf{I}\, z$. Das gilt natürlich nur, sofern der eben angegebene Algorithmus terminiert. Wie man sich überlegen kann, terminiert dieser genau dann, wenn $y \in \mathbb{R}^+$ ist.

Stichwortverzeichnis

Minlog-Befehle

Literaturverzeichnis

[1] Josef Berger. Logik. http://www.mathematik.uni-muenchen.de/~jberger/logik.pdf, 2017. [Online; aufgerufen am 01.09.2017].

[2] Errett Bishop. *Constructive Analysis*. Springer-Verlag Berlin Heidelberg, 1985.

[3] Alberto Ciaffaglione. Certified reasoning on real numbers and objects in co-inductive type theory. http://users.dimi.uniud.it/~alberto.ciaffaglione//Papers/phd-thesis03.pdf, 2003. [Online; aufgerufen am 06.06.2017].

[4] Kenji Miyamoto. Program extraction in exact real arithmetic. http://www.mathematik.uni-muenchen.de/~schwicht/papers/tucker12/average13.pdf, 2013. [Online; aufgerufen am 01.09.2017].

[5] Kenji Miyamoto. The Minlog System. http://www.mathematik.uni-muenchen.de/~logik/minlog/index.php, 2017. [Online; aufgerufen am 30.07.2017].

[6] Helmut Schwichtenberg. Mathematical Logic. http://www.mathematik.uni-muenchen.de/~schwicht/lectures/logic/ss10/ml.pdf, 2010. [Online; aufgerufen am 01.09.2017].

[7] Helmut Schwichtenberg. *Proofs and Computations*. Cambridge University Press, Dezember 2011.

[8] Helmut Schwichtenberg. Logic for exact real numbers. http://www.mathematik.uni-muenchen.de/~schwicht/papers/bridges16/m4c.pdf, 2016. [Online; aufgerufen am 06.06.2017].

[9] Helmut Schwichtenberg. Logik 2. http://www.mathematik.uni-muenchen.de/~schwicht/lectures/logic/ss16/ml.pdf, 2016. [Online; aufgerufen am 23.12.2016].

[10] Till Überrück Fries. Program extraction for exact real numbers: Stream multiplication. http://www.mathematik.uni-muenchen.de/~schwicht/seminars/semws16/Thesis.pdf, 2016. [Online; aufgerufen am 01.09.2017].

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst habe. Ich versichere, dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahren gewesen ist.

München, 20. September 2017
Ort, Datum

Franziskus Wiesnet