Minlogeinführung

Franziskus Wiesnet

30. April 2017

Vorbemerkungen

Dieses Skript dient der Einführung in Minlog für die Besucher der Vorlesung *Mathematische Logik II* bei Prof. Helmut Schiwchtenberg und ist auch in Zusammenarbeit mit ihm entstanden. Es handelt sich dabei um eine vorläufige Version der ersten beiden Kapitel meiner Masterarbeit. Aus diesem Grund wird es auch häufiger Änderungen geben.

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 [3], welches 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 Regel für $\rightarrow$ und $\forall$:

Annahmeregel: Ist A eine Formel, so ist

$$u: A$$

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

$\rightarrow^-$-Regel: Sind A und B Formel, M eine Herleiung 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. Ein 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 diese 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 Annahmemenge Γ und N eine Herleitung von B mit Annahmemenge Δ, dann ist

$$\frac{\begin{array}{cc} |M & |N \\ A & B \end{array}}{A \wedge B} \wedge^+$$

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

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

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 weiter Formel, dann ist

$$\frac{\begin{array}{c} |M \\ A \end{array}}{A \vee B} \vee_0^+$$

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

$$\frac{\begin{array}{c} |M \\ A \end{array}}{B \vee A} \vee_1^+$$

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

$$\frac{\begin{array}{ccc} |M & |N & |L \\ A \vee B & C & C \end{array}}{C} \vee^-$$

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

$$\frac{\begin{array}{cc} & |M \\ r & A(r) \end{array}}{\exists_x A(x)} \exists^+$$

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 Variablenbedienungen $x \notin FV(\Delta \setminus \{(u:A)\})$ 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 Disjuktion $\tilde{\vee}$ 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 $\vdash \exists_x A \to \tilde{\exists}_x A$:

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

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

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. Diese kann man mit Hilfe von Gegenmodellen zeigen, welche aber nicht Gegenstand dieser Arbeit sind. Bei Interesse sei aber auf Kapitel 1.3 von [3] verwiesen.

Motivation 1.1.9. Im natürlichen Schließen hat das Falsum $\bot$ keine weitere Bedeutung, sondern ist nur eine Prädikatenvariable. Insbesondere überlege man sich, dass das Ex falso quodlibet, das heißt $\bot \to A$, nicht für jede Formel A gilt. 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 dem 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 Konstruktortypen 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 $\vec{\alpha}$. Wir fordern zusätzlich, dass κ_0 die Form $\kappa_0 = \vec{\rho} \to \xi$ hat.

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 dieses hochgestellt dazu: $C_i^{\kappa_i(\iota)}$
Einige Konstruktoren haben auch standardisierte Namen. Zum Beispiel wird der einzige Konstruktor von $\mathbb{U}$ mit $\mathbf{u}$ 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 $\mathrm{nil}^{\mathbb{L}(\alpha)}$ den Konstruktor der leeren Liste und den zweistelligen Konstruktor bezeichnet man entweder mit $\mathrm{cons}^{\alpha\to\mathbb{L}(\alpha)\to\mathbb{L}(\alpha)}$ oder als Infixnotation mit $\_ :: \_$. Für den Summentyp $\alpha + \beta$ werden die beiden Konstruktoren durch $\mathrm{inl}^{\alpha\to\alpha+\beta}$ und $\mathrm{inr}^{\beta\to\alpha+\beta}$ 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 einer 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 $\mapsto_\beta$ sowie die Relation η-Konversion $\mapsto_\eta$ zwischen Terme vom selben Typ durch: $\lambda_x M(x) N \mapsto_\beta M(N)$ und, ist $x \notin FV(M)$, dann gilt $\lambda_x(Mx) \mapsto_\eta M$.
Mit $\mapsto$ 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 $\vec{M} = (M_j^{\delta_j})_{j<n}$, 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 $\vec{x}$ genauso so viele Komponenten mit entsprechende 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

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

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, so 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 Konstruktorentypen 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 Schrittypen $\delta_0 = \tau$ 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 Konstruktertyp $\kappa_0 = \alpha \to \beta \to \xi$ 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:

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

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 natürlichen Zahlen n und m definiert man

$$m+0 := m$$
$$m+(Sn) := S(m+n).$$

Mit dem Rekursionsoperator definieren wir daher

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

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 $\mapsto$ 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:

$$G(0) := \mathrm{tt}$$
$$G(SN) := \neg G(N)$$

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 natürlichen Zahlen direkt als Term zu betrachten, der die obigen vier Konversionsregeln erfüllt. Aus diesem Grund führen wir nun Programmkonstanten ein, zu denen man auch den bereits definierten Rekursionsoperator zählen kann. 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 für alle $i \neq j$ $FV(P_i) \cap FV(P_j) = \varnothing$, 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 „Programmkostante“, 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 Konsistenzbedigung: 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 Rekurionsoperator, 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:

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

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.
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

$$\mathsf{tt}\ \mathtt{andb}\ y := y$$
$$\mathsf{ff}\ \mathtt{andb}\ y := \mathsf{ff}$$
$$x\ \mathtt{andb}\ \mathsf{tt} := x$$
$$x\ \mathtt{andb}\ \mathsf{ff} := \mathsf{ff}.$$

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 $\doteq$ ergeführt. Damit haben wir nun eine Sprache, in der wir Programmextraktion aus Beweisen betreiben können. Was uns noch fehlt sind die Formel 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 Beiweisterm 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 fehlt lediglich die Regeln, die den Rekursionsoperratoren 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 $\vec{Y}$ bezeichen. 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 $\forall_x B$ eine 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. Wir fordern, dass in einer Prädikatenform der Form $\mu_X(K_0,\dots,K_{n-1})$ die Klauselform K_0 keine Rekursionsprämissen hat. 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 gebundene 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 $\forall_{x^\rho}\mathrm{Eq}xx$. 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:

Eq ist eine Kurzschreibweise für Eq(ρ), wenn ρ klar oder egal ist.

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

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

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

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 $T^+ / \doteq$, den logischen Verknüpfungen und Regeln aus Definition 1.1.1 also nur für $\forall$ und $\to$ und die Axiome dieser Theorie sind die Einführungs- und Eliminationaxiome aus Definition 1.4.2.

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

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

Man beachte, dass A keine Prädikatenparameter haben darf.

Beweis. Wir beweisen zunächst, wenn x^ρ, y^ρ Terme vom selben Typ sind, so gilt $\mathbf{TCF} \vdash \mathbf{F} \to \mathsf{Eq}xy$: Nach Einführingsaxiom für Eq gilt $\mathsf{Eq}(\mathscr{C}^\rho_{\mathbb{B}}\mathsf{tt}xy)(\mathscr{C}^\rho_{\mathbb{B}}\mathsf{tt}xy)$. Damit gilt nach Lemma 1.4.4 $\mathsf{Eq}(\mathscr{C}^\rho_{\mathbb{B}}\mathsf{ff}xy)(\mathscr{C}^\rho_{\mathbb{B}}\mathsf{tt}xy)$ also $\mathsf{Eq}xy$.
Nun beweisen wir die Aussage $\mathbf{TCF} \vdash \mathbf{F} \to A$ 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 keine Rekurionsprämissen und damit die Form $\forall_x(\vec{B} \to I\vec{t})$. Nach Induktionshypothese folgen aus $\mathbf{F}$ alle $\vec{B}$ und damit $I\vec{t}$. Zusammen mit Lemma 1.4.4 und der gerade gezeigten Gleichheit aller Terme erhalten wir also $I\vec{s}$. Ist $A = B \to C$ oder $A = \forall_x C$, gilt nach Induktionsvoraussetzung $\mathbf{TCF} \vdash \mathbf{F} \to C$ und damit sicher $\mathbf{TCF} \vdash \mathbf{F} \to B \to C$ und weil in **TCF** keine Axiome mit freien Variablen sind, folgt auch $\mathbf{F} \to \forall_x C$. □

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 $\mathsf{Eq}(s, \mathsf{tt})$ auf.

Beispiel 1.4.9. Wir haben in **TCF** nur die logischen 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 $\exists_x A := Ex(\{x^\rho | A\})$ setzten. 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. Man kann leicht 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 definieren Prädikaten haben wir uns für diese eine Regel entschieden.
Für die Disjuktion 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 Anfängervorlesung ist sicherlich noch bekannt, dass man Aussagen über natürlichen 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 äquivalent zum 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 [3] 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 $\mathbf{T}_{\mathbb{N}}0$ und ist n total, so soll auch Sn total sein, also $\forall_n(\mathbf{T}_{\mathbb{N}}n \to \mathbf{T}_{\mathbb{N}}Sn)$. Damit ist $\mathbf{T}_{\mathbb{N}} = \mu_X(X0, \forall_n(Xn \to XSn))$ und wir haben als Eliminationsaxiom

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

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.

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

und

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

Für die strukturelle Totalität definieren wir

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

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 Variable 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. Dass Ersteres nicht gilt, sieht man anhand des Listentyps aus Beispiel 1.5.2. Ein Gegenbeispiel für die zweite Aussage ist 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 total aber nicht strukturell total.

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.

Lemma 1.5.6. Es bezeichne andb die in Definition 1.2.18 gegebene Programmkonstante auf der booleschen Algebra, dann gilt $\forall_{a^{\mathbb{B}},b^{\mathbb{B}}}.\mathbf{G}_{\mathbb{B}}a \to \mathbf{G}_{\mathbb{B}}b \to a\ \mathtt{andb}\ b \to a \wedge b$ und $\forall_{a^{\mathbb{B}},b^{\mathbb{B}}}.\mathbf{G}_{\mathbb{B}}a \to \mathbf{G}_{\mathbb{B}}b \to \mathbf{G}_{\mathbb{B}}\ (a\ \mathtt{andb}\ b)$.

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

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

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

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

Bemerkung 1.5.7. 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 1.5.8. Es sei ι eine finitäre Algebra, dann gilt $\forall_{x^\iota,y^\iota}.\mathbf{G}_\iota x \to \mathbf{G}_\iota y \to \mathbf{G}_{\mathbb{B}} x = y$.

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

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

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

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

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.

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

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

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

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

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

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

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 1.5.6 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 1.5.9. 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 $\iota =: \mu_\xi(\kappa_0,\dots,\kappa_{k-1})$, 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

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

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

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

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 setzten 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 uns

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

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

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

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

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

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 1.5.6 $\vec{u}_l = \vec{x}_l$ 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 $\to$ und $\forall$ 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 zum 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 Beweisen 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 Gehalten 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 rechnerischen 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 $\to$ 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 $\forall^{nc}_x A$ 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 gilt 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, schriebt man für $\to$ auch $\to^c$ und analog schreibt man $\forall^c$ für $\forall$. Ist beides möglich, schreiben wir $\to^{c/nc}$ bzw. $\forall^{c/nc}$.

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 Implikationspfeils 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 $\forall_x^{nc} B$ eine Formelformen mit Parameter $\vec{Y}$.

- Sind $\vec{A}$ Formelformen mit Parameter $\vec{Y}$ und sind $\vec{B_0},\ldots,\vec{B_{n-1}}$ 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 $K_0,\ldots,K_{k-1}$ Klauselformen, dann ist $\mu_X^{nc}(K_0,\ldots,K_{k-1})$ ein Prädikat .

Man beachte, dass jede Rekursionsprämisse rechnerisch eingeht.

Ist $\mu_X(K_0,\ldots,K_{k-1})$ 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 $I^{nc}:=\mu_X^{nc}(K_0,\ldots,K_{k-1})$, sind die Einführungsaxiome und das Eliminationsaxiom genauso wie bei rechnerischen induktive 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 angebenden 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, so 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 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 $\mathbf{G}$ bzw. $\mathbf{T}$ 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 $I = \mu_X(K_0,\dots,K_{k-1})$ ein induktiv definiertes Prädikat, so ist

$$\tau(I) := \mu_\xi(\tau(K_0),\dots,\tau(K_{k-1})).$$

- 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 $\{\vec{x}|A\}$ eine Prädikat mit einer Formel A, dann ist $\tau(\{\vec{x}|A\}) := \tau(A)$.
- Sind A und B Formel, 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 Variable 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 Variable 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 dieses 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, sodass es hier auch passt, dass wir nur den Einheitstyp als rechnerischen Gehalt haben.
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 Disjuktion 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, das wir in jedem Fall die Information bekommen, welche Seite der Disjuktion 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 Annahmevariablen 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 nicht-rechnerischen Aussage A, dann setzten 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 $I = \mu_X(K_0,\dots,K_{k-1})$ 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 $\exists^l_x Eq(\mathbb{N})(x,1)$. Eine Herleitung dieser Aussage sieht wie folgt aus:

$$\left(\left(Ex^l\left(\{x|Eq(x,1)\}\right)_0^+\right)^{\forall^c_x(Eq(x,1)\to^{nc}\exists^l_x Eq(x,1))} 1\right)^{Eq(1,1)\to^{nc}\exists_x Eq(x,1)} \left(\left(Eq_0^+\right)^{\forall^{nc}_x Eq(x,x)} 1\right)^{Eq(1,1)}$$

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

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

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 als extrahierten Term 1. 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 $(1,\mathbf{u})$, 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

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

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 haben 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 Gewünscht ist. Um zu beantworten, was das Gewünschte ist, definieren wir das Realisierungsprädikat einer Aussage. Wir haben uns schon am Beispiel des Existenzquantor überlegt, das ein Term t eine Aussage der Form $\exists^l_x A(x)$, dann realisieren sollte, wenn $A(t)$ gilt. Er ist damit im übertragenen Sinne ein Zeuge für die Aussage $\exists^l_x A(x)$.

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

$$t\ \mathbf{r}\ (A\to^c B) := \forall^{nc}_x(x\ \mathbf{r}\ A\to^{nc} tx\ \mathbf{r}\ B),$$
$$t\ \mathbf{r}\ (A\to^{nc} B) := \forall^{nc}_x(x\ \mathbf{r}\ A\to^{nc} t\ \mathbf{r}\ B),$$
$$t\ \mathbf{r}\ \forall^c_x A := \forall^{nc}_x(tx\ \mathbf{r}\ A),$$
$$t\ \mathbf{r}\ \forall^{nc}_x A := \forall^{nc}_x(t\ \mathbf{r}\ A).$$

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

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

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

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

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

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 $I^{\mathbf{r}} := \mu^{nc}_X(K_0^{\mathbf{r}}, \ldots, K_{k-1}^{\mathbf{r}})$ ist dann das Zeugenprädikat von I.

Beispiel 1.8.7. Betrachten wir als Beispiel zunächst die Zeugenprädikate für einige induktiv definierten 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:

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

In jedem dieser Fälle 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 ein 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 $(u_i : B_i)_{i<n}$. Dann gilt $\mathbf{TCF} \cup \{x_{u_i}\ \mathbf{r}\ B_i | i < n\} \vdash et(M)\ \mathbf{r}\ A$.

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 $et(M) = \lambda_{x_u} et(N)$ und wir haben die Aussage $\lambda_{x_u} et(N)\ \mathbf{r}\ A \to^c B$ zu zeigen. Diese ist Äquivalent zu $\forall_x^{nc}(x\ \mathbf{r}\ A \to^{nc} \lambda_{x_u} et(N) x\ \mathbf{r}\ B)$ und es ist $\lambda_{x_u} et(N) x = et(N)[x_n := x]$. Nach Umbenennen der lokalen Variablen x zu x_u haben wir dann einfach $\forall_{x_u}^{nc}(x_u\ \mathbf{r}\ A \to^{nc} et(N)\ \mathbf{r}\ B)$. Nach Induktionshypothese haben wir, dass $et(N)\ \mathbf{r}\ B$ gilt unter der zusätzlichen Annahme $x_u\ \mathbf{r}\ A$ und, weil das Realisierungsprädikat nicht rechnerisch ist, folgt die zu zeigende Aussage.

$M = (\lambda_x N^A)^{\forall_x^c A}$**:** Hier ist $et(M) = \lambda_x et(N)$ und die Aussage $\forall_x^{nc}(et(N)\ \mathbf{r}\ A)$ 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 $\forall_x^{nc}(et(N)\ \mathbf{r}\ A)$.

$M = N^{A \to^c B} L^A$**:** 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 $\forall_x^{nc}(x\ \mathbf{r}\ A \to^{nc} et(N)x\ \mathbf{r}\ B)$ 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.

$M = N^{\forall_x^c A(x)} t$**:** Gefragt ist nun eine Herleitung von $et(Nt)\ \mathbf{r}\ A(t)$. Nach der Induktionshypothese haben wir eine Herleitung von $et(N)\ \mathbf{r}\ \forall_x^c A(x)$, dies ist äquivalent zu $\forall_x^{nc}(et(N)x\ \mathbf{r}\ A(x))$. 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 $\forall_y^{nc}(y\ \mathbf{r}\ A \to^{nc} et(N)\ \mathbf{r}\ B)$. 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.

$M = (\lambda_x N^A)^{\forall_x^{nc} A}$**:** 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 $et(N)\ \mathbf{r}\ \forall_x^{nc} A$ und damit $et(\lambda_x N)\ \mathbf{r}\ \forall_x^{nc} A$ 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 $\forall_y^{nc}(y\ \mathbf{r}\ A \to et(N)\ \mathbf{r}\ A)$ sowie von $et(L)\ \mathbf{r}\ A$. Die gibt uns genau $et(N)\ \mathbf{r}\ B$.

$M = N^{\forall_x^{nc} A(x)} t$**:** 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 $et(N)\ \mathbf{r}\ \forall_x^{nc} A(x)$ also $\forall_x^{nc}(et(N)\ \mathbf{r}\ A(x))$. Setzten 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 $I = \mu_X(K_0, \ldots, K_{k-1})$ zu nächst 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 **r** $K_i(I)$ gilt. Die folgenden Zeilen sind äquivialent zueinander wegen der Definitionen des Realisierungsprädikats und der Tatsache, dass das Realisierungsprädikat nicht rechnerisch ist, weswegen die Dekorationen irrelevant sind:

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

Das ist wiederum ist äquivalent zu

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

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

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

ist. Es seien nun $\vec{x}$, w gegeben und es gelte w **r** $I\vec{x}$ also $I^{\mathbf{r}}w\vec{x}$. Weiter seien nun $\vec{w}$ gegeben und es gelte w_i **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 mehrfachen 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 $\mathscr{R}w\vec{w}$ **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 $\mathscr{R}w\vec{w}$ **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 $\vec{u}$ **r** $\vec{A}$, $(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j$ **r** $\vec{B}_j \to I^{\mathbf{r}} f_j \vec{y}_j \vec{v}_j \vec{s}_j)_{j<n}$ sowie $(\forall_{\vec{y}_j,\vec{v}_j}.\vec{v}_j$ **r** $\vec{B}_j \to Q f_j \vec{y}_j \vec{v}_j \vec{s}_j)_{j<n}$. Wir wollen $QC_i\vec{x}\vec{u}\vec{f}\vec{t}$ also $\mathscr{R}C_i\vec{x}\vec{u}\vec{f}\vec{w}$ **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 Prädikate 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 definierten. 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 ε **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 $\exists_{\vec{y}}(\vec{y}$ **r** $\vec{A})$. Damit folgt die obige Aussage aus dem Eliminationsaxiom von I, wobei wir das Prädikat $\{\vec{x}|z$ **r** $P\vec{x}\}$ in das Eliminationsaxiom einsetzten.
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 Beweisassitenten Minlog vermitteln. Minlog bassiert 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. Im Minlog-Tutorium [1] ist auch eine Einführung in Minlog gegeben. Einige Beispiel in diesem Kapitel sind auch hieraus entnommen. Man findet das Minlog-Tutorium ebenso im Ordner `doc` des Minlog-Verzeichnisses. Dort gibt es auch das Minlog-Referenz-Manual [2].

2.1 Grundlegende Befehle in Minlog

2.1.1 Deklaration von Prädikatenvariablen

Um Variablen aller Art, d.h. Typvariablen, Termvariablen sowie Prädikatenvariablen, 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 wollen 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 haben wir dafür in Beispiel 1.1.3 angegeben. Schreiben wir einen Beweis in Minlog, ist es nicht so, dass wir bei jeden Beweis einen Herleitungsbaum oder gar einen Herleitungsterm schreiben 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 Verfügen 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 Prämisse 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.

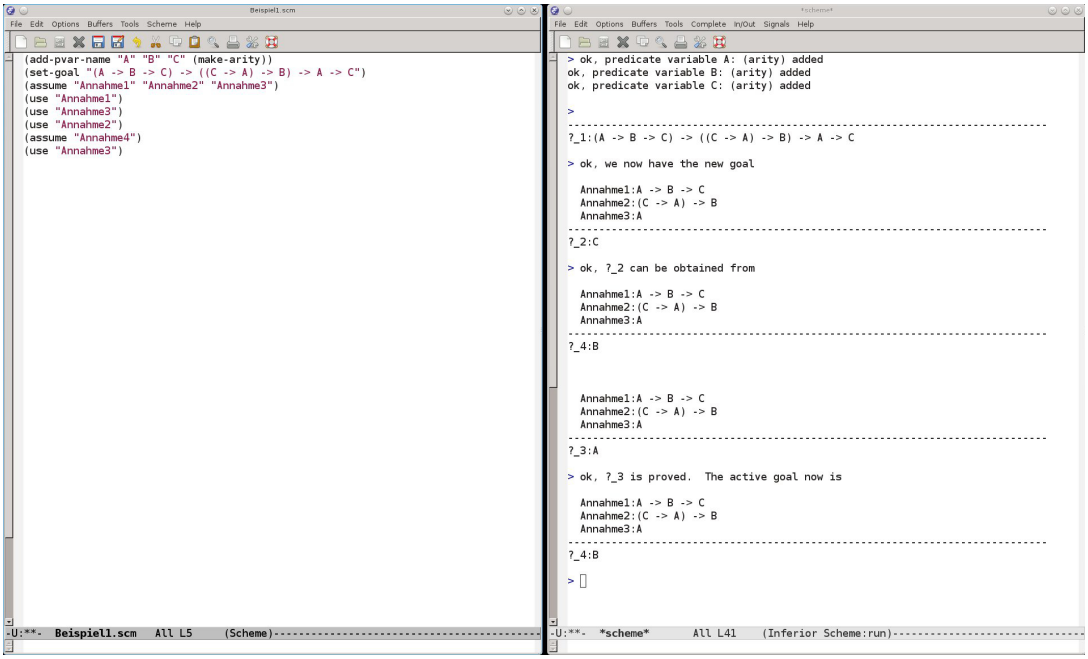
```
ok, ?_3 is proved.  The active goal now is

  Annahme1:A -> B -> C
```

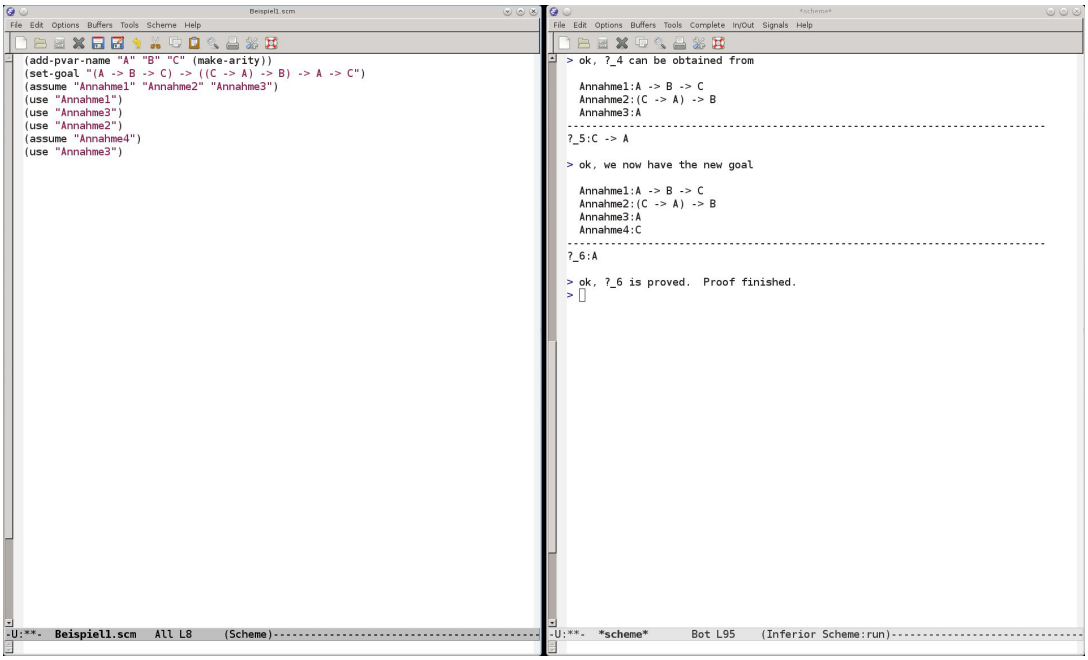
```
  Annahme2:(C -> A) -> B
  Annahme3:A
-------------------------------------------------------------------
?_4:B
```

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 folgedenen 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),
```

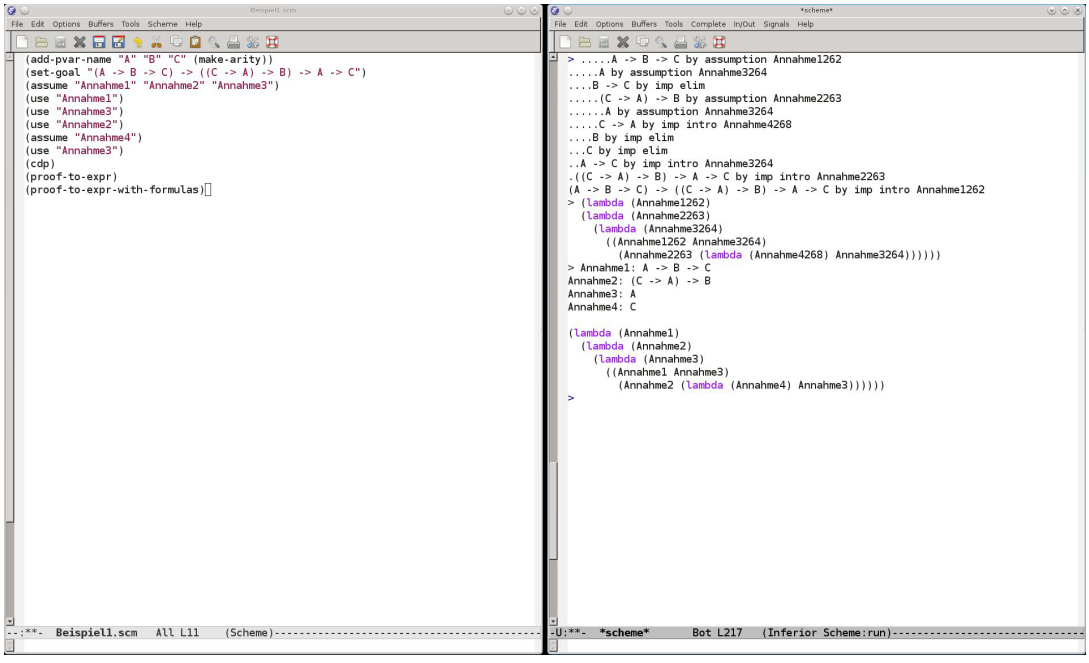
was eine Kurzform für `(check-and-display-proof)` ist. Hier wird der Beweis angezeigt und zusätzlich auf Korrektheit überprüft.
Wollen wir den Beweis als Herleitungsterm ausgeben lassen, so können wir den Befehl

```
(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 Variable 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.
Ein 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 List 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 $\forall_n(Pn \to Qn) \to \forall_x Pn \to \forall_n Qn$, 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 vielem anderen festgelegt, dass die Algebra der natürlichen Zahlen mit `nat` bezeichnet wird, und dass `n` und `m` Variablen von 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")
```

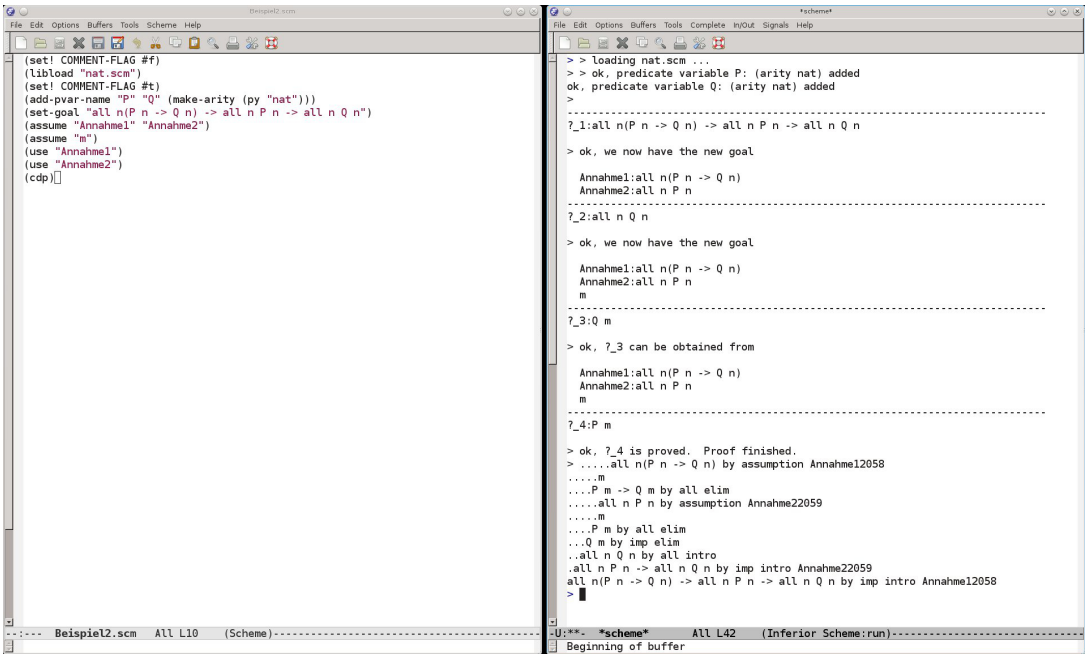
und das Programm gibt

```
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 könnten und hätten 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")`, 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 die 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 gleich 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 $\forall_n(Pn \to Qn) \to \forall_x Pn \to \forall_n Qn$ 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 die selben 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 Kapitel nun 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

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

> 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 ein 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 auf diese Weise 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 diese 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 angezeigt. Ist die Liste leer, das heißt schreiben wir nur (display-global-assumptions), so wird jede globale Annahme angezeigt. 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-Falsum 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 bemerken wir in einem Beweis, dass wir ein Teilziel dieses Beweises als globale Annahme setzten wollen. 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 auf 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 kann.
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 auf den natürlichen Zahlen gegeben ist:

Die Liste ist eigentlich viel Länger. Anstelle von den Punkten 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 induktive 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 allgemeinster 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 dafür 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 von 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 mir 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 sehen, wenn wir `(display-alg "list")` eingeben:

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

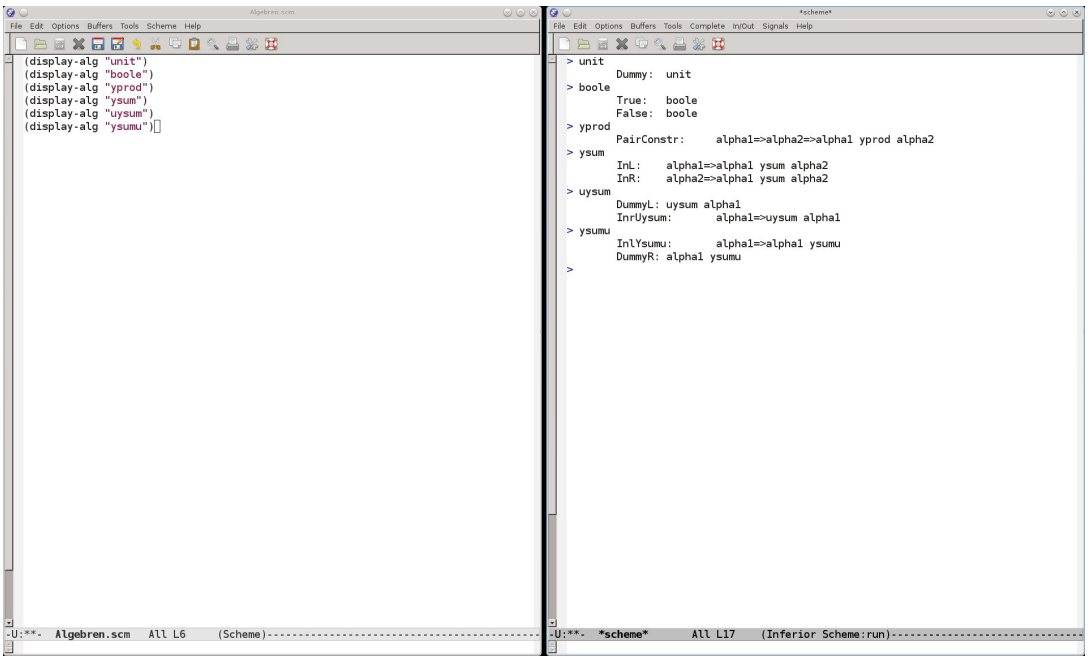
Dort wird der Parameter immer mitangegeben. Man beachte, dass der Parameter hinter dem Namen der Algebra steht. Möchte man lieber die Darstellung `alpha list` haben, so erreicht man dies mit dem hinzufügen von `'postfix-typeop` nach dem Namen der Algebra. Beim Listentyp sieht dies wie folgt aus:

```
(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 Namen 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. Dieses besitzt Argumente, deren Typen man 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 die selben 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 mit

```
(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. Mit der Definition des Eliminationsaxioms 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)))
```

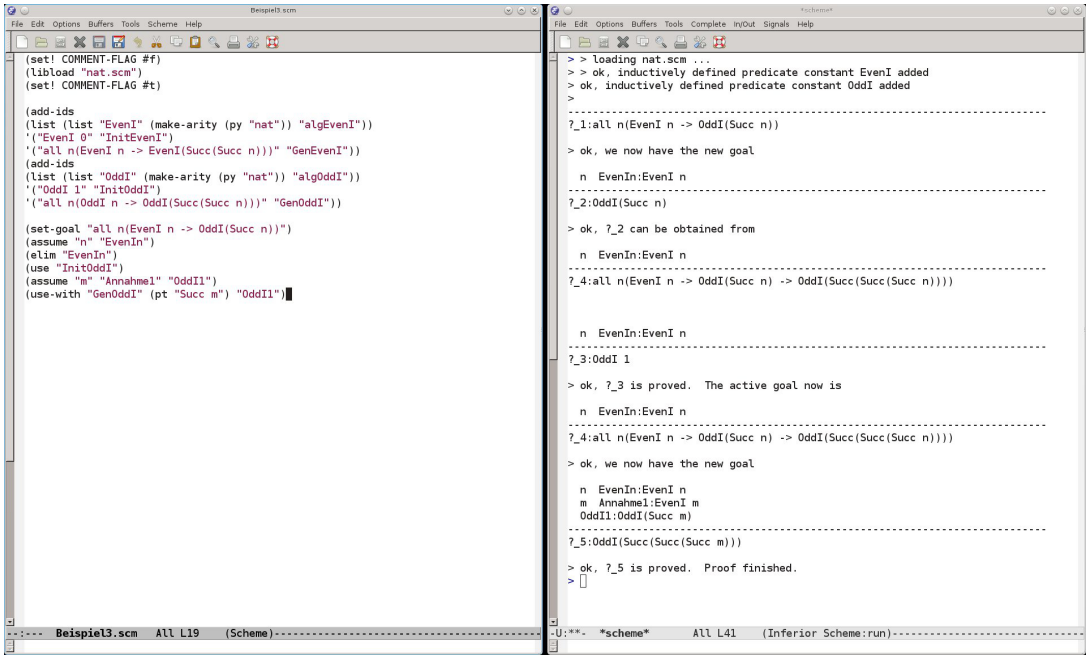
durch das zweite Einführungsaxiom vom `OddI` beweisen. Dieses ist gegeben durch

```
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 Einfühungsaxiom verwendet haben.

2.3 Dekorationen in Minlog

Beim Umgang mit den dekorierten logischen Symbolen ist sehr nützlich, einen Computer als Unterstützung zu verwenden. Denn, wie man in 1.6.2 sieht, muss bei der $\rightarrow^{nc}$- 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 daher 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 $\forall_n(Pn \rightarrow Qn) \rightarrow \forall_x Pn \rightarrow \forall_n Qn$ bewiesen. Jetzt möchten wir die dekorierte Form $\forall_n^{nc}(Pn \rightarrow Qn) \rightarrow \forall_x^{nc} Pn \rightarrow \forall_n^{nc} Qn$ 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")
```

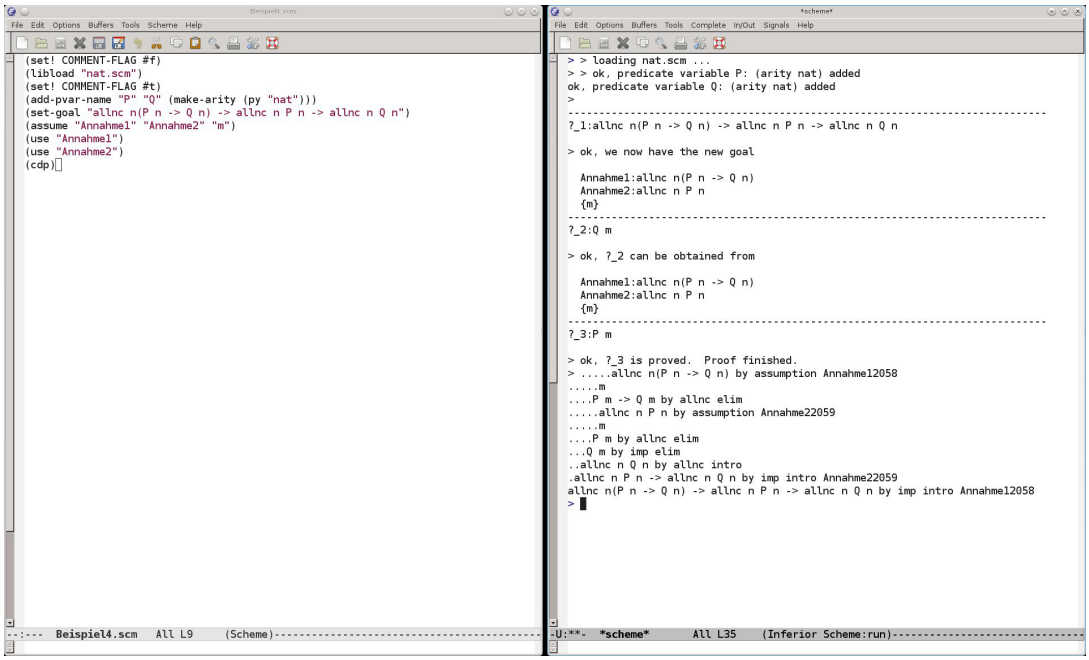
ein, um unsere Zielformel festzulegen. Die Ausgabe des Programms ist dann wie erwartet

```
------------------------------------------------------------------
?_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 fällt jedoch auf, dass um die Variable m im Kontext geschweiften 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 nochmal 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

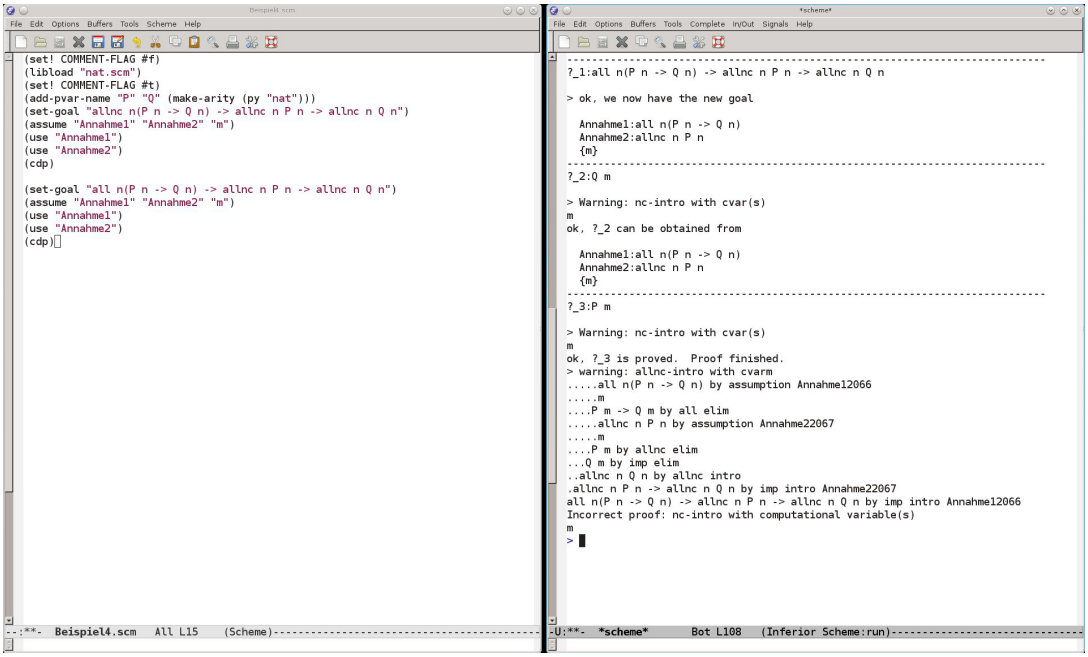
```
(set-goal "all n(P n -> Q n) -> allnc n P n -> allnc n Q n").
```

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 begründen, dass an dieser Stelle noch nicht klar ist, ob es sich wirklich um eine falsche Anwendung handelt. So würde man moglicherweise korrekte Beweise vorzeitig abbrechen. Außerdem gibt es von der logischen Struktur aus betrachtet es keinen Unterschied zwischen $\forall$ und $\forall^{nc}$. Insbesondere wäre der Beweis ohne Dekorationen natürlich korrekt. Beenden wir nun den „Beweis“ und verwenden wir `(cdp)`, so erhalten wir auch einen Beweisbaum, bekommen aber mitgeteilt, dass der „Beweis“ inkorrekt ist, weil wir eine nicht-rechnerische Alleinführung mit einer rechnerischen Variable 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. Wir werden später die vollständig dekorierte Aussage noch betrachten.

Der nicht-rechnerische Implikationspfeil wird in Minlog mit `-->` bezeichnet und besitzt die gleichen syntaktischen Regeln wie `->`. Als Anwednungsbeispiel 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.

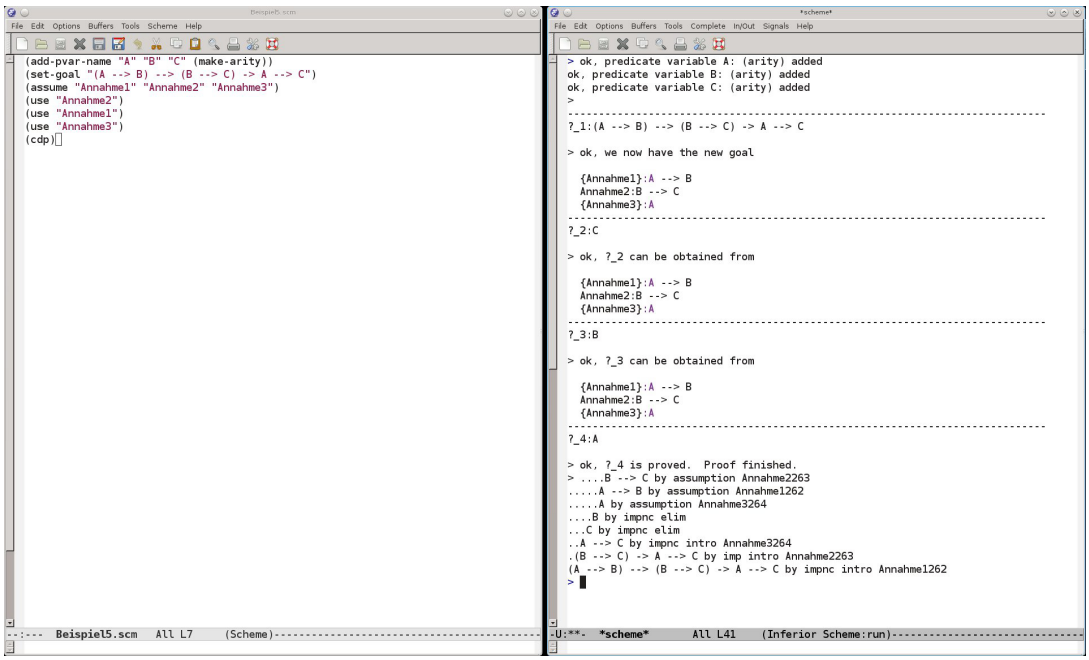
```
ok, we now have the new goal
```

```
  {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-recherischen 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, was 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, würden 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 einen 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 Prädikat 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, die 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 daher 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:)
```

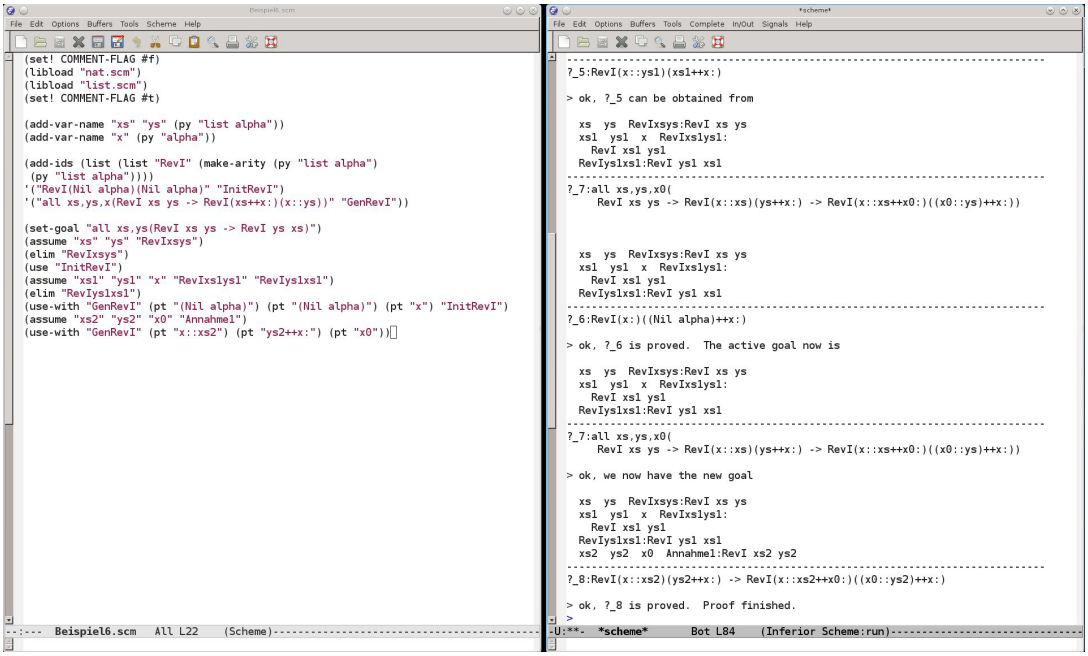
Wir wissen, dass `x:` und `(Nil alpha)++x:` die selben Terme in unserer Theorie sind. Das System erkennt dies auch. Nach `InitRevI` gilt `RevI(Nil alpha)(Nil alpha)` und mit `GenRevI` erhalten wir dann `RevI x: x:`. Minlog erkennt jedoch mit dem `use`-Befehl nicht sofort, wie `GenRevI` anzuwendenden 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 $f(n) = n \to f(n) = f(f(n))$. Wir geben also

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

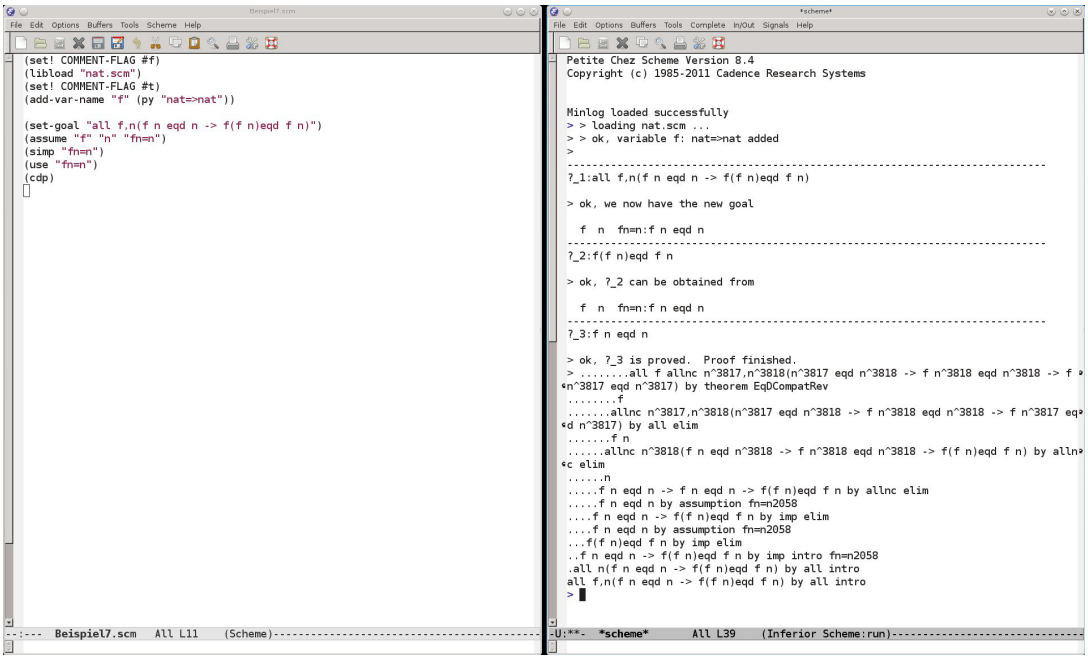
ein und nehmen mit `(assume "f" "n" "fn=n")` die Variablen und die Prämisse in den Kontext. Zu zeigen ist nun folgendes:

```
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 Disjuktion, 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 nicht-rechnerische 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 "Annahme1Inst")` gefolgt von `(use "Annahme2")` 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 Disjuktion gibt es keine besonderen Befehle. Die Einfühungsaxiome und das Eliminationsaxiom lassen auch jeweils leicht anwenden.
Von diesen drei induktiv definierten Prädikaten gibt es jeweils noch „interaktive“ Varianten `andi`, `exi` und `ori`. Dabei handelt es nicht um neue induktiv definierte Prädikate, sondern das System such jeweils die passende Variante aus. Wollen wir Beispielsweise die Aussage

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

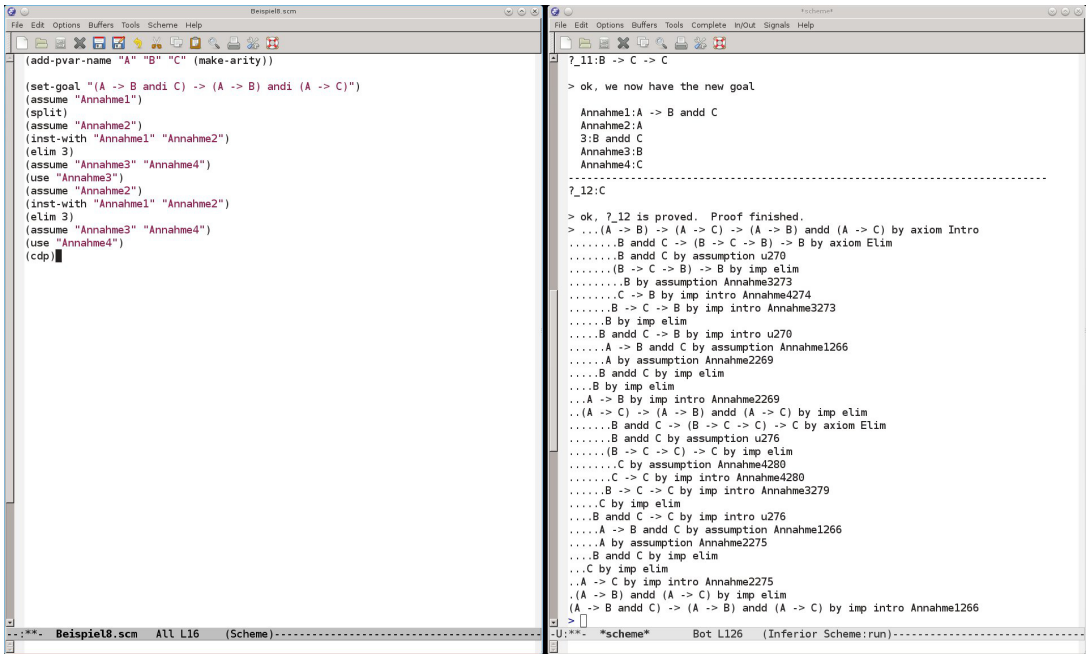
beweisen, so können wir das mit

```
(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 tun durch

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

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 nun 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 Konstruktermustern ist und M ein Term ist, in dem höchstens die Variablen frei sind, die auch in $\vec{P}$ frei sind. Für `TERM1` setzten wir $D\vec{P}$ ein und für `TERM2` setzten 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 etwas abkürzende Schreibweise. Der Befehl

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

bewirkt das selbe wie die beiden Befehle oben, ist nur etwas kürzer, da man `pt` weglassen kann und in jede Zeile eine neue Berechnungsregel schreibt, ohne nochmal `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
  NatPlus n Zero n
  NatPlus n(Succ m) Succ(NatPlus 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 1.5.7 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
  True andb boole^ boole^
  boole^ andb True boole^
  False andb boole^ False
  boole^ andb False False
OrConst
 comprules
  True orb boole^ True
  boole^ orb True True
  False orb boole^ boole^
  boole^ orb False boole^
ImpConst
 comprules
  False impb boole^ True
  True impb boole^ boole^
  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 TERM1 TERM2).
```

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 zu Anwendung kann man aus einen 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 Beispielsaussage 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` reduzieren, beweisen.

2.4.6 Normalisierung

In der Theorie **TCF** gelten Terme 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. 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. So können wir also explizit nach Berechnungsregeln suchen. Wollen wir zum Beispiel alle Berechnungsregeln für die booleschen Operatoren, 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 dem 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 einen 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 hinzufügen, 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, hinzugefügt. 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.

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 setzen wir `(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 Einfühungsregel von EvenI die entsprechende Aussage zu zeigen. Die Aussage exl m m+m=0 zeigen wir, indem wir das Einfühungsaxiom 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 extrahieren 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)))
```

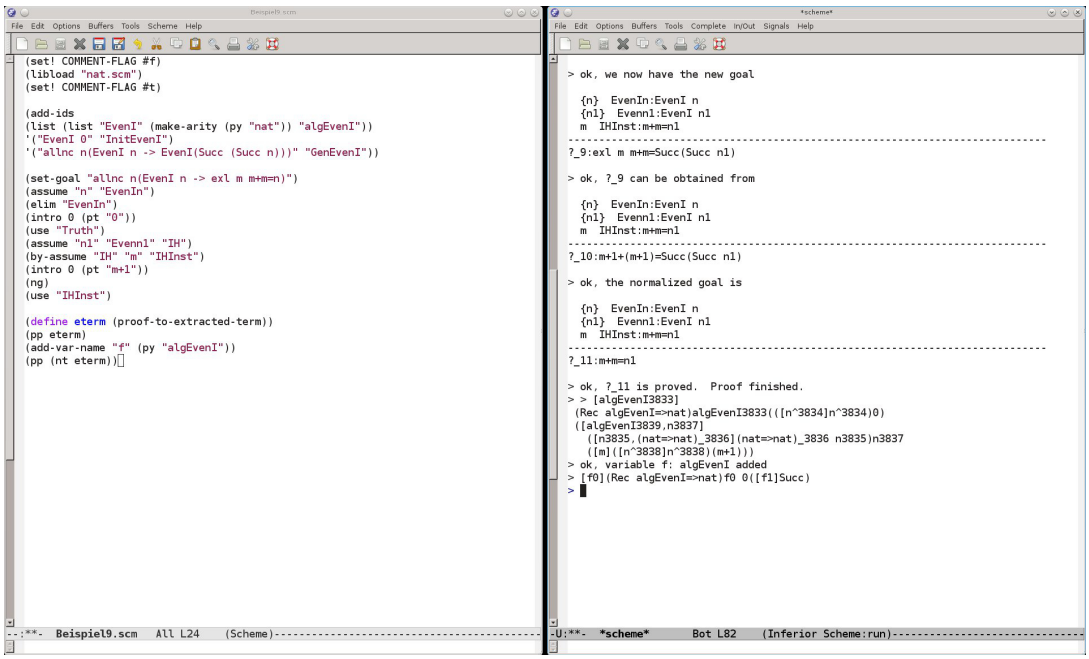
Die normalisierte Form ist deutlich besser lesbar. Es empfiehlt sich, allgemein jeden extrahieren Term zu normalisieren. Wir geben also (pp (nt eterm)) und erhalten die Ausgabe:

```
[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 die extrahieren Term erneut.

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

Das Auftretens 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 Terme wäre sogar ein Isomophismus 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 anderen 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 an Stelle 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 Prädikatenvariable `(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 Variable 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 haben wird als Beispiel die Totalität der Addition auf den natürlichen Zahlen also der Programmkonstante NatPlus. 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 Einfühungsaxiom `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 genau 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 1.5.6 und Bemerkung 1.5.7 besprochen, sind für totale boolesche Terme a und b die logischen Junktoren $\rightarrow, \vee$ 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 von Abschnitt 2.4.5 überlegt hat, dass dies im allgemeinen Fall nicht bewiesen werden kann, ist dies vollkommen richtig. An dieser Stelle war die implizite Angabe der Totalität auch 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 sinnvollem Beweis, wenn jeweils beide Terme total sind. Bemerkenswert ist hierbei noch, dass wir durch `(cdp)` einen sehr lagen 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 1.5.9 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 Variable verwenden, ist dies auf diese Weise sehr kompliziert: Man müsste zuerst beispielsweise $\forall_x A$ zu $\forall_{\widehat{x}}^{nc}(\mathbf{G}\widehat{x} \to A(\widehat{x}))$ 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 setzten 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 Annahmen 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)
```

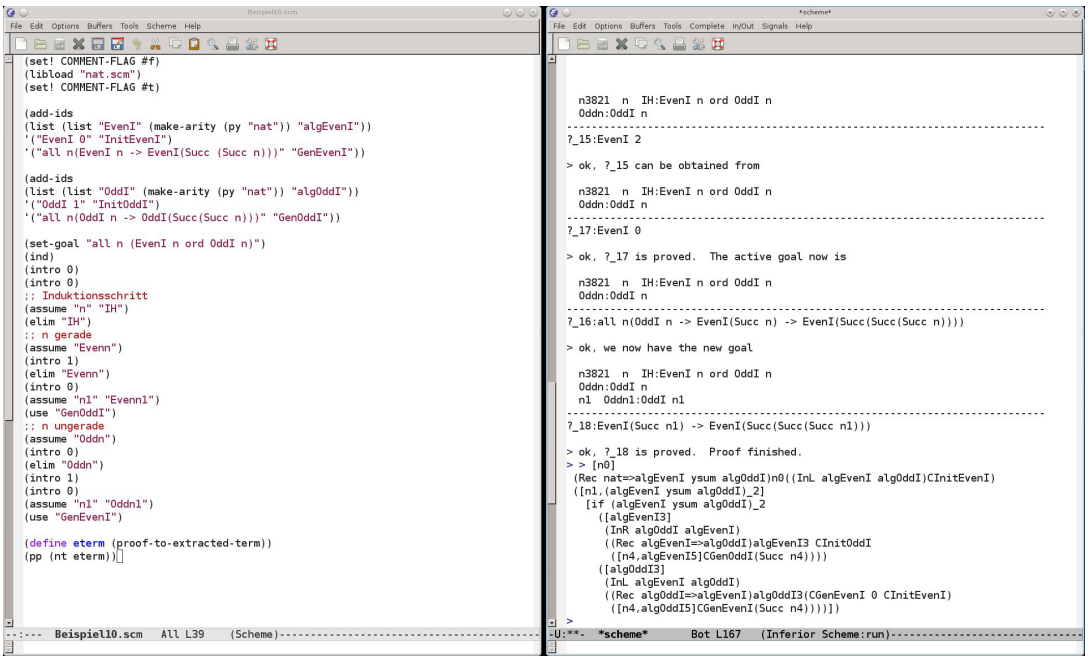
```
  n3965  n  IH:EvenI n ord OddI n
------------------------------------------------------------------
?_6:EvenI n -> EvenI(Succ n) ord OddI(Succ n)
```

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 gesammten Befehle und den extrahieren 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 extrahieren 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

Häuft 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 sich es 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 Reursionsoperator im extrahieren 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` in obigen Term nicht verwendet. Der Rekursionoperator 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 deklarien `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. In diesem Fall gibt es nur den einen Fall, dass `x = n pair m` ist für natürliche Zahlen `n` und `m`. 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))
```

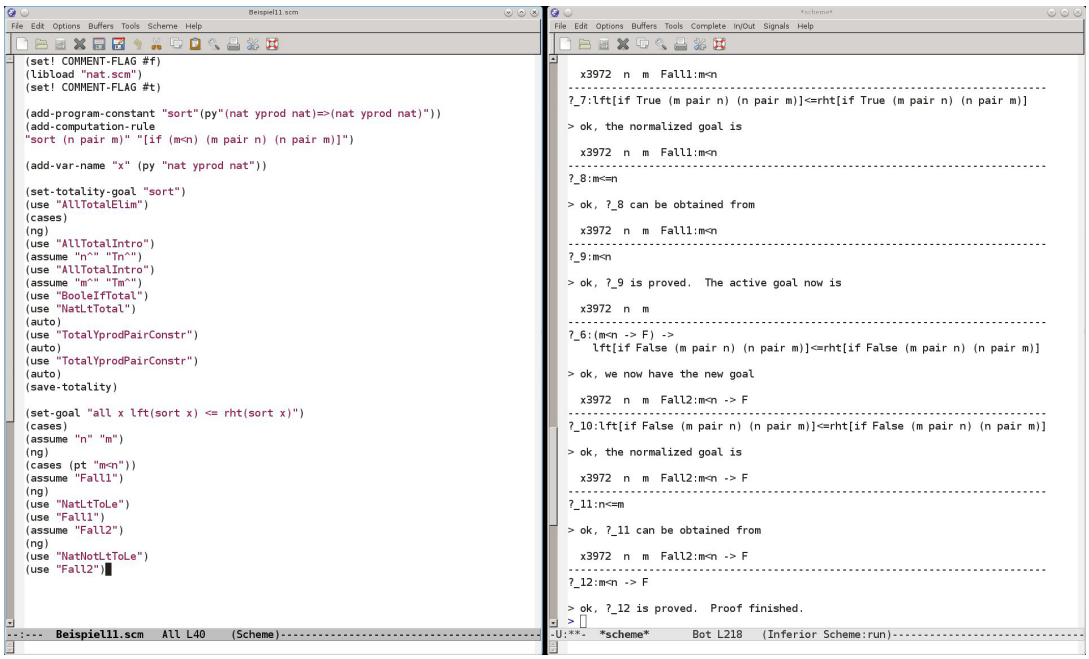
und wir haben erreicht, dass `x` zerlegt wurde. Nun nehmen wir die Variablen `n` und `m` in den Kontext mit `(assume "n" "m")` und normalisieren die Zielformel durch `(ng)`. Da `sort` durch Fallunterscheidung nach `m<n` definiert ist, liegt es nahe, den Beweis auch über Fallunterscheidung nach `m<n` 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 die entsprechende Annahme in die Prämisse durch `(assume "Fall1")`. 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.



Stichwortverzeichnis

Minlog-Befehle

Literaturverzeichnis

[1] Laura Crosilla. A tutorial for minlog, version 5.0. https://www.researchgate.net/publication/262348119_A_TUTORIAL_FOR_MINLOG_VERSION_50, 2014. [Online; aufgerufen am 23.12.2016].

[2] Helmut Schwichtenberg. Minlog reference manual. http://www.mathematik.uni-muenchen.de/~logik/download/ref.pdf, 2011. [Online; aufgerufen am 23.12.2016].

[3] Helmut Schwichtenberg. *Proofs and Computations*. Cambridge University Press, Dezember 2011.

[4] Helmut Schwichtenberg. Logik 2. http://www.mathematik.uni-muenchen.de/~schwicht/lectures/logic/ss16/ml.pdf, 2016. [Online; aufgerufen am 23.12.2016].