Induktive Definitionen in Minlog

Christoph-Simon Senjak

07. 07. 2010

ミンログ：すみません

Abstrakt

Ziel dieser Arbeit ist es, für den interaktiven Beweisassistenten Minlog Funktionen bereit zu stellen, um diesen ontologisch auf der Minimallogik mit Allquantoren und Implikationen, und durch induktiv definierte Prädikatkonstanten, aufzubauen. Dazu wurde darauf hin gearbeitet, dass die Dinge, die bisher atomar fest in Minlog eingearbeitet waren, durch Definitionen ersetzt werden können, die mit diesen Hilfsmitteln auskommen. Wir nutzen dabei die Theorie, die in [1] dargestellt wird. Da sämtliche theoretischen Begriffe auf diesem Buch basieren, wird im Folgenden nicht immer explizit auf dieses verweisen.

Weiterhin wird auf einige andere Systeme, insbesondere auf den Beweisassistenten Coq, eingegangen, die ähnliche Konstrukte aufweisen wie die hier implementierten.

1. Minimallogik in Minlog

Grundsätzlich arbeitet Minlog in der Minimallogik, mit dem Kalkül des natürlichen Schließens. Minlog kennt dabei alle gängigen Regeln, auch für Konjunktionen, Disjunktionen und Existenzquantoren, und diese sind atomar für Aussagen implementiert. Allerdings lassen sich diese durch induktiv definierte Prädikate ersetzen, und ein langfristiges Ziel der Entwicklung von Minlog ist es, diese noch atomar definierten Junktoren und Quantoren später vollständig durch induktive Definitionen zu ersetzen, in diesem Sinne ist auch diese Projektarbeit, weshalb hier auf eine Besprechung der betreffenden Regeln und Fälle gänzlich verzichtet wird.

1.1. Natürliches Schließen

Beweise in Minlog werden im Kalkül des natürlichen Schließens formuliert. Dieser besteht aus folgenden Regeln:

$$u : A$$

Hierbei bezeichnet $u : A$ die Annahmevariable. $u : A$ ist in diesem Fall freie Variable (oder Konstante).

$$\frac{\begin{array}{c}[u : A] \\ \vdots \\ B\end{array}}{A \to B}$$

Diese Regel bezeichnet man als Implikationseinführungsregel. Hierbei wird die Annahme $u : A$ *gebunden*, was man gängiger weise dadurch darstellt, sie in eckige Klammern zu schreiben (diese Notation hat den Vorteil, beim händischen Entwickeln eines Beweisbaums nach und nach Annahmevariablen binden zu können). Ein Beweis im intuitiven Sinne liegt vor, wenn alle Annahmen gebunden sind.

$$\frac{A}{\forall x A}$$

Diese Regel bezeichnet man als Alleinführungsregel. Man darf diese Regel nur dann anwenden, wenn x nicht frei in einer freien Annahme in der Herleitung von A vorkommt.

$$\frac{A \to B \qquad A}{B}$$

Diese Regel bezeichnet man als Implikationseliminationsregel.

$$\frac{\forall x A(x) \qquad [t]}{A(t)}$$

Diese Regel bezeichnet man als Alleliminationsregel. t ist hierbei ein Term, der vom selben Typ wie die Variable x sein muss.

1.2. Rechnerischer Gehalt und Dekorationen

1.2.1. Lambda-Notation

Eine spezielle Zuordnung von typisierten λ-Termen zu Beweisen erzeugt Algorithmen zur Lösung von Problemen, deren Lösbarkeit intuitionistisch beweisbar ist, und liefert eine einfache eindimensionale Notation für Beweise. Annahmevariablen $u : A$ und Annahmekonstanten (Axiome) $c : A$ bekommen dabei als Terme typisierte Variablen u^A bzw. c^A zugeordnet. Die Implikationseinführung als Term mit Typangaben ist $(\lambda_{u^A} M^B)^{A \to B}$, wobei M den Beweis von B aus A darstellt. Implikationselimination wird $(M^{A \to B} N^A)^B$. Bei der Alleinführung kommen Terme eines Objekttyps σ ins Spiel, die Darstellung ist $(\lambda_{x^\sigma} M^A)^{\forall x^\sigma A}$, wobei $\forall x^\sigma A$ zumeist als $\sigma \to A$ interpretiert wird. Allelimination wird durch $(M^{\forall x^\sigma A(x)} t^\sigma)^{A(t)}$ dargestellt.

1.2.2. Dekorationen

Um Programme aus Beweisen zu extrahieren werden diese Beweise vorher *dekoriert*. Dabei werden die Implikationspfeile $\to$ und Allquantoren $\forall$ nochmals unterteilt in jeweils eine rechnerische Variante $\to^c$, $\forall^c$, und eine nicht-rechnerische Variante $\to^{nc}$, $\forall^{nc}$. Die Regeln des natürlichen Schließens gelten für die rechnerischen Varianten unverändert. Die Eliminationsregeln gelten auch für die nicht rechnerischen Varianten. Lediglich die Einführungsregeln unterscheiden sich durch Zusatzbedingungen. Hierzu werden zunächst die Funktionen CA und CV jeweils über Beweisbäumen wie folgt definiert.

$$
\begin{aligned}
CV(c^A) &:= \emptyset \text{ (c Axiom)} \\
CV(u^A) &:= \emptyset \\
CV((\lambda_{u^A} M^B)^{A\to^c B}) &:= CV((\lambda_{u^A} M^B)^{A\to^{nc} B}) := CV(M) \\
CV((M^{A\to^c B} N^A)^B) &:= CV(M) \cup CV(N) \\
CV((M^{A\to^{nc} B} N^A)^B) &:= CV(M) \\
CV((\lambda_x M^A)^{\forall^c x A}) &:= CV((\lambda_x M^A)^{\forall^{nc} x A}) := CV(M)\backslash\{x\} \\
CV((M^{\forall^c x A(x)} r)^{A(r)}) &:= CV(M) \cup FV(t) \\
CV((M^{\forall^{nc} x A(x)} r)^{A(r)}) &:= CV(M) \\
CA(c^A) &:= \emptyset \text{ (c Axiom)} \\
CA(u^A) &:= \{u\} \\
CA((\lambda_{u^A} M^B)^{A\to^c B}) &:= CA(M^A)\backslash\{u\} \\
CA((\lambda_{u^A} M^B)^{A\to^{nc} B}) &:= CA(M^A)\backslash\{u\} \\
CA((M^{A\to^c B} N^A)^B) &:= CA(M) \cup CA(N) \\
CA((M^{A\to^{nc} B} N^A)^B) &:= CA(M) \\
CA((\lambda_x M^A)^{\forall^c x A}) &:= CA(M) \\
CA((\lambda_x M^A)^{\forall^{nc} x A}) &:= CA(M) \\
CA((M^{\forall^c x A(x)} r)^{A(r)}) &:= CA(M) \\
CA((M^{\forall^{nc} x A(x)} r)^{A(r)}) &:= CA(M)
\end{aligned}
$$

Die Einführungsregeln für $\to^{nc}$ und $\forall^{nc}$ lauten dann:

- Ist M^B eine Ableitung und $u^A \notin CA(M)$, dann ist auch $(\lambda_{u^A} M^B)^{A\to^{nc} B}$ eine Ableitung.
- Ist M^A eine Ableitung und kommt x nicht frei in einer freien Annahmevariable von M vor, und gilt $x \notin CV(M)$, so ist $(\lambda_x M^A)^{\forall^{nc} x A}$ eine Ableitung.

1.3. Natürliches Schließen und Dekorationen in Minlog

Zur Bildung von Formeln kennt Minlog elementare Befehle wie `make-allnc` und `make-imp` und etliche Befehle zur Bildung von Termen, gängigerweise verwendet man aber den bereitgestellten Parser, der sich durch `(pf string)` aufrufen lässt. Alle benutzten Variablennamen müssen vorher deklariert werden, die wichtigsten Funktionen hierbei sind `add-pvar-name` und `add-var-name` zur Deklaration von Objektvariablen und Aussagevariablen.

Die Syntax ist ansonsten weitestgehend klar und selbsterklärend. `allnc variablenname aussage` und `all variablenname aussage` bezeichnen den nicht-rechnerischen und rechnerischen Allquantor, `->` steht für die rechnerische Implikation, `-->` für die nicht rechnerische Implikation.

Weitere Funktionen sind `pt` zum Parsen von Termen, `pv` zum Parsen von Variablen, und `py` zum Parsen von Typen.

Zum Verfassen von Beweisen stellt Minlog zwar einen interaktiven Beweisassistenten zur Verfügung, der in der Scheme-REPL lebt, dieser ist allerdings für diese Projektarbeit von untergeordneter Rolle, und soll deshalb nicht näher besprochen werden. Zum nicht-interaktiven Erzeugen von Beweisen gibt es für jede Regel des natürlichen Schließens eine entsprechende Funktion.

Zur Einführung einer Annahmevariable kann die Funktion `(make-proof-in-avar-form annahmevariable)` verwendet werden. Hierbei ist zu beachten, dass `annahmevariable` eine Annahmevariable sein muss, das heißt ein Objekt, von dem Minlog weiß, dass es eine solche ist. Um aus einer Formel eine neue Annahmevariable zu bekommen nutzt man die Funktion `formula-to-new-avar`. Um Beispielsweise einen Beweis der Form $u : B$ auszudrücken, schreibt man in Minlog `(make-proof-in-avar-form (formula-to-new-avar (pf "B")))`.

Zur Einführung eines nicht-rechnerischen Allquantors gibt es die Funktion `(make-proof-in-allnc-intro-form variable beweis)`. `beweis` ist dabei der bisher geführte Beweis, `variable` ist die Variable, die gebunden werden soll. Zur Elimination eines nicht-rechnerischen Allquantors gibt es die Funktion `(make-proof-in-allnc-elim-form beweis term)`. `beweis` ist dabei der bisher geführte Beweis, `term` der Term, der für die durch den Allquantor gebundene Variable eingesetzt werden soll. Analog gibt es für rechnerische Allquantoren die Funktionen `make-proof-in-all-intro-form` und `make-proof-in-all-elim-form`.

Zur Einführung eines nicht-rechnerischen Implikationspfeils gibt es die Funktion `(make-proof-in-impnc-intro-form annahmevariable beweis)`. Dabei bezeichnet `beweis` den bisherigen Beweis, und `annahmevariable` die zu bindende Annahmevariable. Dabei ist zu beachten, dass es sich bei `annahmevariable` weder um eine Formel selbst, noch um einen Beweis der Form $u : A$ handeln darf, sondern tatsächlich um eine Annahmevariable (die man z.B. mit `formula-to-new-avar` erhalten kann) handeln muss. Zur Elimination eines nicht-rechnerischen

Implikationspfeils gibt es die Funktion `(make-proof-in-impnc-elim-form beweis-von-implikation beweis-von-praemisse)`. Die beiden Argumente sind jeweils Beweise, das erste Argument beweist die Implikation die eliminiert werden soll, das zweite Element die Prämisse dieser Implikation (anschaulich $A \to B$ und A).

2. Das Typsystem von Minlog

Das Typsystem von Minlog ist zu komplex als dass es im Rahmen dieser Projektarbeit sinnvoll vollständig betrachtet werden könnte. Hier sollen deshalb nur die wichtigen Aspekte davon behandelt werden. Grundsätzlich existieren definierbare Grundtypen ρ, σ, ..., und zu jeweils zwei existierenden Typen α, β existiert der Pfeiltyp $\alpha \to \beta$. Die Grundtypen wiederum ähneln den Typen, die man von Standard ML kennt, in dem Sinne, dass man sie nur durch Konstruktoren definiert. Zusätzlich zu Typen gibt es noch Typformen, die von Typparametern abhängen, um generelle Transformationen von Typen in andere Typen zu abstrahieren.

2.1. Typformen, Algebraformen, Konstruktortypformen

Wir definieren nun induktiv die Mengen $\mathsf{Ty}(\vec{\alpha})$, $\mathsf{Alg}(\vec{\alpha})$ und $\mathsf{KT}_\xi(\vec{\alpha})$, dabei ist $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ eine Sequenz von Typparametern, und $\langle\rangle$ bezeichne die leere Sequenz.

$$\frac{}{\alpha_i \in \mathsf{Ty}(\vec{\alpha})}\ ,\quad \frac{\iota \in \mathsf{Alg}(\vec{\alpha})}{\iota \in \mathsf{Ty}(\vec{\alpha})}\ ,\quad \frac{\rho \in \mathsf{Ty}(\langle\rangle) \qquad \sigma \in \mathsf{Ty}(\vec{\alpha})}{\rho \to \sigma \in \mathsf{Ty}(\vec{\alpha})}\ ,$$

$$\frac{\kappa_1, \ldots, \kappa_n \in \mathsf{KT}_\xi(\vec{\alpha})}{\mu_\xi(\kappa_1, \ldots, \kappa_n) \in \mathsf{Alg}(\vec{\alpha})}\ (n \geq 1, \text{ mindestens ein } \kappa_i \text{ nullär})\ ,$$

$$\frac{\vec{\rho} \in \mathsf{Ty}(\vec{\alpha}) \qquad \vec{\sigma}_1, \ldots, \vec{\sigma}_n \in \mathsf{Ty}(\langle\rangle)}{\vec{\rho} \to (\vec{\sigma}_1 \to \xi) \to \ldots \to (\vec{\sigma}_n \to \xi) \to \xi \in \mathsf{KT}_\xi(\vec{\alpha})}\ (n \geq 0)$$

Dabei sei $\to$ stets rechtsassoziativ, auch innerhalb eines Ausdrucks mit Vektorpfeilen, unabhängig von den dortigen Bündelungen. Weiterhin heißen die $\vec{\rho}$ die *(nichtrekursiven) Parameterargumenttypen* und die $(\vec{\sigma}_i \to \xi)$ *rekursive Parameterargumenttypen* einer Konstruktortypform $\vec{\rho} \to (\vec{\sigma}_1 \to \xi) \to \ldots \to (\vec{\sigma}_n \to \xi) \to \xi$. Eine Konstruktortypform ohne rekursive Parameter heißt *nullär*. Die Forderung dass es einen solchen nullären Konstruktor geben muss verhindert, dass es leere Typen gibt, und gibt später einen Ansatzpunkt für induktive Definitionen. Die Elemente aus $\mathsf{Ty}(\langle\rangle)$ heißen *Typen*.

Wichtige Typen sind die boole'schen Werte $\mathbb{B} := \mu_\xi(\xi, \xi)$, die natürlichen Zahlen $\mathbb{N} := \mu_\xi(\xi, \xi \to \xi)$ und die Ordinalzahlen $\mathbb{O} := \mu_\xi(\xi, \xi \to \xi, (\mathbb{N} \to \xi) \to \xi)$. Eine wichtige Typform ist die Typform der Listen $\mathbb{L}(\alpha) := \mu_\xi(\xi, \alpha \to \xi \to \xi)$.

2.2. Interpretation

Die Theorie hinter Minlog interpretiert Typen durch freie Algebren. Eine genaue Besprechung würde den Rahmen dieser Projektarbeit weit überschreiten, und ist zudem nicht notwendig um die Ziele zu verstehen. Grundsätzlich kann man jedem Typen ξ für jeden seiner Konstruktoren κ_i ein Konstruktorsymbol C_i^ρ zuordnen, wobei ρ den Konstruktortypen bezeichne. Weiterhin gebe es Symbole $\star_\rho$. Wir ordnen Ketten von solchen Symbolen einen Typ in folgender (offensichtlicher) Weise zu: $\star_\rho$ hat den Typ ρ, wobei wir im Folgenden den Typindex weglassen, jedes andere Singleton C_i^ρ hat den Typ ρ, habe die Kette $a^{\rho\to\sigma}$ den Typen $\rho \to \sigma$ und die Kette b^ρ den Typen ρ, so hat die Kette $a^{\rho\to\sigma}b^\rho$ den Typen σ. Den Typen ξ interpretieren wir dann als Menge aller Symbolketten vom Typ ξ, sie so genannte *freie Algebra* des Typs ξ. Nennen wir die Konstruktoren von $\mathbb{N}$ zum Beispiel 0 und S, so ist diese Menge $\{0, S0, SS0, \ldots, \star, S\star, SS\star, \ldots\}$. $\{0, S0, \ldots\}$ entsprechen dabei den natürlichen Zahlen im "herkömmlichen" Sinne, den *totalen Zahlen* für die das Totalitätsprädikat, welches in 4 eingeführt wird, gilt, $\{\star, S\star, \ldots\}$ können anschaulich als nicht vollständig bestimmte Zahlen angesehen werden, ähnliche Konstrukte findet man zum Beispiel bei Lazy Lists und allgemein Lazy Structures in diversen funktionalen Sprachen.

Damit hat man eine Interpretation der so genannten *finitistischen Typen* ξ, das sind die Typen, deren nicht-nulläre Konstruktoren nur rekursive Prämissen der Form ξ haben. Andere Typen können ebenfalls interpretiert werden, eine Besprechung der Interpretation dieser Typen würde allerdings den Rahmen dieser Projektarbeit sprengen, obgleich die hier implementierten Funktionen auch mit so gearteten Typen zurechtkommen. Für eine grundsätzliche Intuition reicht es im Rahmen dieser Projektarbeit aus, diese Typen als Mengen normalisierter typisierter Terme anzusehen.

Es ist anzumerken, dass man zwei Algebren auch simultan, rekursiv abhängig voneinander definieren kann. Dieser Fall wird in dieser Projektarbeit jedoch grundsätzlich nicht betrachtet, und die Definitionen, auch in Minlog, sind nicht kompatibel damit.

2.3. API

Einen Typen definiert man in Minlog direkt durch dessen Algebra. Der gängigste Weg dies zu tun ist die Verwendung des Befehls `add-param-alg` für parametrisierte Algebren, und `add-alg` für solche ohne Parametrisierung. Die natürlichen Zahlen lassen sich zum Beispiel definieren durch

```
(add-alg "Natural" '("NZero" "Natural") '("NSucc" "Natural=>Natural"))
```

Ordinalzahlen durch

```
(add-alg "Ordinal"
 '("OZero" "Ordinal")
 '("OSucc" "Ordinal=>Ordinal")
 '("OSup" "Natural=>Ordinal"))
```

Listen durch

```
(add-param-alg "MyList"
  'alg-typeop 1
  '("MyNil" "MyList")
  '("MyCons" "alpha1=>MyList=>MyList"))
```

Dabei gibt `'alg-typeop` an um welchen Token-Type es sich handeln soll bei dem Algebra-Namen (nur wichtig für den Parser), und 1 gibt an, dass genau ein Typparameter, `alpha1`, vorhanden ist.

2.4. Ähnliche Typsysteme

Induktive Typen in Coq

Das Typsystem von Coq ist dem von Minlog sehr ähnlich. Ein Beispiel für die Definition von Listen ist gegeben in 4.5.

Rekursive Typen in StandardML

Die funktionale Programmiersprache SML (Standard Meta Language) hat ein in einigen Aspekten vergleichbares Typsystem. Die `datatype`-Deklarationen erwarten eine Auflistung von Konstruktoren. Anders als in Minlog, wo direkt Typen der Konstruktoren angegeben werden, werden in SML nur Argumenttypen angegeben. Deshalb können Tupel der Form $\mu_\xi(\rho_1 \to \ldots \to \rho_n \to \xi)$ nicht auf diese Weise definiert werden, und sind dementsprechend atomar vorgegeben. Ansonsten sind die Deklarationen ähnlich. Natürliche Zahlen lassen sich definieren durch

```
datatype Nat = Zero | Succ of Nat;
```

Ordinalzahlen damit durch

```
datatype Ordinal = OZero | OSucc of Ordinal | OSup of Nat->Ordinal;
```

Die Ordinalzahl ω_0 kann zum Beispiel definiert werden durch

```
let fun nto n =
  case n of Zero   => OZero
          | Succ m => OSucc (nto m) in
  OSup nto end;
```

Als Beispiel für Typen mit Typparametern können Listen definiert werden durch:

```
datatype 'a List = Nil | Cons of 'a * ('a List);
```

3. Induktiv definierte Prädikatkonstanten

Durch induktiv definierte Prädikatkonstanten lassen sich die Junktoren $\wedge$, $\vee$ und der $\exists$-Quantor aus dem $\to$-Junktor und $\forall$-Quantor definieren, und alle ihre Herleitungsregeln, die man ansonsten als Kalkülregeln bräuchte, lassen sich dadurch definieren. Jedes induktiv definierte Prädikat bekommt dabei Einführungs- und Eliminationsklauseln.

3.1. Prädikatparameter, Prädikatformen, Formelformen, Klauselformen

Seien $X, \vec{Y}$ paarweise verschiedene Prädikatvariablen, wobei wir die Y_i *Prädikatparameter* nennen. Wir definieren nun induktiv die Menge der *Formelformen* $F(\vec{Y})$, *Prädikatformen* $\mathrm{Preds}\,(\vec{Y})$ und *Klauselformen* $\mathrm{Cl}_X(\vec{Y})$. Wieder bezeichne $\langle\rangle$ die leere Sequenz.

$$\frac{}{Y_i\vec{r} \in F(\vec{Y})}\,,\quad \frac{A \in F(\langle\rangle) \qquad B \in F(\vec{Y})}{A{\to}B \in F(\vec{Y})}\,,\quad \frac{A \in F(\vec{Y})}{\forall x A \in F(\vec{Y})}\,,\quad \frac{C \in F(\vec{Y})}{\{\vec{x}\|C\} \in \mathrm{Preds}\,(\vec{Y})}\,,$$

$$\frac{P \in \mathrm{Preds}\,(\vec{Y})}{P\vec{r} \in F(\vec{Y})}\,,\quad \frac{K_0, \ldots, K_{k-1} \in \mathrm{Cl}_X(\vec{Y})}{\mu_X(K_0, \ldots, K_{k-1}) \in \mathrm{Cl}_X(\vec{Y})}\ (k \geq 1, \text{mindestens ein } K_i \text{ nullär})\,,$$

$$\frac{\vec{A} \in F(\vec{Y}) \qquad \vec{B}_0, \ldots, \vec{B}_{n-1} \in F(\langle\rangle)}{\forall\vec{x}(\vec{A}{\to}\forall_{\vec{y}_1}(\vec{B}_1{\to}X\vec{s}_1){\to}\ldots{\to}\forall_{\vec{y}_n}(\vec{B}_n{\to}X\vec{s}_n){\to}X\vec{t}) \in \mathrm{Cl}_X(\vec{Y})}\ (n \geq 0)\,.$$

Es fällt sofort auf, dass diese Definitionen stark an die Definitionen von Seite 7 erinnern. Jeder induktiv definierten Prädikatkonstante kann nämlich deren Realisierer-Algebra zugeordnet werden. Diese wird genauer in 5.2 besprochen.

Es ist auch hier anzumerken, dass man mehrere induktiv definierte Prädikate simultan voneinander abhängig machen kann. Dieser Fall wird - wie schon bei den Algebren - in dieser Projektarbeit nicht beachtet. Wir gehen also davon aus, dass eine induktiv definierte Prädikatkonstante in den Klauseln nur sich selbst, eventuelle Parameter, und ansonsten höchstens bereits fertig definierte andere Prädikatkonstanten enthält.

3.2. Induktive Definitionen in Minlog

Zur Erzeugung induktiv definierter Prädikate stellt Minlog die Funktion `add-ids` bereit. Für eine genaue Besprechung der Syntax sei auf [2] verwiesen. Wir definieren in unserem Code einige induktive Prädikate.

In der Datei `predconsts.scm` sind die in dieser Projektarbeit benötigten induktiven Prädikatkonstanten definiert.

An dieser Stelle seien auch gleich die zugehörenden Realisierer-Algebren der Prädikate angegeben, die in 5.2 definiert werden.

Zunächst werden zwei Versionen desselben Prädikates, `Even` und `EvenCR` definiert. Es ist $Even = \mu_X(X0, \forall^{nc} n^{\mathbb{N}}.Xn \to^c X(n+2))$ und $EvenCR = \mu_X(X0, \forall^c n^{\mathbb{N}}.Xn \to^c X(n+2))$. Das Prädikat beschreibt die geraden natürlichen Zahlen. In der ersten Version ist die zugehörige Algebra $\mu_\xi(\xi, \xi \to \xi)$ und damit isomorph zu $\mathbb{N}$, in der zweiten Version ist der Allquantor rechnerisch, die Algebra ist somit $\mu_\xi(\xi, \mathbb{N} \to \xi \to \xi)$. Die Algebren heißen jeweils `algEven` und `algEvenCR` mit den Konstruktoren `InitEven` bzw. `InitEvenCR` und `GenEven` bzw. `GenEvenCR`. Die Definitionen lauten somit

```
(add-ids (list (list "Even" (make-arity (py "nat")) "algEven"))
 '("Even 0" "InitEven")
 '("allnc n^.Even n^ -> Even(n^ +2)" "GenEven"))

(add-ids (list (list "EvenCR" (make-arity (py "nat")) "algEvenCR"))
 '("EvenCR 0" "InitEvenCR")
 '("all n^.EvenCR n^ -> EvenCR(n^ +2)" "GenEvenCR"))
```

Es werden auch zwei Versionen der Leibnizgleichheit definiert, ebenfalls eine nichtrechnerische und eine rechnerische Variante, die rechnerische Variante nur für natürliche Zahlen, $leib = \mu_X(\forall^{nc} x^\alpha Xxx)$, $LeibCR = \mu_X(\forall^c n^{\mathbb{N}} Xnn)$. Der nichtrechnerischen Variante wird keine Algebra zugeordnet, denn sie soll keinen rechnerischen Gehalt haben.

```
(add-ids (list (list "leib" (make-arity (py "alpha") (py "alpha"))))
 '("allnc x^ leib x^ x^" "initLeib"))

(add-ids (list (list "LeibCR" (make-arity (py "nat") (py "nat"))
              "algLeibCR"))
 '("all n^ . LeibCR n^ n^" "initLeibCR"))
```

Weiterhin definiert werden ein Existenzquantor für natürliche Zahlen, der Graph des Even-Prädikats und eine rechnerische Variante des Totalitätsprädikats für natürliche Zahlen.

3.2.1. Leibnizgleichheit, Falsum

Leibnizgleichheit und Falsum werden hauptsächlich im Zusammenhang mit Ex-Falso-Quodlibet benötigt, dementsprechend befinden sich die Beweise und Definitionen dazu in der Datei `efq.scm`.

Das Eliminationsaxiom der Leibnizgleichheit $\mu_X(\forall^{nc} x^\rho X(x^\rho, x^\rho))$ ist $\forall x \forall y.\mathrm{leib}(x,y) \to \forall x P x x \to P x y$

Zunächst gelten Transitivität und Reflexivität, die sich in Minlog beweisen lassen durch

```
"leibSymm"
(set-goal "allnc x^, y^ ( leib x^ y^ -> leib y^ x^ )")
(assume "x^" "y^")
(elim)
(use "initLeib")
(save "leibSymm")

"leibTrans"
(set-goal "allnc x^,y^,z^ ( leib x^ y^ -> leib y^ z^ -> leib x^ z^ )")
(assume "x^" "y^" "z^")
(elim)
(assume "t^")
(use "AfromA")
(save "leibTrans")
```

Ursprünglich gab es einige Probleme bei der Implementierung verschiedener Beweise und Syntaxtransformationen im Zusammenhang mit Typparametern und Substitution. Die Implementierung einiger Funktionen in dieser Arbeit war allerdings nach der Lösung dieses Problems bereits zu weit fortgeschritten, als dass es sinnvoll gewesen wäre, sie nochmals anders aufzubauen, zumal der interaktive

Beweisassistent schlussendlich nur eine andere Notation darstellt, um die selben Beweisbäume zu erzeugen.

Und so wurden einige Beweise, statt sie interaktiv einzugeben und anschließend durch entsprechende Substitution zu nutzen, nichtinteraktiv, durch direkte Programmierung der Beweisbäume, geführt.

`proof-leib-compat-for` erwartet zwei Argumente `form` und `var`. Es soll die Kompatibilität $\mathrm{leib}(x,y)\to Qx\to Qy$ beweisen. Dabei gibt `form` die entsprechende Form Qz, und `var` die zu ersetzende Variable z an.

```
> (cdp (proof-leib-compat-for (pf "P alpha^") (pv "alpha^")))
; .......allnc x^2553,x^2552(leib x^2553 x^2552 ->
  allnc x^(P x^ -> P x^) -> P x^2553 -> P x^2552) by axiom Elim
; .......x^2547
; ......allnc x^2552(leib x^2547 x^2552 ->
  allnc x^(P x^ -> P x^) -> P x^2547 -> P x^2552) by allnc elim
; ......x^2548
; .....leib x^2547 x^2548 -> allnc x^(P x^ -> P x^) -> P x^2547 ->
  P x^2548 by allnc elim
; .....leib x^2547 x^2548 by assumption u1062
; ....allnc x^(P x^ -> P x^) -> P x^2547 -> P x^2548 by imp elim
; ......P x^2549 by assumption u1064
; .....P x^2549 -> P x^2549 by imp intro u1064
; ....allnc x^2549(P x^2549 -> P x^2549) by allnc intro
; ...P x^2547 -> P x^2548 by imp elim
; ..leib x^2547 x^2548 -> P x^2547 -> P x^2548 by imp intro u1062
; .allnc x^2548(leib x^2547 x^2548 -> P x^2547 -> P x^2548)
  by allnc intro
; allnc x^2547,x^2548(leib x^2547 x^2548 -> P x^2547 -> P x^2548)
  by allnc intro
```

Die bisherigen Beweise sind Minimallogisch, über die induktive Definition der Leibnizgleichheit kann nun aber ein Falsum definiert werden, für das mit den bisherigen Möglichkeiten das Ex-Falso-Quodlibet hergeleitet werden kann, es ergibt sich also ein System der intuitionistischen Logik. Hierzu sei die oben definierte Algebra $\mathbb{B} := \mu_\xi(\xi,\xi)$ der Boole'schen Werte herangezogen, deren Konstruktoren mit $\frown$ und $\frowny$, bzw. `True` und `False` in Minlog, benannt seien, und das Falsum als $\mathfrak{F} := \mathrm{leib}(\frowny,\frown)$ definiert. Das Verum wird als $\mathfrak{T} := \mathrm{leib}(\frown,\frown)$ definiert.

Man kann nun die Aussage $\forall^{nc}x^\alpha\forall^{nc}y^\alpha(\mathfrak{F}\to\mathrm{leib}(x,y))$ für beliebiges α herleiten. Diese Aussage nennen wir EQD. Der Beweis wird in `proof-leib-eqd-for` implementiert. Die Funktion erwartet als Argument einen Typen `tp`. Für den Beweis sei auf [1] verwiesen.

```
>  (pp (normalize-formula (proof-to-formula
```

```
        (proof-leib-eqd-for (py "alpha")))))
allnc x^2589,x^2590(leib False True -> leib x^2589 x^2590)
>  (pp (normalize-formula
        (proof-to-formula (proof-leib-eqd-for (py "nat")))))
allnc n^2602,n^2603(leib False True -> leib n^2602 n^2603)
>  (pp (normalize-formula
        (proof-to-formula (proof-leib-eqd-for (py "list nat")))))
allnc (list nat)^2617,(list nat)^2618(
 leib False True -> leib(list nat)^2617(list nat)^2618)
```

Die Funktion `eqd-proof-for` erwartet nun zwei Terme gleichen Typs und eliminiert die Allquantoren dieses Beweises mit diesen Termen.

3.2.2. Ex-Falso-Quodlibet für die Basisfälle

Basierend auf den Funktionen und Definitionen von 3.2.1 werden in der Datei `efq.scm` nun Funktionen zur Erzeugung von Beweisen für EFQ definiert. Zunächst für die Basisfälle.

Zunächst kann man in üblicher Weise aus den Aussagen $\mathfrak{F}\to A$ und $\mathfrak{F}\to B$ die Aussagen $\mathfrak{F}\to A\to B$ und $\mathfrak{F}\to\forall x A$ gewinnen, und somit per struktureller Induktion das Problem der Herleitung bis zu den atomaren Formeln hindurchschleifen. Als atomare Formeln bleiben dann genau die Prädikatkonstanten und Prädikatvariablen.

Die Implementierung des EFQ beginnt in der Funktion `efq-for`. Diese erwartet zwei Argumente, `form`, welches die Formel A, zu der $\mathfrak{F}\to A$ hergeleitet werden soll, angibt, und `avars`, welches eine Liste von Annahmevariablen der Form $\mathfrak{F}\to B_i$ sein soll, welche bei der Herleitung von $\mathfrak{F}\to A$ verwendet werden können. `avars` ist deshalb nötig, weil der Fall einer vollständig bestimmten Aussage eher selten ist, sondern meistens Prädikatvariablen in Aussagen vorkommen, für die das EFQ auf diese Weise nicht nachgewiesen werden kann. `avars` wird an alle weiteren Funktionen weitergegeben. Zunächst überprüft `efq-for`, ob die gewünschte Form bereits in den Annahmevariablen enthalten ist, und falls dem so ist, liefert sie den trivialen Beweis zurück. Dies ist vor Allem sinnvoll, da `efq-for` rekursiv, auch von den anderen Funktionen, die es nutzen und diese Prüfung nicht durchführen, aufgerufen wird. Tritt die übergebene Form dort nicht auf, wird nach der Struktur unterschieden.

Die Herleitungen für die Allquantoren lauten formal

$$\dfrac{\dfrac{\dfrac{}{\mathrm{leib}(\circledcirc,\circledcirc)\to A}\,\text{Voraussetzung} \qquad u : \mathrm{leib}(\circledcirc,\circledcirc)}{A}\,{\to}-}{\dfrac{\dfrac{A}{\forall x A}\,\forall{+}_x}{\mathfrak{F}\to\forall x A}\,{\to}{+}_u}$$

Sie werden in den Funktionen `allnc-efq-for` und `all-efq-for` implementiert. Die Herleitungen für die Implikationen lauten Formal

$$\dfrac{\dfrac{\dfrac{}{\mathfrak{F}\to B}\,\text{Voraussetzung} \qquad u:\mathfrak{F}}{B}\,{\to}{-} \qquad B\to A\to B}{\dfrac{A\to B}{\mathfrak{F}\to A\to B}\,{\to}{+}_u}\,{\to}{-}$$

Sie werden implementiert in `imp-efq-for` und `impnc-efq-for`.

3.2.3. Ex-Falso-Quodlibet für induktiv definierte Prädikatkonstanten

Es bleibt der Fall, dass es sich um eine induktiv definierte Prädikatkonstante handelt. Dieser ist der komplexeste Unterfall, und stellt die eigentliche Schwierigkeit bei der Implementierung von EFQ dar. Die aufzurufende Funktion lautet `idpredconst-efq-for`, und erwartet als erstes Argument eine Prädikatform, deren Prädikat induktiv definiert ist, und eine Liste von Annahmevariablen analog zu `efq-for`. Für den formalen Beweis sei auf [1] verwiesen.

Der Beweis für die Kompatibilität wurde in der Funktion `proof-leib-compat-for` implementiert. Diese erwartet als Argument eine Formel und eine Variable, die in dieser Formel vorkommt, für die anschließend ein Kompatibilitätsbeweis formuliert und zurückgeliefert wird. Um die verallgemeinerte Kompatibilität zu beweisen, wird die entsprechende Aussage zunächst mittels `create-basic-implication` erzeugt. Diese Funktion erwartet eine Implikation der Form $Q\vec{x}\to Q\vec{y}$, und erzeugt für diese eine Form $\forall\vec{x}\forall\vec{y}(\text{leib}(x_n,y_n)\to\ldots\to\text{leib}(x_1,y_1)\to Q\vec{x}\to Q\vec{y})$. Anschließend wird diese übergeben an die Funktion `efq-basic-elimination`, welche einen Beweis dafür zurückliefert.

```
> (add-pvar-name "Z"
   (make-arity(py"alpha")(py"alpha")(py"alpha")(py"alpha")))
> (define *bi*
    (create-basic-implication
     (pf "Z x^1 x^2 x^3 x^4 -> Z x^5 x^6 x^7 x^8")))
> (pp *bi*)
leib x^4 x^8 ->
leib x^3 x^7 ->
leib x^2 x^6 -> leib x^1 x^5 -> Z x^1 x^2 x^3 x^4 -> Z x^5 x^6 x^7 x^8
> (pp (proof-to-formula (efq-basic-elimination *bi*)))
leib x^4 x^8 ->
leib x^3 x^7 ->
leib x^2 x^6 -> leib x^1 x^5 -> Z x^1 x^2 x^3 x^4 -> Z x^5 x^6 x^7 x^8
```

Die Funktion `eqd-proof-for` erwartet zwei Terme x und y gleichen Typs ρ und beweist $\mathrm{leib}(\circledminus,\circledcirc)\to\mathrm{leib}(x^\rho,y^\rho)$. `efq-basic-implication` nutzt diese Funktion anschließend, um aus der mit `efq-basic-elimination` erhaltenen Formel zu beweisen $\mathrm{leib}(\circledminus,\circledcirc)\to\forall\vec{x}\forall\vec{y}(Q(\vec{x})\to Q(\vec{y}))$. Dabei erwartet `efq-basic-implication` eine Prädikatkonstante als Argument.

```
> (pp (proof-to-formula
       (efq-basic-implication (make-idpredconst "Even" '() '()))))
allnc n^2636,n^2635(leib False True -> Even n^2635 -> Even n^2636)
```

Nun nutzt `idpredconst-efq-for-nullary-clause` ein nulläres Einführungsaxiom der Prädikatkonstante, das heißt ein Axiom, das keine rekursiven Prämissen enthält, um zu beweisen $\mathrm{leib}(\circledminus,\circledcirc)\to Q(\vec{t})$ für bestimmte Terme t. Nulläre Einführungsaxiome haben die Form $\forall\vec{x}(\vec{A}\to Q(\vec{s}))$ für bestimmte Terme $\vec{s}$, die auch von $\vec{x}$ abhängen können. Durch Elimination der Allquantoren erreicht man somit eine Aussage der Form $\vec{A}\to Q(\vec{t})$. Für die $\vec{A}$ kann jeweils mittels der vorhandenen Funktionen das EFQ bewiesen werden, und mit dem Einführungsaxiom folgt somit auch $\mathrm{leib}(\circledminus,\circledcirc)\to Q(\vec{t})$.

```
> (cdp
    (idpredconst-efq-for-nullary-clause
      (make-idpredconst "Even" '() '()) '()))
; .Even 0 by axiom Intro
; leib False True -> Even 0 by imp intro u1076
```

Schließlich verwendet `idpredconst-efq-for` die so erhaltene Aussage und folgert daraus mittels verallgemeinertem EQD das EFQ.

3.2.4. Stabilität

Anders als EFQ lässt sich die Stabilität nicht generell beweisen, da man zum Beispiel den Existenzquantor induktiv definieren kann, aber die Stabilität für Existenzaussagen nicht generell intuitionistisch nachweisen kann. Zumindest kann man aber, wie bei dem Beweis von EFQ, das Problem bis zu den Annahmevariablen und induktiven Prädikaten hindurchschleifen. Dieses wurde in den Funktionen `*stab-for` implementiert. Die Namensgebung ist analog zu der der Implementierung von EFQ.

3.2.5. Tests

Ein Test für Stabilität lautet

```
> (cdp (stab-for
  (pf "all x . allnc y . A -> B --> A")
  (map formula-to-new-avar
       (list (make-imp (make-not (make-not (pf "A"))) (pf "A"))
     (make-imp (make-not (make-not (pf "B"))) (pf "B"))))))
```

und liefert einen korrekten Beweis zurück für

```
; ((all x allnc y(A -> B --> A) -> leib False True) -> leib False True)
  -> all x allnc y(A -> B -> A) by imp intro u981
```

Test für die wichtigsten Basisfälle von EFQ befinden sich außerdem in `tests.scm`.

3.2.6. Induktiv definierte Prädikatkonstanten in Coq

Der Beweisassistent Coq kennt ebenfalls induktiv definierte Prädikatkonstanten. Die obige Prädikatkonstante `Even` lässt sich darin beispielsweise definieren durch

```
Inductive even: nat->Prop :=
  | even_init: even 0
  | even_gen: forall m, even m -> even (S(S(m))).
```

Daraufhin definiert Coq als Eliminationsaxiom

```
even_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall m : nat, even m -> P m -> P (S (S m))) ->
       forall n : nat, even n -> P n
```

auf das man in Minlog in der Regel nicht direkt, sondern für eine spezielle Implikation mittels `imp-formulas-to-elim-aconst` zugreift, da Minlog keine Quantifikation über Prädikatvariablen erlaubt. Analog ist die Leibnizgleichheit definierbar durch

```
Inductive leib (T:Type) : T->T->Prop :=
 | leib_intro : forall x:T, leib(T) x x.
```

Anders als in Minlog muss hier der Typparameter explizit angegeben werden und wird nicht automatisch ermittelt. Anders als in Minlog benötigt man aber die

Leibnizgleichheit nicht, um eine intuitionistische Theorie aufzubauen, dies ist in Minlog eine Konsequenz der Tatsache, dass induktive Definitionen eine nulläre Klausel haben müssen. Coq hat diese Beschränkung nicht, somit kann man das Falsum einfach durch $\mu_X()$ definieren, das Eliminationsaxiom stellt dann das EFQ dar. In Coq geht dies mittels

```
Inductive falsum:Prop := .
```

Das berechnete Eliminationsaxiom dazu ist

```
falsum_ind
     : forall P : Prop, falsum -> P
```

4. Totalität

Intuitiv gibt die Totalität eines Objektes an, dass dieses in gewisser Weise "endlich aufgebaut" ist. Beispielsweise sind die totalen Zahlen genau die natürlichen Zahlen im üblichen Verständnis. Da Algebren Parameter haben können, unterscheidet man grob zwischen zwei Arten von Totalität, der strukturellen Totalität und der absoluten Totalität.

4.1. Absolute Totalität

Intuitiv bedeutet Totalität dass man ein Objekt durch endlichfache Anwendung von Konstruktoren auf wiederum totale Objekte oder Funktionen von Objekten in totale Objekte erhält. Beispielsweise sind die totalen natürlichen Zahlen genau die natürlichen Zahlen im herkömmlichen Sinne, also die Zahlen, die durch endlichfache Anwendung des "Succ"-Konstruktors auf den "Zero"-Konstruktor erzeugbar sind.

Die absolute Totalität zu einem Typen lässt sich jeweils induktiv, und somit auch in Minlog definieren.

Für die genaue formale Definition sei auf [1] verwiesen.

4.2. Strukturelle Totalität

Die strukturelle Totalität bezeichnet Totalität bezüglich der Konstruktoren des Typs selbst, ohne die Totalität der Objekte, die aus Parametertypen stammen, zu beachten. Beispielsweise ist eine Liste von Zahlen absolut total wenn sie endlich lang ist und

jede in ihr enthaltene Zahl total ist, aber bereits als Liste strukturtotal wenn sie endlich ist. Auch die strukturelle Totalität lässt sich induktiv definieren. Für die genaue formale Definition sei auf [1] verwiesen.

4.3. Implementierung in Minlog

Die Implementierung der Totalität in Minlog befindet sich in `totality.scm`. Sie unterscheidet nicht direkt zwischen struktureller und absoluter Totalität, sondern geht grundsätzlich davon aus, dass man Totalität für alle angegebenen Typen haben will, das heißt alle Typen, die keine Typvariable darstellen. Beispielsweise wird für den Typen `list alpha` nur Strukturtotalität erzeugt, da `alpha` eine Typvariable ist, während für `list nat` absolute Totalität erzeugt wird. Um Strukturtotalität bezüglich der Listenstruktur einer `list nat` zu erzeugen muss man zunächst die Totalität für `list alpha` definieren, und erhält eine induktiv definierte Prädikatkonstante, die `alpha` als Typparameter hat, in den man anschließend `nat` einsetzen kann.

4.3.1. Namensschema

Bei der Definition von Totalitätsprädikaten ergibt sich das Problem, dass in Minlog jede Prädikatkonstante einen eindeutigen Namen braucht, die selbe Algebra aber viele verschiedene Totalitätsprädikate haben kann, außerdem werden auch Totalitätsprädikate für Pfeiltypen benötigt. Des weiteren besteht Interesse daran, dass möglichst jedes solche Totalitätskonstante nur einmal definiert wird.

Namen werden deshalb zentral gespeichert und deren Typen zugeordnet in Form einer assoc-list, die in der Variable `TOTALITY-PRED-NAMES` gespeichert wird.

Weiterhin werden die Namen durchnummeriert. Der Counter hierfür ist `TOTALITY-PRED-CNUM`. Da Bezeichner keine Zahlen enthalten dürfen, werden diese mittels der in Minlog vordefinierten Funktion `number-to-alphabetic-string` in Wörter umgewandelt.

Das Generieren der Namen erledigt die Funktion `totality-predicate-name`, die als Argument einen Typen erwartet. Sie erzeugt zunächst einen Namen entsprechend des Typen. Weiterhin sucht sie, ob zu dem entsprechenden Typen bereits ein Totalitätsprädikat existiert. Sollte es existieren, wird dessen Name zurückgegeben. Wenn nicht, wird der neu erzeugte Name gespeichert, und anschließend zurückgegeben.

4.3.2. Generelles Framework, Typvariablen, Pfeiltypen

Zur generellen Erzeugung von Totalitätsprädikaten für alle definierten Typen dient die Funktion `type-to-totality-idpredconst`. Sie erwartet als Argument einen Typen, für den ein Totalitätsprädikat definiert werden kann, wobei dies mit `type-has-totality-idpredconst` überprüft werden kann, und immer dann der Fall ist, wenn es sich entweder um eine Algebra selbst handelt, oder um einen Pfeiltypen, für dessen Werttyp ein Totaliätsprädikat definiert werden kann.

Die Definition der Totalität für Pfeiltypen $\alpha \to \beta$ lautet
$T_{\alpha\to\beta} := \mu_X((\forall x^\alpha T_\alpha(x) \to T_\beta(f^{\alpha\to\beta}(x))) \to Xf^{\alpha\to\beta})$, dabei bezeichnen die T_ξ jeweils die Totalitätsprädikate für ξ, sofern diese existieren. Falls T_α nicht definiert ist, definieren wir $T_{\alpha\to\beta} := \mu_X(\forall x^\alpha T_\beta f^{\alpha\to\beta} x^\alpha \to Xf^{\alpha\to\beta})$, ist T_β nicht definiert, ist $T_{\alpha\to\beta}$ auch nicht definiert. T_α ist auch dann nicht definiert, wenn α eine Typvariable ist.

Algebraformen α stellen einen Spezialfall dar, in dem diese Funktion die Funktion `alg-name-and-types-to-totality-idpredconst` aufruft, welche als erstes Argument einen Algebra-Namen, und anschließend die einzusetzenden Typen erwartet. Mit diesen wird mittels `make-alg` zunächst eine Algebraform erstellt.

Die Funktion `alg-form-to-typed-constr-names`, an die diese Algebraform übergeben wird, erwartet genau eine Algebraform als Argument, und liefert eine assoc-list zurück, die Konstruktornamen deren entsprechende Typen zuordnet.

```
> (alg-form-to-typed-constr-names (py "nat"))
(("Zero" (alg "nat")) ("Succ" (arrow (alg "nat") (alg "nat"))))
```

Jedem dieser typisierten Konstruktornamen kann man nun deren entsprechendes Einführungsaxiom des Totalitätsprädikats zuordnen. Hierzu dient die Funktion `alg-name-and-typed-constr-name-and-types-and-pvar-to-totality-clause`. Sie erwartet als Argument einen Algebranamen, einen typisierten Konstruktornamen, die entsprechenden eingesetzten Typen und eine Prädikatvariable, die anstelle des noch zu definierenden Totalitätsprädikats eingesetzt wird.

```
> (pp
  (alg-name-and-typed-constr-name-and-types-and-pvar-to-totality-clause
       "nat" '("Succ" (arrow (alg "nat") (alg "nat"))) '()
       (make-pvar (make-arity (py "nat")) -1 0 0 "TotalNatPvar")))
allnc n^2667(TotalNatPvar n^2667 -> TotalNatPvar(Succ n^2667))
```

Die so erzeugten Klauseln werden anschließend an die Minlog-Funktion `add-ids` übergeben, womit das Totalitätsprädikat erzeugt wird.

4.4. Tests

Um die Totalität für $\mathbb{N}\to\mathbb{N}$ zu erhalten, nutzen wir den Befehl `type-to-totality-idpredconst`.

```
> (map pp
   (map car
    (idpredconst-name-to-clauses-with-names
     (cadr (type-to-totality-idpredconst (py "nat=>nat"))))))
allnc (nat=>nat)^2342(
 allnc n^2335(TotalnatZero n^2335 ->
  TotalnatZero((nat=>nat)^2342 n^2335)) ->
   (Pvar nat=>nat)_513(nat=>nat)^2342)
```

In dieser Ausgabe steht `TotalnatZero` für die Totalität von $\mathbb{N}$. Weitere Tests finden sich in `tests.scm`.

4.5. Totalität in Coq

Für Induktive Typen in Coq werden automatisch Induktionsaxiome erzeugt, alle solchen Typen in Coq sind also Total. Totalitätsprädikate gelten also Implizit. Beispielsweise lassen sich Listen definieren durch

```
Inductive list (T:Type) : Type :=
 | Nil : list (T)
 | Cons : T -> list (T) -> list (T).
```

und es wird automatisch definiert

```
list_ind
     : forall (T : Type (* Top.55 *)) (P : list T -> Prop),
       P (Nil T) ->
       (forall (t : T) (l : list T), P l -> P (Cons T t l)) ->
       forall l : list T, P l
```

also ein Objekt das einen Typen und eine Aussage über Listen dieses Typs erwartet, und dann das Induktionsaxiom dafür liefert.

5. Realisierer-Interpretation

Wir ordnen nun jeder Formel A einen Typen $\tau(A)$ zu.

$$\begin{aligned}
\tau(A{\to}^{c}B) &:= \tau(A){\to}\tau(B)\\
\tau(A{\to}^{nc}B) &:= \tau(B)\\
\tau(\forall^{c}x^{\rho}A) &:= \rho{\to}\tau(A)\\
\tau(\forall^{nc}x^{\rho}A) &:= \tau(A)
\end{aligned}$$

Prädikatvariablen erhalten Typvariablen als zugeordneten Typen. Für rechnerisch irrelevante Prädikate I definieren wir $\tau(I) := \circ$, und definieren weiterhin $\circ{\to}\circ = \circ$, $\circ{\to}\rho = \rho$, $\rho{\to}\circ = \circ$. Für Prädikate mit rechnerischem Gehalt $\mu_X(K_1,\dots,K_n)$ definieren wir den Typen als $\mu_{\tau(X)}(\tau(K_1),\dots,\tau(K_n))$. Die zugehörige Algebra heißt *Zeugenalgebra*.

Zu den definierten Beweis-Termen definieren wir eine *Realisiererrelation* zwischen Termen und Formeln, durch strukturelle Rekursion. Für die Basisfälle definieren wir

$$\begin{aligned}
t\ r\ (A{\to}^{c}B) &:= \forall^{nc}x(x\ r\ A{\to}^{nc}tx\ r\ B)\\
t\ r\ (A{\to}^{nc}B) &:= \forall^{nc}x(x\ r\ A{\to}^{nc}t\ r\ B)\\
t\ r\ (\forall^{c}xA) &:= \forall^{nc}x(tx\ r\ A)\\
t\ r\ (\forall^{nc}xA) &:= \forall^{nc}x(t\ r\ A)
\end{aligned}$$

Für induktive Prädikate mit rechnerischem Gehalt definieren wir diese Relation über Zeugenprädikate. Jedem Prädikat I wird dabei ein Zeugenprädikat I^r zugeordnet.

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

Das *Zeugenprädikat* I^r zu I wird definiert durch die Prädikatkonstante, die für jedes Einführungsaxiom

$$\forall\vec{x}(\vec{A}{\to}(\forall\vec{y}_\nu(\vec{B}_\nu{\to}X\vec{s}_\nu))_\nu{\to}^{c}X\vec{t})$$

von I das Einführungsaxiom

$$\forall^{nc}\vec{x},\vec{u},\vec{f}(\vec{u}\ r\ \vec{A}{\to}^{nc}(\forall^{nc}\vec{y}_\nu\vec{v}_\nu(\vec{v}_\nu\ r\ \vec{B}_\nu{\to}^{nc}I^{r}(f_\nu\vec{y}_\nu\vec{v}_\nu,\vec{s}_\nu)))_\nu{\to}I^{r}(C\vec{x}\vec{u}\vec{f},\vec{t}))$$

haben soll, wobei nur die x_j hinter einem $\forall^c$ und die u_i mit rechnerisch relevantem A_i vor einem $\to^c$ in K als Argumente in $C\vec{x}\vec{u}\vec{f}$ vorkommen sollen, analog für $\vec{y}_\nu$, $\vec{v}_\nu$ und $f_\nu\vec{y}_\nu\vec{v}_\nu$.

5.1. Zeugenprädikate in Minlog

Zeugenprädikate zu implementieren ist erschwert, wenn die Prädikatkonstante freie Typparameter und Prädikatparameter hat, die wiederum von diesen Typparametern

abhängen, ein Beispiel ist der Existenzquantor $\mu_X(\forall^c x^\rho P(x^\rho) \to X)$. Für solche Fälle wurden Zeugenprädikate nicht implementiert.

Da Zeugenprädikate einer sehr komplexen syntaktischen Transformation bedürfen, wurde die Implementierung in viele kleine Teile aufgeteilt. Die Implementierung befindet sich in `witnessingpreds.scm`.

Das Namensschema ist ähnlich dem der Totalitätsprädikate (Siehe 4.3.1), mit dem Unterschied, dass keine Typnamen hinzugefügt werden, da sonst die Namen teils sehr unübersichtlich lang werden. Verwaltet werden die Namen Analog in den Variablen `WITNESSING-PRED-CNUM` und `WITNESSING-PRED-NAMES`, und mit der Funktion `witnessing-predicate-name`.

Es folgen nun einige Funktionen, die die Variablen, über die in den Axiomen der Realisiererrelationen quantifiziert werden muss, zurückgeben. Die Benennungen der Funktionen orientieren sich dabei an den Bezeichnungen in [1] bzw. 5.1.

Die Funktion `idpredconst-clause-to-all-allnc-vars-x` erwartet als Argument ein Einführungsaxiom einer induktiv definierten Prädikatkonstante, und liefert eine Funktion zurück. Die zurückgelieferte Funktion erwartet als Argument eine Formel, und fügt ihr nichtrechnerisch allquantifiziert alle Variablen hinzu, über die das Einführungsaxiom quantifiziert ist. `idpredconst-clause-to-all-vars-x` gibt eine Liste der Variablen darunter zurück, die mit einem rechnerisch relevanten Allquantor quantifiziert wurden.

```
> (pp ((idpredconst-clause-to-all-allnc-vars-x (pf "Even 0"))
       (pf "A")))
A
> (pp
   ((idpredconst-clause-to-all-allnc-vars-x
     (pf "allnc n^. Even n^ -> Even (Succ(Succ(n^)))"))
     (pf "A")))
allnc n^ A
```

Die Funktion `idpredconst-clause-to-all-allnc-vars-u` liefert eine Liste von Paaren zurück, deren erstes Element jeweils eine Variable und deren zweites Element jeweils ein generalized boolean ist, das angibt, ob die Variable rechnerisch relevant ist. Die Variablen werden dabei mittels `(lambda (A) (type-to-new-partial-var (formula-to-et-type A)))` aus den nichtrekursiven Prämissen des jeweiligen Axioms erzeugt. Diese Variablen werden in [1] bzw. 5.1 mit u bezeichnet.

```
> (write (idpredconst-clause-to-all-allnc-vars-u
    (pf "A -> B -> (AndD (cterm () A) (cterm () B))")
    "AndD"))
(((var (tvar 72 "alpha") 2347 0 "") . #t)
 ((var (tvar 82 "alpha") 2348 0 "") . #t))
```

Analoges macht die Funktion `idpredconst-clause-to-all-allnc-vars-f` mit den rekursiven Prämissen, auch hier werden die Variablen in [1] bzw. 5.1 mit `f` bezeichnet.

```
> (idpredconst-clause-to-all-allnc-vars-f
    (pf "allnc n^. Even n^ -> Even (Succ(Succ n^))")
    "Even")
(((var (alg "algEven") 2335 0 "") . #t))
```

Die erste aufwendige Umformung erledigt `recursive-idpc-premise-to-witnessing-idpc-premise`, welche dazu dient, zu einer rekursiven Prämisse eines Einführungsaxioms die entsprechende Prämisse des entsprechenden Einführungsaxioms des Zeugenprädikats zu berechnen. Die Funktion erwartet fünf Argumente. Das erste Argument `premise` ist dabei die Prämisse, das zweite Argument `idpredconst-name` ist der Name der zugehörigen Prädikatkonstante. Das dritte Argument `realizer-f` soll die mit `idpredconst-clause-to-all-allnc-vars-f` berechnete Variable sein. Das vierte Argument `realizer-predicate-gen` soll eine Funktion sein, die zu einem gegebenen Prädikat das Realisierungsprädikat zurückliefert. `witnessing-predicate-pvar` soll eine Prädikatvariable der Stelligkeit sein, zu der die noch zu erzeugende induktive Prädikatkonstante gehört, und wird anstelle von dieser benutzt.

Diese Funktion wird von `idpredconst-clause-to-witnessing-idpredconst-clause` verwendet, welches ebenfalls fünf Argumente erwartet, und ein Einführungsaxiom in das entsprechende Einführungsaxiom des Zeugenprädikats umwandeln soll. Das erste Argument `clause` soll das Einführungsaxiom sein, das zweite Argument `idpredconst-name` der Name der Prädikatkonstante, `constructor-term` der Term des zum Einführungsaxiom gehörenden Konstruktors der entsprechenden realisierenden Algebra, `realizer-predicate-gen` und `witnessing-predicate-pvar` analog zu `recursive-idpc-premise-to-witnessing-idpc-premise`.

```
> (add-pvar-name "X" (make-arity (py "algEven") (py "nat")))
; ok, predicate variable X: (arity algEven nat) added
> (pp
   (idpredconst-clause-to-witnessing-idpredconst-clause
    (pf "allnc n^.Even n^ -> Even(Succ (Succ n^))")
    "Even" (pt "cGenEven") (lambda (P t) (pf "A"))
    (cadr (pf "X cInitEven 0"))))
allnc n^,algEven^2336(
 X algEven^2336 n^ --> X(cGenEven algEven^2336)(Succ(Succ n^)))
```

Um an die Einführungsaxiome zu kommen, wird `idpredconst-to-clauses-with-names` aufgerufen. Es erwartet eine

Prädikatkonstante, einen Variablen-Benennungskontext und einen Prädikatvariablen-Benennungskontext als Argumente. Die Kontexte werden zur Substitution gebraucht, um sicherzugehen, dass gleichen Prädikatvariablen immer gleiche Typvariablen zugeordnet werden. Für eine genauere Besprechung sei auf [2] verwiesen. `idpredconst-to-alg` berechnet die realisierende Algebra (mit allen Typsubstitutionen) zu einer induktiv definierten Prädikatkonstante.

Schließlich berechnet, definiert und liefert `idpredconst-to-mr-idpredconst` die Zeugenprädikatkonstante der ihr übergebenen Prädikatkonstante.

```
> (idpredconst-to-mr-idpredconst (make-idpredconst "Even" '() '()))
; ok, inductively defined predicate constant WitEvenOne added
(idpredconst "WitEvenOne" () ())
> (idpredconst-to-mr-idpredconst (make-idpredconst "Even" '() '()))
(idpredconst "WitEvenOne" () ())
```

5.2. Realisierer-Relation in Minlog

Die Implementierung der Realisiererrelation in Minlog ist, sobald man Zeugenprädikate hat, straightforward, und wird in der Funktion `form-to-realizing-form` realisiert, die zwei Argumente `form` - die Formel - und `term` - den realisierenden Term - erwartet. Sie befindet sich in der Datei `witnessingpreds.scm`.

Es sei bemerkt dass es in Minlog bereits eine Funktion `real-and-formula-to-mr-formula` gibt, die prinzipiell das Gleiche leistet. Allerdings erzeugt diese Aussagen mit totalen Variablen, wo zur Erzeugung von Korrektheitsbeweisen partielle Variablen benötigt werden.

5.3. Einschränkungen

Die Realisierer-Relation ist nicht implementiert für Prädikatvariablen, deren Axiome $\rightarrow^{nc}$ und $\forall^{nc}$ enthalten. Sie funktioniert in den meisten Fällen, wurde aber nicht dafür implementiert und getestet. Außerdem funktioniert sie nicht in Fällen, wo Komprehensionsterm-Parameter von Typparametern abhängen, zum Beispiel im Fall des allgemeinen Existenzquantors.

5.4. Realisierungsbeweis

Zur Erzeugung von Korrektheitsbeweisen ist es von Interesse, beweisen zu können, dass Konstruktoren einer Realisierer-Algebra die Einführungsaxiome realisieren, und

die Rekursionsoperatoren die Eliminationsaxiome realisieren. Letzteres ist aufwendig, und wurde für viele Fälle in der Funktion `proof-that-recop-realizes-elim` implementiert. Sie implementiert grundsätzlich den Beweis aus [1], wobei an einigen Stellen etwas kompliziertere, aber gleichbedeutende Teil-Beweise genutzt werden.

Der Beweis benutzt ein Zusammenspiel aus dem Eliminationsaxiom und der Realisiererformel, allerdings arbeitet er mit umgeformten Realisiererformeln, bei denen die Quantoren jeweils nach vorne geshiftet sind. Dieses Shiften der Allquantoren erledigt `forward-shift-all-quantifiers`. Es erwartet als Argument zunächst den Beweis einer eventuell allquantifizierten Implikationskette, und eine Anzahl, die angibt, über wie viele Implikationen hinweg die Funktion Quantoren shiftet (dies wird gebraucht, da an einigen Stellen der letzte Teil einer formalen Implikationskette von Aussagen die unverändert bleiben sollen wieder eine Implikation ist).

```
> (cdp (forward-shift-all-quantifiers
        (make-proof-in-avar-form
         (formula-to-new-avar
          (pf "allnc x^ . A -> allnc y^ . B -> allnc z^ . C")))
        2))
[...]
; allnc x^2380,y^2383,z^2384(A -> B -> C) by allnc intro
> (cdp (forward-shift-all-quantifiers
        (make-proof-in-avar-form
         (formula-to-new-avar
          (pf "allnc x^ . A -> allnc y^ . B -> allnc z^ . C")))
        1))
[...]
; allnc x^2378,y^2379(A -> B -> allnc z^ C) by allnc intro
```

Ein weiteres Problem das sich stellt ist, dass bei Eliminationen bisweilen der rechnerische Gehalt nicht zusammenpasst. Um diesem Problem zu begegnen, wurde die Funktion `unify-computational-relevance` definiert. Sie erwartet zwei Argumente A und B, die sich bis auf rechnerischen Gehalt der Implikationen und Quantoren nicht unterscheiden, und liefert, falls möglich, einen Beweis für $A \to B$.

```
> (cdp (unify-computational-relevance
        (pf "allnc x^. A --> B --> C")
        (pf "all x^. A -> B -> C")))
; ..........C by assumption u1053
; .........C -> C by imp intro u1053
; ..........B --> C by assumption u1050
; ..........B by assumption u1051
; .........C by impnc elim
```

```
; ........C by imp elim
; .......B -> C by imp intro u1051
; ......(B --> C) -> B -> C by imp intro u1050
; .......A --> B --> C by assumption u1048
; .......A by assumption u1049
; ......B --> C by impnc elim
; .....B -> C by imp elim
; ....A -> B -> C by imp intro u1049
; ...(A --> B --> C) -> A -> B -> C by imp intro u1048
; ....allnc x^(A --> B --> C) by assumption u1047
; ....x^
; ...A --> B --> C by allnc elim
; ..A -> B -> C by imp elim
; .all x^(A -> B -> C) by all intro
; allnc x^(A --> B --> C) -> all x^(A -> B -> C) by imp intro u1047
```

5.5. Dokumentation von proof-that-recop-realizes-elim

Die Bezeichnungen wurden aus dem Beweis von [1] übernommen. Die Funktion erwartet als Argumente den Rekursionsoperator und das Eliminationsaxiom, und berechnet zunächst die eigentliche Formel und das Prädikat, sowie dessen Argumente.

```
(define (proof-that-recop-realizes-elim recop elim-axiom)
  (let*
      ((elim-form (aconst-to-formula elim-axiom))
       (idpc-with-argvars
(imp-impnc-form-to-premise
 (all-allnc-form-to-final-kernel elim-form)))
       (idpc (predicate-form-to-predicate idpc-with-argvars))
       (idpc-argvars
(map term-in-var-form-to-var
     (predicate-form-to-args idpc-with-argvars)))
```

Weiterhin wird eine Schablone erzeugt, um einen Ansatzpunkt zu haben, betreffende Teile einer Formel zu berechnen.

```
       (elim-constraint-aconst
(imp-formulas-to-elim-aconst
 (make-imp (make-predform idpc)
   (pf "A"))))
       (elim-constraint (aconst-to-formula elim-constraint-aconst))
```

Anschließend wird das Zeugenprädikat berechnet.

```
       (wit-idpc (idpredconst-to-mr-idpredconst idpc))
```

Nun wird ein Komprehensionsterm berechnet, der angibt, welche Aussage aus dem Eliminationsaxiom folgt.

```
       (p-cterm
(apply make-cterm
       (append
```

Die Argumentvariablen sollen dabei die sein, die auch der Prädikatkonstante übergeben wurden.

```
idpc-argvars
```

Die Schablone wird verwendet, um an die Aussage zu kommen. Man erhält sie, indem man alle Allquantoren und Implikations-Prämissen entfernt, sofern sie selbst keine Implikation ist. Ist sie selbst eine Implikation würde man zu viele Prämissen entfernen. Deshalb arbeitet die Funktion mit der Schablone, von der sie weiß, dass es sich nicht um eine Implikation handelt. Die Schablone und die eigentliche Aussage werden hierbei parallel destrukturiert.

```
(list
 (let recur ((theimp
      (all-allnc-form-to-final-kernel elim-form))
     (constraint
      (all-allnc-form-to-final-kernel
       elim-constraint)))
   (if (imp-impnc-form? constraint)
       (recur
(imp-impnc-form-to-conclusion theimp)
(imp-impnc-form-to-conclusion constraint))
       theimp))))))
```

Auf vergleichbare Weise werden die Argumenttypen des Rekursionsoperators berechnet.

```
       (arg-types
       ;; calculate the arg-types. the first of them should be the
       ;; type of the inductive predicate's algebra. we will mostly
       ;; not want it.
(let rec ((type-constraint
   (formula-to-et-type elim-constraint))
  (remaining (term-to-type recop)))
```

```
  (if (arrow-form? type-constraint)
      (cons (arrow-form-to-arg-type remaining)
    (rec (arrow-form-to-val-type type-constraint)
 (arrow-form-to-val-type remaining)))
      '()))
       (ws (map type-to-new-partial-var arg-types))
       (w-terms (map make-term-in-var-form ws))
```

Im Beweis von [1] kommt vor $Qw\vec{x} = Rw\vec{w}$ r $P\vec{x}$. Selbiges berechnet die folgende Definition:

```
       (q-pred (lambda (w xs)
 (form-to-realizing-form
  (apply cterm-apply p-cterm xs)
  (let rec ((ret (make-term-in-app-form recop w))
    (wt (cdr w-terms)))
    (if (and (pair? wt))
(rec
 (make-term-in-app-form ret (car wt))
 (cdr wt))
ret)))))
```

Weiterhin werden die Eliminationsaxiome des Zeugenprädikats berechnet und deren Argumentvariablen gespeichert.

```
       ;; Elim-Axiom for witnessing predicate wit-idpc
       ;; we have to save the X-es used to allnc-intro later.
       (xs-vars
(map type-to-new-partial-var
     (cdr
      (arity-to-types
       (idpredconst-to-arity wit-idpc)))))
       (xs (map make-term-in-var-form xs-vars))
       (the-xs xs)
       (elim-ir
(imp-formulas-to-elim-aconst
 (make-imp
  (apply make-predicate-formula wit-idpc (car w-terms) xs)

  (q-pred (car w-terms) xs))))
       (elim-ir-form (aconst-to-formula elim-ir))
```

Weiterhin werden die Teilklauseln $K_i^r(I^r, Q)$ benötigt im Beweis von [1]. Diese werden hier berechnet. Dabei wird anhand einer Schablone das Eliminationsaxiom destrukturiert. Zunächst werden alle Allquantoren eliminiert die führend sind, und

das führende Prädikat wird entfernt. Anschließend werden die Prämissen der Implikationskette gesammelt, welche unsere Teilklauseln darstellen.

```
       (the-k-i
(let recur ((theimp
     (imp-impnc-form-to-conclusion
      (all-allnc-form-to-final-kernel elim-ir-form)))
    (constraint
     (imp-impnc-form-to-conclusion
      (all-allnc-form-to-final-kernel
       elim-constraint)))
    (ret '()))
  (if (imp-impnc-form? constraint)
      (recur
       (imp-impnc-form-to-conclusion theimp)
       (imp-impnc-form-to-conclusion constraint)
       (cons (imp-impnc-form-to-premise theimp) ret))
      (reverse ret))))
```

Analog berechnen wir w_i r $K_i(I, P)$ aus dem Beweis, indem wir anhand der Schablone die entsprechenden Prämissen $K_i(I, P)$ aus dem Eliminationsaxiom extrahieren, und anschließend die Realisierer-Relation bilden.

```
       (the-other-k-i-clean
(let recur ((theimp
     (imp-impnc-form-to-conclusion
      (all-allnc-form-to-final-kernel elim-form)))
    (constraint
     (imp-impnc-form-to-conclusion
      (all-allnc-form-to-final-kernel
       elim-constraint)))
    (ret '()))
  (if (imp-impnc-form? constraint)
      (recur
       (imp-impnc-form-to-conclusion theimp)
       (imp-impnc-form-to-conclusion constraint)
       (cons (imp-impnc-form-to-premise theimp) ret))
      (reverse ret))))
       (the-other-k-i
(map form-to-realizing-form
     the-other-k-i-clean
     (cdr w-terms)))
```

Bis auf äquivalentes herumshiften von Allquantoren sind nun `the-k-i` und `the-other-k-i` jeweils gleich. Um auch diesen Unterschied zu ändern nutzen wir

jetzt unsere Funktion forward-shift-all-quantifiers. Wir speichern allerdings die aktuellen Aussagen als Annahmevariablen, weil wir später mit ihnen Implikationen eliminieren müssen.

```
       (the-other-k-i-avars
(map formula-to-new-avar the-other-k-i))
       (the-other-k-i-2-proofs
        ;; ok, first remove (and re-add afterwards) the universal
        ;; quantifiers in the beginning. we will replace them by new
        ;; variables, to make sure they will not produce any problems.
(map (lambda (x type-constraint)
       (let ((avarp
      (make-proof-in-avar-form x)))
 (if (arrow-form? type-constraint)
     (forward-shift-all-quantifiers
      avarp (- (length
     (arrow-form-to-arg-types
      type-constraint)) 2))
     avarp)))
     the-other-k-i-avars
     (cdr (arrow-form-to-arg-types
   (formula-to-et-type elim-constraint)))))
       (the-other-k-i-2 (map proof-to-formula the-other-k-i-2-proofs))
```

Die w_i r $K_i(I, P)$ enthalten allquantifizierte Variablen $\vec{g}$, in die Terme der Form $(\lambda_{\vec{y_\nu}, \vec{\nu_\nu}} R(f_\nu \vec{y_\nu} \vec{\nu_\nu}) \vec{w})_{\nu<n}$ eingesetzt werden müssen. Diese Terme generieren wir und speichern sie in `the-g`. Wir ermitteln zunächst die Anzahl der allquantifizierten $\vec{g}$, indem wir die Anzahlen der allquantoren vor w_i r $K_i(I, P)$ und $K_i^r(I^r, Q)$ subtrahieren, und genau diese Anzahl an allquantifizierten Variablen, von hinten beginnend, aus $K_i^r(I^r, Q)$ benutzen, um an die Argumente $\vec{f}$ zu gelangen.

```
       (the-g
(map ;; the-other-k-i-2 the-k-i
       (lambda (k tk)
 (let* ((all-args-tk (all-allnc-form-to-vars tk))
(r-num (length all-args-tk))
(all-args (all-allnc-form-to-vars k))
(a-num (length all-args))
(f-num (- a-num r-num))
(f-s (last all-args-tk f-num))
(f-terms (map make-term-in-var-form f-s))
(f-types (map term-to-type f-terms))
(f-argtypes
 (map arrow-form-to-arg-types f-types))
(f-argtype-vars
 (map (lambda (x)
```

```
(map type-to-new-partial-var x))
      f-argtypes))
(f-argtype-terms
 (map (lambda (x)
(map make-term-in-var-form x))
      f-argtype-vars)))
```

Anschließend generieren wir damit die Terme $(\lambda_{\vec{y_\nu},\vec{\nu_\nu}} R(f_\nu \vec{y_\nu} \vec{\nu_\nu})\vec{w})_{\nu<n}$.

```
   (map ;; f-terms f-argtype-vars f-argtype-terms
    (lambda (f-term f-argtv f-argte)
      (let rec ((r (apply
    mk-term-in-app-form recop
    (apply mk-term-in-app-form f-term
   f-argte)
    (cdr w-terms)))
(vs f-argtv))
(if (pair? vs)
    (make-term-in-abst-form (car vs)
    (rec r (cdr vs)))
    r)))
    f-terms f-argtype-vars f-argtype-terms)))
       the-other-k-i-2
       the-k-i))
```

Anschließend eliminieren wir die Allquantoren entsprechend des Beweises mit $\vec{x}$,$\vec{u}$,$\vec{f}$ und den Termen in `the-g`.

```
       (allnc-eliminated
(map
 (lambda (kp ; <- the-other-k-i-2-proofs
  k  ; <- the-other-k-i-2, just for efficiency
  ki ; <- the-k-i
  g  ; <- the-g
 )
   (let recur ((kp kp) (k k) (ki ki) (g g) (ai id))
     (cond
      ((all-allnc-form? ki)
       (recur
(mk-proof-in-elim-form
 kp (make-term-in-var-form
     (all-allnc-form-to-var ki)))
(all-allnc-form-to-kernel k)
(all-allnc-form-to-kernel ki)
g
```

```
(lambda (x) (ai ((if (all-form? ki)
     make-proof-in-all-intro-form
     make-proof-in-allnc-intro-form)
 (all-allnc-form-to-var ki) x)))))
      ((pair? g)
       (recur
(mk-proof-in-elim-form kp (car g))
k ki (cdr g) ai))
      (#t (ai kp))))) the-other-k-i-2-proofs the-other-k-i-2
      the-k-i the-g))
```

Beim schlussendlichen Eliminieren tritt das Problem auf dass die geshifeten Variablen nicht notwendig geordnet in $\vec{x}$ und $\vec{u}$, auftreten. Dementsprechend werden die Argumente des Realisiererprädikats in einer Liste gespeichert, um später zu erkennen, welche Variablen womit eliminiert werden müssen.

```
       (realizing-form (form-to-realizing-form elim-form recop))
       (the-alien-xs-terms
(cdr
 (predicate-form-to-args
  (imp-impnc-form-to-premise
   (all-allnc-form-to-final-kernel elim-ir-form)))))
       (dont-terminate #t)
       (the-alien-xs
(map term-in-var-form-to-var the-alien-xs-terms))
```

Zunächst werden also alle Allquantoren eliminiert, wobei Fallunterschieden wird, ob die betreffende Variable zu `the-alien-xs-terms` gehört, also $\vec{x}$, oder zu $\vec{u}$, und dementsprechend eingesetzt wird. Sind alle Allquantoren aufgebraucht, wird das Flag `dont-terminate` gesetzt, um zu verhindern, dass - sollte die zu folgernde Formel allquantifiziert sein - dort am Ende weiter-eliminiert wird.

```
       (realizing-form-proof
(let recur ((elim-irp (make-proof-in-aconst-form elim-ir))
    (elim-irf elim-ir-form) ;; for efficienty
    (xs xs)
    (w w-terms)
    (ae
     (cons
      (make-proof-in-avar-form
       (formula-to-new-avar
(form-to-realizing-form
 (apply make-predicate-formula idpc xs)
 (car w-terms))))
      allnc-eliminated)))
```

```
  (cond
   ((and dont-terminate (all-allnc-form? elim-irf))
    (let ((m (member (all-allnc-form-to-var elim-irf)
     the-alien-xs)))
      (recur
       (mk-proof-in-elim-form elim-irp
      (if m (car xs) (car w)))
       (all-allnc-form-to-kernel elim-irf)
       (if m (cdr xs) xs)
       (if m w (cdr w))
       ae)))
   ((pair? ae)
    (set! dont-terminate #f)
```

Nach diesem Schritt setzen wir die Terme für $\vec{g}$ ein. An dieser Stelle normalisierte Minlog ursprünglich bei der Implikationselimination nicht automatisch. Somit wurde der interaktive Beweisassistent an dieser Stelle als Workaround verwendet.

```
    (let*
((second-proof
  (mk-proof-in-elim-form
   (unify-computational-relevance
    (proof-to-formula (car ae))
    (imp-impnc-form-to-premise
     (proof-to-formula elim-irp)))
   (car ae)))
 (second-proof-form (proof-to-formula second-proof))
 (third-proof
  (let ((ocf COMMENT-FLAG))
                    (set! COMMENT-FLAG #f)
                    (set-goal
     (make-imp second-proof-form
       (imp-impnc-form-to-premise
(proof-to-formula elim-irp))))
                    (auto)
                    (set! COMMENT-FLAG ocf)
    (elim-alls (current-proof))))
 (fourth-proof
  (make-proof-in-imp-elim-form
   third-proof second-proof)))
      (recur
       (mk-proof-in-elim-form elim-irp fourth-proof)
       (imp-impnc-form-to-conclusion elim-irf)
     '() '() (cdr ae))))
```

Schlussendlich werden nun die Klauseln zu einer Implikationskette zusammengefügt und mit Allquantoren versehen, um die Realisiererformel nachzuweisen.

```
   (#t
    (let recur2 ((xsv xs-vars))
      (if (pair? xsv)
  (make-proof-in-allnc-intro-form
   (car xsv)
   (recur2 (cdr xsv)))
  (make-proof-in-allnc-intro-form
   (car ws)
   (make-proof-in-impnc-intro-form
    (formula-to-new-avar
     (form-to-realizing-form
      (apply make-predicate-formula idpc the-xs)
      (car w-terms)))
    (let recur
((wsv (cdr ws)) (kia the-other-k-i-avars))
      (if (pair? wsv)
  (make-proof-in-allnc-intro-form
   (car wsv)
   (make-proof-in-impnc-intro-form
    (car kia)
    (recur (cdr wsv) (cdr kia))))
  elim-irp))))))))))
    realizing-form-proof))
```

5.6. Einschränkungen von proof-that-recop-realizes-elim

Die Implementierungen für Einführungs- und Eliminationsaxiome sind nur für Prädikatkonstanten vorgesehen, die vollständig rechnerisch sind. Teilweise funktionieren sie auch für andere Beweise, da gewisse Annahmen dennoch zutreffen, dies hängt aber von den jeweiligen Interna ab. Um die Funktionen zu verallgemeinern sind auch entsprechende Modifikationen an den Implementierungen von Zeugenprädikaten und der Realisiererrelation notwendig.

5.7. Tests

Die Tests hierfür haben eine sehr umfangreiche Ausgabe, es sei auf die Source-Datei `tests.scm` verwiesen, die unter Anderem den folgenden Test enthält:

```
(define *elim-aconst-3*
```

```
  (imp-formulas-to-elim-aconst
   (pf "TotalNatCR n^ -> Even n^ -> (ExistsNat (cterm (m^) Twice m^ n^))")))

(define *elim-formula-3* (aconst-to-formula *elim-aconst-3*))
(define *elim-proof-3* (make-proof-in-aconst-form *elim-aconst-3*))
(define *elim-recop-3* (proof-to-extracted-term *elim-proof-3*))

(cdp (proof-that-recop-realizes-elim *elim-recop-3* *elim-aconst-3*))
```

6. Einfache Beweise und Definitionen - die Datei common.scm

In der Datei `common.scm` werden einige kleinere Hilfsfunktionen definiert, die im sonstigen Code gebraucht werden, aber der Übersichtlichkeit halber dort nicht gespeichert werden.

Zunächst werden einige Variablennamen deklariert, und die identische Funktion `id`, die, gerade als Anfangspunkt für Rekursionen, oft sinnvoll ist.

Weiterhin werden erzeugende Funktionen für Beweise für Aussagen der Form $A \to A$, $A \to B \to A$ und $A \to B, B \to C \vdash A \to C$ definiert, die später gebraucht werden.

```
> (cdp (a-from-a (pf "A")))
; .A by assumption u1035
; A -> A by imp intro u1035
> (cdp (imp-trans (make-proof-in-avar-form
   (formula-to-new-avar (pf "A -> B")))
  (make-proof-in-avar-form
   (formula-to-new-avar (pf "B -> C")))))
; ..B -> C by assumption u1039
; ...A -> B by assumption u1038
; ...A by assumption u1040
; ..B by imp elim
; .C by imp elim
; A -> C by imp intro u1040
> (cdp (make-proof-in-a-imp-b-imp-a-form (pf "A") (pf "B")))
; ..A by assumption u1043
; .B -> A by imp intro u1042
; A -> B -> A by imp intro u1041
> (cdp (make-proof-in-a-impnc-b-impnc-a-form (pf "A") (pf "B")))
; ..A by assumption u1046
; .B --> A by impnc intro u1045
; A --> B --> A by impnc intro u1044
```

Die Funktion `remove-all-quantors` erwartet als Argument eine Formel, von der es alle führenden Allquantoren entfernt, sowohl rechnerisch relevante als auch rechnerisch irrelevante Allquantoren sind betroffen.

```
> (pp (remove-all-quantors (pf "allnc x. all y. allnc z,t. A")))
A
```

Vergleichbares tut `elim-alls` mit Beweisen. Es erwartet einen Beweis als Argument, und optional beliebig viele Terme als Argumente. Dann eliminiert es alle vorhandenen Allquantoren der bewiesenen Aussage mit den übergebenen Termen, solange diese vorhanden sind, und ansonsten mit den Variablentermen zu den quantifizierten Variablen selbst. Eine vergleichbare Funktion namens `mk-proof-in-elim-form` wird zwar von Minlog bereitgestellt, allerdings eliminiert diese nur so viele Quantoren, wie Terme übergeben werden.

```
> (add-pvar-name "Z"
   (make-arity(py"alpha")(py"alpha")(py"alpha")(py"alpha")))
; ok, predicate variable Z: (arity alpha alpha alpha alpha) added
> (cdp
   (elim-alls
    (make-proof-in-avar-form
     (formula-to-new-avar
      (pf "allnc x^,y^,z^,t^ . Z x^ y^ z^ t^"))) (pt "alpha^")))
; ....allnc x^,y^,z^,t^ Z x^ y^ z^ t^ by assumption u1041
; ....alpha^
; ...allnc y^,z^,t^ Z alpha^ y^ z^ t^ by allnc elim
; ...y^
; ..allnc z^,t^ Z alpha^ y^ z^ t^ by allnc elim
; ..z^
; .allnc t^ Z alpha^ y^ z^ t^ by allnc elim
; .t^
; Z alpha^ y^ z^ t^ by allnc elim
```

`my-cterm-substitute` und `my-formula-substitute` sind Analoga zu den Minlog-Funktionen `cterm-substitute` und `formula-substitute`, die Benennungskontexte akzeptieren. Dies wird für die Korrektheitsbeweise benötigt.

Die Funktion `cterm-apply` ist eine Convenience-Funktion zum Einsetzen von Werten in Komprehensionsterme. Sie erwartet als erstes Argument einen Komprehensionsterm, und anschließend entsprechend viele einsetzbare Werte.

```
> (pp (cterm-apply (make-cterm
    (pv "m") (pv "n") (pv "k")
    (pf "M k n m"))
```

```
   (pt "1") (pt "2") (pt "3")))
M 3 2 1
```

Die Funktion `last` ist in Petite Chez Scheme vordefiniert, in PLT Scheme nicht, deshalb wurde sie entsprechend definiert. Sie erwartet als erstes Argument eine Liste, und als zweites Argument die Anzahl der Elemente, die sie - von hinten beginnend - von der Liste zurück gibt.

```
> (last '(1 2 3 4 5) 3)
(3 4 5)
```

Bei `make-predform` handelt es sich ebenfalls nur um eine Convenience-Funktion, die aus einem Prädikat eine Prädikatform macht, die autogenerierte Variablenterme als Argumente bekommt.

```
> (pp (make-predform (make-idpredconst "Even" '() '())))
Even n^2455
```

A. Graphviz-Ausgabe von Beweisbäumen

Graphviz ist ein System zum erzeugen von Diagrammen. Es eignet sich damit auch in gewissen Fällen für die Darstellung von Beweisbäumen. In der Datei `graphviz.scm` werden Funktionen definiert, die einen Quellcode aus einem Beweis erzeugen, der mit Graphviz in einen Graphen umgeformt werden kann. Graphviz wird hier nur oberflächlich eingeführt, für eine genauere Besprechung sei auf die Dokumentation in [3] verwiesen.

A.1. Der Quellcode

Der Quellcode arbeitet mit einem Vector mit acht Elementen, der in `make-node-context` erzeugt wird. Da jedes Objekt in Graphviz eine eigene ID braucht, wird als erstes Element dieses Vectors eine Zahl gespeichert, die benutzt wird, um IDs zu vergeben, und jedes mal erhöht wird, wenn eine ID benötigt wurde. Ansonsten werden Listen von Knoten verschiedener Bedeutungen gespeichert, zum Beispiel freie und gebundene Annahmevariablen und Annahmekonstanten. Weiterhin Knoten für einstellige und zweistellige Ableitungen.

Die darauf folgenden Funktionen dienen dazu, die entsprechenden Kontexte zu erzeugen und zu modifizieren. Entgegen des funktionalen Paradigmas wurde hier mit Zuständen gearbeitet, da sich dies als erheblich übersichtlicher herausstellte.

Graphviz erlaubt Labels, die HTML-Code enthalten. HTML-Code kann Unicode-Zeichen enthalten, dafür wurde die Funktion `formula-to-graphviz-html` erzeugt, die mit Unicode-Entities wie `&forall;` und `&#8319;` in den HTML-Code entsprechende mathematische Symbole einfügt. An dieser Stelle wäre freilich eine LaTeX-Ausgabe schöner gewesen, diese wäre aber ungleich aufwendiger.

```
> (formula-to-graphviz-html (pf "allnc x^ . all y^ . A -> B --> C"))
"&forall;&#8319;x^(&forall;y^(A&#x2192;B&#x2192;&#8319;C))"
```

Die Funktion `node-context-to-graphviz-code` erwartet als Argument Namen und Label des Beweises, da dies Parameter sind, die Graphviz haben will. Sie können getrost auf leere Strings gesetzt werden. Außerdem wird ein `node-context` erwartet, der vorher aus dem Beweis erzeugt wurden, und eine `finalnode`, das heißt das Element, das am Ende hergeleitet werden soll.

Aussagen werden in verschiedenen Umrandungen dargestellt, je nach Bedeutung in dem dargestellten Beweis. Diese werden mittels dem `shape`-Argument im Graphviz-Code angegeben. Gebundene Annahmevariablen erhalten hier `box3d`, eine Box mit 3d-Effekt in der Umrandung, freie Annahmevariablen werden - um sie besonders hervorzuheben, da sie ein Indiz für einen unvollständigen Beweis sein können - mit `egg` ausgestattet, ein leicht verzerrtes Oval. Annahmekonstanten erhalten ebenfalls eine `box3d`, diese ist aber zusätzlich grau gefüllt, um sie von den Annahmevariablen abzugrenzen.

Einen weiteren Knotentyp stellen die Knoten mit dem Shape `diamond` dar. Diese werden benutzt, um binäre Inferenzen hervorzuheben, als $\forall+$ und $\rightarrow+$.

Pfeile werden zum Einen genutzt um Ableitungsrichtungen anzugeben, und um Bindungen bei Annahmevariablen anzugeben. Letztere werden nur gepunktet dargestellt. Das `weight`-Argument der Ableitungspfeile wird sehr groß gesetzt, für die anderen Pfeile sehr klein, damit die von Graphviz erzeugten Graphen vor Allem darauf achten, Beweise möglichst entsprechend ihrer Baumstruktur darzustellen, und nicht darauf zu achten, dass Pfeile, die auf gebundene Annahmevariablen zeigen, die Struktur des Graphen beeinflussen.

Die Funktion `proof-to-graph` erzeugt einen `node-context` aus einem Beweis. Schließlich kann mit `proof-to-graphviz-code` aus einem Beweis ein Graphviz-Code erzeugt werden. Das zweite und dritte Argument sind dabei `label` und `name`, die in Graphviz angegeben werden, aber auf leere Strings gesetzt werden können.

A.2. Beispiel

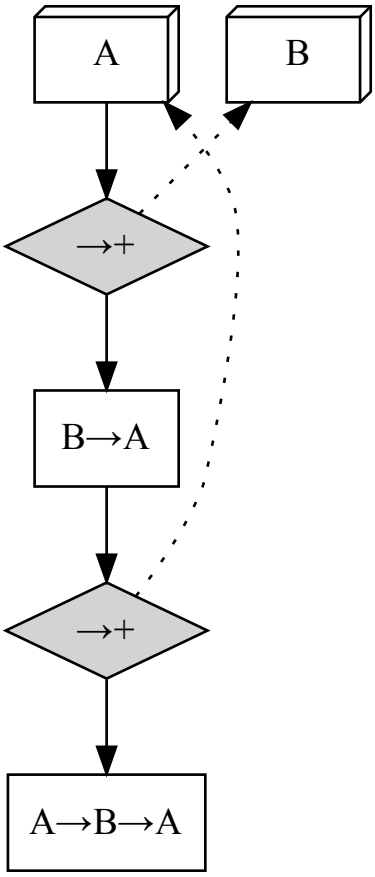
Als Beispiel sei der in `common.scm` definierte Beweis von $A \rightarrow B \rightarrow A$ herangezogen. Wir erhalten den Graphviz-Code durch

```
> (display
   (proof-to-graphviz-code
    (make-proof-in-a-imp-b-imp-a-form (pf "A") (pf "B"))
    "" ""))
digraph  { ratio = "auto" ; mincross = 2.0 ; label = "" ;
"Zero"[label="A",shape=box3d,style=filled,fillcolor=white];
"One"[label="B",shape=box3d,style=filled,fillcolor=white];
"Four"
[label="A&#x2192;B&#x2192;A",
shape=box,style=filled,fillcolor=white];
"Two"[label="B&#x2192;A",shape=box,style=filled,
fillcolor=white];
"Five"[label="&#x2192;+",shape=diamond,style=filled];
"Five"->"Zero"[weight=1,style=dotted,shape=onormal,
constraint=false];
"Two"->"Five"[weight=8,shape=onormal];
"Five"->"Four"[weight=8,shape=onormal];
"Three"[label="&#x2192;+",shape=diamond,style=filled];
"Three"->"One"[weight=1,style=dotted,shape=onormal,
constraint=false];
"Zero"->"Three"[weight=8,shape=onormal];
"Three"->"Two"[weight=8,shape=onormal];
}
```

Der ausgegebene Code muss nun in eine Datei kopiert werden, beispielsweise `aimpbimpa.dot`. Die erzeugten Quellcodes sind für das dot-Backend von Graphviz geschrieben. Aus der erzeugten Datei kann nun mittels

```
$ dot -Tpdf aimpbimpa.dot > aimpbimpa.pdf
```

eine PDF-Datei erzeugt werden, die folgenden Inhalt hat.



B. Generelle Informationen

Die Projektarbeit wurde mit der SVN-Revision 2373 des Minlog-Systems getestet. Ein Checkout in dieser Revision liegt bei. Sie wurde unter Racket v5.0 (früher MzScheme) getestet, in dessen R5RS-Kompatibilitätsmodus.

Inhaltsverzeichnis

Literatur

[1] Helmut Schwichtenberg and Stanley S. Wainer, *Proofs and Computations*. Buchmanuskript 2010

[2] *Minlog Reference Manual*, http://www.mathematik.uni-muenchen.de/ minlog/minlog/ref.pdf, 2010

[3] *The Graphviz Homepage*, http://www.graphviz.org/

[4] Eduardo Giménez and Pierre Castéran, *A Tutorial on [Co-]Inductive Types in Coq*, May 1998 — January 18, 2010