Computer-gestützte Beweisführung

Alexander Bentkamp

23. Januar 2024

1 Einleitung

- Was sind Beweisassistenten?
  - auch Interaktiver Theorem-Beweiser genannt
  - Software, um Beweise in einer formalen Logik durchzuführen
  - Der Beweis wird dabei vom Computer überprüft
- Erfolgsgeschichten
  - Mathematik:
    - Vier-Farben-Satz, formalisiert von Gonthier [6]
    - Satz von Feit-Thompson, formalisiert von Gonthier et al. [7]
    - Beweis der Kepler-Vermutung von Hales et al. [9]
    - Definition Perfektoider Räume, formalisiert von Buzzard et al. [3].
    - Sphere eversion, formalisiert von Van Doorn et al. [21]
    - Unabhängigkeit der Kontinuumshypothese, formalisiert von Han et al. [10]
    - Liquid tensor experiment, formalisiert von Commelin et al. [4, 12]
  - Informatik:
    - Hardware-Verification bei AMD [19] und Intel [11]
    - Verifizierte Betriebssystemkerne seL4 [13] und CertiKOS [8]
    - Verifizierte Compiler CompCert [16], JinjaThreads [17] und CakeML [14].
  - Computerlinguistik: Typentheorie wird genutzt um die Semantik natürlicher Sprache zu erklären [22, 15]. Zum Beispiel zur Analyse von anaphorischen Ausdrücken und Präsuppositionen. Manchmal werden dabei auch Beweisassistenten eingesetzt [5].

Dieses Vorlesungsskript basiert in Teilen auf Peter Selingers *Lecture Notes on the Lambda Calculus* [20], *Theorem Proving in Lean 4* [1] und dem *Hitchhiker's Guide to Logical Verification* [2], der wiederum auf *Concrete Semantics* [18] basiert.

- Es gibt Dutzende Beweisassistenten und sie verwenden unterschiedliche Logiken, z.B.:
  - Mengenlehre: Isabelle/ZF, Metamath, Mizar
  - Einfache Typentheorie: HOL4, HOL Light, Isabelle/HOL
  - Abhängige Typentheorie: Agda, Automath, Coq, Lean, Matita, PVS
- In dieser Vorlesung schauen wir uns abhängige Typentheorie an, die von vielen populären Beweisassistenten genutzt wird. Im Fall von Lean (wie auch Coq und Matita) handelt es sich genauer um den Calculus of Inductive Constructions. In den Übungen werden wir Lean verwenden, um das ganze praktisch auszuprobieren.

2 λ-Kalkül

Bevor wir uns mit der Logik von Lean, dem Calculus of Inductive Constructions, befassen, schauen wir uns zunächst eine einfacheren Formalismus an: den λ-Kalkül. Der λ-Kalkül ist ein Formalismus, den wir nutzen werden, um mathematische Ausdrücke und Funktionen formal niederzuschreiben.

2.1 Motivation

Schauen wir uns zur Einführung die gewöhnlichen mathematische Notation von arithmetischen Ausdrücken an: Diese bestehen aus Variablen (x, y,...), Operationen ($+$, $\cdot$, etc.) und Konstanten (0, 1, 2, π, e etc.). Der Ausdruck $x + y$ steht für das *Ergebnis* einer Addition (im Gegensatz zu einer Anweisung, dass etwas addiert werden soll, oder einer Aussage, dass etwas addiert wird). Das hat den Vorteil, dass man die Ausdrücke verschachteln kann, z.B.

$$a = ((x + y) \cdot z) + 1.$$

Ohne diese Möglichkeit zu verschachteln müssten wir schreiben

Sei $u = x + y$. Außerdem sei $v = u \cdot z$. Setze $a = v + 1$.

Der λ-Kalkül überträgt diese Idee auch auf Funktionen. Anstelle von

Sei $f(x) = 2 \cdot x + 1$. Setze $a = f(3)$.

schreiben wir

$$a = (\lambda x.\ 2 \cdot x + 1)\ (3).$$

wobei $(\lambda x.\ 2 \cdot x + 1)$ für eine Funktion steht, die eine gegebene Zahl x auf $2 \cdot x + 1$ abbildet (im Gegensatz zu der Aussage, dass x auf $2 \cdot x + 1$ abgebildet wird). Auf diese Weise müssen wir also nicht jeder Funktion einen Namen geben. Der Name der Variablen x ist nur innerhalb des Ausdrucks $(\lambda x.\ ...)$ von Bedeutung. Es macht also keinen Unterschied, ob wir $(\lambda x.\ 2 \cdot x + 1)$ oder $(\lambda y.\ 2 \cdot y + 1)$ schreiben.

Im λ-Kalkül können wir auch leicht Funktionen höherer Ordnung notieren, d.h. Funktionen, deren Argumente ebenfalls Funktionen sind. Hier ist beispielsweise eine Funktion, deren Argument eine Funktion f ist und die dann den Funktionswert an der Stelle 3 zurückgibt:

$$\lambda f.\ (f\ (3))$$

Als weiteres Beispiel hier eine Funktion, deren Argument eine Funktion f ist und die dann eine Funktion zurückgibt, die f zweimal auf ein gegebenes Argument anwendet:

$$\lambda f.\ (\lambda x.\ (f\ (f\ (x))))$$

Wir können diese Funktion zum Beispiel auf das Argument $(\lambda y.\ 2 \cdot y + 1)$ anwenden und das Ergebnis dann auf das Argument 3, also

$$((\lambda f.\ (\lambda x.\ (f\ (f\ (x)))))\ (\lambda y.\ 2 \cdot y + 1))\ (3)$$

Bei der Auswertung dieses Ausdrucks wird $(\lambda y.\ 2 \cdot y + 1)$ zweimal auf 3 angewendet. Wir erhalten also $2 \cdot (2 \cdot 3 + 1) + 1 = 15$.

2.2 Ungetypter λ-Kalkül

Das Einführungsbeispiel mit Zahlen und arithmetischen Ausdrücken ist hilfreich, um die Idee des λ-Kalküls zu verstehen, aber der λ-Kalkül selbst betrachtet Funktionen auf einer abstrakteren Ebene.

Syntax Die Ausdrücke des λ-Kalküls heißen λ-Terme. Zunächst benötigen wir eine unendliche Menge $V = \{x, y, ...\}$ von *Variablen* und eine Menge $K = \{\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, ...\}$ von *Konstanten*. Die Menge aller Terme Λ wird dann induktiv definiert durch:

- Ist $x \in V$, dann $x \in \Lambda$. (Variable)
- Ist $\mathsf{a} \in K$, dann $\mathsf{a} \in \Lambda$. (Konstante)
- Sind $s, t \in \Lambda$, dann $(s\ t) \in \Lambda$. (Anwendung)
- Ist $x \in V$ und $t \in \Lambda$, dann $(\lambda x.\ t) \in \Lambda$. (Abstraktion)

Das Wort „induktiv“ bedeutet hier, das diese vier Optionen die einzigen Möglichkeiten sind, wie λ-Terme gebildet werden können. In diesem Skript werden Konstanten sans-serif geschrieben und Variablen *kursiv*.

(Bemerkung: Im *reinen* λ-Kalkül ist $K = \emptyset$, es gibt also keine Konstanten.)

Für diese Art von Definition gibt es auch eine Kurzschreibweise, die Backus-Naur-Form:

$$\Lambda ::= V \mid K \mid (\Lambda\ \Lambda) \mid (\lambda V.\ \Lambda)$$

Hier sind einige Beispiele:

$$(\lambda x.\ x) \qquad ((\lambda x.\ (x\ y))\ (\lambda z.\ (\mathsf{f}\ z))) \qquad \lambda z.\ (\lambda x.\ (z\ (z\ x)))$$

Auch Konstanten können angewendet werden. Ist zum Beispiel a eine Konstante, ist $\mathsf{a}\ \mathsf{a}$ ein Term. Das Wort „Konstante“ ist im Gegensatz zu „Variable“ gemeint, nicht im Gegensatz zu „Funktion“.

Wir schreiben also $(\mathsf{f}\ x)$ für die Anwendung von f auf x, nicht wie sonst üblich $\mathsf{f}(x)$. Das wird uns in Kombination mit den folgenden Konventionen einige Klammern ersparen:

- Die äußersten Klammern können weggelassen werden, also $s\ t$ statt $(s\ t)$ und $\lambda x.\ t$ statt $(\lambda x.\ t)$.
- Anwendung ist links-assoziativ, also schreiben wir $s\ t\ r$ statt $(s\ t)\ r$.
- Der Bezugsbereich einer Abstraktion erstreckt sich so weit nach rechts, wie die Klammern es erlauben. Der Term $\lambda x.\ t\ s$ steht also für $\lambda x.\ (t\ s)$, nicht für $(\lambda x.\ t)\ s$. Und der Term $\lambda x.\ \lambda y.\ t$ steht für $\lambda x.\ (\lambda y.\ t)$. (Die Notation $s\ \lambda y.\ t$ für $s\ (\lambda y.\ t)$ wäre auch denkbar, ist aber eher ungewöhnlich und wird in diesem Skript nicht verwendet.)
- Mehrere aufeinander folgende Abstraktionen können zu einer vereint werden: $\lambda x\ y.\ t$ statt $\lambda x.\ \lambda y.\ t$.

Currying Auf den ersten Blick scheint es, als könnten wir im λ-Kalkül keine Funktionen mit mehreren Argumenten definieren. Wir haben aber schon eine Funktion gesehen, die gewissermaßen zwei Argumente annimmt:

$$\lambda z.\ (\lambda x.\ (z\ (z\ x)))$$

Statt dies als eine Funktion zu sehen, die eine Funktion zurückgibt, können wir sie genauso als eine Funktion betrachten, die zwei Argumente annimmt. Daher auch die Schreibweise

$$\lambda z\ x.\ z\ (z\ x)$$

Wenn wir die Funktion auf zwei Argumente anwenden, z.B. zwei Konstanten f und a, dann passiert dies strenggenommen in zwei Schritten. Erst wird die Funktion auf das erste Argument angewendet, was eine weitere Funktion ergibt, die dann auf das zweite Argument angewendet werden kann:

$$((\lambda z\ x.\ z\ (z\ x))\ \mathsf{f})\ \mathsf{a}$$

Wir können diesen Prozess aber auch als Anwendung auf zwei Argumente betrachten. Unsere Klammer-Konventionen helfen und bei dieser Sichtweise, denn wir können den obigen Term auch schreiben als:

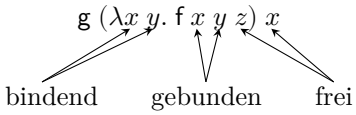
$$(\lambda z\ x.\ z\ (z\ x))\ \mathsf{f}\ \mathsf{a}$$

Diese Art Funktionen mit mehreren Argumenten darzustellen, nennt sich *Currying*, benannt nach Haskell Brooks Curry.

2.2.1 Freie, bindende und gebundene Variabelvorkommen

Vorkommen von Variablen in λ-Termen können in drei Kategorien eingeteilt werden: freie, gebundene und bindende. *Bindende* Vorkommen von Variablen sind solche, die direkt nach einem λ stehen. *Gebundene* Vorkommen einer Variable x sind solche, die sich im Inneren einer λ-Abstraktion $(\lambda x.\ ...)$ von x befinden. Alle anderen Vorkommen von Variablen heißen *frei*.

Beispiel



Wir schreiben $t[x := s]$ für den Term, der aus t entsteht, indem man alle freien Vorkommen von x in t durch s ersetzt.

2.2.2 Äquivalenzregeln

Üblicherweise werden einige λ-Terme als äquivalent betrachtet. Die Notation $s \equiv t$ bedeutet, dass die Terme s und t äquivalente λ-Terme sind.

α-Äquivalenz Die grundlegendste Form solcher Äquivalenzen ist die α-Äquivalenz. Sie besagt, dass die Namen von gebundenen Variablen keine Rolle spielen. Genauer gesagt:

$$\lambda x.\ t \equiv \lambda y.\ t[x := y]$$

wenn y in t nicht frei vorkommt und die freien Vorkommen von x in t nicht im Inneren einer λ-Abstraktion $(\lambda y.\ ...)$ von y liegen. Diese und die anderen Äquivalenzregeln sind auch wiederholt und auch auf alle Teilterme anwendbar: Wird ein Teilterm durch einen äquivalenten Teilterm ersetzt, dann entsteht ein äquivalenter Term.

Beispiele

$$\mathsf{f}\ (\lambda x.\ x)\ (\lambda x.\ x) \equiv \mathsf{f}\ (\lambda y.\ y)\ (\lambda z.\ z)$$
$$\lambda z.\ y \equiv \lambda x.\ y \not\equiv \lambda y.\ y$$
$$\lambda z.\ (\lambda y.\ z) \equiv \lambda x.\ (\lambda y.\ x) \not\equiv \lambda y.\ (\lambda y.\ y)$$

β-Äquivalenz Die β-Äquivalenz besagt, dass eine λ-Abstraktion auf einen Term angewendet wird, indem alle freien Vorkommen der Variable durch das Argument ersetzt werden. Wird die β-Äquivalenz von links nach rechts genutzt, sprechen wir von β-Reduktion.

$$(\lambda x.\ t)\ s \equiv t[x := s]$$

wenn alle freien Vorkommen von Variablen in s auch frei bleiben, d.h. es darf keine freie Variable y in s geben, sodass ein freies Vorkommen von x in t im Inneren einer λ-Abstraktion $(\lambda y.\ ...)$ von y liegt. Um dies sicherzustellen, dürfen Teilterme zunächst durch α-äquivalente ersetzt werden.

Beispiele

$$
\begin{aligned}
(\lambda x.\ \mathsf{f}\ x\ x)\ (\mathsf{g}\ y) &\equiv \mathsf{f}\ (\mathsf{g}\ y)\ (\mathsf{g}\ y) \\
(\lambda x.\ x\ y)\ (\lambda z.\ \mathsf{g}\ z) &\equiv \mathsf{g}\ y \\
(\lambda x.\ \lambda y.\ x)\ y &\not\equiv (\lambda y.\ y) \\
(\lambda x.\ \lambda z.\ x)\ y &\equiv (\lambda z.\ y) \\
(\lambda x.\ \mathsf{a})\ (\lambda x.\ \mathsf{a}) &\equiv \mathsf{a}
\end{aligned}
$$

Achtung: β-Reduktion kann nur bei Anwendung einer λ-Abstraktion auf einen anderen Term angewendet werden. Terme wie $\mathsf{a}\ \mathsf{a}$ mit einer Konstanten a oder $x\ x$ mit einer Variablen x können nicht β-reduziert werden.

η-Äquivalenz η-Äquivalenz ist eine Form von Extensionalität, der Idee, dass Funktionen gleich sind, wenn bei gleichen Argumenten gleiche Werte zurückgeben. Wird sie von links nach rechts angewandt, sprechen wir von η-Reduktion, von rechts nach links sprechen wir von η-Expansion.

$$(\lambda x.\ t\ x) \equiv t$$

wenn x nicht frei in t vorkommt.

Beispiele

$$
\begin{aligned}
(\lambda x.\ \mathsf{f}\ \mathsf{a}\ x) &\equiv \mathsf{f}\ \mathsf{a} \\
(\lambda x\ y.\ \mathsf{f}\ x\ y) &\equiv \mathsf{f} \\
(\lambda x.\ \mathsf{f}\ x\ x) &\not\equiv \mathsf{f}\ x
\end{aligned}
$$

2.3 Einfach getypter λ-Kalkül

Der ungetypte λ-Kalkül ist elegant, aber manchmal etwas zu freigiebig. Ein Problem ist zum Beispiel, dass β-Reduktion nicht immer terminiert, wie etwa bei dem Term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Die Einführung von Typen kann uns dabei helfen, die Menge von zulässigen λ-Termen etwas einzuschränken. Die Idee ist einfach: Jeder Term hat einen Typ. Dabei haben manche Terme einen Funktionstyp, der uns vorschreibt, welche Argumente und Rückgabewerte zulässig sind. Typen ähneln ein wenig mathematischen Mengen. Ein entscheidender Unterschied ist jedoch, dass jeder Term nur einem Typ zugeordnet werden kann.

2.3.1 Typen

Der einfach getypte λ-Kalkül unterscheidet Typen und Terme. Die verfügbaren Typen hängen ab von einer Menge von Basistypen $\mathbb{T}_0$. Dann sind die **Typen** wie folgt induktiv definiert:

- Jeder Basistyp ist ein Typ.
- Sind A und B Typen, dann ist auch der Funktionstyp $(A \to B)$ ein Typ.

Formal können wir diese Definition der Typen $\mathbb{T}$ auch schreiben als:

$$\mathbb{T} ::= \ \mathbb{T}_0 \mid (\mathbb{T} \to \mathbb{T})$$

Beispiele Sind zum Beispiel Nat und Bool unsere Basistypen, dann sind Nat, Bool, $\mathsf{Bool} \to \mathsf{Bool}$, $\mathsf{Nat} \to \mathsf{Bool}$, $\mathsf{Nat} \to (\mathsf{Bool} \to \mathsf{Bool})$, $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}$, und viele mehr unsere Typen.

Notation Um Currying leichter notieren zu können, schreiben wir den Funktionspfel rechts-assoziativ, wir schreiben also $A \to B \to C$ für $A \to (B \to C)$. Bei $(A \to B) \to C$ sind die Klammern aber notwendig!

2.3.2 Terme

Die verfügbaren Terme hängen ab von einer Menge von Konstanten K, jede assoziiert mit einem Typ (Achtung: Trotz des Namens können Konstanten auch mit einem Funktionstyp asoziiert werden.). Außerdem benötigen wir eine unendliche Menge an Variablen V. Die Terme des einfach getypten λ-Kalküls sind wie folgt definiert:

$$\Lambda_{\mathbb{T}} ::= K \mid V \mid (\Lambda_{\mathbb{T}}\ \Lambda_{\mathbb{T}}) \mid (\lambda V : \mathbb{T}.\ \Lambda_{\mathbb{T}})$$

Der einzige Unterschied zur Syntax der ungetypten λ-Terme ist also, dass die bindenden Variablen jeweils mit einer Typangabe „: $\mathbb{T}$“ versehen sind. Zur besseren Lesbarkeit werden wir diese Typangabe manchmal weglassen, wenn sie sich aus dem Kontext ergibt.

Beispiele Sind a, b Konstanten und x, y Variablen, dann sind folgende Ausdrücke Terme:

$$\mathsf{a}, \quad x, \quad \mathsf{a}\ x, \quad \lambda x : \mathsf{Nat}.\ \mathsf{a}, \quad \lambda x : \mathsf{Bool}.\ (\mathsf{a}\ x), \quad ((\lambda x : \mathsf{Nat}.\ x)\ \mathsf{a})\ (y\ \mathsf{b}), \quad \ldots$$

Notation Wir nutzen weiterhin die Notationen, die wir für den ungetypten Lambda-Kalkül eingeführt haben. Bei mehreren aufeinander folgenden Abstraktionen setzen wir Klammern, um die einzelnen Variablen zu trennen; so steht zum Beispiel

$$\lambda(x : \mathsf{Nat})\ (y : \mathsf{Bool}).\ x \quad \text{für} \quad \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Bool}.\ x$$

Wenn Variablen vom selben Typ sind, verwenden wir auch Klammern und nur eine einzige Typannotation:

$$\lambda x\ y : \mathsf{Nat}.\ x \quad \text{steht für} \quad \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ x$$

2.3.3 Typinferenz

Manchen Termen t kann ein Typ T zugewiesen werden. Wir schreiben das als

$$t : T$$

Ein Term heißt *wohlgetypt*, wenn ihm ein Typ zugewiesen werden kann.

Der Typ eines Terms wird durch Typisierungsregeln bestimmt. Intuitiv sagen diese Regeln folgendes aus:

- Wenn $t : A \to B$ und $s : A$, dann ist $t\ s : B$.
- Wenn $t : B$, dann ist $(\lambda x : A.\ t) : A \to B$.
- Jede Konstante c ist global mit einem Typ T assoziiert.
- Der Typ einer Variable hängt von dem Typ ab, der ihr vom umgebenden λ zugewiesen wurde.

Um dies formal zu fassen, benutzen wir einen *Kontext*: eine Liste von Variablen und ihren assoziierten Typen. Ist Γ solch ein Kontext, dann schreiben wir $\Gamma, x : T$ für den Kontext, der Γ um die Variable x, assoziiert mit dem Typ T, erweitert. Wir schreiben $\Gamma(x)$ für den Typ, der mit dem letzten Vorkommen von x in der Liste Γ assoziiert ist.

Wenn in einem Kontext Γ der Term t Typ T hat, schreiben wir dieses *Typurteil* als

$$\Gamma \ \vdash \ t : T$$

Die Typisierungsregeln lauten:

$$\text{Konst}\ \frac{}{\Gamma \ \vdash \ \mathsf{c} : T} \qquad \text{wenn } \mathsf{c} \text{ eine Konstante von Typ } T \text{ ist}$$

$$\text{Var}\ \frac{}{\Gamma \ \vdash \ x : \Gamma(x)}$$

$$\text{Anw}\ \frac{\Gamma \ \vdash \ t : S \to T \quad \Gamma \ \vdash \ s : S}{\Gamma \ \vdash \ t\ s : T}$$

$$\text{Lam}\ \frac{\Gamma,\ x : S \ \vdash \ t : T}{\Gamma \ \vdash \ (\lambda x : S.\ t) : S \to T}$$

Diese Regeln sind zu lesen als: „Wenn alle Typurteile über dem Strich gelten, dann gilt auch das Typurteil unter dem Strich“.

Beispiel Hier ist ein Beispiel, wie wir den Typ von

$$(\lambda x : \mathsf{A} \to \mathsf{A}.\ (\lambda y : \mathsf{A}.\ (x\ y)))$$

feststellen können (wobei A ein Basistyp ist):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{A} \to \mathsf{A},\ y : \mathsf{A} \vdash x : \mathsf{A} \to \mathsf{A}}\,\text{Var} \quad \dfrac{}{x : \mathsf{A} \to \mathsf{A},\ y : \mathsf{A} \vdash y : \mathsf{A}}\,\text{Var}}{x : \mathsf{A} \to \mathsf{A},\ y : \mathsf{A} \vdash x\ y : \mathsf{A}}\,\text{Anw}}{x : \mathsf{A} \to \mathsf{A} \vdash (\lambda y : \mathsf{A}.\ (x\ y)) : \mathsf{A} \to \mathsf{A}}\,\text{Lam}}{\vdash (\lambda x : \mathsf{A} \to \mathsf{A}.\ (\lambda y : \mathsf{A}.\ (x\ y))) : (\mathsf{A} \to \mathsf{A}) \to (\mathsf{A} \to \mathsf{A})}\,\text{Lam}$$

Dabei dekonstruieren wir den Term erst von unten nach oben und füllen dann die Typen von oben nach unten ein. Ein solcher Baum nennt sich Typderivation.

Seien nun A und B Basistypen und seien $\mathsf{a} : \mathsf{A}$ und $\mathsf{g} : \mathsf{A} \to \mathsf{B}$ Konstanten. Wir bestimmen den Typ von $(\lambda x : \mathsf{A} \to \mathsf{B}.\ x\ \mathsf{a})\ (\lambda z : \mathsf{A}.\ \mathsf{g}\ z)$ mittels einer Typderivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{A} \to \mathsf{B} \vdash x : \mathsf{A} \to \mathsf{B}}\,\text{Var} \quad \dfrac{}{x : \mathsf{A} \to \mathsf{B} \vdash \mathsf{a} : \mathsf{A}}\,\text{Konst}}{x : \mathsf{A} \to \mathsf{B} \vdash x\ \mathsf{a} : \mathsf{B}}\,\text{Anw}}{\vdash \lambda x : \mathsf{A} \to \mathsf{B}.\ x\ \mathsf{a} : (\mathsf{A} \to \mathsf{B}) \to \mathsf{B}}\,\text{Lam} \quad \dfrac{\dfrac{\dfrac{}{z : \mathsf{A} \vdash \mathsf{g} : \mathsf{A} \to \mathsf{B}}\,\text{Konst} \quad \dfrac{}{z : \mathsf{A} \vdash z : \mathsf{A}}\,\text{Var}}{z : \mathsf{A} \vdash \mathsf{g}\ z : \mathsf{B}}\,\text{Anw}}{\vdash \lambda z : \mathsf{A}.\ \mathsf{g}\ z : \mathsf{A} \to \mathsf{B}}\,\text{Lam}}{\vdash (\lambda x : \mathsf{A} \to \mathsf{B}.\ x\ \mathsf{a})\ (\lambda z : \mathsf{A}.\ \mathsf{g}\ z) : \mathsf{B}}\,\text{Anw}$$

3 Calculus of Inductive Constructions

Wir schauen uns nun verschiedene Erweiterungen des λ-Kalküls an, die uns letztendlich zum Calculus of Inductive Constructions führen werden, der Logik von Lean.

3.1 Definitionen

Mit Definitionen können wir neue Konstanten einführen, die für einen komplexeren Term stehen. Ist $\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ etwa eine Konstante, die die Addition natürlicher Zahlen repräsentiert, dann können wir eine Konstante zum Verdoppeln einer natürlichen Zahl einführen:

$$\textbf{def}\ \ \mathsf{verdoppeln} : \mathsf{Nat} \to \mathsf{Nat}\ :=\ \lambda x.\ \mathsf{add}\ x\ x$$

Zur besseren Lesbarkeit schreiben wir stattdessen auch:

$$\textbf{def}\ \ \mathsf{verdoppeln}\ (x : \mathsf{Nat}) : \mathsf{Nat}\ :=\ \mathsf{add}\ x\ x$$

Definitionen kommen mit einer neuen Äquivalenzregel, der δ-Äquivalenz, die besagt, dass definierte Konstanten jederzeit durch ihre Definition ersetzt werden können.

Beispiel

$$\mathsf{verdoppeln\ a} \equiv (\lambda x.\ \mathsf{add}\ x\ x)\ \mathsf{a} \equiv \mathsf{add\ a\ a}$$

Im Calculus of Inductive Constructions sprechen wir deshalb auch von *definitionell gleichen* Termen (statt von äquivalenten Termen), wenn sie äquivalent bezüglich unserer Äquivalenzregeln sind (α, β, η, δ, und einige, die wir noch nicht kennengelernt haben).

3.2 Polymorphismus, Typkonstruktoren und abhängige Typen

Einige Terme ergeben in jedem Typ Sinn, zum Beispiel die Identität:

$$\begin{aligned}
&\textbf{def}\ \ \mathsf{idNat} : \mathsf{Nat} \to \mathsf{Nat} := \lambda x : \mathsf{Nat}.\ x \\
&\textbf{def}\ \ \mathsf{idBool} : \mathsf{Bool} \to \mathsf{Bool} := \lambda x : \mathsf{Bool}.\ x \\
&\textbf{def}\ \ \mathsf{idNatToNat} : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat}) := \lambda (x : \mathsf{Nat} \to \mathsf{Nat}).\ x \\
&\vdots
\end{aligned}$$

Wenn wir im Allgemeinen über die Identitätsfunktion sprechen müssen, unabhängig vom konkreten Typ, ist es praktisch, λ-Abstraktionen auch über Typen zu erlauben:

$$\textbf{def}\ \ \mathsf{id} := \lambda X : \mathsf{Type}.\ \lambda x : X.\ x$$

Damit können wir dann z. B. $\mathsf{id\ Nat}$ für das oben definierte idNat schreiben. Diese Erweiterung des λ-Kalküls nennt sich **Polymorphismus**: Terme, die von Typen abhängen.

In ähnlicher Weise gibt es Familien von Typen, die man manchmal zusammenfassen möchte, z.B. Folgen:

$$\begin{aligned}
\textbf{def}\ \ \mathsf{SequenceOfBools} : \mathsf{Type} &:= \mathsf{Nat} \to \mathsf{Bool} \\
\textbf{def}\ \ \mathsf{SequenceOfNats} : \mathsf{Type} &:= \mathsf{Nat} \to \mathsf{Nat} \\
\textbf{def}\ \ \mathsf{SequenceOfInts} : \mathsf{Type} &:= \mathsf{Nat} \to \mathsf{Int} \\
&\vdots
\end{aligned}$$

Hier ist es praktisch auch λ-Funktionen über Typen zu erlauben, die auch Typen zurückgeben:

$$\textbf{def}\ \ \mathsf{Sequence} : \mathsf{Type} \to \mathsf{Type} := \lambda (X : \mathsf{Type}),\ \mathsf{Nat} \to X$$

Damit können wir dann z. B. $\mathsf{Sequence\ Bool}$ für das oben definierte $\mathsf{SequenceOfBools}$ schreiben. Solche Funktionen nennen sich **Typkonstruktoren**: Typen, die von Typen abhängen.

Schließlich macht es auch manchmal Sinn, Typen zu definieren, die von Termen abhängen, z.B. der Typ $\mathsf{Vec}\ n$ von Vektoren der Länge $n : \mathsf{Nat}$. Solche Typen heißen **abhängige Typen**.

Der Calculus of Inductive Constructions erlaubt alle diese Konstruktionen und hebt somit die Unterscheidung zwischen Typen und Termen fast vollständig auf. Damit hängen allerdings noch ein paar Schwierigkeiten zusammen, die wir uns jetzt anschauen:

Abhängige Funktionstypen Was ist der Typ von id, das wir oben definiert haben?

$$\textbf{def}\ \ \mathsf{id} := \lambda X : \mathsf{Type}.\ \lambda x : X.\ x$$

Ein erster Versuch, id einen Typ zu geben, wäre

$$\mathsf{id} : \mathsf{Type} \to X \to X$$

Beim näheren hinschauen merkt man aber, dass das keinen Sinn macht, denn es ist nicht klar, was X ist, denn X soll ja gerade mit dem ersten Argument von id identifiziert werden. Um das klarzustellen, nutzen wir die folgende Schreibweise:

$$\mathsf{id} : (\Pi X : \mathsf{Type}.\ X \to X)$$

Im Allgemeinen steht $(\Pi x : T.\ S)$ für den Typ einer Funktion, die ein Argument von Typ T annimmt und einen Wert von Typ S zurückgibt. Im Gegensatz zum nicht-abhängigen Funktionstyp $T \to S$, darf S nun aber vom Argument $x : T$ abhängen. Streng genommen brauchen wir die Schreibweise $T \to S$ damit nicht mehr: Sie steht einfach für $(\Pi x : T.\ S)$, wenn x nicht in S vorkommt.

Eine vollständige Definition von id lautet also:

$$\textbf{def}\ \ \mathsf{id} : (\Pi X : \mathsf{Type}.\ X \to X) := \lambda X : \mathsf{Type}.\ \lambda x : X.\ x$$

Wie oben schon für nicht-abhängige Typen, schreiben wir zur besseren Lesbarkeit für solche Definitionen auch kurz

$$\textbf{def}\ \ \mathsf{id}\ (X : \mathsf{Type}) : X \to X := \lambda x : X.\ x$$

oder sogar

$$\textbf{def}\ \ \mathsf{id}\ (X : \mathsf{Type})\ (x : X) : X := x$$

Implizite Argumente Im Kontext von abhängigen Funktionstypen treffen wir häufig auf Argumente, die sich in den meisten Fällen aus dem Kontext ergeben. Ist zum Beispiel zero eine Konstante vom Typ Nat, dann ist das erste Argument im Term $\mathsf{id}\ \mathsf{Nat}\ \mathsf{zero}$ redundant. Um die Lesbarkeit zu erhöhen, möchten wir daher das erste Argument von id implizit lassen. Wir markieren solche impliziten Argumente mit geschweiften Klammern:

$$\textbf{def}\ \ \mathsf{id} : (\Pi\{X : \mathsf{Type}\}.\ X \to X) := \lambda\{X : \mathsf{Type}\}.\ \lambda x : X.\ x$$

oder

$$\textbf{def}\ \ \mathsf{id}\ \{X : \mathsf{Type}\}\ (x : X) : X := x$$

Wir schreiben dann $\mathsf{id}\ x$ statt $\mathsf{id}\ X\ x$ und lassen das erste Argument implizit. In manchen Fällen möchten wir alle Argumente explizit angeben. Dann schreiben wir $@\mathsf{id}\ X\ x$.

Der Typ des Typs der Typen Mit der Einführung von Type als dem Typ von Typen, stellt sich die Frage, was der Typ von Type ist. Der naive Ansatz $\mathsf{Type} : \mathsf{Type}$ führt leider zu einem widersprüchlichen System ähnlich wie in der naiven Mengenlehre (siehe Girards Paradoxon). Der Calculus of Inductive Constructions löst dies durch eine Typenhierarchie:

$$\begin{aligned} \mathsf{Type} &: \mathsf{Type}_1 \\ \mathsf{Type}_1 &: \mathsf{Type}_2 \\ \mathsf{Type}_2 &: \mathsf{Type}_3 \\ &\vdots \end{aligned}$$

wobei Type identisch mit Type_0 ist. Die Indizes $0, 1, 2, 3, ...$ heißen *Universe-Levels*.

Häufig möchte man Definitionen für Type_u mit beliebigen u machen. Dazu gibt es in Lean die Möglichkeit, Konstanten abhängig von einem Universe-Level u zu definieren, z.B.:

$$\textbf{def}\ \ \mathsf{id}.\{u\} : (\Pi\{X : \mathsf{Type}_u\}.\ X \to X) := \lambda\{X : \mathsf{Type}_u\}.\ \lambda x : X.\ x$$

Formal definiert dies dann unendlich viele Konstanten $\mathsf{id}.\{0\}$, $\mathsf{id}.\{1\}$, ..., aber das Universe-Level kann weggelassen und von Lean automatisch ermittelt werden, sodass id dann auf beliebigen Universe-Levels funktioniert.

3.3 Typisierungsregeln (ohne Aussagen)

Jetzt können wir eine erste Fassung der Typisierungsregeln des Calculus of Constructions aufschreiben. Später werden wir diese Regeln noch mit Aussagen und deren Beweisen erweitern. Gegeben sei eine unendliche Menge V von Variablen und eine zunächst leere, aber erweiterbare Menge K von Konstanten. Die Ausdrücke des Calculus of Constructions (ohne Aussagen) sind:

$$E ::= K \mid V \mid E\ E \mid \lambda V : E.\ E \mid \mathsf{Type}_u \mid \Pi V : E.E$$

Jede Konstante ist assoziiert mit einem Ausdruck, den wir den Typ der Konstanten nennen. Beim Einführen von Konstanten muss der assoziierte Typ T der Konstanten das Typurteil $\vdash T : \mathsf{Type}_u$ für ein beliebiges u erfüllen.

Die Typisierungsregeln (ohne Aussagen) lauten:

$$\text{Konst}\quad \frac{}{\Gamma \vdash \mathsf{c} : T} \qquad \text{wenn } \mathsf{c} \text{ eine Konstante von Typ } T \text{ ist}$$

$$\text{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{Anw}\quad \frac{\Gamma \vdash t : \Pi x : S.\, T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[x := s]}$$

$$\text{Lam}\quad \frac{\Gamma,\ x : S \vdash t : T \qquad \Gamma \vdash S : \mathsf{Type}_u}{\Gamma \vdash (\lambda x : S.\, t) : (\Pi x : S.\, T)}$$

$$\text{Type}\quad \frac{}{\Gamma \vdash \mathsf{Type}_u : \mathsf{Type}_{u+1}}$$

$$\text{Pi}\quad \frac{\Gamma \vdash S : \mathsf{Type}_u \qquad \Gamma,\ x : S \vdash T : \mathsf{Type}_v}{\Gamma \vdash \Pi x : S.\, T : \mathsf{Type}_{\max(u,v)}}$$

$$\text{DefGl}\quad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : S} \qquad \text{wenn } T \equiv S$$

Die Regeln Konst und Var sind dieselben wie im einfach-getypten λ-Kalkül. Die Regeln Anw und Lam verallgmeinern die entsprechenden Regeln aus dem einfach-getypten λ-Kalkül, indem der Typ T nun eine Variable x vom Typ S enthalten darf. Dementsprechend nutzen wir jeweils die Π-Notation statt $\to$ und der Typ von $t\ s$ ist nun $T[x := s]$ statt T. Die Regel Lam enthält eine zusätzliche Voraussetzung, um sicherzustellen, dass nur über Ausdrücke vom Typ Type_u (für beliebiges u) abstrahiert werden kann.

Die Regel Type legt unsere Typhierarchie fest. Die Regel Pi legt fest, in welches Universe-Level Funktionstypen eingeordnet werden müssen. Wir nehmen hier das größere der beiden Universe-Levels $max(u, v)$ zwischen dem Universe-Level u des Argumenttyps und dem Universe-Level v des Rückgabetyps.

Schließlich legt die Regel DefGl fest, dass Typen nur bis auf definitonelle Gleichheit eindeutig sind. Wir können jederzeit einen Typ durch einen definitionell gleichen Typ austauschen.

Beispiele

$$\dfrac{\dfrac{\dfrac{}{X : \mathsf{Type}, x : X \vdash x : X}\,\text{Var}}{X : \mathsf{Type} \vdash (\lambda x : X.\, x) : (X \to X)}\,\text{Lam} \qquad \dfrac{}{\mathsf{Type} : \mathsf{Type}_1}\,\text{Type}}{\vdash (\lambda X : \mathsf{Type}.\, \lambda x : X.\, x) : (\Pi X : \mathsf{Type}, X \to X)}\,\text{Lam}$$

$$\dfrac{\dfrac{}{\vdash\ @\mathsf{id} : \Pi X : \mathsf{Type}, X \to X}\,\text{Konst} \quad \dfrac{}{\vdash\ \mathsf{Nat} : \mathsf{Type}}\,\text{Konst}}{\dfrac{\vdash\ @\mathsf{id}\ \mathsf{Nat} : \mathsf{Nat} \to \mathsf{Nat}}{\vdash\ @\mathsf{id}\ \mathsf{Nat}\ \mathsf{zero} : \mathsf{Nat}}\,\text{Anw} \quad \dfrac{}{\vdash\ \mathsf{zero} : \mathsf{Nat}}\,\text{Konst}}\,\text{Anw}$$

$$\dfrac{\dfrac{}{\vdash\ \mathrm{Type}_4 : \mathrm{Type}_5}\,\text{Type} \quad \dfrac{}{x : \mathrm{Type}_4 \vdash\ \mathrm{Type}_7 : \mathrm{Type}_8}\,\text{Type}}{\vdash\ (\mathsf{Type}_4 \to \mathsf{Type}_7) : \mathrm{Type}_8}\,\text{Pi}$$

$$\dfrac{\dfrac{}{n : \mathsf{Nat} \vdash\ n : \mathsf{Nat}}\,\text{Var}}{n : \mathsf{Nat} \vdash\ n : \mathsf{id}\ \mathsf{Nat}}\,\text{DefGl}$$

(Im letzten Beispiel ist id Nat die Kurzform für @id.{1} Type Nat, was definitionell gleich mit Nat ist.)

3.4 Induktive Typen

Induktive Typen sind ein vielseiteiger Mechanismus, um neue Typen zu definieren. (Bemerkung für Kategorientheoretiker: In der Kategorientheorie heißen induktive Typen auch initiale Algebren.)

Ein induktiver Typ wird wie folgt definiert:

$$\begin{aligned}&\textbf{inductive}\ \ \mathit{Name}\ [\mathit{Parameter}] : [\mathit{Indizes} \to]\ \mathsf{Type}_u :=\\ &\quad \mid \mathit{Konstruktor}_1 : \mathit{Konstruktortyp}_1\\ &\quad \vdots\\ &\quad \mid \mathit{Konstruktor}_n : \mathit{Konstruktortyp}_n\end{aligned}$$

Die Konstruktoren des Induktiven Typs sind Konstanten, mit denen man verschiedene Elemente des neuen Typs konstruieren kann, möglicherweise basierend auf anderen so konstruierten Elementen. Dabei ist der Grundsatz von induktiven Typen, dass die durch die Konstruktoren erzeugten Elemente die einzigen Elemente des Typs sind.

Die optionalen Parameter und Indizes können genutzt werden, um ganze Familien von induktiven Typen auf einmal zu definieren.

Aufzählungen Die einfachste Art von induktivem Typ sind Aufzählungen von endlich vielen möglichen Werten. Zum Beispiel können wir so einen Typ von Booleans definieren:

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{Bool} : \mathsf{Type} :=\\ &\quad \mid \mathsf{false} : \mathsf{Bool}\\ &\quad \mid \mathsf{true} : \mathsf{Bool}\end{aligned}$$

Die neuen Konstanten true und false sind die Konstruktoren von Bool.

Das Motto der induktiven Typen lautet: “No junk, no confusion”:

- No junk = Der Typ enthält keine Elemente, die nicht durch die Konstruktoren dargestellt werden können.
- No confusion = Elemente, die durch eine unterschiedliche Kombination von Konstruktoren erzeugt wurden, sind unterschiedlich.

Induktive Typen erlauben uns, Funktionen per Fallunterscheidung zu definieren, indem wir einen Wert der Funktion für jeden Konstruktor angeben. Dieses Prinzip heißt *Pattern-Matching*.

$$\begin{aligned}&\textbf{def}\ \ \mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}\\ &\quad \mid \mathsf{false} \Rightarrow \mathsf{true}\\ &\quad \mid \mathsf{true} \Rightarrow \mathsf{false}\end{aligned}$$

Auch bei Definitionen mit Pattern-Matching gelten dann die definitionellen Gleichheiten $\mathsf{not}\ \mathsf{false} \equiv \mathsf{true}$ und $\mathsf{not}\ \mathsf{true} \equiv \mathsf{false}$. Dies basiert auf der ι-Äquivalenzregel, die wir in Abschnitt 3.9 besprechen werden.

Die natürlichen Zahlen Hier ist ein weiteres Beispiel: die natürlichen Zahlen. Es ist ein induktiver Typ mit zwei Konstruktoren: Der Zahl 0 (zero) und der Nachfolgerfunktion (succ).

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{Nat} : \mathsf{Type} :=\\ &\quad \mid \mathsf{zero} : \mathsf{Nat}\\ &\quad \mid \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}\end{aligned}$$

Wie man hier sieht, darf der Typ eines Konstruktors also auch eine Funktion sein.

Mit den beiden Konstruktoren von Nat können wir alle natürlichen Zahlen konstruieren:

$$\begin{aligned}0 &= \mathsf{zero}\\ 1 &= \mathsf{succ}\ \mathsf{zero}\\ 2 &= \mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero})\\ 3 &= \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))\\ &\vdots\end{aligned}$$

Zur besseren Lesbarkeit schreiben wir im Folgenden auch manchmal 0 für zero, 1 für succ zero, 2 für succ (succ zero), etc.

Wieder gilt hier: „No junk, no confusion“.

- No junk: Jedes Element von Nat ist entweder zero (also 0) oder von der Form succ n für $n : \mathsf{Nat}$ (also der Nachfolger einer anderen natürlichen Zahl).

- No confusion: Es gilt $\mathsf{zero} \neq \mathsf{succ}\ n$ für alle $n : \mathsf{Nat}$ und wenn $n \neq m$, dann ist auch $\mathsf{succ}\ m \neq \mathsf{succ}\ n$

Unendliche Elemente wie etwa $\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ} \cdots))$ sind nicht erlaubt.

Über einfache Fallunterscheidungen hinaus können wir auf induktiven Typen mittels Pattern-Matching rekursive Funktionen definieren:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ &\mid x,\ \mathsf{zero} \Rightarrow x\\ &\mid x,\ \mathsf{succ}\ y \Rightarrow \mathsf{succ}\ (\mathsf{add}\ x\ y)\end{aligned}$$

Bei dieser Funktion machen wir die Fallunterscheidung nur auf dem zweiten Argument. Wichtig ist, dass mit jedem rekursiven Aufruf mindestens eins der Argumente strukturell kleiner werden muss.

Es gelten die definitionellen Gleichheiten $\mathsf{add}\ s\ \mathsf{zero} \equiv s$ und $\mathsf{add}\ s\ (\mathsf{succ}\ t) \equiv \mathsf{succ}\ (\mathsf{add}\ s\ t)$ für beliebige Terme s und t (basierend auf der ι-Äquivalenzregel). Achtung: Ist beispielsweise $z : \mathsf{Nat}$ eine Variable, dann gilt nicht die definitionelle Gleichheit $\mathsf{add}\ \mathsf{zero}\ z \equiv z$, denn diese Gleichheit kann nur über eine Fallunterscheidung des Wertes von z gezeigt werden, nicht durch reines Umschreiben mit definitionellen Gleichheiten.

Wir können auch verschachtelte Fallunterscheidungen in den Argumenten der Konstruktoren machen:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{minusTwo} : \mathsf{Nat} \to \mathsf{Nat}\\ &\mid \mathsf{zero} \Rightarrow \mathsf{zero}\\ &\mid \mathsf{succ}\ \mathsf{zero} \Rightarrow \mathsf{zero}\\ &\mid \mathsf{succ}\ (\mathsf{succ}\ y) \Rightarrow y\end{aligned}$$

Im Allgemeinen haben rekursive Definitionen die folgende Form:

$$\begin{aligned}&\textbf{def}\ \ \mathit{Name}\ (x_1 : T_1) \cdots (x_m : T_m)\ : S_1 \to \ldots \to S_n \to R\\ &\mid \mathit{Pattern}_{11}, \ldots, \mathit{Pattern}_{1n} \Rightarrow \mathit{Ergebnis}_1\\ &\vdots\\ &\mid \mathit{Pattern}_{k1}, \ldots, \mathit{Pattern}_{kn} \Rightarrow \mathit{Ergebnis}_k\end{aligned}$$

Die Fallunterscheidungen $\mathit{Pattern}_{ij}$ beziehen sich hierbei nur auf die Argumente nach dem ‚:' von den angegebenen Typen $S_1, \ldots, S_n$. Die Pattern können aus Konstruktoren und neuen Variablen bestehen, wobei die Variablen dann auf der rechten Seite zur Verfügung stehen.

Bemerkung: Namespaces Die Konstanten zero, succ, etc. heißen in Lean eigentlich Nat.zero, Nat.succ, etc. Man kann mit dem Befehl **open** Nat den Namespace Nat öffnen, sodass zero, succ, etc. auch ohne den Präfix ‚Nat.' zu Verfügung stehen. In diesem Skript wird die Kurzform und die Form mit Präfix austauschbar verwendet.

Induktive Typen mit Parametern Induktive Typen können *Parameter* haben. Dies sind Argumente, von denen der induktive Typ abhängt. Im Gegensatz zu Indizes müssen diese Argumente innerhalb der Definition des induktiven Types immer gleich sein. Ein Beispiel ist der Typ $\mathsf{List}\ X$, der Listen von Elementen eines anderen Typs X enthält:

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{List}.\{u\}\ (X : \mathsf{Type}_u) : \mathsf{Type}_u :=\\ &\quad \mid \mathsf{nil} : \mathsf{List}\ X\\ &\quad \mid \mathsf{cons} : X \to \mathsf{List}\ X \to \mathsf{List}\ X\end{aligned}$$

Zum Beispiel repräsentiert dann

$$\mathsf{cons}\ 5\ (\mathsf{cons}\ 7\ (\mathsf{cons}\ 2\ \mathsf{nil})) : \mathsf{List}\ \mathsf{Nat}$$

die Liste $5, 7, 2$. Bei Parametern ist zu beachten, dass die Parameter implizite Argumente der Konstruktoren werden. Die Konstruktoren hängen auch automatisch vom Universe-Level u ab.

$$\begin{aligned}&\mathsf{nil}.\{u\} : \Pi\{X : \mathsf{Type_u}\}.\ \mathsf{List}\ X\\ &\mathsf{cons}.\{u\} : \Pi\{X : \mathsf{Type_u}\}.\ X \to \mathsf{List}\ X \to \mathsf{List}\ X\end{aligned}$$

Ohne die impliziten Argumente wären die Typen der Konstruktoren nicht wohlgetypt, denn die Variable X wäre nirgendwo gebunden.

Pattern-Matching innerhalb einer Definition Wir können Pattern-Matching auch innerhalb von Termen durchführen. Dazu verwenden wir die **match**-Notation:

$$\begin{aligned}&\textbf{match}\ \mathit{term}_1, ..., \mathit{term}_n\ \textbf{with}\\ &\mid \mathit{Pattern}_{11}, ..., \mathit{Pattern}_{1n} \Rightarrow \mathit{Ergebnis}_1\\ &\vdots\\ &\mid \mathit{Pattern}_{k1}, ..., \mathit{Pattern}_{kn} \Rightarrow \mathit{Ergebnis}_k\end{aligned}$$

Hier wird eine Fallunterscheidung gemacht, mit welchen Konstruktoren die Terme $\mathit{term}_1, ..., \mathit{term}_n$ konstruiert wurden und das entsprechende $\mathit{Ergebnis}_i$ steht dann für den gesamten **match**-Ausdruck.

Die folgende Funktion zum Beispiel gibt an, für wie viele Elemente einer Liste eine gegebene Funktion p den Wert true zurückgibt:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{bcount}\ \{X : \mathsf{Type}\}\ (p : X \to \mathsf{Bool}) : \mathsf{List}\ X \to \mathsf{Nat}\\ &\quad \mid \mathsf{nil} \Rightarrow 0\\ &\quad \mid \mathsf{cons}\ x\ xs \Rightarrow\\ &\qquad \textbf{match}\ p\ x\ \textbf{with}\\ &\qquad \mid \mathsf{true} \Rightarrow \mathsf{succ}\ (\mathsf{bcount}\ p\ xs)\\ &\qquad \mid \mathsf{false} \Rightarrow \mathsf{bcount}\ p\ xs\end{aligned}$$

Dabei machen wir zunächst eine Fallunterscheidung auf der Liste und dann eine Fallunterscheidung auf $p\ x$.

Induktive Typen mit Indizes Neben Parametern können induktive Typen auch *Indizes* haben. Dies sind auch Argumente von induktiven Typen, die sich aber im Gegensatz zu Parametern auch verändern dürfen. Wir unterscheiden sie, indem wir die Parameter vor dem ‚:' schreiben und die Indizes rechts davon. Indizes werden nicht automatisch implizite Argumente der Konstruktoren.

Als ein Beispiel definieren wir einen Typ, der so viele Elemente enthält, wie durch ein Argument von Typ Nat angegeben:

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{DFin} : \mathsf{Nat} \to \mathsf{Type} :=\\ &\quad \mid \mathsf{fzero} : \Pi\{n : \mathsf{Nat}\}.\ \mathsf{DFin}\ (\mathsf{succ}\ n)\\ &\quad \mid \mathsf{fsucc} : \Pi\{n : \mathsf{Nat}\}.\ \mathsf{DFin}\ n \to \mathsf{DFin}\ (\mathsf{succ}\ n)\end{aligned}$$

Der erste Konstruktor, fzero, sagt uns, dass jeder Typ $\mathsf{DFin}\ m$ ein Element fzero enthält, wenn m von der Form $\mathsf{succ}\ n$ ist. Also enthalten $\mathsf{DFin}\ 1$, $\mathsf{DFin}\ 2$, … das Element fzero, aber $\mathsf{DFin}\ 0$ enthält es nicht. Das ist auch richtig so, denn $\mathsf{DFin}\ 0$ soll ja 0 Elemente haben.

Der Konstruktor fsucc sagt uns, dass wenn $\mathsf{DFin}\ n$ ein Element x enthält, dann enthält $\mathsf{DFin}\ (\mathsf{succ}\ n)$ ein Element $\mathsf{fsucc}\ x$. Damit ergeben sich die folgenden Typen:

$$\begin{aligned}&\mathsf{DFin}\ 0 \text{ enthält keine Elemente}\\ &\mathsf{DFin}\ 1 \text{ enthält } \mathsf{fzero}\\ &\mathsf{DFin}\ 2 \text{ enthält } \mathsf{fzero}, \mathsf{fsucc}\ \mathsf{fzero}\\ &\mathsf{DFin}\ 3 \text{ enthält } \mathsf{fzero}, \mathsf{fsucc}\ \mathsf{fzero}, \mathsf{fsucc}\ (\mathsf{fsucc}\ \mathsf{fzero})\\ &\vdots\end{aligned}$$

Mit impliziten Argumenten sieht das so aus:

$$\begin{aligned}&\mathsf{DFin}\ 0 \text{ enthält keine Elemente}\\ &\mathsf{DFin}\ 1 \text{ enthält } @\mathsf{fzero}\ 0\\ &\mathsf{DFin}\ 2 \text{ enthält } @\mathsf{fzero}\ 1, @\mathsf{fsucc}\ 1\ (@\mathsf{fzero}\ 0)\\ &\mathsf{DFin}\ 3 \text{ enthält } @\mathsf{fzero}\ 2, @\mathsf{fsucc}\ 2\ (@\mathsf{fzero}\ 1), @\mathsf{fsucc}\ 2\ (@\mathsf{fsucc}\ 1\ (@\mathsf{fzero}\ 0))\\ &\vdots\end{aligned}$$

Damit haben wir unseren ersten abhängigen Typ im Sinne von Abschnitt 3.2 definiert: ein Typ, der von einem Term abhängt.

Grundsätzlich können Parameter auch immer als Indizes definiert werden, es verkompliziert aber sowohl Definition als auch Beweise des induktiven Typs.

Strukturen Strukturen sind induktive Typen mit nur einem einzigen Konstruktor. Zum Beispiel können wir für Typen X und Y einen Typ von Paaren $\mathsf{Prod}\ X\ Y$ definieren (der *Produkttyp* von X und Y):

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{Prod}.\{u, v\}\ (X : \mathsf{Type}_u)\ (Y : \mathsf{Type}_v) : \mathsf{Type}_{\max(u,v)} :=\\ &\mid \mathsf{mk} : X \to Y \to \mathsf{Prod}\ X\ Y\end{aligned}$$

Da es nur einen Konstruktor gibt, können wir immer die Argumente diese Konstruktors wieder aus einem Element von $\mathsf{Prod}\ X\ Y$ extrahieren:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{fst}.\{u, v\}\ \{X : \mathsf{Type}_u\}\ \{Y : \mathsf{Type}_v\} : \mathsf{Prod}\ X\ Y \to X :=\\ &\mid \mathsf{mk}\ x\ y \Rightarrow x\end{aligned}$$

$$\begin{aligned}&\textbf{def}\ \ \mathsf{snd}.\{u, v\}\ \{X : \mathsf{Type}_u\}\ \{Y : \mathsf{Type}_v\} : \mathsf{Prod}\ X\ Y \to Y :=\\ &\mid \mathsf{mk}\ x\ y \Rightarrow y\end{aligned}$$

Um solche Definitionen von induktiven Typen mit einem einzigen Konstruktor abzukürzen, schreiben wir für die obigen drei Definitionen kurz:

$$\begin{aligned}&\textbf{structure}\ \ \mathsf{Prod}.\{u, v\}\ (X : \mathsf{Type}_u)\ (Y : \mathsf{Type}_v) : \mathsf{Type}_{\max(u,v)} :=\\ &\quad \mathsf{mk} :: (\mathsf{fst} : X)\ (\mathsf{snd} : Y)\end{aligned}$$

Zusätzlich zu den definitionellen Gleichheiten basierend auf der ι-Äquivalenzregel

$$\begin{aligned}\mathsf{fst}\ (\mathsf{mk}\ s\ t) &\equiv s\\ \mathsf{snd}\ (\mathsf{mk}\ s\ t) &\equiv t\end{aligned}$$

(für alle Terme s und t) fügt Lean bei Strukturen noch eine weitere Äquivalenzregel hinzu:

$$\mathsf{mk}\ (\mathsf{fst}\ s)\ (\mathsf{snd}\ s) \equiv s$$

(für alle Terme s). Wir nennen diese Regel die η-Äquivalenzregel für Strukturen.

3.5 Curry-Howard-Korrespondenz

Lean ist ein Beweisassistent. Wir können:

- mathematische Objekte definieren (wie wir bereits gemacht haben),
- Aussagen über diese Objekte treffen
- und diese Aussagen beweisen

Für alle drei Zwecke verwenden wir die Ausdrücke des Calculus of Inductive Constructions.

Aussagen sind Ausdrücke von einem speziellen Typ Prop. Aussagen funktionieren wie Typen: Sie können auch Elemente enthalten. Ein Element $a : p$ einer Aussage $p : \mathsf{Prop}$ betrachten wir als Beweis von p. Wenn p also Elemente enthält, ist p ein beweisbarer Satz. Wenn p keine Elemente enthält, dann ist p eine falsche oder zumindest unbeweisbare Aussage.

Dieses Prinzip heißt **Curry-Howard-Korrespondenz**. Ihr Motto lautet: „Aussagen als Typen, Beweise als Terme“.

Das wundervolle an der Curry-Howard-Korrespondenz ist, dass die beiden Methoden neue Typen zu konstruieren, Funktionstypen und induktive Typen, auch für Aussagen Sinn ergeben, wie wir im Folgenden sehen werden.

3.5.1 Implikationen als Funktionstypen

Seien $\mathsf{p} : \mathsf{Prop}$ und $\mathsf{q} : \mathsf{Prop}$. Elemente von p sind Beweise von p. Elemente von q sind Beweise von q. Eine Funktion $\mathsf{f} : \mathsf{p} \to \mathsf{q}$ gibt uns also für jeden Beweis von p einen Beweis von q. Wir können f also als Beweis ansehen, dass q aus p folgt. $\mathsf{p} \to \mathsf{q}$ können wir also interpretieren als die Aussage "Aus p folgt q".

Beispiel Wenn eine Aussage p gilt, dann gilt p. Das können wir wie folgt beweisen:

$$\textbf{def}\ \ \mathsf{selbstimplikation}\ (p : \mathsf{Prop}) : p \to p := \lambda x : p.\ x$$

Um solche Definitionen in Prop hervorzuheben, schreiben wir auch **theorem** statt **def**:

$$\textbf{theorem}\ \ \mathsf{selbstimplikation}\ (p : \mathsf{Prop}) : p \to p := \lambda x : p.\ x$$

Im Grunde gibt es aber keinen Unterschied zwischen Definitionen in Prop und Definitionen in Type.

Currying Wie auch bei Funktionen mit mehreren Argumenten nutzen wir Currying um Implikationen mit mehreren Bedingungen zu schreiben. Für „Wenn p und q, dann r" schreiben wir also $p \to q \to r$.

3.5.2 Quantifikationen als abhängige Funktionstypen

Auch abhängige Funktionstypen sind sinnvoll auf Prop. Sei $s : \mathsf{Prop}$ ein Ausdruck, der eine Variable $z : T$ enthält. Eine abhängige Funktion $\mathsf{f} : \Pi z : T.\ s$ gibt uns dann für jedes $z : T$ einen Beweis, dass die Aussage s für dieses z gilt. Wir können f also als Beweis ansehen, dass s für alle z gilt. $\Pi z : T.\ s$ können wir also interpretieren als die Aussage "Für alle z gilt s". Daher schreiben wir ab jetzt auch $\forall$ für Π.

Beispiel Das obige Beispiel enthielt bereits ein Beispiel für einen ‚Für alle'-Quantor, versteckt hinter unserer Kurzschreibweise für Argumente:

$$\textbf{theorem}\ \ \mathsf{selbstimplikation} : \forall p : \mathsf{Prop}.\ p \to p\ :=\ \lambda p : \mathsf{Prop}.\ \lambda x : p.\ x$$

Achtung: Ohne Klammern steht $\forall p : \mathsf{Prop}.\ p \to p$ für $\forall p : \mathsf{Prop}.\ (p \to p)$, nicht für $(\forall p : \mathsf{Prop}.\ p) \to p$,

3.5.3 Und, Oder

Andere logische Ausdrücke können wir als induktive Typen definieren, die aber in Prop statt in Type leben. Solche induktive Typen heißen auch *induktive Prädikate*.

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{And}\ (p\ q : \mathsf{Prop}) : \mathsf{Prop} :=\\ &\quad \mid \mathsf{And.intro} : p \to q \to \mathsf{And}\ p\ q\end{aligned}$$

$$\begin{array}{l}\textbf{inductive}\ \ \mathsf{Or}\ (p\ q : \mathsf{Prop}) := \\ \quad \mid \mathsf{Or.inl} : p \to \mathsf{Or}\ p\ q \\ \quad \mid \mathsf{Or.inr} : q \to \mathsf{Or}\ p\ q\end{array}$$

Die Konstruktoren And.intro, Or.inl und Or.inr werden auch *Introduktionsregeln* von And und Or genannt. Ihr Gegenstück sind Eliminationsregeln, die es uns erlauben, ein Element vom Typ And $p\ q$ oder Or $p\ q$ in seine Bestandteile zu zerlegen. Wir können sie über Fallunterscheidungen definieren:

$$\begin{array}{l}\textbf{theorem}\ \ \mathsf{And.left}\ \{p\ q : \mathsf{Prop}\} : \mathsf{And}\ p\ q \to p \\ \mid \mathsf{And.intro}\ (hp : p)\ (hq : q) \Rightarrow hp\end{array}$$

$$\begin{array}{l}\textbf{theorem}\ \ \mathsf{And.right}\ \{p\ q : \mathsf{Prop}\} : \mathsf{And}\ p\ q \to p \\ \mid \mathsf{And.intro}\ (hp : p)\ (hq : q) \Rightarrow hq\end{array}$$

$$\begin{array}{l}\textbf{theorem}\ \ \mathsf{Or.elim}\ \{p\ q\ r : \mathsf{Prop}\} : \mathsf{Or}\ p\ q \to (p \to r) \to (q \to r) \to r \\ \mid \mathsf{Or.inl}\ (hp : p),\ hpr,\ hqr \Rightarrow hpr\ hp \\ \mid \mathsf{Or.inr}\ (hq : q),\ hpr,\ hqr \Rightarrow hqr\ hq\end{array}$$

Da And ein induktiver Typ mit nur einem Konstruktor ist, können wir ihn und die Eliminationsregeln And.left und And.right kurz auch wie folgt definieren:

$$\begin{array}{l}\textbf{structure}\ \ \mathsf{And}\ (p\ q : \mathsf{Prop}) : \mathsf{Prop} := \\ \quad \mathsf{intro} :: (\mathsf{left} : p)(\mathsf{right} : q)\end{array}$$

Der Typ And ist nahezu identisch mit Prod. Der einzige Unterschied ist, dass And in Prop lebt und Prod in Type.

Die Parameter werden jeweils zu impliziten Argumenten der Konstruktoren. Die vollständigen Typen der Konstruktoren lauten also:

$$\begin{array}{l}@\mathsf{And.intro} : \forall\{p\ q : Prop\}.\ p \to q \to \mathsf{And}\ p\ q \\ @\mathsf{Or.inl} : \forall\{p\ q : Prop\}.\ p \to \mathsf{Or}\ p\ q \\ @\mathsf{Or.inr} : \forall\{p\ q : Prop\}.\ q \to \mathsf{Or}\ p\ q\end{array}$$

Beispiel

$$\begin{array}{l}\textbf{theorem}\ \ \mathsf{or\_of\_and}\ (p\ q : \mathsf{Prop}) : \mathsf{And}\ p\ q \to \mathsf{Or}\ p\ q\ := \\ \quad \lambda h : \mathsf{And}\ p\ q.\ \mathsf{Or.inl}\ (\mathsf{And.left}\ h)\end{array}$$

Wir schreiben auch $p \wedge q$ für And $p\ q$ und $p \vee q$ für Or $p\ q$:

$$\begin{array}{l}\textbf{theorem}\ \ \mathsf{or\_of\_and}\ (p\ q : \mathsf{Prop}) : p \wedge q \to p \vee q\ := \\ \quad \lambda h : p \wedge q.\ \mathsf{Or.inl}\ (\mathsf{And.left}\ h)\end{array}$$

3.5.4 True, False, Negation

True ist eine Aussage, die uneingeschränkt wahr ist:

$$\textbf{inductive}\ \ \mathsf{True} : \mathsf{Prop} :=$$
$$\quad \mid \mathsf{True.intro} : \mathsf{True}$$

True hat nur eine Introduktionsregel True.intro. Es gibt keine Eliminationsregel, denn eine Annahme vom Typ True liefert keinen Mehrwert.

False ist eine Aussage, die uneingeschränkt falsch ist:

$$\textbf{inductive}\ \ \mathsf{False} : \mathsf{Prop} := \textit{[Induktiver Typ ohne Konstruktoren]}$$

False hat also keine Introduktionsregel, denn es existiert kein Beweis von False. In manchen Fällen können wir einen anscheinenden Beweis von False produzieren, wenn die Annahmen bereits falsch oder widersprüchlich sind. False besitzt aber eine Eliminationsregel, sogar eine sehr starke:

$$\textbf{def}\ \ \mathsf{False.elim}\ \{p : \mathsf{Prop}\} : \mathsf{False} \to p$$
$$\textit{[Leere Fallunterscheidung]}$$

Diese Regel ist auch bekannt als: "Aus falschem folgt beliebiges."

Negation wird mithilfe von False definiert:

$$\textbf{def}\ \ \mathsf{Not}\ (p : \mathsf{Prop}) : \mathsf{Prop} := p \to \mathsf{False}$$

Beispiele

$$\textbf{theorem}\ \ \mathsf{widerspruch}\ \{p\ q : \mathsf{Prop}\}\ (hp : p)\ (hnp : \mathsf{Not}\ p) : q :=$$
$$\quad \mathsf{False.elim}\ (hnp\ hp)$$

Wir schreiben auch $\neg p$ für Not p. Hier ein weiteres Beispiel, diesmal mit dieser Kurzschreibweise:

$$\textbf{theorem}\ \ \mathsf{kontraposition}\ \{p\ q : \mathsf{Prop}\}\ (hpq : p \to q)\ (hnq : \neg q) : \neg p :=$$
$$\quad \lambda hp : p.\ hnq\ (hpq\ hp)$$

3.5.5 Äquivalenz

Äquivalenz von Aussagen (auch bekannt als „genau dann wenn") können wir wie folgt definieren:

$$\textbf{structure}\ \ \mathsf{Iff}\ (p\ q : \mathsf{Prop}) : \mathsf{Prop} :=$$
$$\quad \mathsf{Iff.intro} :: (\mathsf{mp} : p \to q)\ (\mathsf{mpr} : q \to p)$$

Die Introduktionsregel von Iff ist also

$$@\mathsf{Iff.intro} : \forall \{p\ q : \mathsf{Prop}\}.\ (p \to q) \to (q \to p) \to \mathsf{Iff}\ p\ q$$

und die Eliminationsregeln von Iff heißen

$$@\mathsf{Iff.mp} : \forall\{p\ q : \mathsf{Prop}\}.\ \mathsf{Iff}\ p\ q \to (p \to q) \qquad \text{(modus ponens)}$$
$$@\mathsf{Iff.mpr} : \forall\{p\ q : \mathsf{Prop}\}.\ \mathsf{Iff}\ p\ q \to (q \to p) \qquad \text{(modus ponens, reversed)}$$

Wir scheiben für $\mathsf{Iff}\ p\ q$ auch $p \leftrightarrow q$.

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{and\_comm}\ (p\ q : \mathsf{Prop}) : p \wedge q \leftrightarrow q \wedge p\ :=\\ &\quad \mathsf{Iff.intro}\\ &\qquad (\lambda h : p \wedge q.\ \mathsf{And.intro}\ (\mathsf{And.right}\ h)\ (\mathsf{And.left}\ h))\\ &\qquad (\lambda h : q \wedge p.\ \mathsf{And.intro}\ (\mathsf{And.right}\ h)\ (\mathsf{And.left}\ h))\end{aligned}$$

3.5.6 Existenz

Auch der Existenz-Quantor kann als induktiver Datentyp repräsentiert werden:

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{Exists}.\{u\}\ \{X : \mathsf{Type}_u\}\ (p : X \to \mathsf{Prop}) : \mathsf{Prop} :=\\ &\quad \mid \mathsf{Exists.intro} : \forall x : X.\ p\ x \to \mathsf{Exists}\ p\end{aligned}$$

Zur Konstruktion eines Elements/Beweises von $\mathsf{Exists}\ p$ brauchen wir also ein konkretes Beispiel $x : X$, für das $p\ x$ gilt. Wieder werden Parameter zu impliziten Argumenten des Konstruktors:

$$@\mathsf{Exists.intro}.\{u\} : \forall\{X : \mathsf{Type}_u\}\ \{p : X \to \mathsf{Prop}\}\ (x : X).\ p\ x \to \mathsf{Exists}\ p$$

Die zugehörige Eliminationsregel lautet:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{Exists.elim}.\{u\}\ \{X : \mathsf{Type}_u\}\ \{p : X \to \mathsf{Prop}\}\ \{q : \mathsf{Prop}\} :\\ &\quad \mathsf{Exists}\ p \to (\forall x : X.\ p\ x \to q) \to q\\ &\quad \mid \mathsf{Exists.intro}\ x\ hp,\ hpq \Rightarrow hpq\ x\ hp\end{aligned}$$

Wenn wir wissen, dass $\mathsf{Exists}\ p$ gilt, können wir also beim Beweisen von einer Aussage q zusätzlich ein Element $x : X$ mit $p\ a$ annehmen. Wir kommen später darauf zurück, warum wir nicht einfach eine Eliminationsregel der Form $\mathsf{Exists}\ p \to X$ definieren können, die uns das konkrete Beispiel wieder zurückgibt.

Für $\mathsf{Exists}\ (\lambda x : X.\ ...)$ schreiben wir auch $\exists x : X.\ ...$ oder $\exists x.\ ....$

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{Exists.map}\ \{p\ q : \mathsf{Nat} \to \mathsf{Prop}\}\ (hex : \exists n.\ p\ n)\ (hpq : \forall x.\ p\ x \to q\ x) :\\ &\quad \exists n.\ q\ n :=\\ &\ \mathsf{Exists.elim}\ hex\ (\lambda a : \mathsf{Nat}.\ \lambda hp : p\ a.\ \mathsf{Exists.intro}\ a\ (hpq\ a\ hp))\end{aligned}$$

3.5.7 Strukturierte Beweise

Das Schreiben von formalen Beweisen kann es schnell unübersichtlich werden. Wir führen daher die folgenden zwei Schreibweisen ein.

Die Schreibweise

$$\textbf{show } t \textbf{ from } s$$

(mit beliebigen Termen t und s) steht für

$$(\lambda x : t,\ x)\ s$$

Die **show**-Notation können wir nutzen, um daran zu erinnern, was genau an einer bestimmten Stelle bewiesen werden soll. Dies kann auch hilfreich sein, um Umformungen von definitionell gleichen Termen explizit zu machen. Zum Beispiel:

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{not\_false} : \neg\ \mathsf{False} :=\\ &\quad \textbf{show}\ \mathsf{False} \to \mathsf{False}\ \textbf{from}\ \lambda h.\ h\end{aligned}$$

Hierdurch machen wir offensichtlicher, dass wir ausnutzen, dass $\neg\ \mathsf{False}$ definitionell gleich $\mathsf{False} \to \mathsf{False}$ ist.

Die Schreibweise

$$\begin{aligned}&\textbf{have } h : t := s\\ &r\end{aligned}$$

(mit beliebigen Termen t, s und r) steht für

$$(\lambda h : t,\ r)\ s$$

Mit der **have**-Notation können wir einen Beweis elegant in kleinere Teile zerlegen. Zum Beispiel können wir den Beweis

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{or\_of\_and}\ (p\ q : \mathsf{Prop})\ (h : p \wedge q) : p \vee q\ :=\\ &\quad \mathsf{Or.inl}\ (\mathsf{And.left}\ h)\end{aligned}$$

nun wie folgt umschreiben:

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{or\_of\_and}\ (p\ q : \mathsf{Prop})\ (h : p \wedge q) : p \vee q\ :=\\ &\quad \textbf{have } hp : p := \mathsf{And.left}\ h\\ &\quad \textbf{show } p \vee q \textbf{ from } \mathsf{Or.inl}\ hp\end{aligned}$$

3.5.8 Gleichheit

Gleichheit können wir wie folgt definieren:

$$\begin{aligned}&\textbf{inductive}\ \ \mathsf{Eq}.\{u\}\ \{X : \mathsf{Type}_u\}\ (x : X) : X \to \mathsf{Prop} :=\\ &\quad \mid\ \mathsf{rfl} : \mathsf{Eq}\ x\ x\end{aligned}$$

Dieser induktive Typ hat nur einen Konstruktor, rfl, der ein Element von $\mathsf{Eq}\ x\ x$ konstruiert. Das erste Argument von Eq ist ein Parameter, das zweite ein Index, damit wir das zweite Argument im Typ des Konstruktors gleich mit dem ersten wählen können. Zunächst scheint es als könnten wir damit nur über syntaktische oder höchstens definitionelle Gleichheit sprechen. Haben wir aber auch Annahmen vom Typ Eq, können wir auch Gleichheit von Termen beweisen, die nicht definitionell gleich sind. Dabei hilft uns die folgende Eliminationsregel:

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{Eq.subst}.\{u\}\ \{X : \mathsf{Type}_u\}\ \{p : X \to \mathsf{Prop}\}\ \{x\ y : X\}\\ &\qquad (heq : \mathsf{Eq}\ x\ y)\ (hp : p\ x) : p\ y :=\\ &\quad \textbf{match}\ heq\ \textbf{with}\\ &\quad \mid \mathsf{rfl} \Rightarrow hp\end{aligned}$$

Diese Definition scheint zunächst nicht wohlgetypt zu sein, denn wir müssen ja eigentlich ein Element von $p\ y$ konstruieren, während hp vom Typ $p\ x$ ist. Aber durch die Fallunterscheidung, in der wir $heq : \mathsf{Eq}\ x\ y$ mit $\mathsf{rfl} : \mathsf{Eq}\ x\ x$ identifizieren, können wir annehmen, dass x und y identisch sind und somit auch $p\ x$ und $p\ y$. Diese Argumentation wird klarer werden, wenn wir uns in Abschnitt 3.9 mit Rekursoren beschäftigen, die die Grundlage von Fallunterscheidungen sind.

Statt $\mathsf{Eq}\ x\ y$ schreiben wir auch $x = y$.

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{Eq.symm}.\{u\}\ \{X : \mathsf{Type}_u\}\ \{x\ y : X\}\ (h : x = y) : y = x :=\\ &\quad @\mathsf{Eq.subst}\ X\ (\lambda z.\ z = x)\ x\ y\ h\ \mathsf{rfl}\end{aligned}$$

Beweisterme über Eq zu schreiben, etwa mit $\mathsf{Eq.subst}$, ist oft mühsam. Die impliziten Argumente von $\mathsf{Eq.subst}$ können von Lean häufig nicht korrekt ergänzt werden. Daher gibt es eine spezielle Notation für $\mathsf{Eq.subst}$, die einfacher zu verwenden ist. Sie lautet:

$$s \blacktriangleright t$$

Hierbei ist $s : u = v$ für Terme u und v, sodass der Typ von t und der erwartete Typ an der Stelle von ‚$s \blacktriangleright t$' bis auf Ersetzungen von u durch v (oder umgekehrt) identisch sind.

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{beispiel}\ (a\ b : \mathsf{Nat})\ (p : \mathsf{Nat} \to \mathsf{Prop})\ (hab : a = b)\ (hpa : p\ a) : p\ b :=\\ &\quad hab \blacktriangleright hpa\end{aligned}$$

Um *Gleichungen* mithilfe von anderen Gleichungen zu zeigen, ist häufig die folgende Syntax eleganter:

$$\textbf{by}\ \mathrm{rw}\ [s]$$

Das ‚rw' steht hier für ‚rewrite', also ‚umschreiben'. Hierbei ist $s : u = v$ für Terme u und v, sodass der erwartete Typ an der Stelle von ‚**by** rw $[s]$' von der Form $t = r$ ist und die Terme t und r bis auf Ersetzungen von u durch v (oder umgekehrt) identisch sind.

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{beispiel}\ (a\ b : \mathsf{Nat})\ (hab : a = b) : \mathsf{succ}\ a = \mathsf{succ}\ b :=\\ &\quad \textbf{by}\ \mathrm{rw}\ [hab]\end{aligned}$$

Die rw-Notation ist ein Beispiel für eine Taktik. Taktiken sind kleine Programme, die versuchen, einen Beweisterm eines bestimmten Typs zu produzieren. Taktiken können auch kombiniert, hintereinander geschaltet und verschachtelt werden, sodass viele Nutzer von Lean fast gar keine Beweisterme mehr von Hand schreiben. Diese Taktiken alle zu beschreiben, sprengt allerdings den Rahmen dieses Skriptes.

Häufig besteht der Beweis einer Gleichung aus mehreren Schritten. Dann ist folgende Notation hilfreich:

$$\begin{aligned}&\textbf{calc}\ s_0\\ &\quad \_ = s_1 \quad := t_1\\ &\quad \vdots\\ &\quad \_ = s_n \quad := t_n\end{aligned}$$

Hier ist jeweils $t_i : s_{i-1} = s_i$ und die gesamte Notation ist dann ein Beweis von $s_0 = s_n$. Achtung: In Lean muss man darauf achten, alle „_“ gleich weit einzurücken, denn sonst kommt es zu verwirrenden Fehlermeldungen.

Beispiel

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{beispiel}\ (a : \mathsf{Nat})\ (f : \mathsf{Nat} \to \mathsf{Nat})\ (h : \forall x.\ f\ x = x) :\\ &\quad f\ (f\ (f\ a)) = a :=\\ &\ \ \textbf{calc}\ f\ (f\ (f\ a))\\ &\quad \_ = f\ (f\ a) := h\ (f\ (f\ a))\\ &\quad \_ = f\ a := h\ (f\ a)\\ &\quad \_ = a := h\ a\end{aligned}$$

3.5.9 Induktion

Wie bei Definitionen in Type können wir auch in Prop rekursive Definitionen machen. Wir nennen solche Definitionen dann Beweise durch Induktion.

Beispiel Zur Erinnerung noch einmal die Definition der Addition auf Nat:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ &\mid x,\ \mathsf{zero} \Rightarrow x\\ &\mid x,\ \mathsf{succ}\ y \Rightarrow \mathsf{succ}\ (\mathsf{add}\ x\ y)\end{aligned}$$

Da also $t + 0$ und t für beliebige Terme t definitionell gleich sind, können wir leicht beweisen, dass $x + 0 = x$:

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{add\_zero}\ (x : \mathsf{Nat}) : x + 0 = x\ \ :=\\&\quad \mathsf{rfl}\end{aligned}$$

Zu zeigen, dass $0 + x = x$, ist allerdings schwieriger:

$$\begin{aligned}&\textbf{theorem}\ \ \mathsf{zero\_add}\ (x : \mathsf{Nat}) : 0 + x = x\ \ :=\\&\quad \textbf{match}\ x\ \textbf{with}\\&\quad \mid 0 \Rightarrow\\&\qquad \textbf{show}\ \ 0 + 0 = 0\ \textbf{from}\ \mathsf{rfl}\\&\quad \mid \mathsf{succ}\ y \Rightarrow\\&\qquad \textbf{calc}\ \ 0 + \mathsf{succ}\ y\\&\qquad \_ = \mathsf{succ}\ (0 + y) := \mathsf{rfl}\\&\qquad \_ = \mathsf{succ}\ y := \textbf{by}\ \mathrm{rw}\ [\mathsf{zero\_add}\ y]\end{aligned}$$

Wir machen eine Fallunterscheidung, welcher Konstruktor die natürliche Zahl x erzeugt hat: zero oder $\mathsf{succ}\ y$ mit einer anderen natürlichen Zahl y.

Im Fall von zero müssen wir dann zeigen, dass $0 + 0 = 0$ (hier wurde in der zu zeigenden Aussage einfach x durch 0 ersetzt). Per Definition von add sind $0 + 0$ und 0 definitionell gleich und somit ist rfl ein gültiger Beweisterm.

Im Fall von $\mathsf{succ}\ y$ müssen wir zeigen, dass $0 + (\mathsf{succ}\ y) = \mathsf{succ}\ y$. Per Definition von add ist $0 + \mathsf{succ}\ y$ definitionell gleich $\mathsf{succ}\ (0 + y)$. Der Beweis für diese Gleichheit ist also einfach rfl. Es verbleibt dann zu zeigen, dass $\mathsf{succ}\ (0 + y) = \mathsf{succ}\ y$. Wenn die beiden „$\mathsf{succ}$“s für einen Moment ignorieren, wäre es also hilfreich zu zeigen, dass $0 + y = y$. Das ist genau die Aussage des Gesamttheorems, mit y statt x. Da y strukturell kleiner ist als x, können wir das Theorem $\mathsf{zero\_add}$ rekursiv aufrufen und erhalten mit $\mathsf{zero\_add}\ y$ einen Beweis von $0 + y = y$. Um damit einen Beweis von $\mathsf{succ}\ (0 + y) = \mathsf{succ}\ y$ zu erhalten, nutzen wir die rw-Taktik.

3.5.10 Satz des ausgeschlossenen Dritten

Die Beweise und logischen Regeln, die wir bisher gesehen haben, sind alle konstruktiv. Das heißt, sie sind λ-Terme, die als ausführbare Programme angesehen werden können. Haben wir zum Beispiel die Existenz eines mathematischen Objektes bewiesen, können wir einen Algorithmus daraus herleiten, um dieses Objekt zu berechnen.

Leider können wir konstruktiv aber nicht alles beweisen, was man üblicherweise in der Mathematik als wahr ansieht, z.B. den Satz vom ausgeschlossenen Dritten (= excluded middle):

$$\mathsf{em} : \forall p : \mathsf{Prop}.\ p \vee \neg p$$

Dieser Satz kann mit unseren bisherigen Mitteln nicht bewiesen werden, denn ein Beweis von „Oder“ müsste konkret angeben, ob die linke ($\mathsf{Or.inl}$) oder die

rechte Seite (Or.inr) bewiesen wird. Wenn wir diesen Satz trotzdem verwenden wollen, können wir ihn als Axiom hinzufügen:

$$\textbf{axiom}\ \ \mathsf{em} : \forall p : \mathsf{Prop}.\ p \vee \neg p$$

Im Gegensatz zu Definitionen und induktiven Typen, bergen Axiome allerdings immer die Gefahr, dass wir unser gesamtes System widersprüchlich machen. Nichts hält uns beispielsweise davon ab, ein Axiom

$$\textbf{axiom}\ \ \mathsf{alles\_wahr} : \forall p : \mathsf{Prop}.\ p$$

hinzuzufügen, womit wir dann jede Aussage beweisen könnten.

3.6 Sorten

Wie auch Elemente von Type_u können Elemente von Prop also auch selbst Elemente haben. Wenn wir über Type_u und Prop gemeinsam sprechen wollen, sprechen wir von *Sorten*. Wir legen fest, dass $\mathsf{Prop} : \mathsf{Type}$ und daraus ergibt sich dann eine Hierarchie von Sorten, die unsere bisherige Hierarchie von Typen erweitert:

$$\begin{aligned}
\mathsf{Prop} &: \mathsf{Type} \\
\mathsf{Type} &: \mathsf{Type}_1 \\
\mathsf{Type}_1 &: \mathsf{Type}_2 \\
\mathsf{Type}_2 &: \mathsf{Type}_3 \\
&\vdots
\end{aligned}$$

Wir nennen daher Prop auch Sort_0 und schreiben Sort_{n+1} für Type_n. Somit ist Type also mit Type_0 und Sort_1 identisch.

3.7 Typisierungsregeln

Die Typisierungsregeln des Calculus of Inductive Constructions (Lean-style) lauten:

Konst $\dfrac{}{\Gamma \vdash \mathsf{c} : T}$ wenn c eine Konstante von Typ T ist

Var $\dfrac{}{\Gamma \vdash x : \Gamma(x)}$

Anw $\dfrac{\Gamma \vdash t : \Pi x : S.\ T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[x := s]}$

Lam $\dfrac{\Gamma,\ x : S \vdash t : T \qquad \Gamma \vdash S : \mathsf{Sort}_u}{\Gamma \vdash (\lambda x : S.\ t) : (\Pi x : S.\ T)}$

Sort $\dfrac{}{\Gamma \vdash \mathsf{Sort}_u : \mathsf{Sort}_{u+1}}$

Pi $\dfrac{\Gamma \vdash S : \mathsf{Sort}_u \qquad \Gamma,\ x : S \vdash T : \mathsf{Sort}_v}{\Gamma \vdash \Pi x : S.\ T : \mathsf{Sort}_{\mathrm{imax}(u,v)}}$

wobei

$$\mathrm{imax}(u, v) = \begin{cases} 0 & \text{wenn v} = 0 \\ \max(u, v) & \text{sonst} \end{cases}$$

DefGl $\dfrac{\Gamma \vdash t : T}{\Gamma \vdash t : S}$ wenn $T \equiv S$

Die Unterschiede gengenüber den Typisierungsregeln ohne Aussagen sind nur gering. Die Regeln Konst, Var, Anw und DefGl sind identisch. Die Regel Lam ersetzt Type durch Sort. Die neue Regel Sort veralgemeinert die alte Regel Type, indem sie die neue Typisierung $\mathsf{Prop} : \mathsf{Type}$ mit einschließt.

Die Regel Pi bedarf allerdings einiger Erklärungen. Solange der Rückgabetyp T keine Aussage ist (also solange $T : \mathsf{Sort}_v$ mit $v \neq 0$), bleibt alles beim alten: Für Funktionstypen nehmen wir das größere der beiden Universe-Levels $max(u, v)$ zwischen dem Universe-Level u des Argumenttyps und dem Universe-Level v des Rückgabetyps. Zum Beispiel:

$$\mathsf{Type} \to \mathsf{Type} : \mathsf{Type}_1$$

oder, umgeschrieben in langer Schreibweise:

$$(\Pi x : \mathsf{Sort}_1.\ \mathsf{Sort}_1) : \mathsf{Sort}_2$$

Wie schon eingangs erwähnt, ist dies notwendig, um das gesamte System frei von Widersprüchen zu halten (Girards Paradoxon).

Ist allerdings der Rückgabetyp T eine Aussage (also $T : \mathsf{Prop}$ oder äquivalent $T : \mathsf{Sort}_0$), dann ist der Funktionstyp auch vom Typ Prop. Zum Beispiel:

$$(\forall x : \mathsf{Nat}.\ x = x) : \mathsf{Prop}$$

oder, umgeschrieben in langer Schreibweise:

$$(\Pi x : \mathsf{Nat}.\ \mathsf{Eq}\ x\ x) : \mathsf{Sort}_0$$

und das obwohl $\mathsf{Nat} : \mathsf{Sort}_1$. Das ist gut so, denn wir würden ja gerne $\forall x : \mathsf{Nat}.\ x = x$ als eine Aussage interpretieren und sie sollte deshalb ein Element von Prop sein, nicht von Type. Diese spezielle Eigenschaft von Prop nennt sich *Impredikativität*. Um dies zu erlauben ohne das System widersprüchlich zu machen, müssen wir allerdings einen Kompromiss eingehen:

3.8 Large Elimination und Beweis-Irrelevanz

Bei Pattern-Matching auf Typen (d.h. Sorten Sort_u mit $u > 0$) können wir beliebige Rückgabewerte verwenden. Zum Beispiel

$$\begin{aligned}&\textbf{def}\ \ \mathsf{natOrListNat} : \mathsf{Bool} \to \mathsf{Type}\\ &\mid \mathsf{true} \Rightarrow \mathsf{Nat}\\ &\mid \mathsf{false} \Rightarrow \mathsf{List\ Nat}\end{aligned}$$

Hierbei spielt es keine Rolle, dass der Rückgabetyp $\mathsf{Type} : \mathsf{Type}_1$ in einem höheren Universe-Level lebt als der Typ $\mathsf{Bool} : \mathsf{Type}_0$, auf dem die Fallunterscheidung gemacht wird. Diese Eigenschaft heißt Large Elimination.

Auf Prop ist Large Elimination nicht zulässig. Bei einem Pattern-Matching auf Beweisen (d.h. auf Elementen von induktiven Prädikaten) muss der Rückgabetyp des Pattern-Matching auch vom Typ Prop sein. Mit Pattern-Matching auf Beweisen dürfen also nur andere Beweise konstruiert werden, keine Elemente von Typen. Zum Beispiel ist die folgende Definition ungültig:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{isInl}\ \{p\ q : \mathsf{Prop}\} : \mathsf{Or}\ p\ q \to \mathsf{Bool}\\ &\mid \mathsf{Or.inl}\ (h : p) \Rightarrow \mathsf{true}\\ &\mid \mathsf{Or.inr}\ (h : q) \Rightarrow \mathsf{false}\end{aligned}$$

Dies ist ungültig, da $\mathsf{Bool} : \mathsf{Sort}_1$ ein höheres Universe-Level hat als $\mathsf{Or\ p\ q} : \mathsf{Prop}$.

Ein weiteres gutes Beispiel ist Pattern-Matching auf Beweisen von Exists. Folgende Definition ist ebenfalls ungültig:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{witness}\ \{p : \mathsf{Nat} \to \mathsf{Prop}\} : \mathsf{Exists\ p} \to \mathsf{Nat}\\ &\mid \mathsf{Exists.intro}\ (x : \mathsf{Nat})\ (h : p\ x) \Rightarrow x\end{aligned}$$

Denn $\mathsf{Nat} : \mathsf{Type}$ ist nicht in Prop.

Der gewünschte Effekt hiervon, der Widersprüche verhindert, ist, dass keine zwei Elemente (also Beweise) von einer Aussage $p : \mathsf{Prop}$ unterschieden werden

können. Zum Beispiel sind Exists.intro 7 True.intro und Exists.intro 42 True.intro zwei Elemente von $\exists x : \mathsf{Nat}.\ \mathsf{True}$. Aber wir können keine Funktion $(\exists x : \mathsf{Nat}.\ \mathsf{True}) \to \mathsf{Nat}$ definieren, die für den ersten Beweis 7 und für den zweiten 42 zurückgibt.

Es gibt aber eine Ausnahme: Der Rückgabetyp darf in einem beliebigen Universe-Level sein, wenn das induktive Prädikat höchstens einen Konstruktor hat und jedes Argument des Konstruktors entweder einen Aussagentyp hat oder ein Index ist. Die Idee ist, dass in diesem Fall ein Pattern-Matching auf diesem Konstruktor keine Informationen liefert, um ein Element des induktiven Prädikats von einem anderen unterscheiden zu können. Dieser Spezialfall heißt *Singleton-Elimination*. Er sorgt dafür, dass False und Eq beliebiges Pattern-Matching erlauben.

Über ein Verbot von Large Elimination in Prop hinaus können wir ein Axiom einführen, das besagt, dass zwei Beweise eine Aussage stets gleich sind:

$$\textbf{axiom}\ \ \mathsf{proofIrrel}\ \{p : Prop\}\ (h_1\ h_2 : p) : h_1 = h_2$$

Diese Eigenschaft heißt Beweisirrelevanz. In Lean ist dies allerdings nicht als Axiom realisiert, sondern als zusätzliche definitionelle Gleichheit in Leans Typentheorie eingebaut:

$$\textbf{theorem}\ \ \mathsf{proofIrrel}\ \{p : Prop\}\ (h_1 h_2 : p) : h_1 = h_2 := \mathsf{rfl}$$

3.9 Rekursoren

Fallunterscheidungen und rekursive Funktionen werden intern durch Rekursoren implementiert. Jeder induktive Typ besitzt einen Rekursor. Dies ist eine Konstante ohne Definition, deren Typ von der Spezifikation des induktiven Typs abhängt.

3.9.1 Fallunterscheidungen

Dies ist der Rekursor von Bool:

$$\mathsf{Bool.rec}.\{u\} : \Pi\{motive : \mathsf{Bool} \to \mathsf{Sort}_u\}.\ motive\ \mathsf{false} \to motive\ \mathsf{true} \to \Pi t : \mathsf{Bool}.\ motive\ t$$

In Abschnitt 3.4 haben wir Negation wie folgt definiert:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}\\ &\quad \mid\ \mathsf{false} \Rightarrow \mathsf{true}\\ &\quad \mid\ \mathsf{true} \Rightarrow \mathsf{false}\end{aligned}$$

Eine solche Fallunterscheidung bricht Lean intern auf eine Anwendung des Rekursors herunter. Mit dem Rekursor lautet die Definition dann wie folgt:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{not}\ (b : \mathsf{Bool}) : \mathsf{Bool} :=\\ &\quad @\mathsf{Bool.rec}\ (\lambda\_.\ \mathsf{Bool})\ \mathsf{true}\ \mathsf{false}\ b\end{aligned}$$

Das Motiv *motive* gibt an, welchen Rückgabetyp die Fallunterscheidung haben soll. Dabei wird ein Argument vom Typ Bool zugelassen, damit der Rückgabetyp von den Argumenten abhängen kann und wir somit auch abhängige Funktionen definieren können. Im Fall von not ist der Rückgabetyp aber unabhängig vom Argument stets Bool und daher ist unser Motiv ($\lambda\_.$ Bool). (Der Unterstrich „_“ kennzeichnet hier eine Variable, die im Inneren der λ-Abstraktion nicht verwendet wird und somit keinen eigenen Namen benötigt.) Das zweite und dritte Argument des Rekursors, die *Minor-Prämissen*, geben die Rückgabewerte für die verschiedenen Konstruktoren an. Das zweite ist der Rückgabewert für false und das dritte der Rückgabewert für true. Mit dem Motiv ($\lambda\_.$ Bool) ist der erwartete Typ des zweiten Arguments ($\lambda\_.$ Bool) false $\equiv$ Bool und der des dritten Arguments ist ($\lambda\_.$ Bool) true $\equiv$ Bool. Wir können also die Argumente true und false angeben. Das letzte Argument des Rekursors, die *Major-Prämisse*, gibt den Term an, auf dem die Fallunterscheidung gemacht werden soll.

Definitionelle Gleichheiten von Rekursoren Rekursoren besitzen eine Äquivalenzregel, ι-Äquivalenz, die besagt, dass wenn die Major-Prämisse ein Konstruktor (evtl. mit Argumenten) ist, dass dann die gesamte Anwendung des Rekursors definitionell gleich der entsprechenden Minor-Prämisse ist. Im Fall von Bool also:

$$\begin{aligned} @\mathsf{Bool.rec}\ m\ s_1\ s_2\ \mathsf{false} &\equiv_\iota s_1 \\ @\mathsf{Bool.rec}\ m\ s_1\ s_2\ \mathsf{true} &\equiv_\iota s_2 \end{aligned}$$

3.9.2 Rekursion

Der Rekursor von Nat sieht wie folgt aus:

$$\begin{aligned} &\mathsf{Nat.rec}.\{u\} : \Pi\{\mathit{motive} : \mathsf{Nat} \to \mathsf{Sort}_u\}. \\ &\quad \mathit{motive}\ \mathsf{zero} \to \\ &\quad (\Pi n : \mathsf{Nat}.\ \mathit{motive}\ n \to \mathit{motive}\ (\mathsf{succ}\ n)) \to \\ &\quad \Pi t : \mathsf{Nat}.\ \mathit{motive}\ t \end{aligned}$$

Dadurch, dass der Konstruktor succ ein Argument vom Typ Nat hat, fällt die entsprechende Minor-Prämisse etwas komplexer aus. Sie nimmt ein Argument n vom Typ Nat, das das Argument von succ repräsentiert, und zusätzlich ein Argument vom Typ *motive* n, das für Rekursion verwendet werden kann.

Die entsprechenden ι-Äquivalenzregeln lauten wie folgt:

$$\begin{aligned} @\mathsf{Nat.rec}\ m\ z\ s\ \mathsf{zero} &\equiv_\iota z \\ @\mathsf{Nat.rec}\ m\ z\ s\ (\mathsf{succ}\ n) &\equiv_\iota s\ n\ (@\mathsf{Nat.rec}\ m\ z\ s\ n) \end{aligned}$$

Ist die Major-Prämisse also zero, dann können wir den Rekursor einfach auf die Minor-Prämisse für zero reduzieren. Ist die Major-Prämisse von der Form $\mathsf{succ}\ n$, dann verwenden wir die Minor-Prämisse für succ. Dabei ist das erste Argument der Minor-Prämisse die Zahl n und das zweite Argument ist das Ergebnis der Anwendung desselben Rekursors auf n. Dieses zweite Argument der Minor-Prämisse ermöglicht Rekursion.

Wir nutzen nun den Rekursor, um eine rekursive Funktion zu definieren. Zur Erinnerung ist hier die Definition der Addition aus Abschnitt 3.4.

$$\begin{aligned}&\textbf{def}\ \ \mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ &\mid x,\ \mathsf{zero} \Rightarrow x\\ &\mid x,\ \mathsf{succ}\ z \Rightarrow \mathsf{succ}\ (\mathsf{add}\ x\ z)\end{aligned}$$

Eine solche rekursive Funktion wird intern auch mithilfe des Rekursors implementiert:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{add}\ (x\ y : \mathsf{Nat}) : \mathsf{Nat} :=\\ &\quad @\mathsf{Nat.rec}\ (\lambda\_.\ \mathsf{Nat})\ x\ (\lambda z : \mathsf{Nat}.\ \lambda r : \mathsf{Nat}.\ \mathsf{succ}\ r)\ y\end{aligned}$$

Der Rückgabetyp von add ist stets Nat und daher wählen wir $\lambda\_.\ \mathsf{Nat}$ als Motiv. Die Fallunterscheidung ist auf dem zweiten Argument von add, daher ist die Major-Prämisse (d.h. das letzte Argument vom Rekursor) y. Im Fall $y = \mathsf{zero}$ ist das Ergebnis x und daher ist die entsprechende Minor-Prämisse x. Für die andere Minorprämisse, für den Fall succ, ist eine Funktion $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ gefragt, denn $(\lambda\_.\ \mathsf{Nat})\ n \equiv \mathsf{Nat}$ und $(\lambda\_.\ \mathsf{Nat})\ (\mathsf{succ}\ n) \equiv \mathsf{Nat}$. Ihr erstes Argument, das wir z nennen, ist das Argument von succ. Ihr zweites Argument, das wir r nennen, entspricht dem rekursiven Aufruf $\mathsf{add}\ x\ z$. Um das Ergebnis $\mathsf{succ}\ (\mathsf{add}\ x\ z)$ zu bekommen, wählen wir also die Funktion $(\lambda z : \mathsf{Nat}.\ \lambda r : \mathsf{Nat}.\ \mathsf{succ}\ r)$.

Hierbei ist essentiell, dass beim rekursiven Aufruf das Argument, auf dem die Fallunterscheidung gemacht wird, genau um eins verringert wird und das andere Argument gleich bleibt. Wäre das anders, müssten wir ein komplizierteres Motiv wählen oder eventuell den Rekursor mehrfach aufrufen.

Der Term $\mathsf{add}\ 1\ 1$ lässt sich dann mit der ι-Regel wie folgt auf 2 reduzieren:

$$\begin{aligned}&\mathsf{add}\ 1\ 1\\ \equiv\ &(\lambda x\ y.\ \mathsf{Nat.rec}\ x\ (\lambda z\ r.\ \mathsf{succ}\ r)\ y)\ 1\ 1 && (\delta)\\ \equiv\ &\mathsf{Nat.rec}\ 1\ (\lambda z\ r.\ \mathsf{succ}\ r)\ (\mathsf{succ}\ \mathsf{zero}) && (\beta)\\ \equiv\ &(\lambda z\ r.\ \mathsf{succ}\ r)\ \mathsf{zero}\ (\mathsf{Nat.rec}\ 1\ (\lambda z\ r.\ \mathsf{succ}\ r)\ \mathsf{zero}) && (\iota)\\ \equiv\ &\mathsf{succ}\ (\mathsf{Nat.rec}\ 1\ (\lambda z\ r.\ \mathsf{succ}\ r)\ \mathsf{zero}) && (\beta)\\ \equiv\ &\mathsf{succ}\ 1 && (\iota)\end{aligned}$$

3.9.3 Induktion

Schließlich schauen wir uns noch ein Beispiel an, wie auch Induktion mithilfe des Rekursors verwendet werden kann. Zur Erinnerung hier der induktive Beweis

aus Abschnitt 3.5.9:

```
theorem zero_add (x : Nat) : 0 + x = x :=
  match x with
  | 0 ⇒
    show 0 + 0 = 0 from rfl
  | succ y ⇒
    calc 0 + succ y
    _ = succ (0 + y) := rfl
    _ = succ y := by rw [zero_add y]
```

Auch dieser Beweis wird intern mithilfe des Rekursors implementiert:

```
theorem zero_add (x : Nat) : 0 + x = x :=
  @Nat.rec (λz. 0 + z = z)
    (show 0 + 0 = 0 from rfl)
    (λy : Nat. λh : 0 + y = y.
      calc 0 + succ y
      _ = succ (0 + y) := rfl
      _ = succ y := by rw [h])
    x
```

Hier ist nun der Rückgabetyp $0+x = x$ vom Argument x abhängig. Daher nutzen wir diesmal das Argument im Motiv: $\lambda z.\ 0 + z = z$. Die Fallunterscheidung ist auf x und daher ist unsere Major-Prämisse x. Die erste Minor-Prämisse hat Typ *motive* zero, also mit unserem Motiv $0 + 0 = 0$. Wir können einfach den Fall zero aus dem Pattern-Matching-Beweis übernehmen. Die zweite Minor-Prämisse nimmt zwei Argumente: das Argument y von succ und das Ergebnis h des rekursiven Aufrufs des gesamten Rekursors auf y. Wir können den Fall succ aus dem Pattern-Matching-Beweis verwenden, indem wir den rekursiven Aufruf zero_add y durch h ersetzen.

3.9.4 Parameter

Zur Erinnerung nocheinmal die Definition des Typs List, zur Vereinfachung nur für Type statt Type_u:

```
inductive List (X : Type) : Type :=
  | nil : List X
  | cons : X → List X → List X
```

Der zugehörige Rekursor ist:

$$\begin{aligned}&\mathsf{List.rec}.\{u\} : \Pi\{X : \mathsf{Type}\}.\ \Pi\{\mathit{motive} : \mathsf{List}\ X \to \mathsf{Sort}_u\} \to \\ &\quad \mathit{motive}\ \mathsf{nil} \to \\ &\quad (\Pi(x : X)\ (\mathit{xs} : \mathsf{List}\ X).\ \mathit{motive}\ \mathit{xs} \to \mathit{motive}\ (\mathsf{cons}\ x\ \mathit{xs})) \to \\ &\quad \Pi t : \mathsf{List}\ X.\ \mathit{motive}\ t\end{aligned}$$

Der Parameter $X : \mathsf{Type}$ wird zu einem zusätzlichen Argument des Rekursors. Das Motiv, die Minor-Prämisse für nil und die Major-Prämisse sind ähnlich wie in den vorherigen Beispielen. Die Minor-Prämisse für cons nimmt die Argumente x und xs, die die Argumente von cons repräsentieren. Außerdem nimmt sie ein Argument vom Typ $\mathit{motive}\ \mathit{xs}$, das für Rekursion verwendet werden kann. Für x gibt es kein solches Argument, da x nicht vom Typ List ist.

Als Beispiel definieren wir die Funktion length, die die Länge einer Liste bestimmt, zunächst mit Pattern-Matching und dann mithilfe des Rekursors:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{length}\ \{X : \mathsf{Type}\} : \mathsf{List}\ X \to \mathsf{Nat} \\ &\mid \mathsf{nil} \Rightarrow 0 \\ &\mid \mathsf{cons}\ x\ \mathit{xs} \Rightarrow \mathsf{succ}\ (\mathsf{length}\ \mathit{xs})\end{aligned}$$

Nun mithilfe des Rekursors:

$$\begin{aligned}&\textbf{def}\ \ \mathsf{length} : \{X : \mathsf{Type}\}\ (l : \mathsf{List}\ X) : \mathsf{Nat} := \\ &\quad @\mathsf{List.rec}\ X\ (\lambda\_.\ \mathsf{Nat}) \\ &\qquad 0 \\ &\qquad (\lambda(x : X)\ (\mathit{xs} : \mathsf{List}\ X)\ (r : \mathsf{Nat}).\ \mathsf{succ}\ r) \\ &\qquad l\end{aligned}$$

Das dritte Argument r der Minor-Prämisse von cons entspricht hier dem rekursiven Aufruf length xs.

3.9.5 Indizes

Als nächstes betrachten wir einen induktiven Typ mit einem Index. Der folgende Typ ist ein induktives Prädikat, das die geraden natürlichen Zahlen 0, 2, 4, … identifiziert.

$$\begin{aligned}&\textbf{inductive}\ \mathsf{Even} : \mathsf{Nat} \to \mathsf{Prop} := \\ &\mid \mathsf{base} : \mathsf{Even}\ 0 \\ &\mid \mathsf{step} : \forall(n : \mathsf{Nat}).\ \mathsf{Even}\ n \to \mathsf{Even}\ (\mathsf{succ}\ (\mathsf{succ}\ n))\end{aligned}$$

Das Argument vom Typ Nat ist hier ein Index des induktiven Typs. Auf ähnliche Weise können wir ein Prädikat für die ungeraden Zahlen definieren:

$$\begin{aligned}&\textbf{inductive}\ \mathsf{Odd} : \mathsf{Nat} \to \mathsf{Prop} := \\ &\mid \mathsf{base} : \mathsf{Odd}\ 1 \\ &\mid \mathsf{step} : \forall(n : \mathsf{Nat}).\ \mathsf{Odd}\ n \to \mathsf{Odd}\ (\mathsf{succ}\ (\mathsf{succ}\ n))\end{aligned}$$

Es folgt ein Beispiel von Pattern-Matching auf Even:

```
theorem odd_succ_of_even (m : Nat) : Even m → Odd (succ m)
| Even.base ⇒
  show Odd (succ 0) from Odd.base
| Even.step (n : Nat) (h : Even n) ⇒
  show Odd (succ (succ (succ n))) from
  Odd.step (succ n) (odd_succ_of_even n h)
```

Der Rekursor für Even sieht wie folgt aus:

```
Even.rec : Π{motive : Πn : Nat. Even n → Prop}.
  motive 0 Even.base →
  (Π(n : Nat) (h : Even n). motive n h → motive (succ (succ n)) (Even.step n h)) →
  Π{m : Nat} (t : Even m). motive m t
```

Wie Parameter wird auch der Index zu einem zusätzlichen Argument $m : \mathsf{Nat}$ des Rekursors, das allerdings nur in der Major-Prämisse und im Rückgabetyp des Rekursors verwendet wird. Das Motiv erhält ein eigenes Argument $n : \mathsf{Nat}$, das den Index repräsentiert, was unterschiedliche Werte des Index in den Minorprämissen ermöglicht, wie in den Typen der Konstruktoren vorgegeben. Außerdem fällt auf, dass der Rückgabetyp des Motivs Prop ist und nicht Sort_u. Das liegt an dem Verbot von Large Elimination in Prop.

Mit dem Rekursor kann das obige Pattern-Matching wie folgt umgesetzt werden:

```
theorem odd_succ_of_even (m : Nat) (hm : Even m) : Odd (succ m) :=
  @Even.rec (λ(n : Nat) (h : Even n). Odd (succ n))
    (show Odd (succ 0) from Odd.base)
    (λ(n : Nat) (h : Even n) (r : Odd (succ n)).
      show Odd (succ (succ (succ n))) from
      Odd.step (succ n) r)
    m hm
```

3.9.6 Gleichheit

Zur Erinnerung nocheinmal die Definition des Typs Eq und $\mathsf{Eq.subst}$. Zur Vereinfachung verwenden wir nur Type statt Type_u oder gar Sort_u.

$$\begin{aligned}
&\textbf{inductive}\ \ \mathsf{Eq}\ \{X : \mathsf{Type}\}\ (x : X) : X \to \mathsf{Prop} :=\\
&\quad \mid \mathsf{rfl} : \mathsf{Eq}\ x\ x\\
&\\
&\textbf{def}\ \ \mathsf{Eq.subst}\ \{X : \mathsf{Type}\}\ \{x\ y : X\}\ \{p : X \to \mathsf{Prop}\}\\
&\qquad (heq : \mathsf{Eq}\ x\ y)\ (hp : p\ x) : p\ y :=\\
&\quad \textbf{match}\ heq\ \textbf{with}\\
&\quad \mid \mathsf{rfl} \Rightarrow hp
\end{aligned}$$

Dieser Beweis scheint zunächst nicht wohlgetypt zu sein, denn hp hat Typ $p\ x$ und wir müssen zeigen, dass $p\ y$ gilt. Schauen wir uns aber den zugrundeliegenden Rekursor an, wird klar, wie wir mit der Wahl des richtigen Motivs x und y auswechseln können:

$$\begin{aligned}
&\mathsf{Eq.rec}.\{u\} : \Pi\{X : \mathsf{Type}\}.\ \Pi\{x : X\}.\ \Pi\{motive : \Pi z : X.\ \mathsf{Eq}\ x\ z \to \mathsf{Sort}_u\}.\\
&\quad motive\ x\ \mathsf{rfl} \to \Pi\{y : X\}.\ \Pi t : \mathsf{Eq}\ x\ y.\ motive\ y\ t
\end{aligned}$$

Wie bei den vorherigen Rekursoren werden die Parameter $X : \mathsf{Type}$ und $x : X$ zu zusätzlichen Argumenten des Rekursors, die in Motiv, Minor-Prämisse und Major-Prämisse genutzt werden. Der Index (das dritte Argument von Eq) wird zu einem Argument des Rekursors $y : X$, das aber nur in der Major-Prämisse und dem Rückgabetyp verwendet wird, und zu einem zusätzlichen Argument $z : X$ des Motivs. In der Minor-Prämisse füllt der Parameter x den Platz des Index, wie im Konstruktor von Eq vorgegeben. Außerdem fällt auf, dass der Rückgabewert des Motivs Sort_u ist statt nur Prop. Dies ist dem Spezialfall der Singleton-Elimination zu verdanken (Abschnitt 3.8).

Mithilfe dieses Rekursors können wir $\mathsf{Eq.subst}$ nun wie folgt definieren:

$$\begin{aligned}
&\textbf{def}\ \ \mathsf{Eq.subst}\ \{X : \mathsf{Type}\}\ \{x\ y : A\}\ \{p : X \to \mathsf{Prop}\}\\
&\qquad (heq : \mathsf{Eq}\ x\ y)\ (hp : p\ x) : p\ y :=\\
&\quad @\mathsf{Eq.rec}\ X\ x\ (\lambda z : X.\ \lambda_.\ p\ z)\ hp\ y\ heq
\end{aligned}$$

Als Major-Prämisse möchten wir das gegebene $(heq : \mathsf{Eq}\ x\ y)$ nutzen und daher müssen die ersten beiden Argumente X und x sein und die letzten beiden Argumente müssen y und heq sein. Das Motiv können wir nun nutzen, um anzugeben, wo mithilfe der Gleichung umgeschrieben werden soll: $\lambda z : A.\ \lambda\_.\ p\ z$. Als Minor-Prämisse ist nun ein Argument vom Typ $(\lambda z : X.\ \lambda\_.\ p\ z)\ x\ \mathsf{rfl} \equiv p\ x$ gefragt und dadurch wird klar, warum wir $hp : p\ x$ hier verwenden können.

4 Anwendungen

4.1 Informatik

Siehe `CGBF/Vorlesungen/Vorlesung10_Informatik.lean` im vorlesungsbegeleitenden Repository (`https://github.com/hhu-adam/cgbf2023`)

4.2 Linguistik

Sprache kann auf verschiedenen Ebenen analysiert werden:

- Phonologie beschreibt wie Laute bedeutungstragende Wortteile ergeben, z.B. g + u + t = gut.
- Morphologie beschreibt wie bedeutungstragende Wortteile Wörter formen, z.B. un + genau + es = ungenaues.
- Syntax beschreibt wie Wörter Sätze formen.
- Semantik beschreibt die Bedeutung von Wörtern und ihrer Kombination

Wir werden Phonologie und Morphologie hier ignorieren und uns nur auf das Zusammenspiel von Syntax und Semantik konzentieren.

4.2.1 Syntax

Syntax kategorisiert Wörter in Wortarten, je nachdem, wie die Wörter in Sätzen verwendet werden können. Hier sind einige Wortarten:

- Nomen: Diese lassen sich weiter unterteilen in Gattungsnamen, die normalerweise mit einem Artikel (der/die/das) vorkommen und Eigennamen, die normalerweise ohne Artikel stehen.
  - Gattungsnamen (**GN**):

    $$\mathbf{GN} \longrightarrow \text{Schlumpf}$$
    $$\mathbf{GN} \longrightarrow \text{Geschenk}$$

    (Diese Notation bedeutet, dass **GN** für „Schlumpf“ oder für „Geschenk“ stehen kann)
  - Eigennamen (**EN**):

    $$\mathbf{EN} \longrightarrow \text{Sofia}$$
    $$\mathbf{EN} \longrightarrow \text{Klaus}$$

- Verben: Diese können weiter unterteilt werden in intransitive Verben, die normalerweise ohne (Akkusativ-)Objekt stehen („Er lacht.“), und transitive Verben, die normalerweise ein Objekt brauchen („Er umarmt ihn.“).

– intransitiv (**IV**):

$$\begin{aligned}\mathbf{IV} &\longrightarrow \text{lacht}\\ \mathbf{IV} &\longrightarrow \text{schläft}\end{aligned}$$

– transitiv (**TV**):

$$\begin{aligned}\mathbf{TV} &\longrightarrow \text{umarmt}\\ \mathbf{TV} &\longrightarrow \text{beleidigt}\\ \mathbf{TV} &\longrightarrow \text{versteckt}\end{aligned}$$

- Determinierer (**Det**): Dazu gehören Artikel, aber auch z.B. quantifizierende Determinierer wie „keine“.

$$\begin{aligned}\mathbf{Det} &\longrightarrow \text{ein(e)}\\ \mathbf{Det} &\longrightarrow \text{der/die/das}\\ \mathbf{Det} &\longrightarrow \text{jede(r, n)}\\ \mathbf{Det} &\longrightarrow \text{kein(e, en)}\end{aligned}$$

Syntax beschreibt, wie Wörter, abhängig von ihrer Wortart zu größeren Phrasen zusammengesetzt werden können:

- Nominalphrase (**NP**): Eine Nominalphrase kann aus einem einzelnen Eigennamen (**EN**) gebildet werden oder durch die Kombination eines Determinierers mit einem Gattungsnamen (**GN**).

$$\begin{aligned}\mathbf{NP} &\longrightarrow \mathbf{EN}\\ \mathbf{NP} &\longrightarrow \mathbf{Det\ GN}\end{aligned}$$

  Beispiele: Sofia, Klaus, der Schlumpf, ein Schlumpf, jeder Schlumpf, kein Geschenk (Wir ignorieren hier, dass Determinierer an das grammatische Geschlecht und Numerus des Nomens angeglichen werden müssen.)

- Verbalphrase (**VP**):

$$\begin{aligned}\mathbf{VP} &\longrightarrow \mathbf{IV}\\ \mathbf{VP} &\longrightarrow \mathbf{TV\ NP}\end{aligned}$$

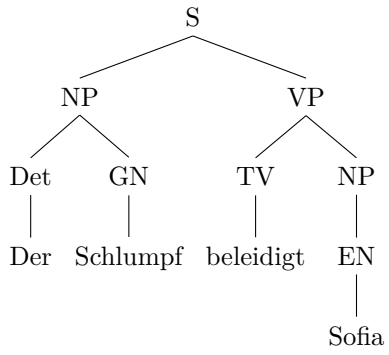
  Beispiele: lacht, schläft, umarmt den Schlupf, versteckt ein Geschenk, beleidigt jeden Schlumpf

- Sätze (**S**):

$$\mathbf{S} \longrightarrow \mathbf{NP\ VP}$$

  Beispiele: Sofia umarmt den Schlupf. Der Schlumpf versteckt ein Geschenk. Jeder Schlumpf lacht.

Häufig wird die Herleitung einer Phrase oder eines Satzes auch als Syntaxbaum dargestellt:



4.2.2 Semantik

Formale Ansätze der Semantik versuchen, syntaktischen Strukturen eine logische Bedeutung zuzuordnen. Dabei ist eine Grundannahme das Kompositionalitätsprinzip. Es besagt, dass sich die Bedeutung eines komplexen Ausdrucks aus den Bedeutungen seiner Teile ergibt. Um zu beschreiben, wie sich die Bedeutung eines komplexen Ausdrucks ergibt, müssen wir zunächst den kleinsten Bausteinen der Syntax, den Wörtern, eine Bedeutung zuordnen. Wir können zum Beispiel einen Typ $\mathsf{Entität} : \mathsf{Type}$ von Entitäten deklarieren und annehmen, dass $\mathsf{sofia} : \mathsf{Entität}$ und $\mathsf{klaus} : \mathsf{Entität}$ Konstanten diesen Typs sind. Wir erweitern nun die syntaktischen Regeln von oben, indem wir jeweils auch eine semantische Regel angeben. Hierbei bezeichnet $[\![X]\!]$ die Bedeutung von X.

$$\begin{aligned} \mathbf{EN} &\longrightarrow \text{Sofia} & [\![\mathbf{EN}]\!] &:= (\mathsf{sofia} : \mathsf{Entität}) \\ \mathbf{EN} &\longrightarrow \text{Klaus} & [\![\mathbf{EN}]\!] &:= (\mathsf{klaus} : \mathsf{Entität}) \end{aligned}$$

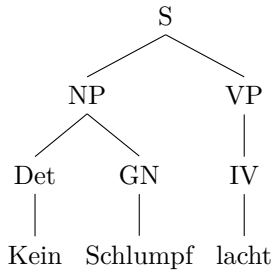
Die Bedeutung von intransitiven Verben wie „lacht“ können wir dann als Funktion $\mathsf{Entität} \to \mathsf{Prop}$ darstellen:

$$\mathbf{IV} \longrightarrow \text{lacht} \qquad [\![\mathbf{IV}]\!] := (\mathsf{lacht} : \mathsf{Entität} \to \mathsf{Prop})$$

Jede Entität lacht oder lacht nicht. Wir ignorieren, dass in der Realität Entitäten auch zu einem Zeitpunkt lachen können und zu einem anderen nicht. Die Aussage $\mathsf{lacht}\ x$ soll nun bedeuten dass die Entität x lacht.

Die Bedeutung des Satzes „Sofia lacht“ können wir nun beschreiben als $\mathsf{lacht\ sofia} : \mathsf{Prop}$. Daher scheint es auf den ersten Blick sinnvoll, einer **NP** ein Element vom Typ $\mathsf{Entität}$ zuzuordnen, einer **VP** ein Element vom Typ $\mathsf{Entität} \to \mathsf{Prop}$ zuzuordnen, und die Bedeutung eines Satzes als $[\![\mathbf{S}]\!] := [\![\mathbf{VP}]\!]\ [\![\mathbf{NP}]\!]$ zu de-

finieren. Allerdings führt dies in Sätzen wie dem folgenden zu Problemen:



Wir können der **NP** „Kein Schlumpf“ keine sinnvolle Bedeutung vom Typ $\mathsf{Entität}$ zuordnen, denn das Wort „Kein“ bedeutet ja gerade, dass es keine Entität gibt, die hier handelt. Eine sinnvolle Beschreibung der Bedeutung dieses Satzes wäre:

$$\neg\, \exists x : \mathsf{Entität}.\ \mathsf{schlumpf}\ x \wedge \mathsf{lacht}\ x$$

wobei $\mathsf{schlumpf} : \mathsf{Entität} \to \mathsf{Prop}$ ein Prädikat ist, das für alle Entitäten wahr ist, die Schlümpfe sind. Diese Beschreibung der Bedeutung ist aber ein Problem für das Kompositionalitätsprinzip, denn kein Teilausdruck steht für die Bedeutung von „Kein Schlumpf“.

Wir können dieses Problem lösen, indem wir die Bedeutung eines Satzes wie folgt definieren:

$$\mathbf{S} \longrightarrow \mathbf{NP}\ \mathbf{VP} \qquad\qquad [\![\mathbf{S}]\!] := [\![\mathbf{NP}]\!]\ [\![\mathbf{VP}]\!]$$

wobei $[\![\mathbf{NP}]\!] : (\mathsf{Entität} \to \mathsf{Prop}) \to \mathsf{Prop}$ und $[\![\mathbf{VP}]\!] : \mathsf{Entität} \to \mathsf{Prop}$. Dann können wir nämlich die Bedeutung der **NP** „Kein Schlumpf“ beschreiben als

$$\lambda p : \mathsf{Entität} \to \mathsf{Prop}.\ \neg\, \exists x : \mathsf{Entität}.\ \mathsf{schlumpf}\ x \wedge p\ x$$

Um dies zu erreichen legen wir folgende Regeln fest:

$\mathbf{Det} \longrightarrow \text{kein}$	$[\![\mathbf{Det}]\!] := \lambda(p\ q : \mathsf{Entität} \to \mathsf{Prop}).\ \neg\, \exists x.\ p\ x \wedge q\ x$
$\mathbf{GN} \longrightarrow \text{Schlumpf}$	$[\![\mathbf{GN}]\!] := (\mathsf{schlumpf} : \mathsf{Entität} \to \mathsf{Prop})$
$\mathbf{NP} \longrightarrow \mathbf{Det}\ \mathbf{GN}$	$[\![\mathbf{NP}]\!] := [\![\mathbf{Det}]\!]\ [\![\mathbf{GN}]\!]$

Schließlich fehlt uns noch die folgende Regel, um den Satz „Kein Schlumpf lacht.“ analysieren zu können:

$$\mathbf{VP} \longrightarrow \mathbf{IV} \qquad\qquad [\![\mathbf{VP}]\!] := [\![\mathbf{IV}]\!]$$

Somit ergibt sich:

$$\begin{aligned}
[\![\text{kein Schlumpf lacht}]\!] &= [\![\text{kein Schlumpf}]\!]\ [\![\text{lacht}]\!] \\
&= [\![\text{kein}]\!]\ [\![\text{Schlumpf}]\!]\ [\![\text{lacht}]\!] \\
&= (\lambda p\ q.\ \neg\, \exists x.\ p\ x \wedge q\ x)\ \mathsf{schlumpf}\ \mathsf{lacht} \\
&\equiv \neg\, \exists x.\ \mathsf{schlumpf}\ x \wedge \mathsf{lacht}\ x
\end{aligned}$$

4.2.3 Abhängige Typentheorie

Betrachten wir nun den folgenden Satz:

> If a boy loses, he cries.

Wir wechseln hier auf Englisch, da deutsche Satzstellung kompliziert ist. Um diesen Satz zu analysieren, brauchen wir eine Regel für „If“. Es scheint sinnvoll, „If“ als Implikation zu übersetzen:

$$\mathbf{S} \longrightarrow \text{If } \mathbf{S}_1\ \mathbf{S}_2 \qquad\qquad [\![\mathbf{S}]\!] := [\![\mathbf{S}_1]\!] \to [\![\mathbf{S}_2]\!]$$

Somit ergibt sich:

$$\begin{aligned}[\![\text{If a boy loses, he cries.}]\!] &= [\![\text{a boy loses}]\!] \to [\![\text{he cries}]\!] \\ &\equiv (\exists x.\ \mathsf{boy}\ x \wedge \mathsf{loses}\ x) \to [\![\text{he}]\!]\ \mathsf{cries}\end{aligned}$$

Die Bedeutung von „he“ sollte eigentlich das x sein, das durch $\exists x$ eingeführt wird, aber $[\![\text{he}]\!]$ ist nicht im Geltungsbereich von $\exists x$. Eine sinnvolle Bedeutung des Satzes wäre

$$\forall x.\ (\mathsf{boy}\ x \wedge \mathsf{loses}\ x) \to \mathsf{cries}\ x$$

aber diese Bedeutung scheint nicht unter Einhaltung des Kompositionalitätsprinzips erreichbar zu sein.

Abhängige Typentheorie kann dieses Problem lösen. Wegen Impredikativität von Prop funktioniert diese Lösung aber nicht in Leans Prop. Wir müssten entweder statt Prop mit Type arbeiten oder eine Typentheorie mit prädikativem Prop verwenden. Für diese Erklärung nehmen wir einfach an, es gäbe eine Funktion Exists.val, die uns für einen Beweis von $\exists x.\ \mathsf{p}\ x$ dieses Element x liefert. Der Trick ist nun, die semantische Regel für „If“ explizit als abhängigen Funktionstyp zu schreiben:

$$\mathbf{S} \longrightarrow \text{If } \mathbf{S}_1\ \mathbf{S}_2 \qquad\qquad [\![\mathbf{S}]\!] := \Pi h : [\![\mathbf{S}_1]\!].\ [\![\mathbf{S}_2]\!]$$

Nun erhalten wir

$$\begin{aligned}[\![\text{If a boy loses, he cries.}]\!] &= \Pi h : [\![\text{a boy loses}]\!].\ [\![\text{he cries}]\!] \\ &\equiv \Pi h : (\exists x.\ \mathsf{boy}\ x \wedge \mathsf{loses}\ x).\ [\![\text{he}]\!]\ \mathsf{cries}\end{aligned}$$

Nun können wir $[\![\text{he}]\!] = \lambda v.\ v\ (\mathsf{Exists.val}\ h)$ wählen, denn $[\![\text{he}]\!]$ liegt im Geltungsbereich von h. Nach β-Reduktion erhalten wir:

$$\Pi h : (\exists x.\ \mathsf{boy}\ x \wedge \mathsf{loses}\ x).\ \mathsf{cries}\ (\mathsf{Exists.val}\ h)$$

Diese Semantik ist äquivalent zu $\forall x.\ (\mathsf{boy}\ x \wedge \mathsf{loses}\ x) \to \mathsf{cries}\ x$.

4.3 Mathematik

Siehe `CGBF/Vorlesungen/Vorlesung12_Mathematik.lean` im vorlesungsbegeleitenden Repository (`https://github.com/hhu-adam/cgbf2023`)

Literatur

[1] J. Avigad, L. de Moura, S. Kong, and S. Ullrich. Theorem proving in Lean 4. 2023. https://leanprover.github.io/theorem_proving_in_lean4.

[2] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, and J. Limperg. The hitchhiker's guide to logical verification, 2020.

[3] K. Buzzard, J. Commelin, and P. Massot. Formalising perfectoid spaces. In J. Blanchette and C. Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020.

[4] D. Castelvecchi et al. Mathematicians welcome computer-assisted proof in 'grand unification' theory. *Nature*, 595(7865):18–19, 2021.

[5] S. Chatzikyriakidis and R. Cooper. Type theory for natural language semantics. In *Oxford Research Encyclopedia of Linguistics*. 2018.

[6] G. Gonthier. Formal proof—the Four-Color Theorem. *Notices AMS*, 55(11):1382–1393, 2008. https://www.ams.org/notices/200811/tx081101382p.pdf.

[7] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the Odd Order Theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. https://hal.inria.fr/hal-00816699/document.

[8] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In K. Keeton and T. Roscoe, editors, *OSDI 2016*, pages 653–669. USENIX Association, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf.

[9] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. http://arxiv.org/abs/1501.02155.

[10] J. M. Han and F. van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 353–366, 2020.

[11] J. Harrison. Formal verification at Intel. In *LICS 2003*, pages 45–54. IEEE Computer Society, 2003. https://ieeexplore.ieee.org/document/1210044.

[12] K. Hartnett. Proof assistant makes jump to big-league math. *Online at https://www. quantamagazine. org/lean-computer-program-confirms-peter-scholze-proof-20210728*, 2021.

[13] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. `https://www.researchgate.net/profile/Gernot_Heiser/publication/220910193_SeL4_Formal_verification_of_an_OS_kernel/links/09e4150f00292a0329000000/SeL4-Formal-verification-of-an-OS-kernel.pdf`.

[14] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *POPL 2014*, pages 179–192. ACM, 2014. `https://cakeml.org/popl14.pdf`.

[15] S. Lappin and C. Fox. *The handbook of contemporary semantic theory*. John Wiley & Sons, 2015.

[16] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009. `https://arxiv.org/pdf/0902.2137.pdf`.

[17] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010. `https://link.springer.com/content/pdf/10.1007/978-3-642-11957-6_23.pdf`.

[18] T. Nipkow and G. Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014.

[19] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999. `https://link.springer.com/content/pdf/10.1023/A:1008669628911.pdf`.

[20] P. Selinger. Lecture notes on the lambda calculus. *arXiv preprint arXiv:0804.3434*, 2008.

[21] F. van Doorn, P. Massot, and O. Nash. Formalising the h-principle and sphere eversion. In R. Krebbers, D. Traytel, B. Pientka, and S. Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 121–134. ACM, 2023.

[22] J. Van Eijck and C. Unger. *Computational semantics with functional programming*. Cambridge University Press, 2010.