Kapitel 10

Typtheorie

Typtheorie ist wie die Mengenlehre eine grundlegende mathematische Theorie. Während es bei der Mengenlehre vor allem darum geht, eine universelle Klasse von mathematischen Objekten bereitzustellen, besteht die Aufgabe der Typtheorie darin, einen flexiblen Rahmen für logische Sprachen zu liefern. Das primäre Ausdrucksmittel typtheoretischer Sprachen sind totale Funktionen $X \to Y$. Der zu einer totalen Funktion gehörige Typ $X \to Y$ beschreibt die zulässigen Argumente und die möglichen Ergebnisse der Funktion. Die Syntax typtheoretischer Sprachen ist so eingeschränkt, dass Funktionen nur auf zulässige Argumente angewendet werden können. Die Ausdrucksstärke typtheoretischer Sprachen beruht ganz wesentlich auf der Verwendung höherstufigen Funktionen (Funktionen, deren Argumente oder/und Ergebnisse Funktionen sind).

Wichtige Beiträge zur Typtheorie stammen von Bertrand Russel [1908] (höherstufige getypte Sprache), Moses Schönfinkel [1924] (kaskadierte höherstufige Funktionen), Alonzo Church [1940] (Lambda-Abstraktion) und Leon Henkin [1950] (Vollständigkeitsergebnis und Reduktion auf Gleichheit). Der eher syntaktische Teil der Typtheorie ist auch unter dem Namen *Lambda-Kalkül* bekannt. Church entwickelte zunächst den *ungetypten Lambda-Kalkül* [1934]. Als sich herausstellte, dass sich dieser wegen der fehlenden Typen nicht als Grundlage für die Beschreibung mathematischer Aussagen eignet (Paradoxe), erweiterte Church den Lambda-Kalkül zum *einfach getypten Lambda-Kalkül*[1] [1940]. Dabei konnte er auf die älteren Arbeiten Russells [1908] zurückgreifen, in dessen logischen Sprachen höherstufige Typen bereits eine wichtige Rolle spielten.

Das klassische Anwendungsfeld der Typtheorie sind Sprachen für die Formulierung mathematischer Aussagen und Theorien. Einfache Vertreter dieser Sprachen sind als *prädikatenlogische Sprachen erster Stufe* bekannt. Ein zweites Anwendungsfeld ist die Theorie von Programmiersprachen. Seit um 1965 klar wurde, dass getypte Lambda-Kalküle eine wichtige Grundlage für den Entwurf und die Analyse von Programmiersprachen bilden [Landin 1963, 1965, 1966; Stra-

[1] Simply-typed Lambda Calculus.

chey 1966], hat sich Typtheorie zu einem der aktivsten Forschungsgebiete der theoretischen Informatik entwickelt.

Im Folgenden betrachten wir ein einfaches typtheoretisches System, dass wir *ETT* nennen (für Elementare Typtheorie). ETT ist eine didaktisch motivierte Variante des einfach getypten Lambda-Kalküls.

ETT ist ein parametrisiertes System. Zunächst können einige Mengen als Typen vorgegeben werden. Darüber hinaus können ausgewählte Elemente der Typen durch Konstanten verfügbar gemacht werden. Mithilfe von Funktionsanwendung und Lambda-Abstraktion[2] können Konstanten und Variablen zu Ausdrücken kombiniert werden.

Durch die Vorgabe geeigneter Typen und Konstanten kann ETT zu einer Vielzahl von logischen Sprachen instantiiert werden. Wenn man eine logische Sprache mit ETT definiert, spart man viel Arbeit, da nur noch die wesentlichen Dinge definiert werden müssen, und gleichzeitig viele wichtige Eigenschaften aus den allgemeinen Eigenschaften von ETT folgen. Alle logischen Sprachen, die wir in dieser Vorlesung betrachten werden, lassen sich als Instanzen von ETT einführen. Prägnant können wir ETT als eine *universelle logische Sprache* oder als eine *Metalogik* bezeichnen.

Die Programmiersprache Standard ML ist zu einem beträchtlichen Maß ein Produkt der programmiersprachlichen Typtheorie. Sie realisiert Typstrukturen, die sehr viel mächtiger sind als die, die wir im Rahmen von ETT kennenlernen werden.

In diesem Kapitel kommt es uns vor allem darauf an, einen mathematischen Rahmen für die Syntax und die Semantik einer großen Klasse von logischen Sprachen bereitzustellen. Im Mittelpunkt stehen dabei klassische höherstufige prädikatenlogische Sprachen. Die grundlegenden typtheoretischen Ideen führen wir zunächst auf semantischer Grundlage ein. Die Angabe einer formalen Syntax erfolgt in einem zweiten Schritt. Auf die Angabe deduktiver Systeme verzichten wir.

Literaturverweise

- Peter B. Andrews. *Classical Type Theory*. In: Alan Robinson und Andrei Voronkov, Handbook of Automated Reasoning; Elsevier, 2001.
- Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2002. (Erste Auflage 1986 bei Academic Press.)
- M. J. C. Gordon and T. F. Melham. *Introduction to HOL; A theorem proving environment for higher order logic. Cambridge University Press, 1993.*
- J. Roger Hindley, *Basic Simple Type Theory.* Cambridge University Press, 1997.

[2] Lambda-Abstraktion kennen wir bereits unter dem Namen Lambda-Notation.

- John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.
- Frank Pfenning. *Logical Frameworks*. In: Alan Robinson und Andrei Voronkov, Handbook of Automated Reasoning; Elsevier, 2001.
- Benjamin Pierce, *Types and Programming Languages*, The MIT Press, 2002.
- Allen Stoughton, Substitution Revisited. Theoretical Computer Science 59:317-325, 1988.

10.1 Funktionale Darstellung mathematischer Aussagen

Wir wollen jetzt genauer verstehen, nach welchen Regeln mathematische Aussagen gebildet werden. Zunächst stellen wir fest, dass die mathematische Sprache eine *natürliche Sprache* ist, die wie die übliche Sprache schrittweise und anwendungsgetrieben entstanden ist. Dagegen handelt es sich bei Programmiersprachen und logischen Sprachen um *artifizielle Sprachen*, die das Ergebnis eines expliziten Entwurfs darstellen.

Bei der Analyse einer natürliche Sprache geht es darum, nachträglich einen „Bauplan“ zu erfinden, der die Funktionsweise der Sprache auf einleuchtende Weise erklärt. Selbstverständlich kann es für eine Sprache mehrere Möglichkeiten geben, sie zu analysieren und zu erklären. Die hier dargestellte Analyse mathematischer Aussagen wurde von Alonzo Church entwickelt (Publikation 1940). Wir bezeichen sie als *funktionale Analyse*.

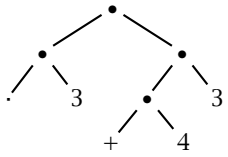
Arithmetische Ausdrücke
Wir beginnen mit der Analyse des arithmetischen Ausdrucks

$$3 \cdot (4 + 3)$$

Dieser Ausdruck ist aus vier *Konstanten* gebildet:

$3 \in \mathbb{N}$	die Zahl 3
$4 \in \mathbb{N}$	die Zahl 4
$\cdot \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$	die Funktion, die ihre Argumente multipliziert
$+ \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$	die Funktion, die ihre Argumente addiert

Jede Konstante ist zusammen mit ihrem *Typ* angegeben. Bei zwei der Konstanten handelt es sich um Funktionen. Der Ausdruck wird dadurch gebildet, dass die Konstanten durch *Funktionsapplikation* kombiniert werden. Die Struktur des Ausdrucks lässt sich graphisch durch einen Baum darstellen:



Die mit • markierten Knoten des Baums stellen Funktionsapplikationen dar. Die Konstanten erscheinen an den Blättern des Baums. Wenn man die Funktionsapplikationen von den Blättern ausgehend schrittweise *auswertet*, bekommt man den *Wert des Ausdrucks* (die Zahl 21).

Die funktionale Analyse des Ausdrucks unterscheidet sich in zwei Punkten von der üblichen Analyse:

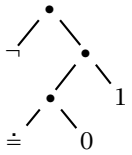
- Funktionsapplikation ist eine explizite binäre Operation.
- Addition und Multiplikation werden als kaskadierte Funktionen modelliert.

Diese Sichtweise geht auf Schönfinkel [1924] zurück.

Einfache Aussagen
Einfache Aussagen sind nach denselben Regeln wie arithmetische Ausdrücke gebildet. Wir zeigen das am Beispiel der Aussage

$$0 \neq 1$$

Diese Aussage ist mittels Funktionsapplikation



aus den Konstanten

$$0 \in \mathbb{N}, \qquad 1 \in \mathbb{N}, \qquad \doteq \in \mathbb{N} \to \mathbb{N} \to \mathbb{B}, \qquad \neg \in \mathbb{B} \to \mathbb{B}$$

gebildet. Der Wert der Aussage ist die Zahl 1.

Aussagen sind Ausdrücke, für die nur die Werte 0 und 1 möglich sind. Wenn eine Aussage den Wert 1 hat, bezeichnet man sie als *gültig*.

Aussagen mit Quantoren
Auch Aussagen mit Quantoren können als Ausdrücke dargestellt werden. Quantifizierung wird dazu auf Lambda-Abstraktion zurückgeführt. Als Beispiel betrachten wir die Aussage

$$\forall x \in \mathbb{N}\colon\ 1 \cdot x = x$$

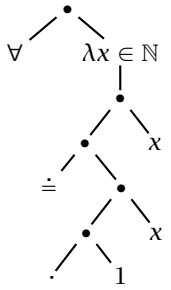
Den Allquantor stellen wir mit der Funktion

$$\forall \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$$
$$\forall f = \min\{f x \mid x \in \mathbb{N}\}$$

dar. Offensichtlich gilt $\forall f = 1$ genau dann, wenn die Funktion f für alle Argumente den Wert 1 liefert. Die Aussage können wir jetzt durch den Ausdruck

$$\forall\ (\lambda x \in \mathbb{N}.\ 1 \cdot x \doteq x)$$

darstellen. Die *Bindung* der quantifizierten Variablen x wir dabei mithilfe einer Lambda-Abstraktion realisiert. Die Baumdarstellung des Ausdrucks ist



Der Wert des Ausdrucks ist 1. Das bedeutet, dass die Aussage gültig ist.

Der Existenzquantor kann analog zum Allquantor modelliert werden:

$$\exists \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$$
$$\exists f = \max\{f x \mid x \in \mathbb{N}\}$$

Offensichtlich gilt $\exists f = 0$ genau dann, wenn die Funktion f für alle Argumente den Wert 0 liefert.

Die Idee, quantifizierte Aussagen als Applikation einer Quantorfunktion auf eine durch eine Lambda-Abstraktion beschriebene Funktion zu analysieren stammt von Church [1940].

Parametrisierte Aussagen

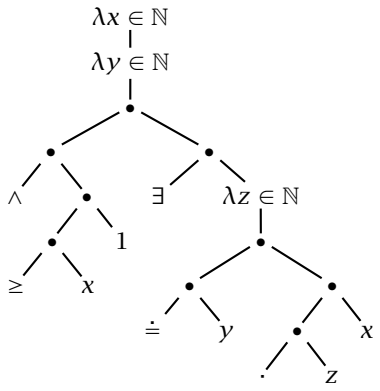
Der Ausdruck

$$\lambda x \in \mathbb{N}.\ \lambda y \in \mathbb{N}.\ (x \text{ ist Teiler von } y)$$

stellt eine parametrisierte Aussage des Typs $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$ dar. Der Ausdruck

$$\lambda x \in \mathbb{N}.\ \lambda y \in \mathbb{N}.\ x \geq 1\ \wedge\ \exists\ (\lambda z \in \mathbb{N}.\ y \doteq z \cdot x)$$

beschreibt eine äquivalente parametrisierte Aussage. Er hat die folgende Baumdarstellung:



Mengen versus Funktionen
Mit Funktionen kann man Mengen beschreiben. Gegeben eine Menge M, können wir jede Teilmenge $Y \subseteq X$ eindeutig durch ihre **charakteristische Funktion**

$$(\lambda\, x \in X \,.\, \text{if } x \in Y \text{ then } 1 \text{ else } 0) \;\in\; X \to \mathbb{B}$$

beschreiben. Umgekehrt können wir jede Funktion $f \in X \to \mathbb{B}$ eindeutig durch die Menge $\{\, x \in X \mid fx = 1 \,\}$ beschreiben. Es gilt also

$$\mathcal{P}(X) \cong X \to \mathbb{B}$$

Sei $f \in X \to \mathbb{B}$ die charakteristische Funktion für eine Menge $Y \in \mathcal{P}(X)$. Dann:

$$\forall x \in X\colon\; x \in Y \iff fx = 1$$

Das bedeutet, dass die Applikation der charakterischen Funktion auf x den Wert der Aussage $x \in Y$ liefert.

Die Menge aller $x \in X$, für die die Aussage $A(x)$ gilt, wird typischerweise durch

$$\{\, x \in X \mid A(x) \,\}$$

beschrieben. Wenn wir annehmen, dass Aussagen Ausdrücke sind, die einen Wert in $\mathbb{B}$ liefern, können wir die charakteristische Funktion für diese Menge durch

$$\lambda\, x \in X \,.\, A(x)$$

beschreiben. Wir können also Mengenbeschreibungen der Bauart $\{\, x \in X \mid A(x) \,\}$ als eine Variante der Lambda-Notation ansehen. Die Lambda-Notation ist flexibler als Mengenbeschreibungen der Bauart $\{\, x \in X \mid A(x) \,\}$, da man mit ihr nicht nur Funktionen $X \to \mathbb{B}$ beschreiben kann, sondern auch Funktionen $X \to Y$ mit einem beliebigen Zieltyp Y.

Höherstufige Quantifizierung
Wir wagen uns jetzt an eine komplexe Aussage, die die Korrektheit der Beweistechnik *Natürliche Induktion* formuliert:[3]

> *Sei* $P \subseteq \mathbb{N}$ *und* $\forall x \in \mathbb{N}\colon\ \{y \in \mathbb{N} \mid y < x\} \subseteq P \implies x \in P$.
> *Dann* $P = \mathbb{N}$.

Zunächst stellen wir fest, dass wir Teilmengen $P \subseteq \mathbb{N}$ durch ihre charakteristischen Funktionen $\mathbb{N} \to \mathbb{B}$ darstellen können. Mit dieser Erkenntnis können wir die Aussage wie folgt formulieren:

$$\forall P \in \mathbb{N} \to \mathbb{B}\colon\ [\forall x \in \mathbb{N}\colon (\forall y \in \mathbb{N}\colon y < x \implies Py) \implies Px] \implies \forall x \in \mathbb{N}\colon Px$$

Der erste Allquantor $\forall P \in \mathbb{N} \to \mathbb{B} \cdots$ unterscheidet sich von den anderen Quantoren dadurch, dass er eine Variable mit einem *höherstufigen Typ* quantifiziert. Für seine Darstellung benötigen wir daher eine andere Funktion als für den *nullstufigen* Quantor $\forall x \in \mathbb{N} \cdots$. Das ist kein Problem. Sei X eine nichtleere Menge. Wenn wir über eine Variable $x \in X$ quantifizieren wollen, können wir Lambda-Abstraktion und eine der folgenden Funktionen benutzen:

$$\forall_X \in (X \to \mathbb{B}) \to \mathbb{B} \qquad \exists_X \in (X \to \mathbb{B}) \to \mathbb{B}$$
$$\forall_X(f) = \min\{fx \mid x \in X\} \qquad \exists_X(f) = \max\{fx \mid x \in X\}$$

Leibnizsche Charakterisierung der Gleichheit
Sei X eine Menge. Das **Gleichheitsprädikat** für X ist wie folgt definiert:

$$\doteq_X \in X \to X \to \mathbb{B}$$
$$(x \doteq_X y) = \text{if } x = y \text{ then } 1 \text{ else } 0$$

Dann gilt für alle $x, y \in X$:

$$(x \doteq_X y) \ =\ \forall f \in X \to \mathbb{B}\colon\ fx \implies fy$$

Für $x = y$ liefern beide Seiten der Gleichung 0. Für $x \neq y$ liefert die linke Seite 0. Die rechte Seite liefert 0, falls es ein $f \in X \to \mathbb{B}$ gibt mit $fx \wedge \neg(fy) = 1$. Das ist für die charakteristische Funktion der Menge $\{x\}$ der Fall.

Von Leibniz stammt die Feststellung, dass zwei Objekte genau dann gleich sind, wenn für sie dieselben Eigenschaften gelten. Wenn man Eigenschaften für die Elemente einer Menge X als Funktionen $X \to \mathbb{B}$ auffasst, entspricht die obige Gleichhung gerade der Leibnizsche Charakterisierung der Gleichheit.

[3] Die Aussage wurde erstmals 1889 von Peano als sogenanntes „Induktionsaxiom“ formuliert.

Reduktion auf Gleichheit
Die Booleschen Primitive und die Quantoren lassen sich auf das Gleichheitsprädikat zurückführen. Betrachten Sie dazu die folgenden Gleichungen, die für alle $x, y \in \mathbb{B}$ und $f \in X \to \mathbb{B}$ gelten ($X \neq \emptyset$):

$$\begin{aligned} 1 &= (\doteq) \doteq (\doteq) \\ 0 &= (\lambda x.\, x) \doteq (\lambda x.\, 1) \\ \neg x &= x \doteq 0 \\ x \wedge y &= (\lambda g \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}.\, gxy) \doteq (\lambda g.\, g11) \\ \forall f &= f \doteq (\lambda x.\, 1) \end{aligned}$$

Wir haben auf die Angabe der Typindizes für die Gleichheitsprädikate verzichtet, da diese durch die restlichen Konstituenten der Gleichungen hinreichend bestimmt sind. Woimmer möglich, haben wir auch auf die Typangaben für die Variablen verzichtet.

Am überraschendsten ist wahrscheinlich die Gleichung für $x \wedge y$. Um diese Gleichung zu zeigen, beginnen wir mit der Gleichung

$$(\lambda g.\, gxy) \doteq (\lambda g.\, g11) \;=\; \forall g.gxy \doteq g11$$

Diese Gleichung gilt, da zwei Funktionen genau dann gleich sind, wenn sie für alle Argumente dieselben Ergebnisse liefern. Es bleibt jetzt zu zeigen, dass

$$\forall g \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}\colon\ gxy = g11$$

genau dann gilt, wenn $x = y = 1$ gilt. Sicherlich gilt die obige Aussage für $x = y = 1$. Sei also $x = 0$ oder $y = 0$. Es genügt zu zeigen, dass es eine Funktion $g \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ gibt mit $gxy \neq g11$. Offensichtlich ist Konjunktion $\wedge$ eine solche Funktion.

Wenn man für jeden Typ X das Gleichheitsprädikat $\doteq_X \in X \to X \to \mathbb{B}$ zur Verfügung stellt, kann man damit die Booleschen Primitive und die Quantoren gemäß der obigen Gleichungen darstellen. Diese Beobachtung findet sich in der Arbeit [Henkin 1963].

10.2 Lambda-Ausdrücke

Wir wissen jetzt, dass sich viele mathematische Aussagen durch Ausdrücke darstellen lassen, die mit den folgenden Primitiven gebildet sind:

- Konstanten (z. B. 0, $+$, $\doteq$, $\neg$, $\Longrightarrow$, $\forall$).
- Funktionsapplikation.

$\forall x \in \mathbb{N}\colon\ 1 \cdot x \doteq x$

$\forall\ (\lambda\, x \in \mathbb{N}\,.\, 1 \cdot x \doteq x)$

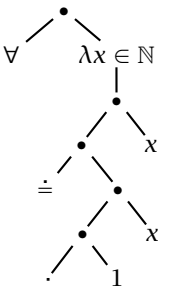


Abbildung 10.1: Drei Beschreibungen eines Lambda-Ausdrucks

- Lambda-Abstraktion und Variablen.

Wir bezeichnen solche Ausdrücke als *Lambda-Ausdrücke*.

Im Zusammenhang mit einem Lambda-Ausdruck unterscheiden wir die folgenden Dinge:

1. Den Ausdruck selbst.
2. Den Wert des Ausdrucks.
3. Verschiedene Beschreibungen des Ausdrucks.

Abbildung 10.1 zeigt drei verschiedene Beschreibungen ein und desselben Lambda-Ausdrucks. Links findet man zwei *textuelle Beschreibungen*. Rechts findet man eine graphische *Baumdarstellung* des Ausdrucks.

Beschreibung von Lambda-Ausdrücken in Standard ML
Die Programmiersprache Standard ML ist so entworfen, dass man mit ihr beliebige Lambda-Ausdrücke beschreiben kann. Betrachten Sie dazu die Signatur in Abbildung 10.2. Im Kontext dieser Signatur kann man den Lambda-Ausdruck $\forall x \in \mathbb{N}\colon 1 \cdot x = x$ wie folgt beschreiben:

```
all (fn x:Nat => eq (mul one x) x)
```

Die Beschreibung von Lambda-Ausdrücken in Standard ML ist aus zwei Gründen interessant. Einerseits haben wir damit eine maschinenverarbeitbare konkrete Syntax für Lambda-Ausdrücke. Andererseits realisiert Standard ML eine *Typdisziplin*, die die Bildung von Lambda-Ausdrücken so einschränkt, dass Funktionsanwendung immer wohldefiniert ist. Beispielsweise werden die Beschreibungen `one one` und `neg one` automatisch als unzulässig erkannt.

De Bruijnsche Darstellung gebundener Variablen
Betrachten Sie die Aussagen

$\forall\ (\lambda\, x \in \mathbb{N}\,.\, 1 \cdot x \doteq x)$ und $\forall\ (\lambda\, y \in \mathbb{N}\,.\, 1 \cdot y \doteq y)$

`type Nat`	$\mathbb{N}$
`type Bool`	$\mathbb{B}$
`val null : Nat`	0
`val one : Nat`	1
`val add : Nat -> Nat -> Nat`	$+$
`val mul : Nat -> Nat -> Nat`	$\cdot$
`val eq : Nat -> Nat -> Bool`	$\doteq_{\mathbb{N}}$
`val not : Bool -> Bool`	$\neg$
`val and : Bool -> Bool -> Bool`	$\wedge$
`val all : (Nat -> Bool) -> Bool`	$\forall_{\mathbb{N}}$
`val all' : ((Nat -> Bool) -> Bool) -> Bool`	$\forall_{\mathbb{N}\to\mathbb{B}}$

Abbildung 10.2: Eine Standard ML Signatur

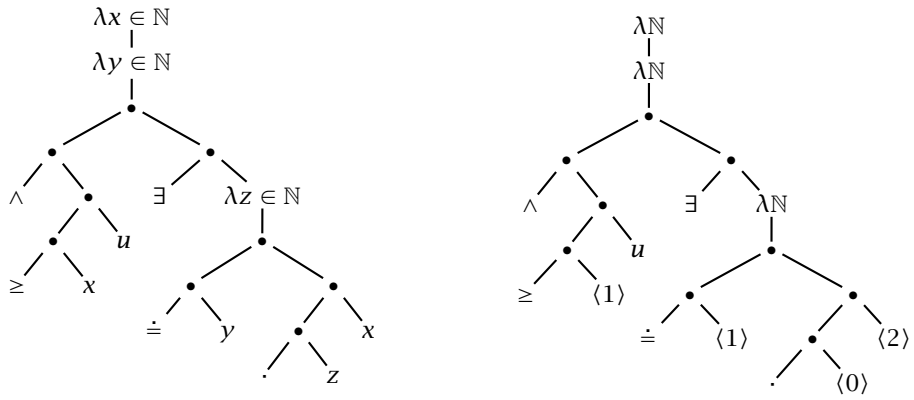


Abbildung 10.3: Normale und De Bruijnsche Darstellung gebundener Variablen

Die zwei Aussagen unterscheiden sich nur in der Wahl der *gebundenen Variablen*. Ob die gebundene Variable mit x oder y realisiert wird, spielt offensichtlich für die Bedeutung der Aussage keine Rolle. Mithilfe der sogenannten *de Bruijnschen Darstellung* kann man gebundene Variablen ohne Namen darstellen. Wir erklären die Idee dieser Darstellung mit dem Beispiel in Abbildung 10.3. Links erscheint die normale Baumdarstellung des Ausdrucks, rechts erscheint die de Bruijnsche Darstellung des Ausdrucks. Die de Bruijnsche Darstellung beschreibt die benutzenden Auftreten der gebundenen Variablen durch *de Bruijnsche Indizes* $\langle i \rangle$ mit $i \in \mathbb{N}$. Der Index $\langle i \rangle$ besagt, dass das durch ihn dargestellte benutzende Variablenauftreten durch denjenigen Lambda-Knoten gebunden ist, den man auf dem Pfad zur Wurzel des Baums nach Überspringen von i Lambda-Knoten erreicht.

Zwei Ausdrücke heißen *α-äquivalent*, wenn sie die gleiche de Bruijnsche Dar-

stellung haben. Beispielsweise sind $\lambda x \in X.\ x$ und $\lambda y \in X.\ y$ α-äquivalent. Informell gesprochen bedeutet α-Äquivalenz, dass zwei Ausdrücke bis auf konsistente Umbenennung von gebundenen Variablen gleich sind.

Die *freien Variablen* eines Ausdrucks sind die Variablen, die auch in der de Bruijnsche Darstellung des Ausdrucks vorkommen. Der in Abbildung 10.3 dargestellte Ausdruck hat nur u als freie Variable.

Elimination von Lambda-Abstraktionen
Sei $f \in X \to Y$. Dann gilt

$$f = \lambda x \in X.\, fx$$

Diese Gleichung ist als **Eta-Regel** bekannt. Sie liefert uns ein Werzeug für die Elimination überflüssiger Lambda-Abstraktionen.

Schönfinkel [1924] hat entdeckt, dass man auf alle Lambda-Abstraktionen verzichten kann, wenn man bestimmte höherstufige Funktionen zur Verfügung hat.[4]

Seien X, Y und Z Mengen. Wir definieren drei Funktionen:

$$I \in X \to X$$
$$I = \lambda x.\, x$$

$$K \in Y \to X \to Y$$
$$Ky = \lambda x.\, y$$

$$S \in (X \to Y \to Z) \to (X \to Y) \to X \to Z$$
$$Sfg = \lambda x.\, (fx)(gx)$$

Wenn wir diese Funktionen für beliebige Mengen X, Y, Z zur Verfügung haben, sind wir in der Lage, Lambda-Abstraktionen zu eliminieren. Dazu werden die definierenden Gleichungen von rechts nach links angewendet. Hier sind Beispiele:

$$\begin{aligned}
\lambda x.\, gxa &= \lambda x.\, (gx)((\lambda x.\, a)x) \\
&= Sg(\lambda x.\, a) && \text{Definition } S \\
&= Sg(Ka) && \text{Definition } K \\
\\
\lambda x.\, f(gxa) &= \lambda x.\, ((\lambda x.\, f)x)((\lambda x.\, gxa)x) \\
&= S(\lambda x.\, f)(\lambda x.\, gxa) && \text{Definition } S \\
&= S(Kf)(\lambda x.\, gxa) && \text{Definition } K \\
&= S(Kf)(Sg(Ka)) && \text{erste Gleichung}
\end{aligned}$$

[4] Schönfinkel machte diese Entdeckung 10 Jahre bevor Church Lambda-Abstraktionen einführte. Die sehr lesbare Arbeit [Schönfinkel 1924] ist eine interessante Lektüre.

Der aufmerksame Leser wird bemerken, dass die Funktionen S und K in den obigen Gleichungen teilweise mit unterschiedlichen Typen verwendet werden. Streng genommen müssten wir jedes Auftreten von I, K und S mit den richtigen Typen indizieren. Die Situation ist ähnlich wie bei den Gleichheitsprädikaten $\doteq$ und den Quantoren $\forall$ und $\exists$.

Die Funktionen I, K, S können mithilfe von Lambda-Termen ohne Rückgriff auf andere Primitive beschrieben werden:

$$I = \lambda x \in X . x$$
$$K = \lambda y \in Y . \lambda x \in X . y$$
$$S = \lambda f \in X \to Y \to Z . \lambda g \in X \to Y . \lambda x \in X . fx(gx)$$

Die Funktion I kann übrigens auf S und K zurückgeführt werden, wenn man diese für beliebige Typen zur Verfügung hat:

$$\begin{aligned} SKK &\models\!\models \lambda x . Kx(Kx) && \text{Definition S} \\ &\models\!\models \lambda x . x && \text{Definition K} \\ &\models\!\models I && \text{Definition I} \end{aligned}$$

Aufgabe 10.1 Geben Sie mögliche Typen für die Auftreten von S und K in SKK an. □

10.3 Signaturen, Typen und Terme

Wir beginnen jetzt mit der Entwicklung eines formalen typtheoretischen Systems ETT. Dafür benötigen wir zunächst eine formale Syntax, die Lambda-Ausdrücke als mathematische Objekte darstellt.

Zunächst benötigen wir so genannte Signaturen. Die Aufgabe eine Signatur besteht darin, die Konstanten festzulegen, mit denen Typen und Terme gebildet werden können.

Betrachten Sie die Standard ML Signatur in Abbildung 10.2. Sie deklariert zunächst zwei *Typkonstanten* `Nat` und `Bool`. Danach werden mehrere *Termkonstanten* (`null`, `one`, usw.) deklariert. Für jede Termkonstante wird ein Typ vereinbart. Die Typen der Termkonstanten werden aus den den zuvor definierten Typkonstanten und dem Funktionspfeil gebildet.

Wir definieren jetzt Signaturen so wie wir sie für ETT benötigen.

Eine **Signatur** ist ein Tripel (Tyc, Tec, ty) wie folgt:

- Tyc und Tec sind Mengen. Die Elemente von Tyc werden als **Typkonstanten** bezeichnet, und die Elemente von Tec als **Termkonstanten**.

- Es gibt mindestens eine Typkonstante.
- Die Menge Ty der **Typen** ist wie folgt definiert:

$$b \in Tyc \qquad \textit{Typkonstante}$$
$$t \in Ty = b \mid t_1 \to t_2 \qquad \textit{Typ}$$

- ty ist eine Funktion $Tec \to Ty$, die jeder Termkonstanten einen Typ zuordnet.

Beachten Sie, dass es sich bei den Typen einer Signatur um syntaktische Objekte handelt. Wir erläutern diesen Aspekt mit einem Beispiel. Seien $\mathbb{N}, \mathbb{B} \in Tyc$. Dann ist der Typ $\mathbb{N} \to \mathbb{B}$ das Objekt $(2, ((1, \mathbb{N}), (1, \mathbb{B})))$. Dagegen ist die Menge $\mathbb{N} \to \mathbb{B}$ die Menge aller Funktionen $\mathbb{N} \to \mathbb{B}$.

Typen der Bauart b heißen **atomar** und Typen der Bauart $t_1 \to t_2$ heißen **funktional**. Damit wir nicht soviele Klammern schreiben müssen, vereinbaren wir, dass der Operator $\to$ rechtsassoziativ klammert. Zum Beispiel:

$$t_1 \to t_2 \to t_3 \to t_4 \quad \mapsto \quad t_1 \to (t_2 \to (t_3 \to t_4))$$

Jeder funktionale Typ kann eindeutig als $t_1 \to \cdots \to t_n \to b$ mit $b \in Tyc$ dargestellt werden ($n \geq 1$).

Terme

Wir definieren jetzt die Ausdrücke, die mit den Konstanten einer Signatur gebildet werden können. Dafür benötigen wir drei Schritte. Zuerst definieren wir die Menge der Variablen. Jede Variable wird mit einem Typ der Signatur versehen. Danach definieren wir die Menge der *Präterme*. Dabei handelt es sich um alle Ausdrücke, die aus den verfügbaren Termkonstanten und Variablen mit Funktionsapplikation und Lambda-Abstraktion gebildet werden können. Aus der Menge der Präterme filtern wir die Menge der *wohlgetypten* Präterme heraus, die wir als *Terme* bezeichnen.

Sei eine Signatur gegeben.

Die Menge der **Variablen** ist wie folgt definiert:

$$x, y \in Var \stackrel{\text{def}}{=} \mathbb{N} \times Ty \qquad \textit{Variable}$$

Wenn $x = (n, t)$ eine Variable ist, sagen wir, dass n der **Name** und t der **Typ** der Variable x ist. Jede Variable hat also einen „eingebauten“ Typ und für jeden Typ gibt es unendlich viele Variablen.

Die Menge der **Präterme** ist wie folgt definiert:

$$c \in Tec \qquad \textit{Termkonstante}$$
$$x \in Var \qquad \textit{Variable}$$
$$M, N \in PT = c \mid x \mid MN \mid \lambda x.M \qquad \textit{Präterm}$$

Die **Typrelation** $\tau \subseteq PT \times Ty$ ist wie folgt definiert:

$$\frac{ty(c) = t}{(c,\, t) \in \tau} \qquad \frac{x = (n, t)}{(x,\, t) \in \tau}$$

$$\frac{(M,\, t' \to t) \in \tau \quad (N,\, t') \in \tau}{(MN,\, t) \in \tau} \qquad \frac{x = (n, t') \quad (M,\, t) \in \tau}{(\lambda x.M,\, t' \to t) \in \tau}$$

Ein Präterm M heißt **wohlgetypt**, wenn ein Typ t mit $(M,t) \in \tau$ existiert. Wohlgetypte Präterme bezeichnen wir kurz als **Terme**, und die Menge aller Terme mit

$$Ter \stackrel{\text{def}}{=} Dom\,\tau$$

Proposition 10.3.1 *Die Typrelation τ ist eine Funktion $Ter \to Ty$.*

Beweis Zeige $\forall M \in Ter\ \forall t, t' \in Ty\colon\ (M,t) \in \tau \wedge (M,t') \in \tau \Longrightarrow t = t'$ durch strukturelle Induktion über M. □

Statt $(M,t) \in \tau$ schreiben wir auch $M\colon t$.

Für die Beschreibung von Termen vereinbaren wir die folgenden Abkürzungen:

$$\begin{array}{rcll} \lambda x_1 x_2 \cdots x_n.M & \mapsto & \lambda x_1.\lambda x_2.\cdots\lambda x_n.M & (n \geq 2) \\ M_1 M_2 M_3 \cdots M_n & \mapsto & (\cdots((M_1 M_2)M_3)\cdots)M_n & (n \geq 3) \\ \lambda x.MN & \mapsto & \lambda x.(MN) & \end{array}$$

Weiter vereinbaren wir, dass der mit dem Punkt beginnende Rumpf einer Lambda-Abstraktion so weit wie möglich nach rechts ausgedehnt wird.

Die Menge der **freien Variablen eines Terms** ist wie folgt definiert:

$$\begin{array}{rcl} FV \in Ter \to \mathcal{P}_{fin}(Var) & & \\ FV(c) & = & \varnothing \\ FV(x) & = & \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \\ FV(\lambda x.M) & = & FV(M) - \{x\} \end{array}$$

Ein Term heißt

- **Applikation**, wenn er die Form MN hat.
- **Abstraktion**, wenn er die Form $\lambda x.M$ hat.
- **atomar**, wenn er eine Konstante oder eine Variable ist.

- **zusammengesetzt**, wenn er eine Applikation oder Abstraktion ist.
- **kombinatorisch**, wenn er keine Abstraktion enthält.
- **offen**, wenn er mindestens eine freie Variable hat.
- **geschlossen**, wenn er keine freien Variablen hat.

Die **Stufe** $ord(t)$ eines Typs t ist wie folgt definiert:

$$\begin{aligned} ord &\in Ty \to \mathbb{N} \\ ord(b) &= 0 \\ ord(t_1 \to t_2) &= \max\{ord(t_1) + 1,\ ord(t_2)\} \end{aligned}$$

Die **Stufe einer Termkonstanten** c ist $ord(ty(c))$, und die **Stufe eines Terms** M ist $ord(\tau M)$.

Wenn wir eine Signatur mit den Typen $\mathbb{B}$ und $\mathbb{N}$ betrachten, stellt sich die Frage, welche Typen wir den Konstanten 0 und 1 geben sollen. Da nur ein Typ möglich ist, müssen wir uns jeweils für $\mathbb{B}$ oder $\mathbb{N}$ entscheiden. Oft ist es bequem, 0 und 1 den Typ $\mathbb{N}$ zu geben und für $\mathbb{B}$ eigene Konstanten vorzusehen, zum Beispiel **F** und **T**. Diese Lösung findet man auch in vielen Programmiersprachen.

10.4 Strukturen

Typen und Terme sind syntaktische Objekte, die dazu dienen, semantische Objekte zu beschreiben. Die damit zusammenhängenden Fragen werden wir in diesem Abschnitt klären.

Das einem syntaktischen Objekt zugeordnete semantische Objekt bezeichnen wir als seine *Denotation*.

Zunächst müssen wir den Konstanten der zugrundeliegenden Signatur semantische Objekte zuordnen. Den Typkonstanten können wir beliebige nichtleere Mengen zuordnen. Bei der Wahl der Denotationen für die Termkonstanten müssen wir darauf achten, dass sie mit den durch die Signatur festgelegten Typen verträglich sind.

Eine **Struktur für eine Signatur** (Tyc, Tec, ty) ist eine Funktion $\mathcal{A}$ wie folgt:

- $Tyc \cup Tec \subseteq Dom\,\mathcal{A}$.
- Für jede Typkonstante $b \in Tyc$ ist $\mathcal{A}(b)$ eine nichtleere Menge.

- Die Funktion $\mathcal{A}^{\mathrm{Ty}}$, die jedem Typ der Signatur eine Menge zuordnet, ist wie folgt definiert:

$$\begin{aligned} \mathit{Dom}\,\mathcal{A}^{\mathrm{Ty}} &= \mathit{Ty} \\ \mathcal{A}^{\mathrm{Ty}}(b) &= \mathcal{A}(b) \\ \mathcal{A}^{\mathrm{Ty}}(t_1 \to t_2) &= \mathcal{A}^{\mathrm{Ty}}(t_1) \to \mathcal{A}^{\mathrm{Ty}}(t_2) \end{aligned}$$

- Für jede Termkonstante $c \in \mathit{Tec}$ gilt $\mathcal{A}(c) \in \mathcal{A}^{\mathrm{Ty}}(\mathit{ty}(c))$.

Beachten Sie, dass $\mathcal{A}^{\mathrm{Ty}}$ einem funktionalen Typ $t_1 \to t_2$ die Menge aller Funktionen $\mathcal{A}^{\mathrm{Ty}}(t_1) \to \mathcal{A}^{\mathrm{Ty}}(t_2)$ zuordnet. Statt $\mathcal{A}^{\mathrm{Ty}}(t)$ werden wir kürzer $\mathcal{A}(t)$ schreiben.

Da die Konstanten einer Signatur beliebige mathematische Objekte sein können, ist es oft naheliegend, die Konstanten durch sich selbst zu interpretieren (d. h. $\mathcal{A}(b) = b$ und $\mathcal{A}(c) = c$). Beispielsweise können wir die Menge der natürlichen Zahlen $\mathbb{N}$ als Typkonstante verwenden, und die Elemente von $\mathbb{N}$ als Termkonstanten. Auch die Additions- und Multiplikationsfunktion können wir als Termkonstanten verwenden.

Wir nehmen an, dass eine Signatur und eine Struktur $\mathcal{A}$ gegeben sind.

Die Menge

$$|\mathcal{A}| \stackrel{\text{def}}{=} \bigcup_{t \in \mathit{Ty}} \mathcal{A}(t)$$

bezeichnen wir als das durch die Signatur und die Struktur gegebene **Universum**. Die Elemente des Universums bezeichnen wir als **Werte**.

Eine Funktion $\sigma \in \mathit{Var} \to |\mathcal{A}|$ heißt **Belegung**, wenn $\forall x \in \mathit{Var}\colon\ \sigma x \in \mathcal{A}(\tau x)$. Eine Belegung ist also eine Funktion, die jeder Variablen einen typgerechten Wert zuordnet. Die Menge aller Belegungungen bezeichnen wir mir $\Sigma_{\mathcal{A}}$.

Da es Variablen für jeden Typ gibt, und jede Belegung jeder Variablen einen typgerechten Wert zuordnen muss, gibt es nur dann Belegungen, wenn kein Typ die leere Menge denotiert. Das erklärt, warum wir darauf bestehen, dass Typkonstanten mit *nichtleeren* Mengen interpretiert werden.[5]

Die **Denotationsfunktion für Terme** definieren wir mit struktureller Rekursion wie folgt:

$$\begin{aligned} \hat{\mathcal{A}} &\in \mathit{Ter} \to \Sigma_{\mathcal{A}} \to |\mathcal{A}| \\ \hat{\mathcal{A}}(c)\sigma &= \mathcal{A}(c) \\ \hat{\mathcal{A}}(x)\sigma &= \sigma x \\ \hat{\mathcal{A}}(MN)\sigma &= (\hat{\mathcal{A}}(M)\sigma)\,(\hat{\mathcal{A}}(N)\sigma) \\ \hat{\mathcal{A}}(\lambda x.M)\sigma &= \lambda\, \nu \in \mathcal{A}(\tau x)\mathbf{.}\ \hat{\mathcal{A}}(M)(\sigma[\nu/x]) \end{aligned}$$

[5] Wenn man leere Typen zulassen will, kann man mit partiellen Belegungen arbeiten, die nur ausgewählten Variablen Typen zuweisen.

Die definierende Gleichung für Applikationen ist zulässig, da

$$\forall M \in \mathit{Ter}\ \forall \sigma \in \Sigma_{\mathcal{A}}\colon\ \hat{\mathcal{A}}(M) \in \mathcal{A}(\tau M)$$

gilt. Diese Eigenschaft folgt parallel zur Definition mit struktureller Induktion.

Statt $\hat{\mathcal{A}}(M)$ schreiben wir kürzer $\mathcal{A}(M)$.

Proposition 10.4.1 (Koinzidenz) *Sei M ein Term und seien σ und σ' zwei Belegungen, die auf den freien Variablen von M übereinstimmen. Dann gilt $\mathcal{A}(M)\sigma = \mathcal{A}(M)\sigma'$.*

Beweis Durch strukturelle Induktion über M. □

Die durch die Proposition festgestellte Koinzidenzeigenschaft ist essentiell. Sie besagt, dass die Denotation eines Terms nur von den für die freien Variablen gewählten Werten abhängt (bei gegebener Struktur). Daraus folgt insbesondere, dass ein geschlossener Term für alle Belegungen denselben Wert liefert.

Für jede Struktur $\mathcal{A}$ definieren wir auf der Menge der Terme eine Äquivalenzrelation:

$$M \models\!|_{\mathcal{A}}\ N \quad \overset{\text{def}}{\Longleftrightarrow} \quad M, N \in \mathit{Ter}\ \wedge\ \tau M = \tau N\ \wedge\ \mathcal{A}(M) = \mathcal{A}(N)$$

Darüberhinaus definieren wir auf der Menge der Terme die folgende Äquivalenzrelation:

$$M \models\!| N \quad \overset{\text{def}}{\Longleftrightarrow} \quad \forall \text{ Struktur } \mathcal{A}\colon\ M \models\!|_{\mathcal{A}}\ N$$

Wir vereinbaren die folgenden Sprechweisen:

$M \models\!|_{\mathcal{A}}\ N$	M **$\mathcal{A}$-äquivalent zu** N
$M \models\!| N$	M **λ-äquivalent zu** N

Proposition 10.4.2 *Für jede Struktur $\mathcal{A}$ gilt: $\models\!| \ \subseteq\ \models\!|_{\mathcal{A}}$.*

Proposition 10.4.3 (Ersetzungsregel) *Sei $\mathcal{A}$ eine Struktur und seien M und M' zwei Terme, sodass M' aus M erhalten werden kann, indem ein Auftreten des Teilterms N durch den Term N' ersetzt wird. Dann: $N \models\!|_{\mathcal{A}}\ N' \Longrightarrow M \models\!|_{\mathcal{A}}\ M'$.*

Die durch die folgende Proposition formulierte Eta-Regel gibt uns die Möglichkeit, überflüssige Abstraktionen zu eliminieren.

Proposition 10.4.4 (Eta-Regel) *Für jeden Term $\lambda x.Mx$ mit $x \notin FV(M)$ gilt: $\lambda x.Mx \models\!| M$.*

Sei $\mathcal{A}$ eine Struktur. Manchmal ist es hilfreich, die **Konsequenzrelation für** $\mathcal{A}$ zur Verfügung zu haben:

$$M \models_{\mathcal{A}} N \quad \overset{\text{def}}{\iff} \quad M, N \in Ter \ \wedge \ \tau M = \tau N \ \wedge \ \forall \sigma \in \Sigma_{\mathcal{A}} \colon \ \mathcal{A}(M)\sigma \leq \mathcal{A}(N)\sigma$$

Für eine Aussage $M \models_{\mathcal{A}} N$ vereinbaren wir die folgenden Sprechweisen: M **impliziert in** $\mathcal{A}$ N oder **aus** M **folgt in** $\mathcal{A}$ N.

Erwartungsgemäß gilt für jede Struktur $\mathcal{A}$ und alle Terme M, N:

$$M \models\!\mid_{\mathcal{A}} N \quad \iff \quad M \models_{\mathcal{A}} N \wedge N \models_{\mathcal{A}} M$$

10.5 Variablenbindung und Substitution

Bisher haben wir α-Äquivalenz nur informell charakterisiert. Eine formale Definition erfordert etwas Aufwand. Eine Möglichkeit für die formale Definition von α-Äquivalenz wäre, die de Bruijnsche Darstellung eines Terms formal zu definieren. Wir wählen eine zweite Möglichkeit, die α-Äquivalenz mithilfe eines Substitutionsoperators definiert.

Zunächst stellen wir fest, dass wir für die gebundenen Variablen eines Terms eindeutig bestimmte Namen wählen können. Von oben kommend, wählen wir für eine gebundene Variable jeweils den kleinsten Namen, der zu keinem Konflikt mit den freien Variablen führt.

Eine Abstraktion $\lambda(n,t).M$ heißt α-**normal**, wenn gilt:

$$n = \min\{\, m \in \mathbb{N} \mid (m,t) \notin FV(\lambda(n,t).M) \,\}$$

Ein Term heißt α-**normal**, wenn jeder seiner abstrahierenden Teilterme α-normal ist. Hier ist ein Beispiel für einen α-normalen Term:

$$\lambda(0,t).\lambda(2,t).\, x\,(0,t)(1,t)(2,t)$$

Beachten Sie, dass der Name der freien Variable x keine Rolle spielt, da der Typ von x verschieden von t ist (sonst hätten wir einen nicht wohlgetypten Präterm).

Zu jeder de Bruijnschen Darstellung D gibt es offenbar genau einen α-normalen Term M, sodass D die de Bruijnschen Darstellung von M ist. Gegeben D, können wir M berechnen, indem wir für jeden Lambda-Knoten den kanonischen Namen bestimmen. Dabei bearbeiten wir die Lambda-Knoten von oben nach unten gehend.

Substitutionen

Um das Konzept der Variablenumbenennung präzise beschreiben zu können, benötigen wir den Begriff der Substitution.

Eine Funktion $\theta \in \mathit{Var} \to \mathit{Ter}$ heißt **Substitution**, wenn $\forall x \in \mathit{Var}\colon \tau x = \tau(\theta x)$. Eine Substitution ist also eine Funktion, die jeder Variablen einen typgerechten Term zuordnet. Die Menge aller Substitutionen bezeichnen wir mit *Sub*.

Seien $x_1, \ldots, x_n$ paarweise verschiedene Variablen und $M_1, \ldots, M_n$ Terme mit $\tau x_i = \tau M_i$ für $i \in \{1, \ldots, n\}$. Dann bezeichnet

$$[x_1 := M_1, \ldots, x_n := M_n]$$

die Substitution θ mit

- $\forall i \in \{1, \ldots, n\}\colon\ \theta x_i = M_i$.
- $\forall x \in \mathit{Var} - \{x_1, \ldots, x_n\}\colon\ \theta x = x$.

Die Substitution $[\,]$ heißt **Identitätssubstitution**.

Kontextoperator
Der Kontextoperator wendet eine Substitution auf einen Term an, indem er alle Teilterme, die nur aus einer Variable bestehen, gemäß der Substitution ersetzt. Wir definieren den **Kontextoperator** C mit struktureller Rekursion:

$$\begin{aligned} C &\in \mathit{Ter} \to \mathit{Sub} \to \mathit{Ter} \\ C(c)\theta &= c \\ C(x)\theta &= \theta x \\ C(MN)\theta &= (C(M)\theta)(C(N)\theta) \\ C(\lambda x.M)\theta &= \lambda y.C(M)\theta \end{aligned}$$

Die definierende Gleichung für Applikationen ist zulässig, da

$$\forall M \in \mathit{Ter}\ \ \forall \theta \in \mathit{Sub}\colon\ \tau(C(M)\theta) = \tau M$$

gilt. Diese Eigenschaft folgt parallel zur Definition mit struktureller Induktion.

Hier sind zwei Beispiele für die Anwendung des Kontextoperators:

$$\begin{aligned} C((\lambda x.x)y)\,[x := z, y := x] &= (\lambda x.z)x \\ C(\lambda x.yxz)\,[z := x] &= \lambda x.yxx \end{aligned}$$

Die Beispiele zeigen, dass $C(M)\theta$ die *Bindungsstruktur* des Terms M nicht respektiert.

Mithilfe des Kontextoperators können wir die Ersetzungsregel (Proposition 10.4.3) neu formulieren.

Proposition 10.5.1 (Ersetzungsregel) *Sei $\mathcal{A}$ eine Struktur und seien θ, θ' Substitutionen. Dann gilt für jeden Term M:*

$$(\forall x \in \mathit{Var}\colon\ \theta x \models\!|_{\mathcal{A}}\ \theta' x) \implies C(M)\theta \models\!|_{\mathcal{A}}\ C(M)\theta'$$

Aufgabe 10.2 Sei eine Signatur mit mindestens einer Typkonstante b gegeben. Geben Sie zwei λ-äquivalente Terme M und N und eine Substitution θ an, so dass die Terme $C(M)\theta$ und $C(N)\theta$ in keiner Struktur äquivalent sind, die b mit einer mindestens zweielementigen Menge interpretiert. □

Proposition 10.5.2 (Interne Ersetzungsregel) *Sei $\mathcal{A}$ eine Struktur, die die Konstante $\doteq$ als Gleichheitsprädikat und die Konstante $\wedge$ als Konjunktion interpretiert. Weiter sei $M_1 \doteq M_2 \wedge x \doteq M_1 \wedge M$ ein Term. Dann gilt:*

$$M_1 \doteq M_2 \ \wedge\ C(M)[x{:=}\,M_1] \ \models\!|_{\mathcal{A}}\ M_1 \doteq M_2 \ \wedge\ C(M)[x{:=}\,M_2]$$

Substitutionsoperator
Unser nächstes Ziel ist die Definition eines Substitutionsoperators, der eine Substitution so auf einen Term anwendet, dass dabei die Bindungsstruktur des Terms erhalten bleibt. Für kombinatorische Terme soll der Substitutionsoperator dieselben Ergebnisse wie der Kontextoperator liefern. Bei Abstraktionen soll der Substitutionsoperator die gebundene Variable so umbenennen, dass die Abstraktion α-normal wird und die Bindungsstruktur erhalten bleibt.

Für die Definition des Substitutionsoperators benötigen wir eine Hilfsfunktion (canonical variable):

$$cv \in \mathcal{P}_{fin}(Var) \times Sub \times Ty \to Var$$
$$cv(V,\theta,t) \ =\ (\min\{\, n \in \mathbb{N} \mid \forall x \in V\colon\ (n,t) \notin FV(\theta x)\,\},\ t)$$

Wir bezeichnen $cv(V,\theta,t)$ als die **kanonische Variable für** (V,θ,t).

Den **Substitutionsoperator** S definieren wir jetzt mit strukturelle Rekursion wie folgt:

$$\begin{aligned} S &\in Ter \to Sub \to Ter \\ S(c)\theta &= c \\ S(x)\theta &= \theta x \\ S(MN)\theta &= (S(M)\theta)(S(N)\theta) \\ S(\lambda x.M)\theta &= \lambda y.S(M)(\theta[x{:=}\,y]) \quad \text{wobei } y = cv(FV(\lambda x.M),\theta,\tau x) \end{aligned}$$

Die definierende Gleichung für Applikationen MN ist zulässig, da

$$\forall M \in Ter\ \forall \theta \in Sub\colon\ \tau(S(M)\theta) = \tau M$$

gilt. Diese Eigenschaft folgt parallel zur Definition mit struktureller Induktion.

Statt $S(M)\theta$ schreiben wir kürzer $M\theta$.

Lemma 10.5.3 (Substitution) *Sei $\mathcal{A}$ eine Struktur, M ein Term, θ eine Substitution, und $\sigma \in \Sigma_{\mathcal{A}}$ eine Belegung. Dann $\mathcal{A}(M\theta)\sigma \ =\ \mathcal{A}(M)(\lambda\, x \in Var\,.\,\mathcal{A}(\theta x)\sigma)$.*

Beweis Durch strukturelle Induktion über M. □

Proposition 10.5.4 (Einsetzungsregel) *Für jede Struktur $\mathcal{A}$ gilt:*

$$\forall M, N \in Ter \;\; \forall \theta \in Sub\colon \;\; M \models\!|_{\mathcal{A}} N \implies M\theta \models\!|_{\mathcal{A}} N\theta$$

Beweis Folgt aus dem Substitutionslemma. □

Die durch die folgende Proposition formulierte Beta-Regel gibt uns die Möglichkeit, die Applikation von Abstraktionen zu vereinfachen.

Proposition 10.5.5 (Beta-Regel) *Für jeden Term $(\lambda x.M)N$ gilt:*

$$(\lambda x.M)N \models\!| M[x := N]$$

Beweis Folgt aus dem Substitutionslemma. □

Proposition 10.5.6 *Sei M ein Term und θ eine Substitution. Dann:*

1. $FV(M\theta) = \bigcup_{x \in FV(M)} FV(\theta x)$.
2. *Wenn θx für alle $x \in FV(M)$ α-normal ist, dann ist $M\theta$ α-normal.*
3. *$M[\,]$ ist α-normal.*
4. *M ist α-normal genau dann, wenn $M = M[\,]$.*

Beweis Die Beweise sind nicht ganz einfach. Siehe [Stoughton 1988]. □

Alpha-Äquivalenz

Wir sind jetzt in der Lage, Alpha-Äquivalenz formal zu definieren. Informell gesehen sind zwei Terme genau dann alpha-äquivalent, wenn sie dieselbe de Bruijnsche Darstellung haben.

Auf der Menge der Terme definieren wir die folgende Äquivalenzrelation:

$$M \sim_\alpha N \quad \stackrel{\text{def}}{\iff} \quad M, N \in Ter \;\wedge\; M[\,] = N[\,]$$

Zwei Terme heißen **α-äquivalent**, wenn $M \sim_\alpha N$. Ein Term M heißt **α-Normalform** eines Terms N, wenn $M = N[\,]$. Zwei Terme sind also genau dann α-äquivalent, wenn sie dieselbe α-Normalform haben.

Proposition 10.5.7 (Alpha-Äquivalenz) *Für alle Terme M, N gilt:*

$$M \sim_\alpha N \implies M \models\!| N$$

Die wichtigsten Äquivalenzregeln für Terme

Alpha	$M \models\mid M[]$
	$\lambda x.M \models\mid \lambda y.M[x := y]$ falls $y \notin FV(M)$ und $\tau y = \tau x$
Beta	$(\lambda x.M)N \models\mid M[x := N]$
Eta	$\lambda x.Mx \models\mid M$ falls $x \notin FV(M)$
Einsetzung	$M \models\mid_{\mathcal{A}} N \implies M\theta \models\mid_{\mathcal{A}} N\theta$
Ersetzung	$N_1 \models\mid_{\mathcal{A}} N_2 \implies C(M)[x := N_1] \models\mid_{\mathcal{A}} C(M)[x := N_2]$

Beweis Sei $\mathcal{A}$ eine Struktur, $\sigma \in \Sigma_{\mathcal{A}}$ eine Belegung und $M \sim_\alpha N$. Wir zeigen $\mathcal{A}(M)\sigma = \mathcal{A}(N)\sigma$:

$$\begin{aligned} \mathcal{A}(M)\sigma &= \mathcal{A}(M[])\sigma && \text{Substitutionslemma} \\ &= \mathcal{A}(N[])\sigma && \text{da } M \sim_\alpha N \\ &= \mathcal{A}(N)\sigma && \text{Substitutionslemma} \end{aligned}$$

□

Die durch die folgende Proposition formulierte Alpha-Regel gibt uns die Möglichkeit, gebundene Variablen umzubenennen.

Proposition 10.5.8 (Alpha-Regel) *Sei $\lambda x.M$ ein Term und y eine Variable, die denselben Typ wie x hat und nicht frei in M vorkommt. Dann $\lambda x.M \sim_\alpha \lambda y.M[x := y]$.*

Beweis Siehe [Stoughton 1988]. □

10.6 Reduktion

Wir geben jetzt ein einfaches Verfahren an, das entscheidet, ob zwei Terme λ-äquivalent sind. Das Verfahren beruht auf der Tatsache, das zwei Terme genau dann λ-äquivalent sind, wenn sie nach vollständiger Vereinfachung mit den Regeln

$$\begin{array}{lrcll} (\beta) & (\lambda x.M)N & \to & M[x := N] & \\ (\eta) & \lambda x.Mx & \to & M & \text{falls } x \notin FV(M) \end{array}$$

α-äquivalent sind. Das Verfahren ist effektiv, da immer nur endlich viele Vereinfachungsschritte möglich sind und α-Äquivalenz leicht entscheidbar ist. Die

Beta-, die Eta- und die Ersetzungsregel garantieren, dass die Vereinfachung mit den Regeln β und η denotationserhaltend ist. Hier ist ein Beispiel:

$$\begin{aligned}
\lambda x.(\lambda f y g.f g y)(\lambda h.h)xh &\to_\beta \lambda x.(\lambda y g.(\lambda h.h)g y)xh \\
&\to_\beta \lambda x.(\lambda y g.g y)xh \\
&\to_\beta \lambda x.(\lambda g.g x)h \\
&\to_\beta \lambda x.hx \\
&\to_\eta h
\end{aligned}$$

Bei den ersten vier Vereinfachungsschritten haben wir neben der Beta-Regel auch die Ersetzungsregel benutzt. Da wir nicht wissen, wie der Substitutionsoperator gebundene Variablen umbenennt (wir haben nur partielle Information über die gebundenen Variablen), machen die Vereinfachungsschritte unter Umständen auch von der Alpha-Regel Gebrauch.

Die gerade vorgestellte Vereinfachung von Termen bezeichnet man auch als **Reduktion**. Sie hat die wichtige Eigenschaft, dass man zu jedem Term eine eindeutig bestimmte Normalform bestimmen kann. Außerdem gilt, dass zwei Terme genau dann λ-äquivalent sind, wenn sie die gleiche Normalform haben. Konzeptuell gesehen haben die Normalformen für Terme also ähnliche Eigenschaften wie die Primbäume und Primformen für aussagenlogische Formeln.

Normalformen
Wir definieren jetzt, was genau wir unter einer Normalform eines Terms verstehen wollen. Ein Term heißt

- β-**Redex**, wenn er die Form $(\lambda x.M)N$ hat.
- β-**normal**, wenn keinen β-Redex enthält.
- η-**Redex**, wenn er die Form $\lambda x.Mx$ mit $x \notin FV(M)$ hat.
- η-**normal**, wenn keinen η-Redex enthält.
- $\beta\eta$-**normal**, wenn er β- und η-normal ist.
- λ-**normal**, wenn er α-, β- und η-normal ist.

Beachten Sie, dass jeder β-normale Term, der keine Abstraktion ist, eindeutig als $M_0 \ldots M_n$ mit M_0 atomar und $n \geq 0$ dargestellt werden kann.

Proposition 10.6.1 *Wenn M $\beta\eta$-normal ist, dann ist $M[\,]$ λ-normal.*

Ein Term N heißt $\beta\eta$-**Normalform** eines Terms M, wenn N $\beta\eta$-normal ist und $M \models\!\mid N$ gilt. Ein Term N heißt λ-**Normalform** eines Terms M, wenn N λ-normal ist und $M \models\!\mid N$ gilt.

Satz 10.6.2 (Kanonizität) *Zwei λ-normale Terme sind genau dann λ-äquivalent, wenn sie gleich sind.*

Beweis Schwer. Siehe [Mitchell]. □

Korollar 10.6.3 *Ein Term hat höchstens eine λ-Normalform.*

Lambda-Kompatibilität und Kongruenzabschluss
Eine binäre Relation R auf der Menge der Terme heißt

- **rein**, wenn für all $(M,N) \in R$ gilt: $\tau M = \tau N$.
- λ-**kompatibel**, wenn $R \subseteq \models\!|$ gilt.

Mit $\models\!|$, $\models\!|_{\mathcal{A}}$ und $\sim_\alpha$ haben wir bereits drei Beispiele für reine Relationen kennen gelernt. Die Relationen $\models\!|$ und $\sim_\alpha$ sind zudem λ-kompatibel. Beachten Sie, dass jede λ-kompatible Relation rein ist.

Sei R eine reine Relation. Den **Kongruenzabschluss** $CC(R)$ von R definieren wir rekursiv mithilfe der folgenden Inferenzregeln:

$$\frac{(M,M') \in R}{(M,M') \in CC(R)} \qquad \frac{\lambda x.M \in Ter \quad (M,M') \in CC(R)}{(\lambda x.M, \lambda x.M') \in CC(R)}$$

$$\frac{MN \in Ter \quad (M,M') \in CC(R)}{(MN, M'N) \in CC(R)} \qquad \frac{MN \in Ter \quad (N,N') \in CC(R)}{(MN, MN') \in CC(R)}$$

Offensichtlich ist $CC(R)$ eine reine Relation auf Termen mit $R \subseteq CC(R)$. Machen Sie sich klar, dass $(M,N) \in CC(R)$ genau dann gilt, wenn es ein Paar $(M',N') \in R$ gibt, sodass M in N überführt werden kann, indem ein Teiltermauftreten von M' durch N' ersetzt wird.

Den **relativierte Kongruenzabschluss** $ACC(R)$ einer reinen Relation R definieren wir wie folgt:

$$ACC(R) \stackrel{\text{def}}{=} \sim_\alpha \circ CC(R) \circ \sim_\alpha$$

Offensichtlich ist $ACC(R)$ eine reine Relation auf Termen und erfüllt $R \subseteq ACC(R)$.

Proposition 10.6.4 *Sei R eine λ-kompatible Relation. Dann sind $CC(R)$ und $ACC(R)$ λ-kompatibel.*

Beweis Folgt aus der Ersetzungs- und der Alpha-Regel. □

Reduktion
Wir definieren jetzt, was genau wir unter einem Reduktionsschritt verstehen wollen. Dazu definieren wir die folgenden binären Relationen auf der Menge der Terme:

$$\to_\beta \stackrel{\text{def}}{=} ACC(\{\,((\lambda x.M)N,\ M[x:=N]) \mid (\lambda x.M)N \text{ Term}\,\}) \qquad \beta\textbf{-Reduktion}$$

$$\to_\eta \stackrel{\text{def}}{=} ACC(\{\,(\lambda x.Mx,\ M) \mid Mx \text{ Term und } x \notin FV(M)\,\}) \qquad \eta\textbf{-Reduktion}$$

$$\to_{\beta\eta} \stackrel{\text{def}}{=} \to_\beta \cup \to_\eta \qquad \beta\eta\textbf{-Reduktion}$$

Proposition 10.6.5 *Ein Term M ist genau dann*

1. *β-normal, wenn es keinen Term M' gibt mit $M \to_\beta M'$.*
2. *η-normal, wenn es keinen Term M' gibt mit $M \to_\eta M'$.*
3. *$\beta\eta$-normal, wenn es keinen Term M' gibt mit $M \to_{\beta\eta} M'$.*

Proposition 10.6.6 *Die Relationen $\to_\beta$, $\to_\eta$ und $\to_{\beta\eta}$ sind λ-kompatibel.*

Beweis Folgt mit Proposition 10.6.4 aus der Beta- und Eta-Regel. □

Satz 10.6.7 (Terminierung) *Die Relationen $\to_\beta$, $\to_\eta$ und $\to_{\beta\eta}$ sind terminierend.*

Beweis Schwer. Siehe [Mitchell]. □

Satz 10.6.8 (Reduktion)

1. *Zu jedem Term existiert genau eine λ-Normalform. Diese kann durch $\beta\eta$-Reduktion und abschließende Alpha-Normalisierung berechnet werden.*
2. *Zwei Terme sind genau dann λ-äquivalent, wenn sie die gleiche λ-Normalform haben.*

Beweis Folgt aus den Sätzen 10.6.2 und 10.6.7 sowie den Propositionen 10.5.7, 10.6.5 und 10.6.6. □

10.7 Mehr über Quantoren

Wir gehen jetzt genauer auf die Quantoren $\forall$ und $\exists$ ein. Zunächst rufen wir uns die Definition der Quantoren ins Gedächtnis. Sei X eine nichtleere Menge. Der **Allquantor** und der **Existenzquantor** für X sind wie folgt definiert:

$$\forall_X \in (X \to \mathbb{B}) \to \mathbb{B} \qquad \exists_X \in (X \to \mathbb{B}) \to \mathbb{B}$$
$$\forall_X(f) = \min\{fx \mid x \in X\} \qquad \exists_X(f) = \max\{fx \mid x \in X\}$$

Der Allquantor testet, ob eine Funktion $X \to \mathbb{B}$ für alle Argumente 1 liefert. Der Existenzquantor testet, ob eine Funktion $X \to \mathbb{B}$ für mindestens ein Argument 1 liefert.

Für die Applikation von Quantoren auf Lambda-Abstraktionen vereinbaren wir die folgenden Notationen:

$$\forall x \in X \,.\, M \quad \stackrel{\text{def}}{=} \quad \forall_X (\lambda\, x \in X \,.\, M)$$
$$\exists x \in X \,.\, M \quad \stackrel{\text{def}}{=} \quad \exists_X (\lambda\, x \in X \,.\, M)$$

Abbildung 10.4 zeigt die wichtigsten Gesetze für Qantoren. Auf die Angabe der Typen für die quantifizierten Variablen haben wir verzichtet, da diese durch die Typvorgaben für die freien Variablen eindeutig bestimmt sind. Beachten Sie, dass $\exists x \,.\, fx \wedge b$ als $\exists x \,.\, (fx \wedge b)$ zu lesen ist, da nach dem Punkt generell ein möglichst langer Ausdruck zu bilden ist.

Aus der zweispaltigen Darstellung der Gesetze kann man ersehen, dass der Existenz- und der Allquantor in einer Dualitätsbeziehung stehen. Das letzte Vertauschungsgesetz ist zu sich selbst dual.

Der Beweis der Gesetze ist einfach (Übungsaufgabe!). Man benötigt lediglich die Definitionen der Quantoren und der Booleschen Operationen. Einzige Ausnahme sind die Vertauschungsgesetze in der vorletzten Zeile. Die durch das linke Gesetz

$$\forall x \in X \; \exists y \in Y \,.\, hxy \quad = \quad \exists z \in X \to Y \; \forall x \in X \,.\, hx(zx)$$

zugesicherte Existenz einer Funktion $z \in X \to Y$ folgt mit dem Auswahlaxiom. Das duale rechte Gesetz folgt aus dem linken Gesetz mit den de Morganschen Regeln unter Ausnutzung der Tatsache, dass das linke Gesetz für alle $h \in X \to Y \to \mathbb{B}$ gilt.

Abbildung 10.5 zeigt die syntaktische Reformulierung der Quantorengesetze aus Abbildung 10.4. Neu hinzugekommen sind zwei Umbenennungsgesetze, die unmittelbar aus der Alpha-Regel folgen. Die anderen Gesetze folgen aus ihren semantischen Vorbildern in Abbildung 10.4. Beachten Sie, dass die Eliminationsgesetze und die erste Gruppe der Distributionsgesetze nur unter einer Freiheitsbedingung für die quantifizierte Variable gelten.

Introduktion

$$f a \leq \exists f \qquad \forall f \leq f a$$

Elimination

$$\exists x\,.\,b = b \qquad \forall x\,.\,b = b$$

De Morgan

$$\neg(\exists x\,.\,f x) = \forall x\,.\,\neg(f x) \qquad \neg(\forall x\,.\,f x) = \exists x\,.\,\neg(f x)$$

Distribution

$$\exists x\,.\,f x \wedge b = (\exists x\,.\,f x) \wedge b \qquad \forall x\,.\,f x \vee b = (\forall x\,.\,f x) \vee b$$

$$\exists x\,.\,f x \vee b = (\exists x\,.\,f x) \vee b \qquad \forall x\,.\,f x \wedge b = (\forall x\,.\,f x) \wedge b$$

$$\exists x\,.\,f x \vee g x = (\exists x\,.\,f x) \vee (\exists x\,.\,g x) \qquad \forall x\,.\,f x \wedge g x = (\forall x\,.\,f x) \wedge (\forall x\,.\,g x)$$

Vertauschung

$$\exists x \exists y\,.\,h x y = \exists y \exists x\,.\,h x y \qquad \forall x \forall y\,.\,h x y = \forall y \forall x\,.\,h x y$$

$$\forall x \exists y\,.\,h x y = \exists z \forall x\,.\,h x (z x) \qquad \exists x \forall y\,.\,h x y = \forall z \exists x\,.\,h x (z x)$$

$$\exists x \forall y\,.\,h x y \leq \forall y \exists x\,.\,h x y$$

Die Gesetze gelten für alle nichtleeren Mengen X, Y und für alle $a \in X$, $b \in \mathbb{B}$, $f, g \in X \to \mathbb{B}$ und $h \in X \to Y \to \mathbb{B}$.

Abbildung 10.4: Quantorengesetze

Introduktion

$M[x := N] \models_{\mathcal{A}} \exists x.M$ $\quad$ $\forall x.M \models_{\mathcal{A}} M[x := N]$

Elimination $(x \notin FV(M))$

$\exists x.M \models\!\mid_{\mathcal{A}} M$ $\quad$ $\forall x.M \models\!\mid_{\mathcal{A}} M$

De Morgan

$\neg(\exists x.M) \models\!\mid_{\mathcal{A}} \forall x.\neg M$ $\quad$ $\neg(\forall x.M) \models\!\mid_{\mathcal{A}} \exists x.\neg M$

Distribution $(x \notin FV(M))$

$\exists x.M \wedge N \models\!\mid_{\mathcal{A}} (\exists x.M) \wedge N$ $\quad$ $\forall x.M \vee N \models\!\mid_{\mathcal{A}} (\forall x.M) \vee N$

$\exists x.M \vee N \models\!\mid_{\mathcal{A}} (\exists x.M) \vee N$ $\quad$ $\forall x.M \wedge N \models\!\mid_{\mathcal{A}} (\forall x.M) \wedge N$

Distribution

$\exists x.M \vee N \models\!\mid_{\mathcal{A}} (\exists x.M) \vee (\exists x.N)$ $\quad$ $\forall x.M \wedge N \models\!\mid_{\mathcal{A}} (\forall x.M) \wedge (\forall x.N)$

Vertauschung

$\exists x \exists y.M \models\!\mid_{\mathcal{A}} \exists y \exists x.M$ $\quad$ $\forall x \forall y.M \models\!\mid_{\mathcal{A}} \forall y \forall x.M$

$\forall x \exists y.M \models\!\mid_{\mathcal{A}} \exists z \forall x.M[y := zx]$ $\quad$ $\exists x \forall y.M \models\!\mid_{\mathcal{A}} \forall z \exists x.M[y := zx]$

$\exists x \forall y.M \models_{\mathcal{A}} \forall y \exists x.M$

Umbenennung $(y \notin FV(M))$

$\exists x.M \models\!\mid_{\mathcal{A}} \exists y.M[x := y]$ $\quad$ $\forall x.M \models\!\mid_{\mathcal{A}} \forall y.M[x := y]$

Die Gesetze gelten für jede Struktur $\mathcal{A}$, die die logischen Konstanten $\neg$, $\wedge$, $\vee$, $\forall$, und $\exists$ wie üblich interpretiert.

Abbildung 10.5: Quantorengesetze in syntaktischer Formulierung