Kapitel 10

Elementare Typtheorie

In der Typtheorie geht es um logische Sprachen, deren primäres Ausdrucksmittel Funktionen sind, und die mit einer Typdisziplin dafür sorgen, dass Funktionsapplikation sicher ist. Man bezeichnet typtheoretische Sprachen oft als *höherstufig* und meint damit, dass Funktionen Werte von Variablen, Argumente von Funktionen, und Ergebnisse von Funktionen sein können. Die Wurzeln der Typtheorie liegen in der mathematischen Logik [Russell 1903, Curry 1934, Church 1940, Henkin 1950]. Statt von Typtheorie spricht man auch von *getypten Lambda-Kalkülen*.

Im Folgenden betrachten wir ein einfaches typtheoretisches System, dass wir *Elementare Typtheorie*, oder kurz *ETT*, nennen. ETT ist eine didaktisch motivierte Variante des einfach typisierten Lambda-Kalküls.[1]

ETT ist ein parametrisiertes System. Zunächst können einige Mengen als Typen vorgegeben werden. Darüberhinaus 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. Eine einfache Typdisziplin schränkt die Bildung von Ausdrücken so ein, dass die vorkommenden Funktionsapplikationen sicher sind. Das ist schon alles.

Durch Vorgabe geeigneter Typen und Konstanten kann ETT zu einer Vielzahl von logischen Sprachen instantiiert werden. 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.

Wenn man eine logische Sprache mit ETT definiert, spart man viel Arbeit, da nur noch die wesentlichen Dinge definiert werden müssen, und gleichzeitig vie-

[1] Simpy-typed Lambda Calculus.

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

le wichtige Eigenschaften aus den allgemeinen Eigenschaften von ETT folgen. Man kann ETT als eine mathematische Abstraktion sehen, mit der syntaktische und semantische Eigenschaften von logischen Sprache formuliert und untersucht werden können.

Um 1965 wurde klar, dass getypte Lambda-Kalküle eine wichtige Grundlage für den Entwurf und die Analyse von Programmiersprachen bilden [Landin 1963, 1965, 1966; Strachey 1966]. Seitdem hat sich Typtheorie zu einem der aktivsten Forschungsgebiete der theoretischen Informatik entwickelt.

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.

Literaturverweise

Tytheorie ist ein weites Feld. Bei der Darstellung von ETT in diesem Kapitel kommt es mir vor allem darauf an, einen mathematischen Rahmen für die Syntax und denotationale Semantik einer großen Klasse von logischen Sprachen bereitzustellen. Merkwürdigerweise hat diese sehr lohnende Anwendung des einfach getypten Lambda-Kalküls bisher noch keinen Eingang in Lehrbücher gefunden.[3]

Bei der Zusammenstellung dieses Kapitels habe ich mich vor allem auf die folgende Literatur gestützt:

- John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.
- J. Roger Hindley, *Basic Simple Type Theory.* Cambridge University Press, 1997.
- Allen Stoughton, Substitution Revisited. Theoretical Computer Science 59:317-325, 1988.

Ein aktuelles und pragmatisch orientiertes Buch über programmiersprachliche Typtheorie ist:

[3] Die Idee, den einfach getypten Lambda-Kalkül als mathematischen Rahmen für die Syntax und denotationale Semantik von logischen Sprachen zu sehen, ist aber alt:

- Gérard Huet und Bernard Lang, Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31-55, 1978.
- Frank Pfenning und Conal Elliot, Higher-Order Abstract Syntax. ACM PLDI 1988.

- Benjamin Pierce, *Types and Programming Languages*, The MIT Press, 2002.

10.1 Funktionale Analyse 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

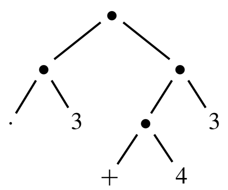
Wir beginnen mit der Analyse des arithmetischen Ausdrucks

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

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.

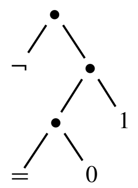
Diese Sichtweise geht auf Schönfinkel zurück (etwa 1920).

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

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

gebildet. Der Wert der Aussage ist die Zahl 1.

Aussagen sind also 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

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

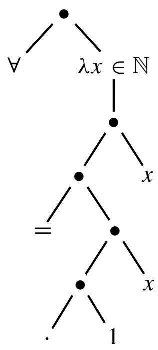
Den Allquantor stellen wir mit der Funktion

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

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

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

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.

Prädikate

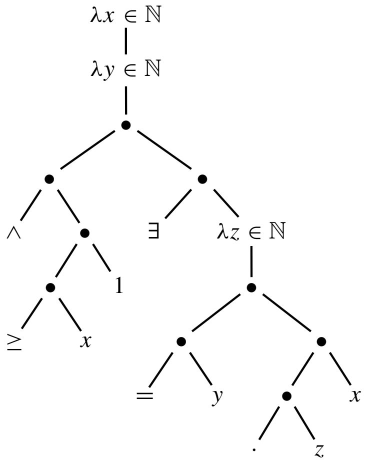
Prädikate sind parametrisierte Aussagen, die Funktionen darstellen, die Werte in $\mathbb{B}$ liefern. Als Beispiel betrachten wir das Prädikat

$x \in \mathbb{N}$ *ist Teiler von* $y \in \mathbb{N}$

Dieses Prädikat stellt die Funktion

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

dar. Diese hat den Typ $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$. Das Prädikat selbst können wir durch den folgenden Ausdruck beschreiben:



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 $\{\, x \in X \mid f\,x = 1 \,\}$ 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:

$$\forall x \in X\colon\ x \in Y \iff f\,x = 1$$

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:[4]

> *Sei* $P \subseteq \mathbb{N}$ *und* $\forall x \in \mathbb{N}\colon\ \{\, y \in \mathbb{N} \mid y < x \,\} \subseteq P \;\Rightarrow\; x \in P$.
> *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:

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

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 M eine Menge. Wenn wir eine Variable $x \in M$ quantifizieren wollen, können wir Lambda-Abstraktion und eine der folgenden Funktionen benutzen:

$$\begin{array}{ll} \forall_M \in (M \to \mathbb{B}) \to \mathbb{B} & \exists_M \in (M \to \mathbb{B}) \to \mathbb{B} \\ \forall_M(f) = \min\{\, fx \mid x \in M \,\} & \exists_M(f) = \max\{\, fx \mid x \in M \,\} \end{array}$$

Leibnizsche Charakterisierung der Gleichheit

Sei M eine Menge. Dann gilt:

$$\forall x, y \in M\colon\ \ x = y \iff \forall f \in M \to \mathbb{B}\colon\ fx = 1 \Rightarrow fy = 1$$

Die Richtung $\Rightarrow$ ist offensichtlich. Die andere Richtung folgt, wenn man für f die charakterische Funktion für $\{x\}$ wählt.

Die obige Äquivalenz bezeichnet man als *Leibnizsche Charakterisierung der Gleichheit*. Sie führt Gleichheit für eine beliebige Menge M mithilfe einer höherstufigen Quantifizierung auf Gleichheit für $\mathbb{B}$ zurück.

Lambda-Ausdrücke

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

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

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

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

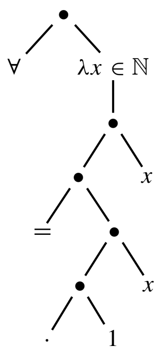


Abbildung 10.1: Drei Beschreibungen eines Lambda-Ausdrucks

- Konstanten (z. B. $0, +, =, \neg, \Rightarrow, \forall$).
- Funktionsapplikation.
- Lambda-Abstraktion und Variablen.

Wir bezeichnen solche Ausdrücke als *Lambda-Ausdrücke*.

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

1. Der Ausdruck selbst.
2. Der Wert des Ausdrucks.
3. Beschreibungen des Ausdrucks.

Abbildung 10.1 zeigt drei verschiedene Beschreibungen ein und desselben Lambda-Ausdrucks. Bei den zwei links stehenden Beschreibungen handelt es sich um *textuelle Beschreibungen*. Bei der rechtsstehenden Beschreibung handelt es sich um eine *graphische Beschreibung*, die wir als *Baumdarstellung* des Ausdrucks bezeichnen.

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)
```

`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`	$=$
`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

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

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

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. Der linke Baum beschreibt einen Ausdruck, den wir bereits bei der Diskussion von Prädikaten gesehen haben. Der rechte Baum beschreibt die de Bruijnsche Darstellung dieses Ausdrucks. Dabei werden die benutzenden Auftreten der gebundenen Variablen durch *de Bruijnsche Indizes* $\langle i\rangle$ mit $i \in \mathbb{N}$ beschrieben. 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.

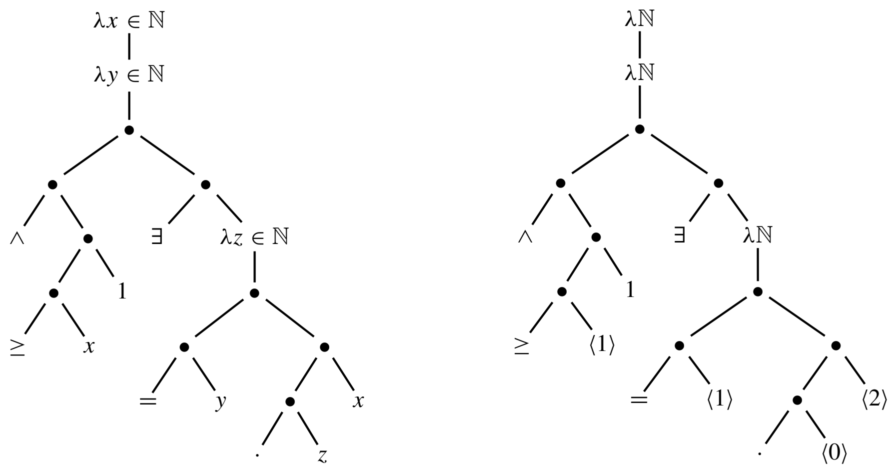


Abbildung 10.3: De Bruijnsche Darstellung eines Lambda-Ausdrucks

10.2 Signaturen und Terme

Unser nächstes Ziel ist Darstellung von Lambda-Ausdrücken durch geeignete mathematische Objekte. Bei der Bildung von Ausdrücken spielen die verfügbaren Konstanten (z. B. 0, $+$, $=$) eine wichtige Rolle. Die Aufgabe einer *Signatur* besteht darin, die verfügbaren Konstanten zusammen mit ihren Typen zu deklarieren. Die Standard ML Signatur in Abbildung 10.2 liefert uns ein gutes Beispiel. Sie deklariert zunächst zwei *Typkonstanten* `Nat` und `Bool`. Danach werden mehrere *Termkonstanten* (`null`, `one`, usw.) deklariert. Für jede Termkonstante wird ein Typ vereinbart.

Definition 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:

$b \in Tyc$	*Typkonstante*
$t \in Ty \;=\; b \;\mid\; t_1 \to t_2$	*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 $\mathbb{N}, \mathbb{B} \in \mathit{Tyc}$. 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 $b \in \mathit{Tyc}$ dargestellt werden ($n \geq 1$).

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:

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

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:

$$\begin{array}{rll} c \in \mathit{Tec} & & \textit{Termkonstante} \\ x \in \mathit{Var} & & \textit{Variable} \\ M, N \in \mathit{PT} & = c \mid x \mid MN \mid \lambda x.M & \textit{Präterm} \end{array}$$

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

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

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

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

$$Ter \overset{\text{def}}{=} Dom\ \tau$$

Proposition 10.2.1 *Die Typrelation τ ist eine Funktion $Ter \to Ty$.*

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

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

Für die Beschreibung von Termen vereinbaren wir die folgenden Abkürzungen:

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

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

$$FV \in Ter \to \mathcal{P}_{fin}(Var)$$

$$\begin{aligned}
FV(c) &= \emptyset \\
FV(x) &= \{x\} \\
FV(MN) &= FV(M) \cup FV(N) \\
FV(\lambda x.M) &= FV(M) - \{x\}
\end{aligned}$$

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 Variablen hat.
- ***geschlossen***, wenn er keine freien Variablen hat.

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

$$ord \in Ty \to \mathbb{N}$$

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

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

10.3 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 der Termkonstanten müssen wir darauf achten, dass sie mit den durch die Signatur festgelegten Typen verträglich sind.

Definition Eine ***Struktur für eine Signatur*** 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:

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

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

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

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

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

Eine Funktion $\sigma \in Var \to |\mathcal{A}|$ heißt ***Belegung***, wenn $\forall x \in Var\colon\ \sigma x \in \mathcal{A}(\tau x)$. 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 Tykonstanten mit *nichtleeren* Mengen interpretiert werden.[5]

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

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

Die definierende Gleichung für Applikationen ist zulässig, da

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

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

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

Proposition 10.3.1 (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. □

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

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:

$$M \models\!\!\!\!\dashv_{\mathcal{A}} N \quad \overset{\text{def}}{\iff} \quad M, N \in \mathit{Ter} \;\wedge\; \tau M = \tau N \;\wedge\; \mathcal{A}(M) = \mathcal{A}(N)$$

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

$$M \models\!\!\!\!\dashv N \quad \overset{\text{def}}{\iff} \quad \forall \text{ Struktur } \mathcal{A}\colon\; M \models\!\!\!\!\dashv_{\mathcal{A}} N$$

Wir vereinbaren die folgenden Sprechweisen:

$M \models\!\!\!\!\dashv_{\mathcal{A}} N$	M ***$\mathcal{A}$-äquivalent zu*** N
$M \models\!\!\!\!\dashv N$	M ***λ-äquivalent zu*** N

Proposition 10.3.2 *Für jede Struktur $\mathcal{A}$ gilt: $\models\!\!\!\!\dashv \;\subseteq\; \models\!\!\!\!\dashv_{\mathcal{A}}$.*

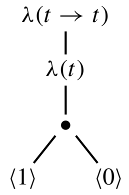
Proposition 10.3.3 (Kompatibilität) *Sei $\mathcal{A}$ eine Struktur und seien M und M' zwei Terme, sodass M' aus M erhalten werden kann, indem ein Auftreten des Teilterms N durch den Term N' ersetzt wird. Dann: $N \models\!\!\!\!\dashv_{\mathcal{A}} N' \Rightarrow M \models\!\!\!\!\dashv_{\mathcal{A}} M'$.*

Proposition 10.3.4 (Eta-Regel) *Für jeden Term $\lambda x.Mx$ mit $x \notin FV(M)$ gilt: $\lambda x.Mx \models\!\!\!\!\dashv M$.*

10.4 Variablenbindung und Substitution

Zwei Terme heißen *α-äquivalent*, wenn sie bis auf konsistenten Umbenennung von gebundenen Variablen gleich sind. Beispielsweise sind $\lambda x.x$ und $\lambda y.y$ α-äquivalent, wenn $\tau x = \tau y$ gilt; und die Terme $\lambda xy.xy$ und $\lambda uv.uv$ sind α-äquivalent, wenn $\tau x = \tau u$ und $\tau y = \tau v$ gilt.

Zwei Terme sind genau dann α-äquivalent, wenn sie die gleiche de Bruijnsche Darstellung haben. Das liegt daran, dass die de Bruijnsche Darstellung gebundene Variablen namensfrei darstellt. Beispielsweise hat der Term $\lambda xy.xy$ mit $\tau y = t$ die de Bruijnsche Darstellung



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.

Definition Ein Term heißt ***α-normal***, wenn für jeden seiner abstrahierenden Teilterme $\lambda(n,t).M$ gilt: $n = \min\{\, m \in \mathbb{N} \mid (m,t) \notin FV(\lambda(n,t).M) \,\}$.

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.

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

Eine Funktion $\theta \in Var \to Ter$ heißt ***Substitution***, wenn $\forall x \in Var\colon \tau x = \tau(\theta x)$. 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

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

Die Substitution [] heißt ***Identitätssubstitution***.

Notation Sei $f \in X \to Y$ und $(x, y) \in X \times Y$. Im Folgenden schreiben wir $f[x := y]$ für $f[y/x]$.

Wir definieren jetzt eine Funktion $\mathcal{C}$, die eine Substitution auf einen Term anwendet, indem sie alle Teilterme, die nur aus einer Variable bestehen, gemäß der Substitution ersetzt. Wir definieren $\mathcal{C}$ mit struktureller Rekursion:

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

Die definierende Gleichung für Applikationen ist zulässig, da

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

gilt. Diese Eigenschaft folgt parallel zur Definition mit struktureller Induktion. Wir bezeichnen C als ***Kontextoperator***.

Hier sind zwei Beispiele für eine Anwendung von $\mathcal{C}$:

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

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

Unser nächstes Ziel ist die Definition eines Substitutionsoperators $\mathcal{S}(M)\theta$, der die Bindungsstruktur des Terms M erhält. Wie das zweite Beispiel für $\mathcal{C}$ zeigt, muss $\mathcal{S}(M)\theta$ durch M gebundene Variablen umbenennen können, um gewährleisten zu können, dass keine der durch θ importierte Variablen durch einen Binder in M gekapert wird. Wir werden $\mathcal{S}(M)\theta$ so definieren, dass alle durch M gebundenen Variablen mit kleinstmöglichen Namen versehen werden. Insbesondere wird $\mathcal{S}(M)[]$ die α-normale Variante von M liefern.

Wir definieren

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

und nennen $cv(V, \theta, t)$ die ***kanonische Variable für*** (V, θ, t).

Den ***Substitutionsoperator*** $\mathcal{S}$ definieren wir mit strukturelle Rekursion wie folgt:

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

Die definierende Gleichung für Applikationen MN ist zulässig, da

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

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

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

Proposition 10.4.1 *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]. □

Auf der Menge der Terme definieren wir die folgende Äquivalenzrelation:

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

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

Proposition 10.4.2 *Sei $\lambda x.M$ ein Term und y eine Variable, die denselben Typ wie x hat und nicht frei in $\lambda x.M$ vorkommt. Dann $\lambda x.M \sim_\alpha \lambda y.(M[x := y])$.*

Lemma 10.4.3 (Substitution) *Sei $\mathcal{A}$ eine Struktur, M ein Term, θ eine Substitution, und $\sigma \in \Sigma_{\mathcal{A}}$ eine Belegung. Dann $\mathcal{A}(M\theta)\sigma = \mathcal{A}(M)(\lambda x \in Var.\ \mathcal{A}(\theta x)\sigma)$.*

Proposition 10.4.4 (Alpha-Regel) *Für alle Terme M, N gilt:*

$$M \sim_\alpha N \ \Rightarrow\ M \models\!\!\!\dashv N$$

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

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

□

Proposition 10.4.5 (Beta-Regel) *Für jeden Term* $(\lambda x.M)N$ *gilt:*

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

Beweis Folgt aus dem Substitutionslemma. □

Proposition 10.4.6 (Stabilität) *Für jede Struktur* $\mathcal{A}$ *gilt:*

$$\forall M, N \in \mathit{Ter}\ \ \forall \theta \in \mathit{Sub}: \quad M \models\!\!\!\dashv_{\mathcal{A}} N \ \Rightarrow\ M\theta \models\!\!\!\dashv_{\mathcal{A}} N\theta$$

Beweis Folgt aus dem Substitutionslemma. □

10.5 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 Propositionen 10.4.5, 10.3.4 und 10.3.3 garantieren, dass Vereinfachung mit den Regeln β und η denotationserhaltend ist.

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.

- λ-***normal***, wenn er α-, β- und η-normal ist.

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

Jeder β-normale Term, der keine Abstraktion ist, kann eindeutig als $M_0 \dots M_n$ mit M_0 atomar dargestellt werden ($n \geq 0$).

Satz 10.5.1 *Zwei λ-normale Terme sind genau dann λ-äquivalent, wenn sie gleich sind.*

Beweis Schwer. Siehe [Mitchell]. □

Korollar 10.5.2 *Ein Term hat höchstens eine λ-Normalform.*

Eine binäre Relation R auf der Menge der Terme heißt ***rein***, wenn für all $(M, N) \in R$ gilt: $\tau M = \tau N$. Für jede reine Relation R definieren wir eine binäre Relation $PE(R)$ auf der Menge der Terme:

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

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

Offensichtlich ist $PE(R)$ rein und erfüllt $R \subseteq PE(R)$. Die ***kompatible Erweiterung*** $CE(R)$ einer reinen Relation R definieren wir wie folgt:

$$CE(R) \stackrel{\text{def}}{=} \sim_\alpha \circ PE(R) \circ \sim_\alpha$$

Offensichtlich gilt $R \subseteq CE(R)$.

Proposition 10.5.3 *Wenn $R \subseteq \models\!\dashv_{\mathcal{A}}$, dann $CE(R) \subseteq \models\!\dashv_{\mathcal{A}}$.*

Wir definieren die folgenden binären Relationen auf der Menge der Terme:

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

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

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

Proposition 10.5.4 $\to_\lambda \ \subseteq\ \models\!\dashv$.

Proposition 10.5.5 *Ein Term M ist genau dann λ-normal, wenn es keinen Term M' gibt mit $M \to_\lambda M'$.*

Satz 10.5.6 (Strenge Normalisierung) *Die Relation $\to_\lambda$ ist terminierend.*

Beweis Schwer. Siehe [Mitchell]. □

Satz 10.5.7

1. *Zu jedem Term existiert genau eine λ-Normalform.*
2. *Zwei Terme sind genau dann λ-äquivalent, wenn sie die gleiche λ-Normalform haben.*

Beweis Folgt aus Satz 10.5.1 und 10.5.6. □