Kapitel 1

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, 1963] (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 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.
- John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

© G. Smolka, 18. 5. 2004

- 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.

1.1 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

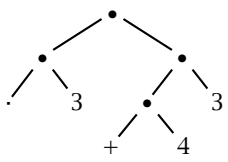
Wir beginnen mit der Analyse des arithmetischen Ausdrucks

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

Dieser Ausdruck ist mit vier **Wertkonstanten** 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 der Wertkonstanten ist zusammen mit ihrem **Typ** angegeben. Die Konstanten + und · haben einen **funktionalen Typ**, der mit der **Typkonstante** $\mathbb{N}$ gebildet ist. Der obige Ausdruck wird dadurch gebildet, dass die angegeben Wertkonstanten 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 Wertkonstanten erscheinen an den Blättern des Baums. Wenn man die Typen der Wertkonstanten entsprechend der Funktionsapplikationen kombiniert, bekommt man den **Typ des Ausdrucks** (die Typkonstante $\mathbb{N}$). 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.

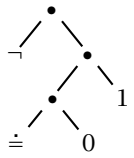
Diese Sichtweise geht auf Schönfinkel [1924] zurück.

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



mit den Wertkonstanten

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

gebildet. Der Typ der Aussage ist $\mathbb{B}$, ihr Wert ist die Zahl 1.

Unter einer **Aussage** wollen wir im Folgenden einen Ausdruck des Typs $\mathbb{B}$ verstehen. Wenn eine Aussage den Wert 1 hat, bezeichnen wir sie als **gültig**. Unter einem **Prädikat** wollen wir eine funktionale Konstante mit dem Ergebnistyp $\mathbb{B}$ verstehen, bei der mindestens ein Argumenttyp verschieden von $\mathbb{B}$ ist (z.B. $\doteq$).

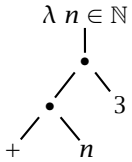
Lambda-Abstraktion

Als Notation für die Beschreibung von Funktionen verwenden wir sogenannte Lambda-Abstraktionen. Als Beispiel betrachten wir den Ausdruck

$$\lambda\, n \in \mathbb{N}.\; n + 3$$

Dieser Ausdruck hat den funktionalen Typ $\mathbb{N} \to \mathbb{N}$. Sein Wert ist die Funktion, die zu $n \in \mathbb{N}$ die Zahl $n + 3$ liefert. Die Struktur des Ausdrucks können wir graphisch wie folgt darstellen:

© G. Smolka, 18. 5. 2004



Der Ausdruck ist mit den Wertkonstanten + und 3, der Typkonstanten $\mathbb{N}$ sowie der **Variablen** *n* mittels Funktionsapplikation und Lambda-Abstraktion gebildet. Die Variable *n* wird im **Kopf der Lambda-Abstraktion** (alles vor dem Punkt) mit dem Typ $\mathbb{N}$ **deklariert** und im **Rumpf der Lambda-Abstraktion** (alles nach dem Punkt) **benutzt**. Man spricht von einer **gebundenen Variable** des Ausdrucks und meint damit, dass sie nur innerhalb des Ausdrucks gültig ist.

Lambda-Abstraktionen wurden von Church [1934] eingeführt.

Aussagen mit Quantoren

Auch Aussagen mit Quantoren lassen sich als Ausdrücke darstellen. Quantifizierung wird dazu auf Lambda-Abstraktion zurückgeführt. Als Beispiel betrachten wir die Aussage

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

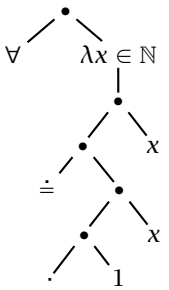
Den Allquantor stellen wir mit der Funktion

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

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

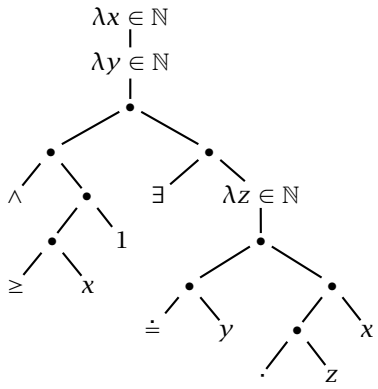
Der Ausdruck

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

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 als Funktionen

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

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

beschreiben. Umgekehrt können wir jede Funktion $f \in X \to \mathbb{B}$ eindeutig durch die Menge $\{ x \in X \mid f x = 1 \}$ beschreiben. Man sagt, dass die Teilmengen von X isomorph zu den Booleschen Funktionen über X sind und schreibt

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

© G. Smolka, 18. 5. 2004

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:[2]

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

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:

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

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

1.2 Lambda-Ausdrücke

Wir haben jetzt alle Beschreibungskonstrukte für Ausdrücke kennen gelernt, die wir im Folgenden betrachten wollen: Typkonstanten, funktionale Typen, Wertkonstanten, Variablen, Funktionsapplikation und Lambda-Abstraktion. Diese Beschreibungsmittel sind uns aus der Programmiersprache Standard ML wohl bekannt. Oft ist es hilfreich, sich die durch Lambda-Abstraktionen beschriebenen Funktionen als Prozeduren vorzustellen.

Wir haben es also mit einer außerordentlich einfachen Sprache zu tun, deren Beschreibungsmittel jedem Programmierer vertraut sind. Wir haben bereits gesehen, dass wir mit dieser sehr einfachen Sprache dennoch sehr komplexe mathematische Aussagen beschreiben können. Dafür gibt es zwei Gründe. Zunächst kann fehlende Ausdruckskraft durch die Wahl geeigneter Konstanten importiert werden. Weiter kann es sich bei diesen Konstanten um Funktionen handeln, deren Argumente und Ergebnisse Funktionen sein können (so genannte **höherstufige Funktionen**).

Ausdrücke, die mit den gerade erwähnten Primitiven gebildet werden können, bezeichnen wir als **Lambda-Ausdrücke**. Schematisch können wir die durch Lambda-Ausdrücke gegebene Sprache wie folgt darstellen:

b	*Typkonstante*
$t ::= b \mid t \to t$	*Typ*
c	*Wertkonstante*
x	*Variable*
$M ::= c \mid x \mid MM \mid \lambda x \in t.\, M$	*Lambda-Ausdruck*

Beschreibung von Lambda-Ausdrücken

Abbildung 1.1 zeigt drei verschiedene Beschreibungen ein und desselben Lambda-Ausdrucks. Bei den zwei linken Beschreibungen handelt es sich um **textuelle Darstellungen**, bei der Beschreibung rechts um eine so genannte **Baumdarstellung**.

Oft ist es bequem, statt $\lambda x \in t.\, M$ kürzer $\lambda x.M$ zu schreiben. Das Weglassen der Typangabe t für die gebundene Variable x ist legitim, wenn t aus dem Kontext erschlossen werden kann. Bei manchen Beispielen lassen wir die Typangaben auch weg, da ihre genaue Form für das Beispiel keine Rolle spielt.

Für die Beschreibung von Lambda-Ausdrücken vereinbaren wir die folgenden Abkürzungen:

$$\lambda x_1 x_2 \cdots x_n.M \quad \rightsquigarrow \quad \lambda x_1.\lambda x_2.\cdots\lambda x_n.M \qquad (n \geq 2)$$
$$M_1 M_2 M_3 \cdots M_n \quad \rightsquigarrow \quad (\cdots((M_1 M_2) M_3)\cdots) M_n \qquad (n \geq 3)$$

© G. Smolka, 18. 5. 2004

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

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

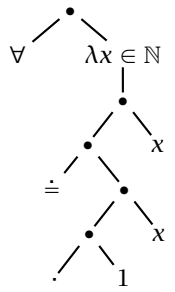


Abbildung 1.1: Drei Beschreibungen eines Lambda-Ausdrucks

Außerdem vereinbaren wir, dass der mit dem Punkt beginnende Teilausdruck eines Ausdrucks $\lambda x.M$ so weit wie möglich nach rechts ausgedehnt wird. Also beschreibt $\lambda x.MN$ den Ausdruck $\lambda x.(MN)$ und nicht den Ausdruck $(\lambda x.M)N$.

Schließlich vereinbaren wir noch eine Klammersparregel für funktionale Typen:

$$t_1 \to t_2 \to \cdots \to t_n \quad \rightsquigarrow \quad t_1 \to (t_2 \to (\cdots t_n)) \qquad\qquad (n \geq 3)$$

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 1.2. 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.

Substitution

Wir unterscheiden zwischen *definierenden* und *benutzenden Variablenauftreten*. Bei benutzenden Variablenauftreten unterscheiden wir wiederum zwischen *freien* und *gebundenen Variablenauftreten*. Der Ausdruck $(\lambda\, x \in \mathbb{N}.\ fx)x$ enthält vier Variablenauftreten:

- Das erste Auftreten von x ist definierend.
- Das zweite Auftreten von x ist benutzend und gebunden.

```
type Nat                                          ℕ
type Bool                                         𝔹
val  null : Nat                                   0
val  one  : Nat                                   1
val  add  : Nat -> Nat -> Nat                     +
val  mul  : Nat -> Nat -> Nat                     ·
val  eq   : Nat -> Nat -> Bool                    ≐ℕ
val  not  : Bool -> Bool                          ¬
val  and  : Bool -> Bool -> Bool                  ∧
val  all  : (Nat -> Bool) -> Bool                 ∀ℕ
val  all' : ((Nat -> Bool) -> Bool) -> Bool        ∀ℕ→𝔹
```

Abbildung 1.2: Eine Standard ML Signatur

- Das dritte Auftreten von x ist benutzend und frei.
- Das Auftreten von f ist benutzend und frei.

Betrachten Sie die Aussagen

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

Die zwei Aussagen unterscheiden sich nur in der Wahl der *gebundenen Variablen*. Ob die gebundene Variable mit x oder y realisiert wird, spielt für die Bedeutung der Aussage keine Rolle.

Seien M und N lambda-Ausdrücke und x eine Variable. Mit $M[x := N]$ bezeichnen wir den Term, den man aus M erhält, wenn man jedes freie Auftreten der Variable x durch den Ausdruck N ersetzt. Die hinter dieser Notation stehende Operation wird als *Substitution* bezeichnet. Hier sind Beispiele für Substitutionen:

$$\begin{aligned}(x + y)[y := x + y] &= x + (x + y)\\ ((\lambda x.x)x)[x := 5] &= (\lambda x.x)5\end{aligned}$$

Bei der Ausführung einer Substitution $M[x := N]$ ist zu beachten, dass die freien Variablen von N beim Einsetzen in M nicht durch übergeordnete Lambdas in M **gekapert** werden dürfen. Um unerlaubtes Kapern zu verhindern, kann es notwendig sein, gebundene Variablen umzubenennen:

$$(\lambda x.(yx)z)[z := x] = \lambda x'.yx'x$$

De Bruijnsche Variablendarstellung

Eine Variable ist ein Name für einen Wert. Die *de Bruijnschen Darstellung* von Ausdrücken stellt Variablen grundsätzlich durch natürliche Zahlen dar. Durch

© G. Smolka, 18. 5. 2004

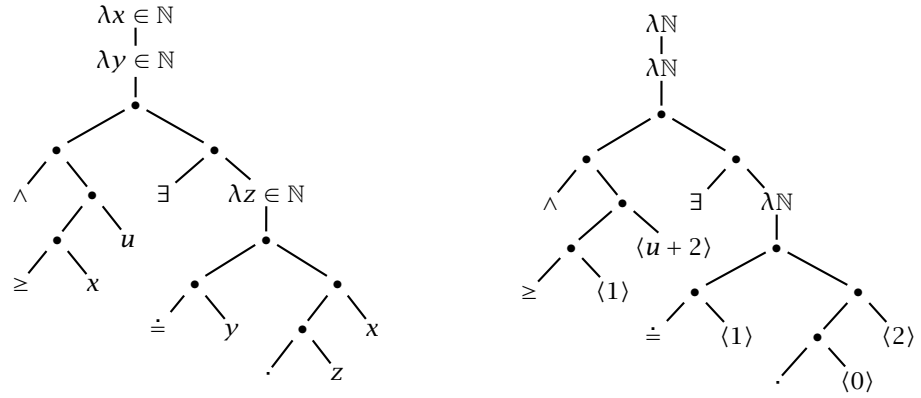


Abbildung 1.3: Normale und De Bruijnsche Variablendarstellung

ein relatives Adressierungsschema werden gebundene Variablen ohne Namen realisiert.

Wir erklären die de Bruijnschen Darstellung mit dem Beispiel in Abbildung 1.3. Links erscheint die normale Baumdarstellung des Ausdrucks, rechts erscheint die de Bruijnsche Darstellung des Ausdrucks. Bei der de Bruijnsche Darstellung werden definierende Variablenauftreten nicht dargestellt, und benutzende Auftreten werden durch so genannte *de Bruijnsche Indizes* $\langle x \rangle$ mit $x \in \mathbb{N}$ dargestellt. Sei l die Zahl der einem Index $\langle x \rangle$ übergeordneten Lambda-Knoten. Wenn $x \geq l$ gilt, stellt der Index ein freies Auftrteten der Variablen $x - l$ dar. Ansonsten stellt der Index ein benutzendes Variablenauftreten dar, das durch denjenigen Lambda-Knoten gebunden ist, den man auf dem Pfad zur Wurzel des Baums nach Überspringen von x Lambda-Knoten erreicht.

1.3 Logische Konstanten

Die Konstanten für die Booleschen Verknüpfungen ($\wedge$, $\vee$, $\Longrightarrow$, $\neg$ usw.), die Gleichheitsprädikate ($\doteq_X$) und die Quantoren ($\exists_X$, $\forall_X$) werden als logische Konstanten bezeichnet. Erwartungsgemäß ist es nicht notwendig, alle logischen Konstanten zur Verfügung zu stellen, da sich logische Konstanten oft mithilfe anderer logischer Konstanten darstellen lassen.

Leibnizsche Charakterisierung der Gleichheit

Sei X eine Menge. Das **Gleichheitsprädikat** für X ist wie folgt definiert:

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

Dann gilt für alle $x, y \in X$:

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

Für $x = y$ liefern beide Seiten der Gleichung 1. 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.

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. Wo immer 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

© G. Smolka, 18. 5. 2004

gemäß der obigen Gleichungen darstellen. Diese Beobachtung findet sich in der Arbeit [Henkin 1963].

Lambda-Elimination

Schönfinkel [1924] hat entdeckt, dass man auf alle Lambda-Abstraktionen verzichten kann, wenn man bestimmte höherstufige Funktionen zur Verfügung hat.[3]

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:

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

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

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-Abstraktionen ohne Rückgriff auf andere Primitive beschrieben werden:

$$\begin{aligned}
I &= \lambda x \in X.\, x \\
K &= \lambda y \in Y.\, \lambda x \in X.\, y \\
S &= \lambda f \in X \to Y \to Z.\, \lambda g \in X \to Y.\, \lambda x \in X.\, fx(gx)
\end{aligned}$$

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

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

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

Aufgabe 1.1 Geben Sie mögliche Typen für die Auftreten von S und K in SKK an. ■

1.4 Lambda-Reduktion

Für Lambda-Ausdrücke gilt die folgende, als β-**Regel** bezeichnete Gleichung:

$$(\beta) \quad (\lambda x.M)N \;=\; M[x := N]$$

Durch Anwendung der β-Regel können wir Lambda-Ausdrücke vereinfachen. Hier ist ein Beispiel:

$$(\lambda x \in \mathbb{N}.\, x + 3)7 \;=\; 7 + 3$$

Sei $f \in X \to Y$. Dann gilt

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

Da diese Gleichung für alle Werte der Variable f gilt, können wir für f einen beliebigen Lambda-Ausdruck M einsetzen, in dem x nicht frei vorkommt:

$$(\eta) \quad (\lambda x.Mx) \;=\; M \quad \text{falls } x \text{ nicht frei in } M$$

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

Mithilfe der β- und η-Regel kann man Lambda-Ausdrücke vereinfachen. Die Regeln werden dabei von links nach rechts angewendet. Hier sind zwei Beispiele:

$$\begin{aligned} (\lambda x.fxx)(\lambda y.gy) &= (\lambda x.fxx)g && \eta\text{-Regel} \\ &= fgg && \beta\text{-Regel} \end{aligned}$$

$$\begin{aligned} \lambda x.(\lambda fyg.fgy)(\lambda h.h)xh &= \lambda x.(\lambda yg.(\lambda h.h)gy)xh && \beta\text{-Regel} \\ &= \lambda x.(\lambda yg.gy)xh && \beta\text{-Regel} \\ &= \lambda x.(\lambda g.gx)h && \beta\text{-Regel} \\ &= \lambda x.hx && \beta\text{-Regel} \\ &= h && \eta\text{-Regel} \end{aligned}$$

Die Vereinfachung von Lambda-Ausdrücken gemäß der β- und der η-Regel wird als **Lambda-Reduktion** bezeichnet. Sie hat zwei wichtige Eigenschaften:

© G. Smolka, 18. 5. 2004

(a) Ein wohlgeformter Lambda-Ausdruck kann nur endlich oft mit der β- und η-Regel vereinfacht werden.

(b) Der Ausdruck, den man durch vollständige Vereinfachung eines Terms M gemäß der β- und η-Regel erhält, ist eindeutig bestimmt und wird als λ-**Normalfom** von M bezeichnet.

Der Beweis dieser Eigenschaften ist schwer.

Wir führen noch einige Begriffe ein. Ein Lambda-Ausdruck 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 $\lambda x.Mx$ hat und x nicht frei in M auftritt.
- η-**normal**, wenn keinen η-Redex enthält.
- λ-**normal**, wenn er β- und η-normal ist.

1.5 Syntax

Wir beginnen jetzt mit der formalen Definition der typtheoretischen Sprache ETT. Zunächst definieren wir die abstrakte Syntax von ETT. Dabei handelt es sich um eine Datenstruktur, mit der wir Lambda-Ausdrücke als mathematische Objekte darstellen.

Wie bei Programmiersprachen unterscheiden wir zwischen **konkreter Syntax** und **abstrakter Syntax**. Die konkrete Syntax legt textuelle Darstellungen für die Objekte der abstrakten Syntax fest. Solche textuellen Darstellungen bezeichnen wir als **Notationen**. Dagegen bezeichen wir die Objekte der abstrakten Syntax als **syntaktische Objekte**.

Wir werden die abstrakte Syntax von ETT formal definieren. Dagegen lassen wir viele Details einer möglichen konkreten Syntax für ETT offen. Mithilfe von Beispielen führen wir jedoch einige Notationen ein, mit denen wir die syntaktischen Objekte von ETT bequem beschreiben können.

Zunächst klären wir, was wir unter Konstanten und Variablen verstehen wollen. Da es uns um die abstrakte Syntax geht, handelt es sich dabei jedenfalls nicht um Symbole. Andererseits ist klar, dass es sich bei Konstanten und Variablen um Namen für Objekte handelt. Wie diese Namen genau funktionieren, wird durch den jeweiligen Kontext geregelt, nicht durch die Namen selbst. Deswegen können wir Namen durch beliebige mathematische Objekte darstellen. Bei Konstanten legen wir uns auf keine bestimmten Objekte fest, bei Variablen wählen wir die natürlichen Zahlen.

1.5.1 Konstanten und Typen

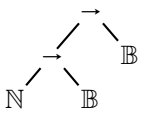
Typen und Ausdrücke werden gemäß einer Signatur gebildet, die die verfügbaren Typ- und Wertkonstanten deklariert. Dieses Vorgehen kennen Sie aus Standard ML. Die Standard ML Signatur in Abbildung 1.2 deklariert zunächst zwei *Typkonstanten Nat* und *Bool*. Danach deklariert sie mehrere *Wertkonstanten* (*null*, *one*, usw.). Für jede Wertkonstante wird ein Typ vereinbart. Die Typen der Wertkonstanten werden aus den zuvor definierten Typkonstanten und dem Funktionspfeil gebildet.

Sei eine Menge *TC* gegeben, deren Elemente wir als Typkonstanten verwenden. Aufbauend auf *TC* können wir die Menge der Typen wie folgt definieren:

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

Bei Typen handelt es sich um binäre Bäume, deren Blätter Typkonstanten und deren innere Knoten funktionale Typen darstellen.

Beispiel 1.5.1 Seien die Mengen $\mathbb{N}$ und $\mathbb{B}$ Typkonstanten. Dann können wir den Typ $(\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$ bilden. Graphisch können wir diesen Typ wie folgt darstellen:



Gemäß unserer Definition ist dieser Typ das mathematische Objekt

$$(2,\ ((2,\ ((1,\mathbb{N}),\ (1,\mathbb{B}))),\ (1,\mathbb{B})))$$

Dieser Darstellung von Typen entspricht die folgende Typdeklaration in Standard ML:

```
datatype ty = TC of tyc | TA of ty*ty
```

Der Konstruktor *TC* bildet Paare mit der Variantenummer 1, der Konstruktor *TA* Paare mit der Variantenummer 2. ■

Wir haben jetzt alles beisammen, um Signaturen so zu definieren, wie wir sie für ETT benötigen.

Definition 1.5.1 Eine **Signatur** ist ein Tripel (TC, VC, ty) wie folgt:

- *TC* und *VC* sind Mengen. Die Elemente von *TC* werden als **Typkonstanten** bezeichnet, und die Elemente von *VC* als **Wertkonstanten**.
- Es gibt mindestens eine Typkonstante.

© G. Smolka, 18. 5. 2004

- Die Menge Ty der **Typen** ist wie folgt definiert:

$$\begin{array}{ll} b \in TC & \textit{Typkonstante} \\ t \in Ty \;=\; b \;\mid\; t_1 \to t_2 & \textit{Typ} \end{array}$$

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

Typen der Bauart b heißen **atomar** und Typen der Bauart $t_1 \to t_2$ heißen **funktional**. Die **Stufe** $ord(t)$ eines Typs t ist wie folgt definiert:

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

Die **Stufe einer Wertkonstanten** c ist $ord(ty\, c)$. ■

Notationelles

Proposition 1.5.1 Jeder funktionale Typ kann eindeutig als $t_1 \to \cdots \to t_n \to b$ mit $n \geq 1$ und $b \in TC$ dargestellt werden.

Proposition 1.5.2 $ord(t_1 \to \cdots \to t_n \to b) = 1 + \max\{-1,\ ord\ t_1\ , \ldots,\ ord\ t_n\}$.

Wenn wir eine Signatur mit den Typen $\mathbb{B}$ und $\mathbb{N}$ und den Konstanten 0 und 1 betrachten, stellt sich die Frage, welchen Typ wir den Konstanten 0 und 1 geben sollen. Da nur ein Typ möglich ist, müssen wir uns entweder für $\mathbb{B}$ oder $\mathbb{N}$ entscheiden. Meistens geben wir 0 und 1 den Typ $\mathbb{N}$ und führen für $\mathbb{B}$ zwei zusätzliche Konstanten ein, die wir mit **F** und **T** bezeichnen. Durch welche mathematischen Objekte wir **F** und **T** darstellen, spielt nur insofern eine Rolle, dass diese Objekte von den restlichen Wertkonstanten verschieden sein müssen.

Die obige Diskussion bezieht sich auf die abstrakte Syntax. Für die konkrete Syntax haben wir die Möglichkeit, die Symbole 0 und 1 zu **überladen**. Das bedeutet, dass sie je nach Kontext entweder die (abstrakten) Konstanten für $\mathbb{N}$ oder $\mathbb{B}$ bezeichnen.

Das Konzept der Überladung findet man in den meisten Programmiersprachen. Beispielsweise bezeichnet das Symbol + in Standard ML je nach Kontext entweder die Additionsoperation für Festkommazahlen oder die Additionsoperation für Gleitkommazahlen.

1.5.2 Variablen und Präterme

Wir definieren jetzt die Ausdrücke, die mit den Konstanten einer Signatur gebildet werden können. Dafür benötigen wir zwei Schritte. Zuerst definieren wir die

Menge der *Präterme*. Dabei handelt es sich um alle Ausdrücke, die mit den Konstanten der Signatur gebildet werden können, ohne die durch die Typdisziplin auferlegten Einschränkungen zu beachten. Aus der Menge der Präterme filtern wir im zweiten Schritt die Menge der *wohlgetypten* Präterme heraus, die wir als *Terme* bezeichnen.

Variablen stellen wir durch natürliche Zahlen dar. Gebunden Variablen stellen wir gemäß de Bruijn durch Offsets dar.

Definition 1.5.2 Sei eine Signatur gegeben. Die Menge der **Variablen** und die Menge der **Präterme** definieren wir wie folgt:

$$
\begin{array}{rr}
t \in Ty & Ty \\
c \in VC & \textit{Wertkonstante} \\
x \in Var = \mathbb{N} & \textit{Variable} \\
M, N \in PT = c \mid x \mid MN \mid \lambda t.M & \textit{Präterm}
\end{array}
$$

Die **freien Variablen** eines Präterms definieren wir wie folgt:

$$
\begin{aligned}
&FV \in Ter \to \mathcal{P}_{fin}(Var) \\
&FV(c) = \emptyset \\
&FV(x) = \{x\} \\
&FV(MN) = FV(M) \cup FV(N) \\
&FV(\lambda t.M) = \{\, n-1 \mid n \in FV(M) - \{0\} \,\}
\end{aligned}
$$

■

Beispiel 1.5.2 Die freien Variablen des Präterms $(\lambda t.c\langle 0\rangle\langle 2\rangle)\langle 3\rangle$ sind 1 und 3. ■

Ein Präterm heißt

- **Applikation**, wenn er die Form MN hat.
- **Abstraktion**, wenn er die Form $\lambda t.M$ 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.

1.5.3 Typumgebungen und Terme

Eine **Typumgebung** versieht Variablen mit Typen:

$$\Gamma \in Var \rightharpoonup Ty \qquad \textit{Typumgebung}$$

© G. Smolka, 18. 5. 2004

Sei t eine Typ und Γ eine Typumgebung. Dann bezeichnen wir mit $t :: \Gamma$ (lies „t cons Γ“) die wie folgt definierte Typumgebung:

$$t :: \Gamma \stackrel{\text{def}}{=} \{(0,t)\} \cup \{\, (n+1,t') \mid (n,t') \in \Gamma \,\}$$

Eine Typumgebung Γ mit $Dom\ \Gamma = \{1,\ldots,n\}$ kann durch die Liste $[\Gamma 0,\ldots,\Gamma n]$ dargestellt werden. Für solche Typumgebungen entspricht die Cons-Operation der Cons-Operation für Listen.

Beispiel 1.5.3 $t_0 :: \{(0,t_1),\ (3,t_2)\} = \{(0,t_0),\ (1,t_1),\ (4,t_2)\}$. ■

Als Nächstes definieren wir eine als **Typrelation** bezeichnete Relation „$\Gamma \vdash M : t$“, die genau dann gilt, wenn der Präterm M in der Typumgebung Γ wohlgetypt ist und den Typ t hat:

$$\frac{}{\Gamma \vdash c : ty\, c} \qquad \frac{(x,t) \in \Gamma}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash M : t \to t' \qquad \Gamma \vdash N : t}{\Gamma \vdash MN : t'} \qquad \frac{t :: \Gamma \vdash M : t'}{\Gamma \vdash \lambda t.M : t \to t'}$$

Ein Präterm M heißt **wohlgetypt** in einer Umgebung Γ, wenn ein Typ t mit $\Gamma \vdash M : t$ existiert. Wenn für Γ und M ein Typ t mit $\Gamma \vdash M : t$ existiert, dann ist er eindeutig bestimmt und wird als der **Typ von** M **in** Γ bezeichnet.

Sei Γ eine Typumgebung. Unter einem Γ-**Term** verstehen wir einen in Γ wohlgetypten Präterm. Die Menge aller Γ-Terme bezeichnen wir mit

$$Ter(\Gamma) \stackrel{\text{def}}{=} \{\, M \in PT \mid \exists t \in Ty\colon\ \Gamma \vdash M : t \,\}$$

Notationelles

Wenn eine Signatur und eine Typumgebung gegeben sind, sprechen wir kurz von Termen, ohne die Typumgebung zu erwähnen. Weiter sprechen wir kurz vom **Typ eines Terms** und bezeichnen ihn mit τM. Die Menge aller Präterme, die in der gegebenen Typumgebung wohlgetypt sind, bezeichnen wir mit *Ter*. Die **Stufe eines Terms** M ist $ord(\tau M)$.

1.5.4 Substitution

Die Definition der Substitutionsoperation erfordert Sorgfalt. Wir beginnen mit einigen Beispielen:

$$\begin{aligned}
(\lambda t.\langle 3\rangle\langle 0\rangle)\,[2{:=}\langle 7\rangle] &= \lambda t.\langle 8\rangle\langle 0\rangle\\
(\lambda t.\langle 1\rangle\langle 0\rangle)\,[0{:=}\langle 1\rangle] &= \lambda t.\langle 2\rangle\langle 0\rangle\\
(\lambda t.\langle 1\rangle\langle 0\rangle)\,[1{:=}\langle 7\rangle] &= \lambda t.\langle 1\rangle\langle 0\rangle\\
(\lambda t.\langle 1\rangle\langle 0\rangle)\,[0{:=}\lambda t.\langle 1\rangle\langle 0\rangle] &= \lambda t.(\lambda t.\langle 2\rangle\langle 0\rangle)\langle 0\rangle
\end{aligned}$$

Bei einer Substitution $M[x:= N]$ müssen wir darauf achten, dass die freien Variablen von N beim Einsetzen in M gemäß der Anzahl der übergeordneten Lambdas angehoben werden. Dazu definieren wir eine Operation $up\ l\ M$, die die freien Variablen in M, die größer gleich l sind, um 1 anhebt:

$$
\begin{aligned}
&up \in \mathbb{N} \to PT \to PT\\
&up\ l\ c \;=\; c\\
&up\ l\ x \;=\; if\ x \geq l\ then\ x+1\ else\ x\\
&up\ l\ (MN) \;=\; (up\ l\ M)(up\ l\ N)\\
&up\ l\ (\lambda t.M) \;=\; \lambda t.\, up\ (l+1)\ M
\end{aligned}
$$

Wir vereinbaren die Kurzschreibweise

$$\uparrow M \stackrel{\text{def}}{=} up\ 0\ M$$

und definieren die Substitutionsoperation wie folgt:

$$
\begin{aligned}
&sub\colon PT \to Var \to PT \to PT\\
&sub\ c\ y\ N \;=\; c\\
&sub\ x\ y\ N \;=\; if\ x = y\ then\ N\ else\ x\\
&sub\ (MM')\ y\ N \;=\; (sub\ M\ y\ N)(sub\ M'\ y\ N)\\
&sub\ (\lambda t.M)\ y\ N \;=\; \lambda t.\, sub\ M\ (y+1)\ (\uparrow N)
\end{aligned}
$$

Für die Substitutionsoperation vereinbaren wir die Kurzschreibweise

$$M[x:= N] \stackrel{\text{def}}{=} sub\ M\ x\ N$$

Die folgende Propositione besagt, dass die Substitutionsoperation mit der Typdisziplin verträglich ist:

Proposition 1.5.3 Sei $\Gamma \vdash M : t,\ \Gamma \vdash N : t'$ und $\Gamma x = t'$. Dann $\Gamma \vdash M[x:= N] : t$.

1.5.5 Notationelles

Sei eine Typumgebung Γ gegeben. Die folgende Notation für Präterme entspricht der üblichen Notation für Lambda-Ausdrücke:

$$\lambda x.M \stackrel{\text{def}}{=} \lambda(\Gamma x).\,(\uparrow M)[x+1:= \langle 0\rangle]$$

Beispiel 1.5.4 Sei $x = 0$, $y = 1$, $f = 2$, $\Gamma x = \mathbb{N}$, $\Gamma y = \mathbb{B}$ und $\Gamma f = \mathbb{N} \to \mathbb{B} \to \mathbb{B}$. Dann gilt $\lambda y.fxy = \lambda\mathbb{B}.\langle 3\rangle\langle 1\rangle\langle 0\rangle$. ■

Proposition 1.5.4 Sei $\Gamma x = t$ und $\Gamma \vdash M : t'$. Dann $\Gamma \vdash \lambda x.M : t \to t'$.

© G. Smolka, 18. 5. 2004

1.6 Semantik

Sei eine Signatur und ein gemäß dieser Signatur gebildeter Term gegeben. Wenn wir Interpretationen für die Konstanten der Signatur und Werte für die freien Variablen des Terms vorgeben, beschreibt der Term einen Wert, der in der durch den Typ des Terms beschriebenen Menge enthalten ist. Wenn wir beispielsweise die Konstanten $2\colon \mathbb{N}$ und $+\colon \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ wie üblich interpretieren und den Wert 7 für die Variable x vorgeben, beschreibt der Term $x + 2$ den Wert 9.

Die den syntaktischen Objekten in diesem Sinne zugeordneten Objekte bezeichnet man als **semantische Objekte**. Es geht uns jetzt darum, die Zuordnung semantischer Objekte zu syntaktischen Objekten präzise zu definieren. Da intuitiv klar ist, wie diese Zuordnung aussieht, handelt es sich bei dieser Definition vor allem um eine technische Angelegenheit, bei der wir lernen, wie man die **Semantik** einer formalen Sprache definiert.

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

Interpretationen

Sei eine Signatur (TC, VC, ty) gegeben. Zunächst müssen wir den Konstanten der Signatur semantische Objekte zuordnen. Dazu nehmen wir an, dass eine Funktion $\mathcal{D}$ mit den folgenden Eigenschaften gegeben ist:

i) $TC \cup VC \subseteq Dom\,\mathcal{D}$.

ii) Für jede Typkonstante $b \in TC$ ist $\mathcal{D}b$ eine nichtleere Menge.

Jetzt können wir eine Funktion $\hat{\mathcal{D}}$ mit $Dom\,\hat{\mathcal{D}} = Ty$ definieren, die jedem Typ eine nichtleere Menge zuordnet:

$$\begin{aligned} \hat{\mathcal{D}}b &= \mathcal{D}b \\ \hat{\mathcal{D}}(t_1 \to t_2) &= \hat{\mathcal{D}}t_1 \to \hat{\mathcal{D}}t_2 \end{aligned}$$

Beachten Sie, dass $\hat{\mathcal{D}}$ einem funktionalen Typ $t_1 \to t_2$ die Menge aller Funktionen von $\hat{\mathcal{D}}t_1$ nach $\hat{\mathcal{D}}t_2$ zuordnet. Wenn möglich, werden wir statt $\hat{\mathcal{D}}t$ kürzer $\mathcal{D}t$ schreiben.

Wir sind jetzt in der Lage, eine dritte Bedingung für die Funktion $\mathcal{D}$ zu formulieren, die sicherstellt, dass die Denotationen der Wertkonstanten den durch die Signatur deklarierten Typen genügen:

iii) Für jede Wertkonstante $c \in VC$ gilt $\mathcal{D}c \in \mathcal{D}(ty\,c)$.

Unter einer **Interpretation** einer Signatur verstehen wir im Folgenden eine Funktion $\mathcal{D}$, die die Bedingungen i) bis iii) erfüllt.

Da die Konstanten einer Signatur beliebige mathematische Objekte sein können, ist es oft naheliegend, die Konstanten durch sich selbst zu interpretieren (d.h. $\mathcal{D}b = b$ und $\mathcal{D}c = c$). Beispielsweise können wir die Menge der natürlichen

Zahlen $\mathbb{N}$ als Typkonstante verwenden, und die Elemente von $\mathbb{N}$ als Wertkonstanten. Auch die Additions- und Multiplikationsfunktion können wir als Wertkonstanten verwenden.

Werte und Zustände

Sei eine Signatur und eine Interpretation $\mathcal{D}$ gegeben. Das dadurch bestimmte **Universum** ist die Menge

$$Val\,\mathcal{D} \stackrel{\text{def}}{=} \bigcup_{t \in Ty} \mathcal{D}t$$

Die Elemente des Universums bezeichnen wir als **Werte**.

Um die Denotation von Termen zu bestimmen, müssen wir Werte für die freien Variablen vorgeben. Dazu verwenden wir Funktionen $Var \rightharpoonup Val\,\mathcal{D}$, die wir als **Zustände** bezeichnen. Die Menge aller Zustände bezeichnen wir wie folgt:

$$Sta\,\mathcal{D} \stackrel{\text{def}}{=} Var \rightharpoonup Val\,\mathcal{D}$$

Zustände werden manchmal auch als *Wertumgebungen* oder *Belegungen* bezeichnet.

Sei Γ eine Typumgebung. Ein Γ-**Zustand** ist ein Zustand σ mit $Dom\,\Gamma \subseteq Dom\,\sigma$, sodass für alle $x \in Dom\,\Gamma$ gilt: $\sigma x \in \mathcal{D}(\Gamma x)$. Die Menge aller Γ-**Zustände** bezeichnen wir mit

$$Sta(\mathcal{D},\Gamma) \stackrel{\text{def}}{=} \{\, \sigma \in Sta\,\mathcal{D} \mid Dom\,\Gamma \subseteq Dom\,\sigma \;\wedge\; \forall x \in Dom\,\Gamma\colon\ \sigma x \in \mathcal{D}(\Gamma x) \,\}$$

Wir benötigen noch eine Cons-Operation, die einen Zustand σ um einen Wert v erweitert:

$$v :: \sigma \stackrel{\text{def}}{=} \{(0,v)\} \cup \{\, (n+1,v') \mid (n,v') \in \sigma \,\}$$

Denotation von Termen

Wir definieren jetzt eine partielle Funktion, die versucht, einen Präterm gemäß eines Zustands auszuwerten. Die Definition erfolgt durch Rekursion über die Struktur von Prätermen:

$$\begin{aligned}
&\hat{\mathcal{D}} \in PT \to (Sta\,\mathcal{D} \rightharpoonup Val\,\mathcal{D}) \\
&\qquad \hat{\mathcal{D}}c\sigma &&= \mathcal{D}c \\
&\qquad \hat{\mathcal{D}}x\sigma &&= \sigma x \quad \text{falls } x \in Dom\,\sigma \\
&\hat{\mathcal{D}}(MN)\sigma &&= (\hat{\mathcal{D}}M\sigma)\,(\hat{\mathcal{D}}N\sigma) \quad \text{falls } \hat{\mathcal{D}}M\sigma \text{ Funktion, die für } \hat{\mathcal{D}}N\sigma \text{ def. ist} \\
&\hat{\mathcal{D}}(\lambda t.M)\sigma &&= \lambda v \in \mathcal{D}t.\ \hat{\mathcal{D}}M(v :: \sigma)
\end{aligned}$$

Wenn möglich, werden wir statt $\hat{\mathcal{D}}M$ kürzer $\mathcal{D}M$ schreiben.

Die Auswertung eines Terms gemäß eines typgerechten Zustands ist immer erfolgreich und liefert einen Wert, der dem Typ des Terms entspricht.

© G. Smolka, 18. 5. 2004

Proposition 1.6.1 (Auswertung) Sei $\Gamma \vdash M : t$. Dann ist $\mathcal{D}M$ auf jedem Zustand $\sigma \in Sta(\mathcal{D}, \Gamma)$ definiert und $\mathcal{D}M\sigma \in \mathcal{D}t$.

Beweis Durch Induktion über M. Sei $\sigma \in Sta(\mathcal{D}, \Gamma)$. Fallunterscheidung:

1. Sei $M = c$. Dann $\mathcal{D}M\sigma = \mathcal{D}c$ und $ty\, c = t$. Also gilt $\mathcal{D}c \in \mathcal{D}t$ gemäß Bedingung (iii) für $\mathcal{D}$. Also $\mathcal{D}M\sigma \in \mathcal{D}t$.
2. Sei $M = x$. Dann $\Gamma x = t$. Da σ ein Γ-Zustand ist, folgt $\sigma x \in \mathcal{D}t$. Also $\mathcal{D}M\sigma \in \mathcal{D}t$.
3. Sei $M = M_1 M_2$. Dann gibt es einen Typ t' mit $\Gamma \vdash M_1 : t' \to t$ und $\Gamma \vdash M_2 : t'$. Also folgt mit der Induktionsannahme $\mathcal{D}M_1\sigma \in \mathcal{D}(t' \to t)$ und $\mathcal{D}M_2\sigma \in \mathcal{D}t'$. Also ist $\mathcal{D}M_1\sigma$ eine Funktion, die für $\mathcal{D}M_2\sigma$ einen Wert in $\mathcal{D}t$ liefert. Also $\mathcal{D}M\sigma \in \mathcal{D}t$.
4. Sei $M = \lambda t'.M'$. Dann existiert ein Typ t'' mit $t = t' \to t''$ und $t' :: \Gamma \vdash M' : t''$. Sei $v \in \mathcal{D}t'$. Wir müssen $\mathcal{D}M'(v :: \sigma) \in \mathcal{D}t''$ zeigen. Da $v :: \sigma$ ein $(t' :: \Gamma)$-Zustand ist, folgt dies mit der Induktionsannahme. ∎

Im Folgenden werden wir uns grundsätzlich nur für die Denotation von Termen gemäß typgerechter Zustände interessieren. Der obige Beweis zeigt, dass wir dann in einem geschlossenem System arbeiten, in dem die Seitenbedingungen der definierenden Gleichungen von $\hat{\mathcal{D}}$ immer erfüllt sind und auch im Rekursionsfall immer nur Terme gemäß typgerechter Zustände ausgewertet werden müssen.

Beachten Sie, dass wir die Denotationsfunktion für Präterme in einem ungetypten Rahmen definiert haben. Normalerweise versucht man bei der mathematischen Arbeit in einem getypten Rahmen zu arbeiten, um Schwierigkeiten mit undefinierten Funktionsanwendungen prinzipiell zu vermeiden. Bei der Definition der Denotationsfunktion bleibt uns aber keine andere Wahl, da es sich hier um eine „Bootstrap-Situation“ handelt, bei der die getypte Welt auf Grundlage der ungetypten Mengenlehre errichtet wird.

Wenn man einen Term gemäß zweier typgerechter Zustände auswertet, die auf den freien Variablen des Terms übereinstimmen, erhält man denselben Wert:

Proposition 1.6.2 (Koinzidenz) Sei $\Gamma \vdash M : t$ und $\sigma, \sigma' \in Sta(\mathcal{D}, \Gamma)$. Wenn $\sigma x = \sigma' x$ für alle $x \in FV(M)$ gilt, dann $\mathcal{D}M\sigma = \mathcal{D}M\sigma'$.

Beweis Durch Induktion über M. Übung! ∎

Aus der Proposition folgt, dass ein geschlossener Term für alle typgerechten Zustände denselben Wert liefert.

Um die grundlegende Verträglichkeitseigenschaft zwischen Substitution und Denotation zu formulieren, benötigen wir die so genannte **Adjunktionsoperation**, die zu einer Funktion f und einem Paar (a, b) die Funktion liefert, die für a

den Wert b liefert und sich ansonsten genau wie f verhält:

$$f[a:=b] \stackrel{\text{def}}{=} \{(a,b)\} \cup \{\,(u,v) \in f \mid u \neq a\,\}$$

Lemma 1.6.3 (Substitution) Sei $\Gamma \vdash M : t$, $\Gamma \vdash N : t'$, $\Gamma x = t'$ und $\sigma \in \mathit{Sta}(\mathcal{D},\Gamma)$. Dann $\mathcal{D}(M[x:=N])\sigma = \mathcal{D}M(\sigma[x:=\mathcal{D}N\sigma])$.

Auf den Beweis dieses Lemmas verzichten wir, da er wegen der vielen zu betrachtenden Details recht mühselig ist. Er kann wie die Beweise der vorausgehenden Propositionen durch Induktion über M geführt werden.

Die nächste Proposition zeigt, wie die Denotation eines durch $\lambda x.M$ beschriebenen Terms direkt bestimmt werden kann.

Proposition 1.6.4 Sei $\lambda x.M$ ein Γ-Term, t der Typ von x und $\sigma \in \mathit{Sta}(\mathcal{D},\Gamma)$. Dann $\mathcal{D}(\lambda x.M)\sigma = \lambda\, v \in \mathcal{D}t.\ \mathcal{D}M(\sigma[x:=v])$.

Auf den etwas mühseligen Beweis dieser Proposition verzichten wir.

Wir haben jetzt Syntax und Semantik von ETT präzise definiert. Damit haben wir ein mathematisches Modell, mit dem wir grundlegende logische Eigenschaften formulieren und beweisen können.

1.7 Gleichungen und semantischer Abschluss

Das Rechnen und Schließen mit Gleichungen gehört zu den grundlegenden logischen Phänomenen. Mit ETT verfügen wir über eine solide Grundlage, um dieses Phänomen zu erkunden.

Damit wir nicht so viel schreiben müssen, arbeiten wir im Folgenden gemäß eines syntaktischen Standardkontexts, der durch eine nicht weiter bestimmte Signatur (TC, VC, ty) und eine dazu passende Typumgebung Γ_0 gegeben ist. Wenn wir nichts anderes sagen, verstehen wir unter einem Term oder einem Zustand stets einen Γ_0-Term oder einen Γ_0-Zustand. Entsprechend verstehen wir unter dem Typ eines Terms seinen Typ in Γ_0. Wenn wir die Notation $\lambda x.M$ benutzen, wird der Typ von x stets als $\Gamma_0\, x$ angenommen werden. Die Menge aller Γ_0-Terme bezeichnen wir mit Ter.

Definition 1.7.1 Eine **Gleichung** ist ein Paar (M,N) aus zwei Termen, die den gleichen Typ haben. Die Menge aller Gleichungen bezeichnen wir mit Eq. Wenn immer möglich, schreiben wir eine Gleichung (M,N) suggestiver als $M = N$.

Wir sagen, dass eine Interpretation $\mathcal{D}$ eine Gleichung $M = N$ **erfüllt** und schreiben $\mathcal{D} \vDash M = N$, wenn die Terme M und N in $\mathcal{D}$ für alle Zustände den gleichen Wert liefern (d.h. $\mathcal{D}M\sigma = \mathcal{D}N\sigma$ für alle Zustände $\sigma \in \mathit{Sta}(\mathcal{D},\Gamma_0)$). Statt $\mathcal{D}$ erfüllt E sagen wir auch, dass E in $\mathcal{D}$ **gültig** ist.

© G. Smolka, 18. 5. 2004

Wir sagen, dass eine Interpretation $\mathcal{D}$ eine Gleichungsmenge $A \subseteq Eq$ **erfüllt** und schreiben $\mathcal{D} \vDash A$, wenn $\mathcal{D}$ jede Gleichung in A erfüllt. ■

Beachten Sie, dass eine Gleichungsmenge A (d.h. eine Menge von Gleichungen) eine Relation auf der Menge der Terme ist (d.h. $A \subseteq Ter^2$).

Gültige Gleichungen werden oft als *Identitäten* oder *Gesetze* bezeichnet.

Beispiel 1.7.1 Seien die Konstanten $\mathbb{N}$, $+$ und 0 wie üblich getypt und interpretiert und sei x eine Variable des Typs $\mathbb{N}$ (d.h. $\Gamma_0\, x = \mathbb{N}$). Dann handelt es sich bei $x + 0 = x$ um eine Gleichung, die in der zu grunde liegenden Interpretation gültig ist. ■

Die Menge aller Gleichungen, die in einer Interpretation $\mathcal{D}$ gültig sind, bezeichnen wir als die **Theorie von** $\mathcal{D}$:

$$Th\ \mathcal{D} \stackrel{\text{def}}{=} \{ E \in Eq \mid \mathcal{D} \vDash E \}$$

Wir sind jetzt in der Lage, einen *logischen Folgerungsbegriff* präzise zu definieren:

Definition 1.7.2 Wir sagen, dass eine Gleichung E aus einer Menge A von Gleichungen **semantisch folgt** und schreiben $A \vDash E$, wenn jede Interpretation, die A erfüllt, auch E erfüllt. Die Menge aller Gleichungen, die aus einer Menge A von Gleichungen logisch folgen, bezeichnen wir als den **semantischen Abschluß** von A:

$$SC\ A \stackrel{\text{def}}{=} \{ E \in Eq \mid A \vDash E \}$$ ■

Die Theorie einer Interpretation ist bezüglich semantischer Folgerung abgeschlossen:

Proposition 1.7.1 (Semantische Abgeschlossenheit von Theorien) Für jede Gleichungsmenge A und jede Interpretation $\mathcal{D}$ gilt: $SC(Th\ \mathcal{D}) = Th\ \mathcal{D}$.

Des Begriff des semantischen Abschlusses liefert eine Funktion, die Gleichungsmengen auf Gleichungsmengen abbildet und die für logische Folgerungsbegriffe charakteristischen Eigenschaften hat. Wir formulieren diese Eigenschaften mithilfe des abstrakten Begriffs des Abschlussoperators.

Definition 1.7.3 Sei X eine Menge. Ein **Abschlussoperator für** X ist eine Funktion $f \in \mathcal{P}(X) \to \mathcal{P}(X)$, sodass für alle $A, B \subseteq X$ gilt:

- *Expansivität:* $A \subseteq f\,A$
- *Monotonie:* $A \subseteq B \Longrightarrow f\,A \subseteq f\,B$.
- *Idempotenz:* $f(f\,A) = f\,A$. ■

Proposition 1.7.2 Der semantische Abschluss ist ein Abschlussoperator für *Eq*.

Proposition 1.7.3 Sei f ein Abschlussoperator für X. Dann gilt für alle Teilmengen $A, B, C \subseteq X$:

a) $A \subseteq fB \iff f(A \cup B) = fB$

b) $fA = fB \Longrightarrow f(A \cup C) = f(B \cup C)$

c) $f(A \cup B) = f(A \cup C) \iff B \subseteq f(A \cup C) \ \wedge \ C \subseteq f(A \cup B)$

1.8 Deduktiver Abschluss

Unter **Deduktion** versteht man das Ableiten von gültigen Aussagen aus gültigen Aussagen auf rein syntaktischer Basis. Wir führen jetzt ein Deduktionssystem für ETT ein, mit dem man aus einer gegebenen Gleichungsmenge A Gleichungen ableiten kann, die logisch aus A folgen.

Das Deduktionssystem besteht aus den in Abbildung 1.4 gezeigten Ableitungsregeln. Mit diesen so genannten **Gleichheitsregeln** kann man aus Gleichungen mechanisch weitere Gleichungen ableiten. Beispielsweise kann man mit der Symmetrie-Regel

$$\frac{M = N}{N = M}$$

aus einer Gleichung $M = N$ die Gleichung $N = M$ ableiten. Allgemein haben die Gleichheitsregeln die Form

$$\frac{\textit{Prämisse 1} \quad \cdots \quad \textit{Prämisse n} \qquad \textit{Seitenbedingung}}{\textit{Konklusion}}$$

wobei es sich bei den Prämissen und der Konklusion um Beschreibungen von Gleichungen handelt. Die Seitenbedingung ist optional. Die Reflexivitäts-, Beta- und Eta-Regel haben keine Prämissen und liefern daher Gleichungen, ohne dass Gleichungen vorgegeben werden. Die Ersetzungs-, Einsetzungs- und Eta-Regel sind mit einer Seitenbedingung versehen. Die Seitenbedingung der Ersetzungsregel ist mithilfe der folgenden Notation formuliert:

$$M/N \approx M'/N' \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} M' \text{ kann aus } M \text{ gebildet werden, indem genau ein} \\ \text{Auftreten von } N \text{ in } M \text{ durch } N' \text{ ersetzt wird} \end{array}$$

Beispielsweise gilt:

$$x \cdot (y + 0)/y + 0 \ \approx \ x \cdot y/y$$
$$x/x \ \approx \ fxx/fxx$$
$$\lambda x.fxx/x \ \approx \ \lambda x.f(gx)x/gx$$

© G. Smolka, 18. 5. 2004

Reflexivität, Symmetrie und Transitivität

$$\frac{}{M = M} \qquad \frac{M = N}{N = M} \qquad \frac{M = N \quad N = M'}{M = M'}$$

Ersetzung

$$\frac{N = N' \quad M/N \approx M'/N'}{M = M'}$$

Einsetzung

$$\frac{M = M' \quad x \text{ und } N \text{ haben denselben Typ}}{M[x := N] = M'[x := N]}$$

Beta und Eta

$$\frac{}{(\lambda x.M)N = M[x := N]} \qquad \frac{x \notin FV(M)}{\lambda x.Mx = M}$$

Abbildung 1.4: Die Gleichheitsregeln

Das letzte Beispiel zeigt, dass beim Ersetzen Variablen gekapert werden dürfen (was beim Einsetzen nicht erlaubt ist).

Gegeben eine eventuell leere Menge von Gleichungen, können wir mit den Gleichheitsregeln in einem oder mehreren Schritten Gleichungen ableiten. Eine solche Ableitung erfolgt ohne Kenntnis der Interpretation der Konstanten rein mechanisch auf der Basis von syntaktischen Objekten. Wir werden zeigen, dass mit den Gleichheitsregeln aus gültigen Gleichungen nur gültige Gleichungen abgeleitet werden können. Eine Ableitung stellt also einen formalen Beweis dafür dar, dass die abgeleitete Gleichung in jeder Interpretation gültig ist, in der alle Ausgangsgleichungen gültig sind.

Beispiel 1.8.1 Seien $fxx = x$ und $(\lambda y.fy(\lambda z.gz))(fgg) = g$ Gleichungen. Wenn x eine Variable ist, können wir mit den Gleichheitsregeln aus der ersten Gleichung die zweite Gleichung ableiten. Also ist die zweite Gleichung in jedem Kontext gültig, in dem die erste Gleichung gültig. Dabei spielt es keine Rolle, ob f und g Konstanten oder Variablen sind. Die Ableitung der zweiten Gleichung aus der ersten Gleichung können wir wie folgt darstellen:

$$\begin{aligned}
&(\lambda y.fy(\lambda z.gz))(fgg) && \\
&= (\lambda y.fy(\lambda z.gz))g && fxx = x, \text{ Ein, Ers} \\
&= fg(\lambda z.gz) && \beta \\
&= fgg && \eta, \text{ Ers} \\
&= g && fxx = x, \text{ Ein}
\end{aligned}$$

Rechts haben wir die Benutzung der ersten Gleichung und der jeweiligen Gleichheitsregeln ausgewiesen. Dabei bezeichnet „Ein“ die Einsetzungsregel, „Ers“ die Ersetzungsregel, β die Beta-Regel und η die Eta-Regel. Zusätzlich benötigt man noch 3 Anwendungen der Transitivitätsregel, um die abgeleiteten Gleichungen zu $(\lambda y.fy(\lambda z.gz))(fgg) = g$ zu kombinieren.

Machen Sie sich unbedingt bis ins letzte Detail klar, wie die obige Ableitung funktioniert. ■

Definition 1.8.1 Wir sagen, dass eine Gleichung E aus einer Menge A von Gleichungen **ableitbar** ist und schreiben $A \vdash E$, wenn E aus A durch endlich viele Anwendungen der Gleichheitsregeln in Abbildung 1.4 erhalten werden kann. Die Menge aller Gleichungen, die aus einer Menge A von Gleichungen ableitbar ist, bezeichnen wir als den **deduktiven Abschluß** von A:

$$DC\ A \stackrel{\text{def}}{=} \{ E \in Eq \mid A \vdash E \}$$ ■

Proposition 1.8.1 Der deduktive Abschluss ist eine Abschlussoperator für Eq.

Die nächste Proposition formuliert die für deduktive Folgerungsbegriffe charakteristische *Kompaktheitseigenschaft*:

Proposition 1.8.2 (Kompaktheit) Sei A eine Gleichungsmenge und $E \in DC\ A$. Dann existiert eine endliche Teilmenge $A' \subseteq A$ mit $E \in DC\ A'$.

1.9 Korrektheit der Gleichheitsregeln

Wir zeigen jetzt, das man mit den Gleichheitsregeln aus gültigen Gleichungen nur gültige Gleichungen ableiten kann. Diese als *Korrektheit der Gleichheitsregeln* bezeichnete Tatsache ist mit $DC\ A \subseteq SC\ A$ gleichbedeutend.

Wir nehmen an, dass ein Standardkontext durch eine Signatur (TC, VC, ty), eine dazu passende Typumgebung Γ_0 und eine dazu passende Interpretation $\mathcal{D}$ gegeben ist. Wir schreiben $M \equiv N$ um zu sagen, dass die Gleichung $M = N$ in $\mathcal{D}$ gültig ist.

Grundlegende Eigenschaften der Gleichheit sind Reflexivität, Symmetrie und Transitivität. Also ist $\equiv$ eine Äquivalenzrelation auf der Menge der Terme. Die Korrektheit der ersten drei Gleichheitsregeln beruht auf dieser Tatsache und wird durch die folgende Proposition festgestellt.

Proposition 1.9.1 (Äquivalenz) Seien M, M' und N Terme. Dann gilt:

- $M \equiv M$
- $M \equiv M' \implies M' \equiv M$

© G. Smolka, 18. 5. 2004

- $M \equiv N \ \wedge \ N \equiv M' \implies M \equiv M'$

Die nächste Proposition formuliert die Korrektheit der so genannten **Kongruenzregeln**, die die Grundlage für die Ersetzungsregel bilden.

Proposition 1.9.2 (Kongruenz) Seien M, M' und N Terme. Dann gilt:

- $M \equiv M' \implies MN \equiv M'N$
- $N \equiv N' \implies MN \equiv MN'$
- $M \equiv M' \implies \lambda t.M \equiv \lambda t.M'$
- $M \equiv M' \implies \lambda x.M \equiv \lambda x.M'$

Die nächste Proposition stellt die Korrektheit der **Ersetzungsregel** fest:

Proposition 1.9.3 (Ersetzung) Sei $N \equiv N'$ und $M/N \approx M'/N'$. Dann $M \equiv M'$.

Der Beweis dieser Proposition zeigt, dass eine Anwendung der Ersetzungsregel immer durch wiederholtes Anwenden der Kongruenzregeln simuliert werden kann. Sie sollten diesen Beweis im Detail nachvollziehen.

Beweis Durch Induktion über M. Wir unterscheiden die folgenden Fälle.

1. Sei $M = N$. Dann $M = N \equiv N' = M'$.
2. Sei $M = M_1M_2$, $M' = M_1'M_2'$ und $M_1/N \approx M_1'/N'$. Dann folgt $M_1 \equiv M_1'$ mit der Induktionsannahme. Damit liefert die erste Kongruenzregel $M \equiv M'$.
3. Sei $M = M_1M_2$, $M' = M_1'M_2'$ und $M_2/N \approx M_2'/N'$. Dann folgt $M_2 \equiv M_2'$ mit der Induktionsannahme. Damit liefert die zweite Kongruenzregel $M \equiv M'$.
4. Sei $M = \lambda t.M_1$, $M' = \lambda t.M_1'$ und $M_1/N \approx M_1'/N'$. Dann folgt $M_1 \equiv M_1'$ mit der Induktionsannahme. Damit liefert die dritte Kongruenzregel $M \equiv M'$. ■

Wir kommen jetzt zur Korrektheit der Einsetzungsregel. Dazu müssen wir zeigen, dass die Anwendung einer Substitution auf eine gültige Gleichung eine gültige Gleichung liefert.

Proposition 1.9.4 (Einsetzung) Seien M, M', N Terme und x eine Variable, die denselben Typ wie N hat. Dann gilt: $M \equiv M' \implies M[x{:=}N] \equiv M'[x{:=}N]$.

Beweis Sei $M \equiv M'$. Mit Proposition 1.5.3 folgt, dass es sich bei $M[x := N]$ und $M'[x := N]$ um Terme desselben Typs handelt. Sei σ ein Zustand. Wir müssen zeigen, dass $M[x := N]$ und $M'[x := N]$ für σ dieselbe Denotation haben. Mit dem Substitutionslemma 1.6.3 folgt

$$\mathcal{D}(M[x{:=}N])\sigma = \mathcal{D}M(\sigma[x{:=}\mathcal{D}N\sigma])$$
$$\mathcal{D}(M'[x{:=}N])\sigma = \mathcal{D}M'(\sigma[x{:=}\mathcal{D}N\sigma])$$

Da es sich bei $\sigma[x := \mathcal{D}N\sigma]$ um einen typgerechten Zustand handelt, folgt aus $M \equiv M'$, dass $\mathcal{D}M(\sigma[x := \mathcal{D}N\sigma]) = \mathcal{D}M'(\sigma[x := \mathcal{D}N\sigma])$. Also haben $M[x := N]$ und $M'[x := N]$ für σ dieselbe Denotation. ■

Beispiel 1.9.1 Sei $fxx \equiv x$ und sei M ein Term, der denselben Typ wie die Variable x hat. Dann folgt $fMM \equiv M$ mit der Einsetzungsregel. ■

Jetzt fehlen nur noch die Korrektheit der Beta- und der Eta-Regel.

Proposition 1.9.5 (Beta) $(\lambda x.M)N \equiv M[x := N]$ falls $(\lambda x.M)N$ ein Term ist.

Beweis Sei $(\lambda x.M)N$ ein Term und sei σ ein Zustand. Wir müssen $\mathcal{D}((\lambda x.M)N)\sigma = \mathcal{D}(M[x := N])\sigma$ zeigen. Das gelingt mit Proposition 1.6.1 wie folgt:

$$\begin{aligned}
\mathcal{D}((\lambda x.M)N)\sigma &= (\mathcal{D}(\lambda x.M)\sigma)(\mathcal{D}N\sigma) && \text{Definition } \mathcal{D}\\
&= (\lambda\, \nu \in \mathcal{D}(\tau x).\, \mathcal{D}M(\sigma[x := \nu]))(\mathcal{D}N\sigma) && \text{Proposition 1.6.4}\\
&= \mathcal{D}M(\sigma[x := (\mathcal{D}N\sigma)])\\
&= \mathcal{D}(M[x := N])\sigma && \text{Substitutionslemma 1.6.3}
\end{aligned}$$ ■

Proposition 1.9.6 (Eta) $\lambda x.Mx \equiv M$ falls $x \notin FV(M)$ und $\lambda x.Mx$ ein Term ist.

Beweis Sei $\lambda x.Mx$ ein Term, x eine Variable des Typs t, die nicht frei in M auftritt, und σ ein Zustand. Wir müssen zeigen, dass $\mathcal{D}(\lambda x.Mx) = \mathcal{D}M\sigma$. Das gelingt mit Proposition 1.6.1 wie folgt:

$$\begin{aligned}
\mathcal{D}(\lambda x.Mx) &= \lambda\, \nu \in \mathcal{D}t.\, \mathcal{D}(Mx)(\sigma[x := \nu]) && \text{Proposition 1.6.4}\\
&= \lambda\, \nu \in \mathcal{D}t.\, (\mathcal{D}M(\sigma[x := \nu]))(\mathcal{D}x(\sigma[x := \nu])) && \text{Definition } \mathcal{D}\\
&= \lambda\, \nu \in \mathcal{D}t.\, (\mathcal{D}M(\sigma[x := \nu]))\nu && \text{Definition } \mathcal{D}\\
&= \lambda\, \nu \in \mathcal{D}t.\, (\mathcal{D}M\sigma)\nu && \text{Proposition 1.6.2}\\
&= \mathcal{D}M\sigma
\end{aligned}$$ ■

Aus der Korrektheit der einzelnen Regeln folgt sofort die Korrektheit des durch die Gleichheitsregeln gegebenen Deduktionssystems. Da wir über die zugrunde gelegte Interpretation $\mathcal{D}$ keine besonderen Annahmen gemacht haben, folgt:

Satz 1.9.7 (Korrektheit) Für alle $A \subseteq Eq$ gilt: $DC\ A \subseteq SC\ A$.

Korollar 1.9.8 (Deduktive Abgeschlossenheit von Theorien) Für jedes $A \subseteq Eq$ und jede Interpretation $\mathcal{D}$ gilt: $DC(Th\ \mathcal{D}) = Th\ \mathcal{D}$.

Beweis Die Richtung „$\supseteq$“ folgt mit der Expansivität von DC. Die andere Richtung folgt mit der semantischen Abgeschlossenheit von Theorien (Proposition 1.7.1) und dem Korrektheitssatz. ■

© G. Smolka, 18. 5. 2004

Korollar 1.9.9 Für alle $A, A' \subseteq Eq$ gilt: $DC\ A \subseteq DC\ A' \Longrightarrow SC\ A \subseteq SC\ A'$.

Beweis Sei $DC\ A \subseteq DC\ A'$. Mit der Expansivität von DC und dem Korrektheitssatz folgt $A \subseteq DC\ A \subseteq DC\ A' \subseteq SC\ A'$. Also folgt $SC\ A \subseteq SC(SC\ A') = SC\ A'$ mit der Monotonie und Idempotenz von SC. ■

Es stellt sich die Frage, ob umgekehrt zur Korrekheit auch $SC\ A \subseteq DC\ A$ und damit $SC\ A = DC\ A$ gilt. Diese Eigenschaft wird als **Vollständigkeit** des zugrunde liegenden Deduktionssystems bezeichnet. Vollständigkeit bedeutet, dass die der Definition des semantischen Abschlusses zugrunde liegende Information durch die Gleichheitsregeln vollständig syntaktisch erfasst ist.

Bisher konnte der Autor nicht klären, ob die Gleichheitsregeln für ETT im obigen Sinne vollständig sind. Für den Spezialfall $A = \emptyset$ sind die Gleicheitsregeln jedoch vollständig:

Satz 1.9.10 $SC\,\emptyset = DC\,\emptyset$

Der Beweis dieses Satzes ist nicht einfach. Näheres findet man im Zusammenhang mit Theorem A.1.19 in Barendregts Buch über dem Lambda-Kalkül.