

# 5. Lambda-Kalkül und Logik höherer Stufe

## 5.1 Lambda-Kalkül (ungetypt)

- Lambda-Notation, Lambda-Terme
- Konversionsregeln

## 5.2 Einfach getypter Lambda-Kalkül

## 5.3 (Prädikaten-)Logik höherer Stufe

- Motivation
- Syntax
- Semantik
- Schlußregeln
- Nutzen

## 5.1 Lambda-Kalkül

*Lambda-Ausdrücke*: Notation zur Bezeichnung von Werten, die *Funktionen* darstellen  
z.B. statt

$$f(x) = 3 * x + 4$$

können wir schreiben

$$f = \lambda x. 3 * x + 4$$

Der Ausdruck “ $\lambda x. 3 * x + 4$ ” kann an Stellen benutzt werden, wo “gewöhnlich” ein Funktionsname steht; er bezeichnet eine *anonyme Funktion*, im Gegensatz zu einer *deklarierten* und benannten Funktion.

Z.B. bei Funktionsanwendung:

$$(\lambda x. 3 * x + 4)(2 * a)$$

Der Variablenname, der dem Symbol  $\lambda$  folgt, die *Lambda-Variable*, entspricht dem formalen Parameter bei einer herkömmlichen Funktions-Deklaration (wie etwa in Programmiersprachen).

# Lambda-Terme: Syntax

*Syntax* von Lambda-Termen (zunächst ohne Typen):

- Ein Variablen- oder Konstantensymbol ist ein Lambda-Term.
- Ist  $x$  eine Variable und  $t$  ein Lambda-Term, so ist auch  $\lambda x. t$  ein Lambda-Term:  
*Lambda-Abstraktion.*
- Sind  $s$  und  $t$  Lambda-Terme, so ist auch  $s(t)$  ein Lambda-Term:  
*Lambda-Applikation.*

Klammern können hinzugefügt werden, wenn notwendig für Eindeutigkeit.

Abweichend von der strengen Syntax verwenden wir in konkreten Beispielen für gewisse Standard-Funktionssymbole die übliche Infix-Notation.

Die Notation für Lambda-Terme ist nicht einheitlich; insbesondere wird in theoretischen Arbeiten häufig Juxtaposition  $fx$  benutzt, wo wir  $f(x)$  schreiben.

In dem Term  $\lambda x. t$  wird die Variable durch  $\lambda$  gebunden in  $t$ . Die Bindung ist analog der Bindung von Variablen in einer quantifizierten Formel; insbesondere werden die Begriffe *Bindungsbereich*, *freie* und *gebundene Variable* für Lambda-Terme genauso wie für quantifizierte Formeln in PL1 definiert.

# Konversionsregeln

Lambda-Terme können umgeformt oder “konvertiert” werden durch Anwendung der folgenden *Konversionsregeln*:

- $\alpha$ -Regel: Gebundene Variablen können umbenannt werden, sofern keine Namenskonflikte auftreten.

$$(\lambda x. e) = (\lambda y. e[x \leftarrow y])$$

$y$  darf in  $e$  nicht vorkommen, weder frei noch gebunden.

- $\beta$ -Regel: “Auswertung” der Anwendung eines Lambda-Terms auf ein Argument.

$$(\lambda x. e_1)(e_2) = e_1[x \leftarrow e_2]$$

Hierbei ist die Substitution nur zulässig, falls  $e_2$  *frei für  $x$  in  $e_1$*  ist, d.h. falls durch die Substitution keine freien Vorkommen von Variablen in  $e_2$  in dem Term  $e_1[x \leftarrow e_2]$  gebunden werden.

Durch (ein- oder mehrmalige) vorherige Anwendung der  $\alpha$ -Regel auf  $e_1$  kann dies generell vermieden werden.

## Konversionsregeln (2)

Die  $\beta$ -Regel beschreibt, wie ein zusammengesetzter Lambda-Term zu einer einfacheren Form *reduziert* werden kann, d.h. die  $\beta$ -Regel ist eine *Reduktionsregel*.

$\rightsquigarrow$   $\beta$ -Reduktion von Lambda-Ausdrücken

Reduktion = “Auswertung”

Die  $\beta$ -Regel kann als eine Art von “Rechenvorschrift” für Lambda-Terme angesehen werden.

Sie gibt im wesentlichen an, wie ein Lambda-Ausdruck auszuwerten ist.

Terminologie: Ein Ausdruck der Form  $(\lambda x. e_1)(e_2)$  heißt *Redex* und  $e_1[x \leftarrow e_2]$  das dazu gehörende *Kontraktum*.

## Beispiel für $\beta$ -Reduktion

$$\begin{aligned} & (\lambda x. (\lambda f. f(f(x))))(\lambda x. x * x + 1)(2) \\ &= (\lambda f. f(f(x)))(\lambda x. x * x + 1)[x \leftarrow 2] \\ &= (\lambda f. f(f(2)))(\lambda x. x * x + 1) \\ &= f(f(2))[f \leftarrow (\lambda x. x * x + 1)] \\ &= (\lambda x. x * x + 1)((\lambda x. x * x + 1)(2)) \\ &= (x * x + 1)[x \leftarrow ((\lambda x. x * x + 1)(2))] \\ &= ((\lambda x. x * x + 1)(2)) * ((\lambda x. x * x + 1)(2)) + 1 \\ &= ((x * x + 1)[x \leftarrow 2]) * ((x * x + 1)[x \leftarrow 2]) + 1 \\ &= (2 * 2 + 1) * (2 * 2 + 1) + 1 \\ &= \dots = 26 \end{aligned}$$

## Beispiel für $\beta$ -Reduktion (Forts.)

Bemerkungen:

- Die gewählte Auswertungsstrategie spielt offensichtlich eine Rolle bezüglich Effizienz: Unterschied zwischen *call-by-name* und *call-by-value*, *innermost*, *outermost*, links-nach-rechts usw.
- Durch *call-by-name*-Auswertung kann Mehrfachauswertung eines Ausdrucks erforderlich werden.
- Möglichkeit der partiell parallelen Auswertung
- Diese Form der Auswertung funktionaler Ausdrücke ist die Grundlage vieler funktionaler Sprachen.

# Lambda-Kalkül

Der *Lambda-Kalkül* ist ein formaler Kalkül zur Ableitung von Gleichungen zwischen Lambda-Termen:

$$M = N$$

entsprechend den folgenden Regeln:

**Regeln** des Lambda-Kalkül:

- $\alpha$ -Regel (“ $\alpha$ -Konversion”)
- $\beta$ -Regel (“ $\beta$ -Konversion” oder “ $\beta$ -Reduktion”)
- Kongruenz-Abschluß der sich aus diesen Konversionen ergebenden Gleichungen:

$$M = M$$

$$M = N \Rightarrow N = M$$

$$M = N \wedge N = L \Rightarrow M = L$$

$$M = N \wedge L = K \Rightarrow L(M) = K(N)$$

$$M = N \Rightarrow \lambda x. M = \lambda x. N$$

“ $\xi$ -Regel”

# Lambda-Kalkül: Normalformen

Ein Lambda-Term ist *in  $\beta$ -Normalform* (oder einfacher: eine Normalform), wenn er keinen Unterterm der Form  $(\lambda x. M)(N)$  enthält.

Ein Term  $M$  hat eine Normalform, wenn es einen Term  $N$  in Normalform gibt mit  $M = N$ .

Nicht jeder Lambda-Term hat eine Normalform. Beispiel:

$$N \doteq (\lambda x. x(x))(\lambda x. x(x))$$

Normalformen und Ableitbarkeit im Kalkül hängen direkt zusammen:

Sind  $M$  und  $N$  zwei verschiedene Normalformen, dann läßt sich die Gleichung  $M = N$  im Kalkül nicht ableiten.

# Extensionalität

Lambda-Terme bezeichnen (im allgemeinen) *Funktionen*. Im mathematischen Sinn ist es unwichtig, *wie* Funktionswerte ausgerechnet werden, sondern nur, was die Funktionswerte sind.

*Extensionalität* bezeichnet die Eigenschaft, daß Funktionen als gleich angesehen werden, wenn sie für alle Argumente dieselben Werte liefern:

$$(\forall x. f(x) = g(x)) \Rightarrow f = g \quad (\text{Extensionalität})$$

Hier darf  $x$  nicht frei in  $f$  oder  $g$  vorkommen.

Aus dem Extensionalitätsaxiom folgt die “ $\eta$ -Regel”:

$$\lambda x. f(x) = f \quad \eta\text{-Konversion}$$

vorausgesetzt  $x$  kommt nicht frei in  $f$  vor.

## Extensionalität (2)

Mit Hilfe der  $\xi$ -Regel läßt sich zeigen, daß aus dem Lambda-Kalkül mit  $\eta$ -Regel Extensionalität folgt:

Angenommen  $M(x) = N(x)$  mit  $x$  nicht frei in  $M$  oder  $N$ ; mit  $\xi$  folgt

$$(\lambda x. M(x)) = (\lambda x. N(x)),$$

mit  $\eta$  daraus  $M = N$ .

In praktisch verwendeten Kalkülen wird im allgemeinen entweder Extensionalität oder  $\eta$ -Konversion vorausgesetzt.

*Literatur* zum Lambda-Kalkül: Die Standard-Referenz ist

H. P. Barendregt: *The Lambda Calculus. Its Syntax and Semantics*.

North-Holland, 1984 (revised edition).

## 5.2 Einfach getypter Lambda-Kalkül

Ungetypter Lambda-Kalkül erlaubt Terme wie “Selbstanwendung”, die den Regeln “normaler” Typisierung widersprechen.

Der *einfach getypte Lambda-Kalkül* folgt den Typisierungs-Regeln, wie sie für getypte Sprachen üblich sind: jeder Variablen und jedem Term wird ein Typ zugeordnet, und alle Terme, insbesondere Lambda-Applikationen, müssen typ-korrekt sein.

Notation:  $M : T$  der Term  $M$  hat den Typ  $T$

*Typ-Ausdrücke:*

- Typ-Konstanten, speziell Basistypen wie  $bool$ ,  $nat$ , . . .
- Funktionstypen  $S \rightarrow T$  für Typ-Ausdrücke  $S$  und  $T$
- Tupeltypen (“kartesische Produkte”):  $S \times T$  Typ-Ausdrücke  $S$  und  $T$

## Einfach getypter Lambda-Kalkül (2)

Die *Syntax* von Termen ist im wesentlichen dieselbe wie für ungetypte Lambda-Terme, mit dem Unterschied, daß Lambda-Bindungen getypt werden:

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

Die Typangabe für die Lambda-Variable wird häufig fortgelassen, wenn der Typ aus dem Zusammenhang klar ersichtlich ist.

## Regeln für Typkorrektheit von Lambda-Termen

Lambda-Terme sind nur wohlgeformt (und zulässig), wenn sie den Typisierungsregeln entsprechen. Typisierung ist relativ zu einem *Kontext*  $\Gamma$  von Typ-Deklarationen.

$\Gamma, x : T \vdash x : T$  eine Art 'Axiom' der Typisierung

$$\frac{\Gamma \vdash x : S \quad \Gamma, x : S \vdash e : T}{\Gamma \vdash (\lambda x : S. e) : S \rightarrow T}$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash f(e) : T}$$

Die **Konversionsregeln** für den getypten Lambda-Kalkül sind dieselben wie vorher.

## Typisierung von Lambda-Termen

Vielen, aber nicht allen Termen des ungetypten Lambda-Kalküls kann ein Typ zugeordnet werden.

Offensichtlich schließen die Typ-Regeln die Typisierung von Termen wie  $x(x)$  aus.

# Mehrstellige Funktionen

Konzeptionell mehrstellig Funktionen können auf verschiedene Weise im Kalkül dargestellt werden. Jeder Funktion des Typs

$$T_1 := (S_1 \times S_2) \rightarrow S_3$$

entspricht eine Funktion des Typs

$$T_2 := S_1 \rightarrow (S_2 \rightarrow S_3)$$

und umgekehrt durch die Zuordnungen

$$f : T_1 \rightsquigarrow (\lambda x : S_1. \lambda y : S_2. f(x, y)) : T_2$$

$$g : T_2 \rightsquigarrow (\lambda (x, y) : S_1 \times S_2. g(x)(y)) : T_1$$

Der Übergang von  $T_1$  nach  $T_2$  heißt *Currying* und ist ein Isomorphismus zwischen Funktionenräumen.

Streng genommen könnte man auf Produkt-Typen verzichten und sich auf die Curry-Form von Funktionstypen beschränken.

## Getypter Lambda-Kalkül: Normalformen

$\beta\eta$ -Konversion ist die Kombination von  $\beta$ -Konversion und  $\eta$ -Konversion (Anwendung der  $\eta$ -Regel als Reduktionsregel).

### Satz:

- Jeder Term des getypten Lambda-Kalküls hat eine Normalform.
- $\beta$ -Konversion (bzw.  $\beta\eta$ -Konversion) ist für den (einfach) getypten Lambda-Kalkül streng normalisierend:  
Jede Reduktion terminiert und führt zu einer *eindeutigen*  $\beta$ - (bzw.  $\beta\eta$ -) Normalform (modulo  $\alpha$ -Konversion).
- Zwei Terme sind gleich genau dann, wenn ihre Normalformen übereinstimmen (modulo  $\alpha$ -Konversion).

## Getypter Lambda-Kalkül: Normalformen (2)

Die  $\beta\eta$ -Normalform eines Terms hat die allgemeine Gestalt

$$\lambda x_1 : T_1 \dots \lambda x_n : T_n. K(B_1) \dots (B_k)$$

bestehend aus einem *Binderteil*  $\lambda x_1 : T_1 \dots \lambda x_n : T_n. \dots$  und einem *Rumpfteil*  $K(B_1) \dots (B_k)$ , in dem  $K$  ein Konstante oder Variable ist und die Unterterme  $B_j$  jeweils in  $\beta\eta$ -Normalform sind.

Für die Praxis (auch des Beweisens) ist die  $\eta$ -Langform oder  $\eta$ -*Expansion* wichtig: Ein Term in  $\beta$ -Normalform wird so lange  $\eta$ -expandiert, bis der Rumpf einen Basistyp hat.

# Semantik des getypten Lambda-Kalküls

Interpretation der Typen:

- Jedem Basistyp  $T$  wird eine Trägermenge  $\mathcal{M}[T]$  zugeordnet.

$$\mathcal{M}[Bool] := \{W, F\}$$

(Im allgemeinen wird  $\mathcal{M}[T]$  als nicht leer vorausgesetzt.)

- Dem Funktionstyp  $S \rightarrow T$  wird die Menge der Funktion von  $\mathcal{M}[S]$  nach  $\mathcal{M}[T]$  zugeordnet:

$$\mathcal{M}[S \rightarrow T] := \mathcal{M}[T]^{\mathcal{M}[S]}$$

- Dem Produkttyp  $S \times T$  wird das kartesische Produkt der Trägermengen der Komponenten zugeordnet:

$$\mathcal{M}[S \times T] := \mathcal{M}[S] \times \mathcal{M}[T]$$

Die Semantik von Termen wird durch eine Interpretation gegeben, die jedem wohlgetypten Term vom Typ  $S$  ein Element der Menge  $\mathcal{M}[S]$  zuordnet.

## Semantik des getypten Lambda-Kalküls (2)

Für die Interpretation von Termen mit freien Variablen wird außerdem eine Variablenbelegung benötigt:

$$\mathcal{V}_S : V_S \rightarrow \mathcal{M}[S] \quad \text{für jeden Typ } S \text{ (Index meist fortgelassen)}$$

Interpretation von Termen:

- $\mathcal{I}_{\mathcal{M},\mathcal{V}}[v] := \mathcal{V}_S[v]$  für  $v : S$  ( $S$  ergibt sich i.a. aus dem Kontext)
- $\mathcal{I}_{\mathcal{M},\mathcal{V}}[f(t)] := \mathcal{I}_{\mathcal{M},\mathcal{V}}[f](\mathcal{I}_{\mathcal{M},\mathcal{V}}[t])$
- $\mathcal{I}_{\mathcal{M},\mathcal{V}}[\lambda x : S. t] := g$ , wobei für  $d \in \mathcal{M}[S]$   
 $g(d) := \mathcal{I}_{\mathcal{M},\mathcal{V}'}[t]$  mit  $\mathcal{V}' = \mathcal{V}[x := d]$

Diese Struktur ergibt das “Standard-Modell”.

# Bedeutung des Lambda-Kalküls

- Ungetypter Lambda-Kalkül als abstraktes Modell der Berechenbarkeit (entsprechend der These von Church)
- Grundlage vieler (aller?) funktionaler Programmiersprachen, beginnend mit (“reinem”) Lisp (McCarthy)
- Identität von ‘Berechnung’ und ‘Ableitung’ im Kalkül
- Grundlage der “*domain theory*”, die wiederum Grundlage der klassischen *denotationellen Semantik* von Programmiersprachen ist.
- Erweiterungen des (einfach getypten) Lambda-Kalküls
  - Lambda-Kalkül mit polymorphen Typen
  - Lambda-Kalkül höherer Stufe: Abstraktion über Typen  
     $\rightsquigarrow$  diverse *Typentheorien*
- Grundlage (einer Form) der Logik höherer Stufe

## 5.3 Prädikatenlogik höherer Stufe

*Weshalb Logik höherer Stufe?*

### 1. *Volle Charakterisierung von Standard-Modellen*

Axiomatisierung der elementaren Arithmetik über natürlichen Zahlen  $\mathcal{N}_+$  mit Nachfolger  $s$  und Addition  $+$ . Axiomensystem  $AX_1$ :

$$\forall x. \neg(0 = s(x)) \quad (\text{constr})$$

$$\forall x, y. s(x) = s(y) \Rightarrow x = y \quad (\text{inj})$$

$$\forall x. x + 0 = x \quad (\text{add0})$$

$$\forall x, y. x + s(y) = s(x + y) \quad (\text{add1})$$

plus Axiome über Gleichheit

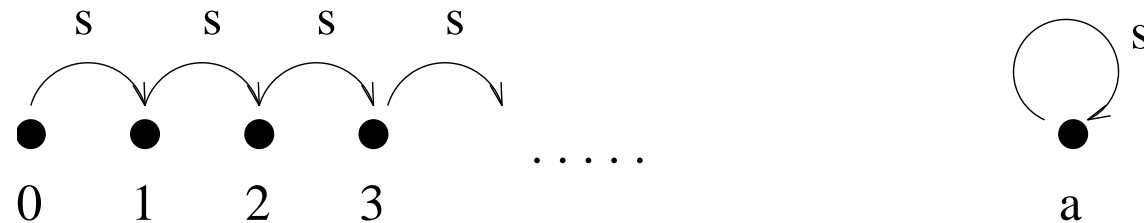
Damit läßt sich etwa ableiten:  $s(0) + s(0) = \dots = s(s(0))$

Frage: Läßt sich auch ableiten  $\forall x. x + s(0) = s(0) + x$  ? (\*)

## Warum Logik höherer Stufe (2)

Die folgende Struktur, bestehend aus  $\mathcal{N}_+$  und einem zusätzlichen Element  $a$  mit den Gleichungen

$$s(a) = a \quad a + x = a \quad x + a = x$$



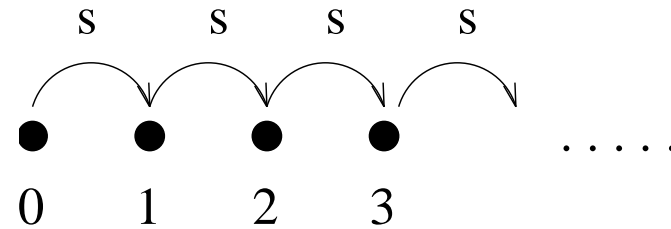
erfüllt die Axiome  $AX_1$ , aber die Formel (\*) ist *nicht* erfüllt.

Schlußfolgerung:  $AX_1$  charakterisiert das gewünschte Modell nicht eindeutig.

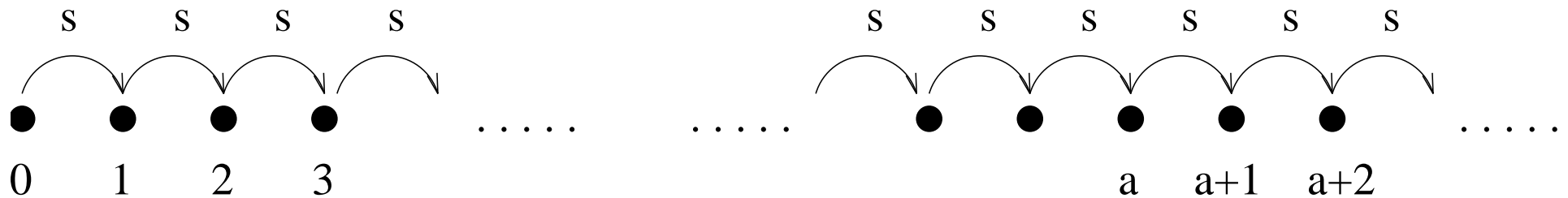
Die volle Charakterisierung des Standard-Modells von  $\mathcal{N}_+$  (eindeutig bis auf Isomorphie) ist in PL1 nicht möglich, selbst wenn das stärkste mögliche "Axiomensystem", die Menge aller in  $\mathcal{N}_+$  gültigen Sätze, gewählt wird.

## Warum Logik höherer Stufe (3)

Standard-Modell:



Nicht-standard-Modelle können durch PL1-Axiome nicht ausgeschlossen werden: in der folgenden Struktur sind dieselben PL1-Aussagen erfüllt wie im Standard-Modell.



Allgemein: In einer PL1-Theorie kann nicht ausgedrückt werden, daß das *kleinste* Modell einer Menge von Formeln gemeint ist.

## Warum Logik höherer Stufe (4)

2. Andere Eigenschaften, die in PL1 nicht axiomatisch formuliert werden können:

- Transitiv Hülle einer Relation (hier wird wiederum eine *kleinste* Menge mit einer bestimmten Eigenschaft gesucht)
- Endliche Strukturen (d.h. die Klasse der endlichen Mengen) lassen sich nicht durch PL1-Formeln axiomatisieren. Generell: mit PL1-Formeln kann man nicht zwischen endlichen und unendlichen Strukturen (als Modellen) unterscheiden.

In der Sprache der Prädikatenlogik 2. Stufe können diese Dinge axiomatisiert werden.

Z.B. Charakterisierung endlicher Mengen als die Menge derjenigen Strukturen, in denen gilt, daß jede injektive Abbildung auch surjektiv ist.

$$\forall f. \text{Injektiv}(f) \Rightarrow \text{Surjektiv}(f)$$

mit

$$\text{Injektiv}(f) := \forall x, y. f(x) = f(y) \Rightarrow x = y$$

$$\text{Surjektiv}(f) := \forall u. \exists v. f(v) = u$$

# Prädikatenlogik höherer Stufe: Syntax

In Prädikatenlogik 1. Stufe (PL1):

- Konstanten und Variable für “Individuen”; Funktions- und Prädikatensymbole bezeichnen Funktions- und Prädikaten*konstanten*.
- Quantifizierung nur über Individuen-Variablen erlaubt:

$$\forall v. \exists w. P(v, w) \wedge Q(w).$$

In Prädikatenlogik 2. Stufe (PL2) auch:

- Variable für Funktionen und Prädikate und Quantifizierung über Funktions- und Prädikaten-Variablen

Beispiel:  $\forall X. \exists Y. X(v, w) \wedge Y(c)$

## Syntax (2):

- Die *Terme* sind die des getypten Lambda-Kalküls, als Erweiterung der Terme der PL1.  
Insbesondere werden Terme auch mit Hilfe von Funktions-*Variablen* gebildet.
- Die *atomaren Formeln* sind einfach applikative Terme des Typs *Bool*.
- Formeln werden wie üblich aus atomaren Formeln und logischen Verknüpfungen gebildet.

Bei quantifizierten Formeln kann über Variablen beliebigen Typs quantifiziert werden.

*Bemerkung:* Streng genommen werden Quantoren über beliebigen Typen nicht benötigt; stattdessen wird für jeden Typ  $S$  eine Prädikat-Konstante  $Q : (S \rightarrow Bool) \rightarrow Bool$  gefordert mit der Bedeutung, daß die Formel  $Q(\lambda X : S. p)$  wahr ist genau dann, wenn  $(\lambda X : S. p)(s)$  für alle  $s : S$  den Wert  $W$  ergibt, d.h. ihre intuitive Bedeutung gerade der von  $\forall X : S. p$  entspricht.

# Beispiele

## Modellierung von Mengen und Mengenoperationen

Es wird ausgenutzt die Äquivalenz von Prädikaten (boolschen Funktionen) und Teilmengen (der Trägermenge eines Typs): Mengen werden durch Prädikate dargestellt.

Operationen auf Mengen: z.B. Inklusion

$$\subseteq := \lambda M : (S \rightarrow Bool). \lambda N : (S \rightarrow Bool). \forall x : S. M(x) \Rightarrow N(x)$$

**Kompakte Formulierung** von Eigenschaften von Funktionen und Prädikaten; z.B.

$$injektiv = \lambda f : T_1 \rightarrow T_2. \forall x, y : T_1. f(x) = f(y) \Rightarrow x = y$$

**Formulierung von Induktionsprinzipien**  $\rightsquigarrow$  kommt später

# Prädikatenlogik höherer Stufe: Semantik

Die Semantik ergibt sich aus der Kombination der Standard-Semantik für Prädikatenlogik erster Stufe (PL1) (Interpretation, Modell) und der Semantik für den getypten Lambda-Kalkül.

Beispiel für Formel  $\psi$ , die die Prädikaten-Variable  $P$  frei enthält:

- $\mathcal{I}[\forall X : S. \psi]$  ist wahr g.d.w.  $\mathcal{I}[\psi]$  wahr ist für alle  $\mathcal{I}[X] \in \mathcal{M}[S]$ .
- $\mathcal{I}[\exists X : S. \psi]$  ist wahr g.d.w.  $\mathcal{I}[\psi]$  wahr ist für mindestens ein  $\mathcal{I}[X] \in \mathcal{M}[S]$ .

*Beispiel* (ohne Typen): Das Universum  $\mathcal{U}$  bestehe aus den Elementen  $a$  und  $b$ .

Die Menge  $U_2$  aller 2-stelligen Relationen über  $\mathcal{U}$  ist die Potenzmenge von  $\{(a, a), (a, b), (b, a), (b, b)\}$ .

Interpretation der Formel

$$\forall v, w. (\exists X. X(v, w)) \Rightarrow (\exists Y. Y(w, v))$$

Ist  $\mathcal{I}[X] = \{(a, a), (a, b)\}$ , so muss sein  $\mathcal{I}[Y] \supseteq \{(a, a), (b, a)\}$ .

# Beweisregeln

Ein Ableitungskalkül für Logik höherer Stufe besteht aus den üblichen Regeln der Prädikatenlogik 1. Stufe und zusätzlichen Regeln für Quantoren höherer Stufe:

$$\frac{\psi[X \leftarrow \sigma], \Gamma \vdash \Delta}{\forall X : S. \psi, \Gamma \vdash \Delta} \forall^\omega \text{L} \qquad \frac{\Gamma \vdash \Delta, \psi[X \leftarrow Y]}{\Gamma \vdash \Delta, \forall X : S. \psi} \forall^\omega \text{R}$$
$$\frac{\psi[X \leftarrow Y], \Gamma \vdash \Delta}{\exists X : S. \psi, \Gamma \vdash \Delta} \exists^\omega \text{L} \qquad \frac{\Gamma \vdash \Delta, \psi[X \leftarrow \sigma]}{\Gamma \vdash \Delta, \exists X : S. \psi} \exists^\omega \text{R}$$

Bemerkung: Beachte die Analogie zu den Quantor-Regeln für PL1.

Hier sind  $X$  und  $Y$  jeweils Prädikatenvariablen;

$\sigma = \lambda x_1, \dots, x_n. R(x_1, \dots, x_n)$  ist ein Prädikatausdruck des richtigen Typs.

Ähnlich wie in PL1 müssen Bedingungen für die Variablen beachtet werden:

- In  $\forall^\omega \text{L}$  und  $\exists^\omega \text{R}$  darf keine Bindung von freien (Individuen- und Prädikaten-) Variablen auftreten.
- Die Variable  $Y$  in  $\forall^\omega \text{R}$  und  $\exists^\omega \text{L}$  darf nicht frei in der Nachbedingung der Regel vorkommen.

# Definition der Gleichheit

In der Logik höherer Stufe läßt sich die Gleichheit auf einem Typ  $S$  definieren:

$$(x = y) := [\forall P : (S \rightarrow Bool). (P(x) \Rightarrow P(y))]$$

Dies ist eine explizite Definition der *Leibniz-Gleichheit*: Zwei Dinge sind gleich, wenn sie in allen Eigenschaften äquivalent sind.

*Bemerkung*: In der Definition genügt die Implikation ( $\Rightarrow$ ) statt Äquivalenz ( $\Leftrightarrow$ ).

**Beispiel-Beweis** für Symmetrie der Gleichheit (ohne Typen):

$$\begin{array}{c} (x = y) \Rightarrow (y = x) \\ \frac{\frac{\frac{\frac{\vdash Y(x) \Rightarrow Y(x) \quad Y(y) \Rightarrow Y(x) \quad \vdash Y(y) \Rightarrow Y(x)}{(\forall X.(X(x) \Rightarrow X(y)) \Rightarrow (\forall X.(X(x) \Rightarrow X(y))) \vdash Y(y) \Rightarrow Y(x))} \Rightarrow L}{\forall X.(X(x) \Rightarrow X(y)) \vdash Y(y) \Rightarrow Y(x)} \forall^\omega L}{\forall X.(X(x) \Rightarrow X(y)) \vdash \forall X.X(y) \Rightarrow X(x)} \forall^\omega R}{\vdash \forall x, y. (\forall X.X(x) \Rightarrow X(y)) \Rightarrow \forall X.X(y) \Rightarrow X(x)} \end{array}$$

Bei der Anwendung der Regel  $\forall^\omega L$  wurde für  $X$  der (einstellige) funktionale Ausdruck  $\sigma := (\lambda z. Y(z) \Rightarrow Y(x))$  eingesetzt.

# Logiken höherer Stufe: Ergänzungen

## Eigenschaften von PL1 und PL2 (1)

Aus welchem Grund existieren in PL1 Nicht-Standard-Modelle?

Für PL1 gilt der **Endlichkeitssatz (Kompaktheitssatz)**:

Für jede Formelmengende  $\Phi$  gilt:  $\Phi$  hat ein Modell gdw. jede endliche Teilmenge von  $\Phi$  ein Modell hat.

(Daraus folgt z.B. auch die Vollständigkeit des Resolutionsverfahrens.)

Daraus läßt sich ableiten: (**“Aufsteigender”**) **Satz von Löwenheim und Skolem**:

Sei  $\Phi$  eine Formelmengende, die über einer unendlichen Menge  $N$  erfüllbar ist. Dann ist  $\Phi$  auch über einer Menge  $K$  erfüllbar mit Mächtigkeit  $|K| > |N|$ .

**Anwendung:**  $N$  die Menge der natürlichen Zahlen,  $K$  ein Nichtstandardmodell.

Lassen sich ähnliche Aussagen in PL2 treffen?

## Eigenschaften von PL1 und PL2 (2)

**Endlichkeitssatz gilt nicht in PL2:** Es gibt eine Formelmengende, die unerfüllbar ist, obwohl jede endliche Teilmenge erfüllbar ist.

*Beweis:* Eine solche Formelmengende ist  $\{\phi_{endl}\} \cup \{\phi_{\geq n} \mid n \geq 2\}$ , wobei

$\phi_{endl} := \forall f : S \rightarrow S. Injektiv(f) \Rightarrow Surjektiv(f)$

$\phi_{\geq 2} := \exists x_1, x_2 : S. x_1 \neq x_2$

$\phi_{\geq n} :=$  "Menge  $S$  hat mindestens  $n$  Elemente" □

Dies ist ein Indiz (noch kein Beweis!) dafür, daß:

- es im allgemeinen nicht möglich ist, die Ungültigkeit einer Aussage durch Angabe einer endlichen widersprüchlichen Formelmengende nachzuweisen.
- der Satz von Löwenheim und Skolem für PL2 nicht gilt.

In der Tat gilt: **Unvollständigkeit von PL2:**

Die Menge der im Standardmodell der natürlichen Zahlen gültigen Sätze ist nicht rekursiv aufzählbar.

## Eigenschaften von PL1 und PL2 (3)

**Charakterisierbarkeit von Kardinalitäten:** Die Menge der natürlichen Zahlen läßt sich bis auf Isomorphie eindeutig charakterisieren.

Z.B. mit den *Peano-Axiomen* (Peano, 1889).

Eines dieser Axiome ist das *Induktionsprinzip* für natürliche Zahlen:

$$\forall P : N \rightarrow Bool.$$

$$(P(0) \wedge \forall x : N.(P(x) \Rightarrow P(s(x)))) \Rightarrow \forall x : N.P(x))$$

Weiterführende Literatur zur Logiken höherer Stufe:

D. Leivant: Higher order logic. In: *Handbook of Logic in AI and LP*, vol. 2, 1994, 229–321.

P.B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

# Zusammenfassung

## Eigenschaften von PL1:

- Mengen lassen sich nicht eindeutig charakterisieren.
- Insbesondere erlaubt die PL1-Formalisierung der natürlichen Zahlen Nicht-Standard-Modelle.
- Im allgemeinen gilt: Hat eine Formelmenge in PL1 ein Modell, so hat sie auch ein Modell größerer Mächtigkeit (Satz von Löwenheim und Skolem).

## Eigenschaften von PL2:

- Charakterisierung von Mengen bis auf Isomorphie möglich.
- Daher: Standardmodell der natürlichen Zahlen in PL2 definierbar.
- Jedoch: Unvollständigkeit von PL2 (plausibel, weil Endlichkeitssatz nicht gilt).
- In PL2 außerdem definierbar: z.B. Gleichheit nach dem Leibniz-Prinzip.

Für PL2 gibt es einen Sequenzen-Kalkül, der den Kalkül für PL1 um Regeln für die Quantoren höherer Ordnung ergänzt.