Rechnergestützte Beweissysteme

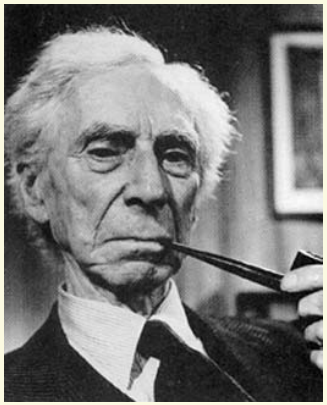
Logik höherer Stufe
λ-Kalkül

Logik höherer Stufe und λ-Kalkül

- Logik höherer Stufe
  - Logik 1. Stufe
  - Logik 2. Stufe
  - Logik beliebiger Stufe
- Funktionsobjekte
  - λ-Terme
  - α-Reduktion
  - β-reduktion
  - η-Regel
- Schlussregeln
  - im Sequenzenkalkül
  - in PVS

Warum nicht „stufenlos“ ?

- Hat man keine Restriktionen an Typen von Sorten bzw. Formeln, so kann man sinnlose Objekte hinschreiben:
  - $\{ P \mid P \notin P \}$ (Russelsche Antinomie)
- Ausweg von Bertrand Russell: Sorten
  - 0.Stufe: Atome: 1, 2, 3, ..., Bob, Hans, rot, blau
  - 1.Stufe: Mengen von Atomen
  - 2. Stufe: Mengen von Objekten 0. oder 1. Stufe
  - n-te Stufe: Mengen von Objekten höchstens (n-1)-er Stufe
- Anderer Ausweg: Zermelo-Fraenkel Mengenlehre
  - Bildung von Mengen durch Operatoren
  - angefangen mit leerer Menge, dann schrittweise
  - Vereinigung, Paarbildung, Produkt, Potenzmenge und „Aussonderung“
- Mengenbildung durch „Aussonderung“ aus einer *bereits existierenden* Menge ***A***:
  - $\{ x \in \mathbf{A} \mid p(x) \}$ p beliebiger logischer Ausdruck



Bertrand Russell

Logik 1. Stufe

- Logik 1. Stufe:
  - Jeweils feste Signatur $\Sigma = (\mathbb{T}, \mathbb{F}, \mathbb{R})$
    - $\mathbb{T}$ Typnamen (τ_i)
    - $\mathbb{F}$ Funktionsnamen mit Funktionssignatur (f_j)
    - $\mathbb{R}$ Relationsnamen mit Relationssignatur (R_k)
  - Quantifikation nur über Objektvariablen erlaubt:
    - $\forall x{:}\tau \,.\, \phi.$
    - $\exists x{:}\tau \,.\, \phi$

    wobei $\tau \in \mathbb{T}$, ϕ eine Formel aus $\mathcal{L}(\Sigma)$

Beispiele und Gegenbeispiele

- Beispielsignatur:

  $\Sigma = (\{nat, bool\}, \{0:nat, s:nat \to nat\}, \{ = : nat \times nat, =: nat \times nat\})$

- In Logik 1. Stufe möglich:
  - $\forall n : nat\ .\ \neg n=0 \to \exists\ y:nat.\ s(y)=n$

- In Logik 1, Stufe nicht möglich:
  - $\forall n:nat.\ \exists\ f:nat \to nat.\ \ f(0) = n \wedge \forall m : nat.\ f(s(m)) = s(f(m-1))$
  - $\forall P:nat \to bool.\ (P(0) \wedge \forall k : nat.\ P(k) \Rightarrow P(k+1)\ ) \Rightarrow \forall k : nat.\ P(k)$
  - $\forall f:nat \to nat.\ \forall\ g:nat \to nat.\ \exists\ h:nat \to nat.\ \forall n:nat.\ h(n)=f(g(n))$
  - Primrec:

    $\forall g:nat \to nat.\ \forall h:nat \to nat \to nat \to nat.\ \exists\ g:nat \to nat.$
    $\forall x,y:nat.$
    $f(0)(y) = g(y) \wedge$
    $f(s(x))(y) = h(x)(y)(g(m)(n))$

Vorteil der Logik 1. Stufe

- Vollständigkeitssatz
  - Γ Menge von Aussagen. Aus $\Gamma \models \varphi$ folgt $\Gamma \vdash \varphi$.
    „Was wahr ist, kann man auch herleiten"
- Kompaktheitssatz
  - Ist Γ eine Menge von Axiomen mit $\Gamma \models \varphi$,

    dann gibt es eine *endliche* Teilmenge $\Gamma_0 \subseteq \Gamma$ mit $\Gamma_0 \models \varphi$.
  - Folgt aus dem Vollständigkeitssatz:
    - Beweise sind endlich, können also auch nur endlich viele Axiome benutzen.

Schwächen der Logik 1. Stufe

- „Endlichkeit" ist nicht ausdrückbar:

  Sei Σ eine Signatur. Es gibt keine Formel $\varphi_{endl} \in \mathcal{L}(\Sigma)$, so dass in jedem Modell S von Σ gilt:

  $$S \models \varphi_{endl} :\Leftrightarrow \text{gdw. S ist endlich.}$$

  - Beweis: Angenommen, φ_{endl} wäre so eine Formel. Betrachte für jede natürliche Zahl n die Formel

    $\varphi_{\geq n} := \exists x_1,\ldots,x_n.\ x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \ \ldots \ \wedge \ x_{n-1} \neq x_n$

    Sei $\Gamma = \{\varphi_{\geq n} | n \in \mathbb{N}\}$. Wenn es eine Formel φ_{endl} gäbe, dann hätte man $\Gamma \models \neg\, \varphi_{endl}$. Wegen des Kompaktheitssatzes müsste es eine endliche Teilmenge $\Gamma_0 \subseteq \Gamma$ geben mit $\Gamma_0 \models \neg\varphi_{endl}$. Widerspruch.

Analog : Transitive Hülle ist nicht ausdrückbar:

Es gibt keine Formel 1. Stufe, die ausdrückt, dass in einem Graph ein Punkt von einem anderen auf einem endlichen Pfad erreichbar ist.

Logik zweiter Stufe

- Feste Signatur $\Sigma = (\mathbb{T}, \mathbb{F}, \mathbb{R})$
  - Aus $\mathbb{T}$ gebildete einfache Funktionstypen erhalten gleiche Rechte wie Objektmengen

    2 (= {0,1} = bool) $\in \mathbb{T}$

    $\tau_1 \rightarrow \tau_2$, wenn τ_1, τ_2, Basistypen

    ,

- Quantifikation über Objekte von einfachen Funktionstypen erlaubt, insbes. auch Quantifikation über Prädikate

  Prädikat P hat Funktionstyp

  $P : \tau \rightarrow \text{bool}$

Logik höherer Stufe

- Higher order logic - HOL
- Feste Signatur $\Sigma = (\mathbb{T}, \mathbb{F}, \mathbb{R})$
  - Beliebige aus $\mathbb{T}$ gebildete Funktionstypen erhalten gleiche Rechte

    $\tau_1, \tau_2, \tau_3 \in \mathbb{T} \Rightarrow$

    $\tau_1 \to (\tau_2 \to \tau_3)$, $(\tau_1 \to \text{bool}) \to (\tau_1 \to \text{bool})$, ... $\in \mathbb{T}$
  - Auch Produkttypen erlaubt (kann man aus Funktionstypen gewinnen)

    $\tau_1 \times \tau_2$, $\tau_1 \times \tau_2 \to \tau_3$, $\tau_1 \times \tau_2 \to$ nat, ...
- Quantifikation über beliebige Funktionstypen erlaubt

Endlichkeit in 2. Stufe ausdrückbar

- Eine Menge ist endlich gdw. jede injektive Selbstabbildung ist surjektiv:
  - $\forall$ **f: S $\rightarrow$ S. injektiv(f) $\rightarrow$ surjektiv(f)**
  - Hierbei:
    - injektiv(f) := $\forall$ x,y:S. f(x)=f(y) $\rightarrow$ x=y (1. Stufe)
    - surjektiv(f) := $\forall$ y:S. $\exists$ x:S. f(x)=y (1. Stufe)
- Endlichkeit in Logik 2. Stufe ausdrückbar !
- Folgerungen :
  - Für die Logik 2. Stufe kann es keinen Kompaktheitssatz geben
  - Für die Logik 2. Stufe kann keinen vollständigen Beweiskalkül geben

Beispiel: Primitive Rekursion

- Beispielsignatur:

  $\Sigma = (\{nat, bool\}, \{0:nat, s:nat \to nat\}, \{= : nat \times nat\})$

- $\forall g:nat \to nat.\ \forall h:nat \times nat \times nat \to nat.\ \exists f:nat \times nat \to nat.$
  $\forall x,n:nat.$
  $f(0,x) = g(x)$
  $\wedge\ f(s(n),x) = h(n,x,f(n,x))$

- Übung:
  - Schreiben Sie in der obigen Signatur (erweitert um $\leq$):
    - „Jede Teilmenge der natürlichen Zahlen hat ein kleinstes Element"
    - „Es gibt unendliche viele gerade Zahlen"
    - „Es gibt eine Funktion $primRec_2$, die aus Abbildungen g,h wie oben eine neue Abbildung $primRec_2(g,h)$ macht"
    - das Gleiche, nur soll g jetzt 2-stellig, h 4-stellig und $primRec_3(g,h)$ 3-stellig sein
  - Geht es auch für beliebiges n: g n-stellig, h (n+2)-stellig und primRec(n,f) jetzt (n+1)-stellig ?

Polymorphe Typen

- Polymorphe Typen sind Typen mit Parametern
  - Beispiele:
    - stack[α] mit α beliebiger Typ
    - array[α,β]
- Erfordert Typvariablen (Java: Generische Typen)
  - α, β, ... stehen für beliebige Typen
  - Damit lassen sich polymorphe Funktions- und Relationszeichen einführen:
    - kleinstes: array[α,nat] $\rightarrow$ nat
- In PVS vorhandene Konstanten mit polymorphem Typ:

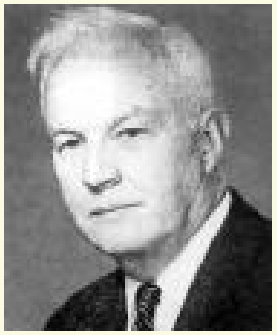
  $=$: $\alpha \times \alpha \rightarrow$ bool

  $\forall$: ($\alpha \rightarrow$ bool) $\rightarrow$ bool

  $\exists$: ($\alpha \rightarrow$ bool) $\rightarrow$ bool

Funktionen

- Seien τ_1 und τ_2 Typen.
  - Was sind Elemente von $\tau_1 \rightarrow \tau_2$?
  - Zwei Möglichkeiten
    - Funktionsdefinition
    - Lambda-Ausdruck



Haskell Curry

- Funktionsdefinition:
  - *myFunc*(x:τ_1) : τ_2 = t
  - t ein Ausdruck vom Typ τ_2, in der nur die Variable x:τ_1 frei vorkommt.

- Beispiel:
  - myFunc(x,y:nat) : nat $\times$ nat = (x div y , x mod y)
  - prim(x:nat) : bool = forall (y:nat): y < x and (x mod y = 0) implies y=0

Anonyme Funktionen

- Da Funktionen jetzt normale Objekte sind, sollten wir sie auch konstruieren können wie andere Objekte

- Eine Funktion aus $[\tau_1 \to \tau_2]$ besteht aus
  - einer Variablen $x : \tau_1$ dem Argument
  - einem Term $t : \tau_2$ dem Körper
  - Der Konstruktor für Funktionen heißt λ.

- $\lambda\, x{:}\tau.t$ ist die Funktion, die einem $a{:}\tau_1$ den Wert $t[x/a]$ zuordnet.
  - Schreibweise in der Mathematik: $x \longmapsto t$

- Beispiele:
  - $\lambda\, x{:}\tau.\ x$ ist die Identität auf τ
  - $\lambda\, x{:}nat.\ x+1$ ist die Nachfolgerfunktion auf nat
  - $\lambda\, x{:}nat.5$ ist die konstante Funktion mit Wert 5
  - $\lambda\, x{:}nat.\ x \bmod 2 = 0$ ist die (charakteristische Funktion der) Menge der geraden Zahlen
  - $\lambda\, x{:}nat.\ s(s(x))$ ist die Abbildung $x \mapsto x+2$
  - $\lambda\, x{:}nat.\ \lambda\, y{:}nat.\ (x + y)/2$ ist die Mittelpunktsfunktion

Syntax – ungetypter Lambda-Kalkül

- Funktionen durch λ-Terme dargestellt

- Abstrakte Syntax für λ-Terme σ:

  var ::= x | y | z | … (Variablen)

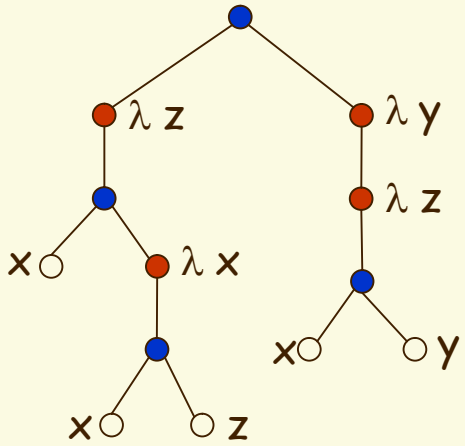
  σ ::= var
  | $\sigma_1\ (\sigma_2)$ (Applikation)
  | λ var.σ (Abstraktion)

  Zusätzliche Klammern dienen der Eindeutigkeit in linearer Darstellung

  σ := (σ)

- Beispiel:

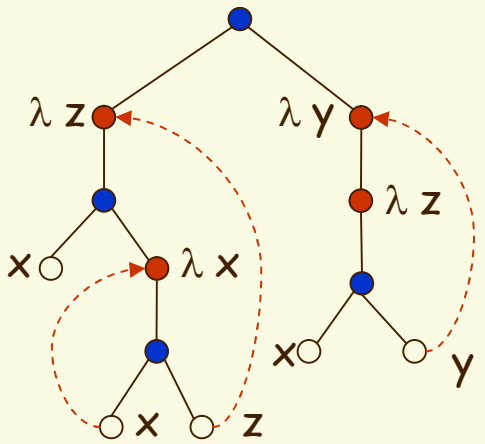
(λ z. x (λ x. x(z))) (λ y. λ z. x y)



Bindung

- λ x.e bindet Variable x im Körper e
- Eine Variable kann in einem λ-Ausdruck mehrfach vorkommen.
  - Ein *Vorkommen der Variablen* v ist frei, wenn es nicht von einem λ v gebunden ist.
  - Beispiel: In
    $(\lambda z.\ x\ (\lambda x.\ x(z))\ )\ (\lambda y.\ \lambda z.\ x\ y)$
    ist das erste und dritte x frei, das zweite gebunden
- Rekursive Definition: *x frei in e :*
  - e Variable v:
    - x frei in v, falls $x \equiv v$
  - $e \equiv e_1(e_2)$:
    - x frei in e_1 oder x frei in e_2
  - $e = \lambda v.e_1$:
    - nicht $x \equiv v$ und x frei in e_1
- Andere Binder (mit analogen Mechanismen) sind:
  - Quantoren : $\forall$ x.P
  - Summen : $\Sigma_{n:N} n^2$
  - Integralen : $\int f(x)\,dx$
  - Methodendefinitionen :

```
int doppelt (int x) {
        return x+x; }
```



Umbenennung von Variablen

- Die Argumentvariable einer Funktion dient nur als Platzhalter
  - Kann konsistent (d.h. in Argument und body) umbenannt werden
  - Die Umbenennung von Variablen nennt man auch α-Reduktion
- **α -Reduktion:** $\lambda\, x{:}\tau.e \;=\; \lambda\, y{:}\tau.e[x/y]$
- Beispiele
  - $\lambda\, x{:}\tau.\, x = \lambda\, y{:}\tau.y$
  - $\lambda\, x{:}nat.s(s(x)) = \lambda\, n{:}nat.s(s(n))$
  - $\lambda\, x{:}nat.\, \lambda\, y{:}nat.\, (x + y)/2 = \lambda\, z{:}nat.\, \lambda\, y{:}nat.\, (z + y)/2$
- Man muss vermeiden,
  - eine vorher nicht gebundene Variable zu binden
    - $\lambda\, y{:}nat.\, (x + y)/2 \neq \lambda\, x{:}nat.\, (x + x)/2$
  - dass die Variable von einem andern λ eingefangen wird
    - $\lambda\, x{:}nat.\, \lambda\, y{:}nat.\, (x + y)/2 \neq \lambda\, y{:}nat.\, \lambda\, y{:}nat.\, (y + y)/2$
- Das kennt man vom Programmieren :
  - ```
    int zahleZinsen (int x){ return x*(1+zinssatz); }
    ≠
    int zahleZinsen (int zinsSatz){ return zinsSatz*(1+zinsSatz); }
    ```
    ```

Anwendung von Funktionen

- Die Anwendung einer Funktion $\lambda\ x{:}\tau.e$ auf ein Argument a besteht darin
  - die Argumentvariable x durch a zu ersetzen
  - diese Ersetzung in e auszuführen
  - diesen Prozess nennt man β-Reduktion
- β-Reduktion: $(\lambda\ x{:}\tau.\ e\ )\ (a) = e[x/a]$
  - Beispiele
    - $(\lambda\ x{:}\tau.\ x)(a) = x[x/a] = a$
    - $(\lambda\ x{:}nat.x+1)(5) = (x+1)[x/5] = 5+1=6$
    - $(\lambda\ x{:}nat.\ x \bmod 2 = 0)(7) \Leftrightarrow (x \bmod 2=0)[x/7] \Leftrightarrow (7 \bmod 2 = 0) \Leftrightarrow$ false
    - $(\lambda\ x{:}nat.\ s(s(x)))(s(y)) = s(s(x))[x/s(y)] = s(s(s(y)))$
    - $(\lambda\ x{:}nat.\ \lambda\ y{:}nat.\ (x + y)/2\ )\ (x+1) = (\lambda\ y{:}nat.\ (x + y)/2\ )[x/x+1] =$ $\lambda\ y{:}nat.(x+1+y)/2$

Gleichheit von Funktionen

- Zwei Funktionen sollen gleich sein, wenn sie auf allen Argumenten übereinstimmen
  - $f:\tau_1 \to \tau_2 = g:\tau_1 \to \tau_2 \quad :\Leftrightarrow \quad \forall x:\tau_1.\ f(x) = g(x)$
  - Die so definierte Gleichheit heißt η-Reduktion
  - Offensichtlich gilt

    für jede Funktion $f:\tau_1 \to \tau_2$

    und jedes Element a: τ_1
    - $(\lambda x:\tau_1.f(x))(a) = f(a)$, also

- η-Reduktion: $\lambda x:\tau.\ f(x) = f$
  - „f ist die Funktion, die einem x den Wert f(x) zuordnet"

Extensionalität

- Die folgenden zwei Regeln sind äquivalent

$$\frac{\Gamma \vdash \Delta,\ \forall x.f(x) = g(x)}{\Gamma \vdash \Delta,\ f = g}$$

Extensionalität

x nicht frei in f, g

$$\frac{}{\Gamma \vdash \Delta,\ \lambda x.e\ x = e}$$

η–Axiom

falls x nicht frei in e

Extensionalitätsregel

- **`(extensionality τ)`**
  - führt ein Extensionalitätslemma für τ in den Beweis ein
  - Beispiel: **`(extensionality "[nat->nat]" )`** erweitert den Antezedenten um die Formel

```
forall (f,g:[nat->nat]):
 (forall(n:nat):f(n)=g(n)) implies f=g))
```

- Eine Funktionsdefinition

$$f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau = e$$

ist syntaktischer Zucker für

$$f : [\tau_1, \ldots, x_n{:}\tau_n \to \tau] = \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$$

daher ist das bereits bekannte **`(expand)`** eine Form der η-Regel.

Extensionalität bei Datentypen

- Datentyp-Definitionen erzeugen automatisch Extensionalitätsregeln
- Beispiel:

```
stack[T:Type]:DATATYPE
Begin
  empty: (emptyStack?)
  push( top:T, pop:stack[T]): (nonEmptyStack?)
End stack
```

erzeugt u.a. das Extensionalitätsxiom

```
stack_push_extensionality:AXIOM
  forall(v1,v2:(nonEmptyStack?)):
    top(v1)=top(v2) & pop(v1)=pop(v2) => v1=v2
```

sowie das Eta-Axiom

```
stack_push_eta :AXIOM
  forall(v:(nonEmptyStack?)):
    push(top(v),pop(v)) = v
```

Regeln für Funktionen

- Auswertung

$$\frac{}{\Gamma \vdash \Delta, (\lambda\ x.\ e)(a) = e[x/a]}$$

β–Axiom

β Regel in PVS

- Der PVS-Befehl **(beta)** wendet die β-Regel auf einen oder mehrere Terme der Form $(\lambda\, x.e)(a)$ an.

- Viele syntaktische Konstrukte verbergen λ-Terme:
  - Eine Menge $\{ x{:}\tau \mid p(x) \}$ ist Syntax für $(\lambda\, x{:}\tau.p(x))$, also
    - $member(u,\{ x{:}\tau \mid p(x) \}) = (\lambda\, x{:}\tau.p(x))\,(u) \rightarrow_\beta p(u)$
  - Ein array[I,V] ist eine Funktion a:[I->V].
    - Eine Array-Referenz a[x] ist nichts anderes als a(x),
    - Ein Array-Update a[e := v] ist Syntax für
      a := λ i:I. if (i=e) then v else a(e)
    - Ein Ausdruck der Form a[e:=v][k] reduziert mittels **(beta)** zu if(k=e)then v else a