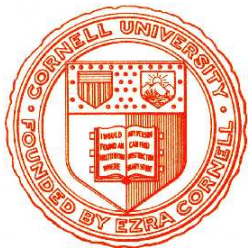
Automatisierte Logik und Programmierung

Einheit 5



Der λ-Kalkül

1. Syntax
2. Operationale Semantik
3. Schließen über Berechnung
4. Ausdruckskraft

Auswertung von Termen

- **Prädikatenlogik und Evidenzen verwenden Termsprache**
  - Prädikatenlogik: uninterpretierte Variablen- und Funktionssymbole
  - Evidenz: Terme der Prädikatenlogik und Konstrukte mit Bedeutung
    $\lambda x.b,\ f(a),\ (a,b),\ x.1, x.2,\ \texttt{inl}(a), \texttt{inr}(b),\ \texttt{case}\ x\ \texttt{of}\ \ldots$
- **Verschiedene Terme können die gleiche Bedeutung haben**
  - Für $f = \lambda x.b$ ist $f(a) = b[a/x]$ und $\lambda x.f(x)$ ist dasselbe wie f
  - $(a,b).1 = a$, $(a,b).2 = b$ und das Paar $(x.1, x.2)$ ist dasselbe wie x
  - $\texttt{case inl}(a)\ \texttt{of inl}(y) \to s \mid \texttt{inr}(z) \to t$ ist dasselbe wie $s[a/y]$
  - Terme auf rechter und linker Seite haben jeweils denselben Wert
- **Termauswertung kann als Kalkül beschrieben werden**
  - Berechnungskalkül interpretiert Terme als Programmiersprache und kalkuliert Wert schrittweise durch Termersetzung (Rewriting)
  - Beweiskalkül beschreibt Effekte der Berechnungsschritte als Gleichheiten in einer Berechnungslogik
  - Die Grundform beider Kalküle ist der λ-Kalkül

Der λ-Kalkül

Logisches Schließen über den Wert von Termen

- **Einfacher mathematischer Mechanismus**
  - Funktionen werden definiert und angewandt
  - Beschreibung des Funktionsverhaltens wird zum Namen der Funktion
  - Funktionswerte werden ausgerechnet durch Einsetzen von Werten
- **Leicht zu verstehende Basiskonzepte**
  1. Definition einer Funktion:

     $\texttt{f} \mathrel{\hat{=}} \lambda\texttt{x.\,2*x+3}$ **λ-Abstraktion**

     Name der Funktion irrelevant für Beschreibung des Verhaltens
  2. Anwendung der Funktion (ohne Auswertung):

     $\texttt{f(4)} \mathrel{\hat{=}} \texttt{(}\lambda\texttt{x.\,2*x+3)(4)}$ **Applikation**
  3. Auswertung einer Funktionsanwendung (tatsächliches Ausrechnen):

     $\texttt{(}\lambda\texttt{x.\,2*x+3)(4)} \xrightarrow{\beta} \texttt{2*4+3} \xrightarrow{*} \texttt{11}$ **Reduktion**
- **Alle anderen Konstrukte können simuliert werden**
  - Turingmächtige funktionale Programmiersprache $\mapsto$ Lisp, ML, Haskell, ...

λ-Kalkül – Syntax

- **Einfache Programmiersprache: λ-Terme**
  - Variablen $x, y, z, x_0, y_0,, \ldots$
  - $\lambda x\,.\,t$, wobei x Variable und t λ-Term **λ-Abstraktion**
    Vorkommen von x in t werden gebunden
  - $f\ t$, wobei t und f λ-Terme **Applikation**
  - (t), wobei t λ-Term
- **Prioritätskonventionen sparen Klammern**
  - Applikation bindet stärker als λ-Abstraktion $\quad \lambda x\,.\,f\ t \mathrel{\hat{=}} \lambda x\,.\,(f\ t)$
  - Applikation ist links-assoziativ: $\quad f\ t_1\ t_2 \mathrel{\hat{=}} (f\ t_1)\ t_2$
- **Beispiele für λ-Terme** (mit impliziten und 'kosmetischen' Klammern)
  - `x` Symbole sind immer Variablen
  - `λf.(λx.(f(x)))` Anwendung einer Funktion
  - `λf.(λg.(λx.(((f g)(g x)))))` Funktionen höherer Ordnung
  - `x(x)` Selbstanwendung
  - `(λx.x(x))(λx.x(x))` Fixpunkt

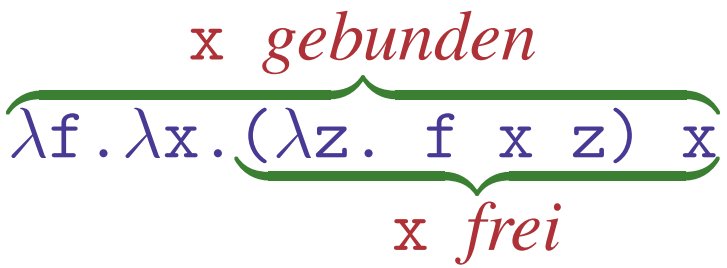
λ-Kalkül – Semantik

- **Interpretation in Zielsprache ist zu kompliziert**
  - Abbildung auf Funktionen in einfachem Universum nicht möglich
  - Modellierung benötigt Theorie vollständiger Halbordnungen (CPOs)
- **Semantik von Termen wird operational erklärt**
  - Entspricht Methodik beim Rechnen auf Zahlen: $4{*}(3{+}2) = 4{*}5 = 20$
  - λ-Terme werden ausgewertet bis irreduzibler Term erreicht ist
  - Irreduzible Terme werden als Werte angesehen
- **Berechnung ist rein syntaktischer Mechanismus**
  - Auswertung ersetzt Funktionsparameter durch Funktionsargumente
  - Formales Konzept: **Reduktion** $(\lambda x\,.\,t)\,(b) \xrightarrow{\beta} t[b/x]$
    Ersetzen der freien Vorkommen der λ-Variablen x durch den λ-Term b
  - Benötigt Erweiterung des Substitutionskonzepts auf λ-Terme

Vorkommen von Variablen in λ-Termen

- **Vorkommen der Variablen x im Term t, informal**
  - Gebunden: x erscheint im Bindungsbereich einer λ-Abstraktion λx
  - Frei: x kommt in t vor, ohne gebunden zu sein
  - t heißt geschlossen falls t keine freien Variablen enthält
  - $t[x_1, .., x_n]$: Term t hat mögliche freie Vorkommen von $x_1, .., x_n$
- **Präzise, induktive Definition**

x	die Variable x kommt frei vor; $y \neq x$ kommt nicht vor
$\lambda x.t$:	beliebige Vorkommen von x in t werden gebunden
	Vorkommen von $y \neq x$ in t bleiben unverändert
$f\ t$	freie Vorkommen von x in t bleiben frei
(t)	gebundene Vorkommen von x bleiben gebunden



Substitution $u[t/x]$ formal

- **Anwendung einer endlichen Abbildung $\sigma = [t/x]$**

$\lfloor x \rfloor [t/x]$	$= t$	$\lfloor x \rfloor [t/y]$	$= x \quad (y \neq x)$
$\lfloor \lambda x \,.\, u \rfloor [t/x]$	$= \lambda x \,.\, u$		
$\lfloor \lambda x \,.\, u \rfloor [t/y]$	$= \lfloor \lambda z \,.\, u[z/x] \rfloor [t/y]$ *	$\lfloor \lambda x \,.\, u \rfloor [t/y]$	$= \lambda x \,.\, \lfloor u[t/y] \rfloor$ **
$\lfloor f \; u \rfloor \sigma$	$= f\sigma \; u\sigma$	$\lfloor (u) \rfloor \sigma$	$= (u\sigma)$

*: $y \neq x$, y *frei in* u, x *frei in* t, z *neue Variable*
**: $y \neq x$, y *nicht frei in* u *oder* x *nicht frei in* t

- **Substitution und Reduktion am Beispiel**

$$
\begin{aligned}
& (\lambda n.\lambda f.\lambda x.\ n\ f\ (f\ x))\,(\lambda f.\lambda x.x) \\
= \; & (\lambda n.(\lambda f.(\lambda x.((n\ f)(f\ x)))))\,(\lambda f.(\lambda x.x)) && \textit{(Reduktion ersetzt n)} \\
\longrightarrow \; & \lambda f.(\lambda x.((\lambda f.(\lambda x.x))\ f)(f\ x)) && \textit{(8 Substitutionsschritte)} \\
\longrightarrow \; & \lambda f.(\lambda x.((\lambda x.x)(f\ x))) \\
\longrightarrow \; & \lambda f.\lambda x.\ f\ x
\end{aligned}
$$

$$
\Downarrow
$$

$$
(\lambda n.\lambda f.\lambda x.\ n\ f\ (f\ x))\,(\lambda f.\lambda x.x) \xrightarrow{3} \lambda f.\lambda x.\ f\ x
$$

Reduktion als Formalismus – wichtige Begriffe

- **α-Konversion:**
  - Umbenennung gebundener Variablen in Termen (erhält Bedeutung)
  - Ersetze Teilterms der Gestalt $\lambda x\,.\,u$ durch $\lambda z\,.\,u[z/x]$ (z neue Variable)
- **Kongruenz $t \cong u$ ("t und u sind α-konvertibel")**
  - u entsteht aus t durch α-Konversion
- **Reduktion $(\lambda x\,.\,u)\ s \xrightarrow{\beta} u[s/x]$:**
  - Ersetze Teilterm der Gestalt $\underbrace{(\lambda x\,.\,u)\ s}_{\text{Redex}}$ durch $\underbrace{u[s/x]}_{\text{Kontraktum}}$
- **Reduzierbarkeit $t \xrightarrow{*} u$:**
  - u entsteht aus t durch endlich viele Reduktionen und α-Konversionen
- **Normalform:**
  - Term u, der nicht weiter reduzierbar (irreduzibel) ist
  - Wert von t: Normalform u mit $t \xrightarrow{*} u$

Regeln für Konversion und Reduktion

- **α-Konversion: $\lambda x\,.\,u \cong \lambda z\,.\,u[z/x]$** ($z$ neu)
  - μ-Konversion: $f\ t \cong f\ u$, falls $t \cong u$
  - ν-Konversion: $f\ t \cong g\ t$, falls $f \cong g$
  - ξ-Konversion: $\lambda x\,.\,t \cong \lambda x\,.\,u$, falls $t \cong u$
  - ρ -Konversion: $t \cong t$
  - σ-Konversion: $t \cong u$, falls $u \cong t$
  - τ-Konversion: $t \cong u$, falls $t \cong s$ und $s \cong u$
- **β-Reduktion: $(\lambda x\,.\,u)\ s \longrightarrow u[s/x]$**
  - μ–Reduktion: $f\ t \longrightarrow f\ u$, falls $t \longrightarrow u$
  - ν–Reduktion: $f\ t \longrightarrow g\ t$, falls $f \longrightarrow g$
  - ξ–Reduktion: $\lambda x\,.\,t \longrightarrow \lambda x.u$, falls $t \longrightarrow u$
  - τ–Reduktion: $t \longrightarrow u$, falls $t \longrightarrow s$ und $s \cong u$ oder $t \cong s$ und $s \longrightarrow u$
- **Reduzierbarkeit: $t \xrightarrow{*} u$,** falls $t \xrightarrow{n} u$ für ein $n \in \mathbb{N}$
  - $t \xrightarrow{0} u$, falls $t \cong u$
  - $t \xrightarrow{n+1} u$, falls $t \longrightarrow s$ und $s \xrightarrow{n} u$

Reduktion ist nicht deterministisch

Naheliegende Reduktionsreihenfolge

$$
\begin{aligned}
& (\lambda f.\lambda x.\ f\ x\ (f\ f))\ (\lambda x.\lambda y.\ y) \\
\longrightarrow\ & \lambda x.\ (\lambda x.\lambda y.\ y)\ x\ ((\lambda x.\lambda y.\ y)\ (\lambda x.\lambda y.\ y)) \\
\longrightarrow\ & \lambda x.\ (\lambda y.\ y)\ ((\lambda x.\lambda y.\ y)\ (\lambda x.\lambda y.\ y)) \\
\longrightarrow\ & \lambda x.\ ((\lambda x.\lambda y.\ y)\ (\lambda x.\lambda y.\ y) \\
\longrightarrow\ & \lambda x.\ \lambda y.\ y
\end{aligned}
$$

Alternative Reduktionsreihenfolgen sind möglich

$$
\begin{aligned}
& (\lambda f.\lambda x.\ f\ x\ (f\ f))\ (\lambda x.\lambda y.\ y) \\
\longrightarrow\ & \lambda x.\ (\lambda x.\lambda y.\ y)\ x\ ((\lambda x.\lambda y.\ y)\ (\lambda x.\lambda y.\ y)) \\
\longrightarrow\ & \lambda x.\ (\lambda x.\lambda y.\ y)\ x\ (\lambda y.\ y) \\
\longrightarrow\ & \lambda x.\ (\lambda y.\ y)\ (\lambda y.\ y) \\
\longrightarrow\ & \lambda x.\ \lambda y.\ y
\end{aligned}
$$

Bekommt man immer das gleiche Ergebnis?

Semantik von λ-Termen präzisiert

- **Bedeutung von λ-Termen ist ihr Wert**
  - **Normalform**: Term ohne keine Redizes als Teilterme
  - **u Normalform von t**: u in Normalform und $t \xrightarrow{*} u$
  - **t normalisierbar**: es gibt eine Normalform u von t
- **Hat jeder λ-Term eine Normalform?**

  Nein: $(\lambda x.\, x\ x)\,(\lambda x.\, x\ x)$ ist nicht normalisierbar
  - Terminierung von λ-Programmen ist nicht garantiert
- **Haben normalisierbare λ-Terme eindeutige Werte?**

  Notwendig für Einsatz des λ-Kalküls als Programmiersprache
  - Führt jede Reduktionsfolge zu einer Normalform?

    Wenn nein, kann man eine Normalform systematisch finden?
  - Ist die Normalform eines λ-Terms eindeutig?

Reduktion normalisierbarer λ-Terme

- **Führt jede Reduktionsfolge zu einer Normalform?**

  Nein: Reduktionsfolgen normalisierbarer Terme müssen nicht terminieren

  $$(\lambda x.\lambda y.y)\ ((\lambda x.xxx)\ (\lambda x.xxx))\ (\lambda x.x)$$
  $$\longrightarrow (\lambda y.y)\ (\lambda x.x)$$
  $$\longrightarrow \lambda x.x$$

  Eine "innermost" Strategie führt bei diesem Term nicht zur Normalform

  $$(\lambda x.\lambda y.y)\ ((\lambda x.xxx)\ (\lambda x.xxx))\ (\lambda x.x)$$
  $$\overset{*}{\longrightarrow} (\lambda x.\lambda y.y)((\lambda x.xxx)\ (\lambda x.xxx)\ (\lambda x.xxx))\ (\lambda x.x)$$
  $$\overset{*}{\longrightarrow} (\lambda x.\lambda y.y)((\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx))\ (\lambda x.x)$$
  $$\vdots \qquad\qquad \vdots$$

- **Kann man eine Normalform immer finden?**

  – **Ja**: Reduktion des äußersten Redex ("leftmost reduction") führt zur Normalform, wenn es eine gibt

  Beweis: Curry & Feys 1958, p142

Eindeutigkeit von Normalformen

- **Was heißt "eindeutig"?**
  - Müssen Normalformen textlich identisch sein (wie $4 = 4$)?
  - Oder sollen sie "nur" den gleichen Wert haben (wie $\lambda x.x$ und $\lambda y.y$)?
  - Gleichwertigkeit zu fordern ist semantisch sinnvoller
- **Semantische Gleichheit von λ-Termen ist extensional**
  - $t = u$: es gibt v mit $t \xrightarrow{*} v$ und $u \xrightarrow{*} v$ (t und u sind konvertierbar)
  - Auch Terme, die nicht normalisierbar sind, können gleich sein
- **Ist die Normalform eines λ-Terms eindeutig?**
  - **Ja**: Reduktion ist konfluent — Beweis: Barendregt 1981 §3.2

    Gilt $t \xrightarrow{*} u$ und $t \xrightarrow{*} v$, so folgt $u = v$ (**Church-Rosser Theorem**)
  - Alle Normalformen eines λ-Terms sind kongruent und können (durch Konversionen) auf denselben Term reduziert werden
  - Semantisch gleiche Terme haben dieselben Normalformen oder keine
  - Normalformen, die nicht kongruent sind, sind nicht semantisch gleich

Semantische Gleichheit

- **Ausdrucksstärker als Reduzierbarkeit**
  - Benötigt keine Normalform für den Vergleich von Termen
    Es reicht, mit Reduktionen/Konversionen den gleichen Term zu erreichen
  - Auch Terme, die nicht normalisierbar sind, können verglichen werden
- **Grundlage des logischen Schließens über Berechnungen**
  - Im (allgemeinen) λ-Kalkül kann automatische Normalisierung zu nicht-terminierenden Reduktionsfolgen führen
    Für typisierbare λ-Termen gibt es immer eine terminierende Reduktionsstrategie
  - Schließen über semantische Gleichheit braucht terminierende Reduktion
  - Beweisregeln müssen einzelne Reduktionsschritte widerspiegeln
- **Der λ-Kalkül ist unabhängig von der Prädikatenlogik**
  - Schließen über semantische Gleichheit benötigt keine logischen Regeln
  - Beweissysteme können separat beschrieben und dann integriert werden

Regeln zum Schliessen über Gleichheit

- **β-Reduktion**
  - Gleichheit mit einem Redex entspricht Gleichheit mit Kontraktum

$$\frac{H \vdash u[s/x] = t}{H \vdash (\lambda x.u)\, s = t}\ \texttt{reduction}$$

- **Kongruenzregeln**
  - Applikationen sind gleich, wenn Funktionen und Argumente gleich sind
  - Funktionen sind gleich, wenn sie auf allen Argumenten gleich sind

$$\frac{H \vdash f = g \quad H \vdash t = u}{H \vdash f\, t = g\, u}\ \texttt{applyEq}$$

$$\frac{H, x' : \mathbb{U} \vdash t[x'/x] = u[x'/y]}{H \vdash \lambda x.t = \lambda y.u}\ \texttt{lambdaEq}$$

x' ist neue Variable, die in H nicht frei vorkommt

- **Reflexivität, Symmetrie, Transitivität, Substitution**

$$\frac{}{H \vdash t = t}\ \texttt{reflexivity}$$

$$\frac{H \vdash t = u}{H \vdash u = t}\ \texttt{symmetry}$$

$$\frac{H \vdash t = u \quad H \vdash t = s}{H \vdash s = u}\ \texttt{transitivity}\ s$$

$$\frac{H \vdash C[t/x] \quad H \vdash t = u}{H \vdash C[u/x]}\ \texttt{subst}\ t{=}u$$

Sequenzenbeweis für eine Reduktion

- **Beweis einer einfachen Reduktion**

$\vdash (\lambda f.\lambda x.fx(ff))(\lambda x.\lambda y.y) = \lambda x.\lambda y.y$	BY reduction
1. $\vdash \lambda x.(\lambda x.\lambda y.y)x((\lambda x.\lambda y.y)(\lambda x.\lambda y.y)) = \lambda x.\lambda y.y$	BY lambdaEq
1.1. $x{:}\mathbb{U} \vdash (\lambda x.\lambda y.y)x((\lambda x.\lambda y.y)(\lambda x.\lambda y.y)) = \lambda y.y$	BY reduction
1.1.1. $x{:}\mathbb{U} \vdash (\lambda y.y)((\lambda x.\lambda y.y)(\lambda x.\lambda y.y)) = \lambda y.y$	BY reduction
1.1.1.1. $x{:}\mathbb{U} \vdash (\lambda x.\lambda y.y)(\lambda x.\lambda y.y) = \lambda y.y$	BY reduction
1.1.1.1.1. $x{:}\mathbb{U} \vdash \lambda y.y = \lambda y.y$	BY reflexivity

- **Beweis einer semantischen Gleichheit**

$\vdash (\lambda f.\lambda x.f\,x)(\lambda y.y) = (\lambda x.\lambda y.y)(\lambda z.z)$	BY reduction
1. $\vdash \lambda x.(\lambda y.y)\,x = (\lambda x.\lambda y.y)(\lambda z.z)$	BY symmetry
1.1. $\vdash (\lambda x.\lambda y.y)(\lambda z.z) = \lambda x.(\lambda y.y)\,x$	BY reduction
1.1.1. $\vdash \lambda y.y = \lambda x.(\lambda y.y)\,x$	BY lambdaEq
1.1.1.1. $x{:}\mathbb{U} \vdash x = (\lambda y.y)\,x$	BY symmetry
1.1.1.1.1. $x{:}\mathbb{U} \vdash (\lambda y.y)\,x = x$	BY reduction
1.1.1.1.1.1. $x{:}\mathbb{U} \vdash x = x$	BY reflexivity

Eigenschaften des Beweiskalküls

- **Korrektheit**
  - Wenn $\vdash u{=}v$ im erweiterten Sequenzenkalkül bewiesen werden kann, dann sind u und v semantisch gleich ($u = v$)
  - *Die neuen Regeln sind korrekt in Bezug auf semantische Gleichheit*
- **Vollständigkeit gilt nur in eingeschränktem Sinne**

  Jede (konkrete) Reduktionsfolge $u \xrightarrow{*} t$ und $v \xrightarrow{*} t$ kann in eine Folge von Beweisschritten für $\vdash u{=}v$ übertragen werden
  - β-Reduktion wird von `reduction` abgedeckt
  - α- und ξ-Konversionen und -Reduktionen werden von `lambdaEq` erfaßt
  - μ- und ν-Konversionen und -Reduktionen werden von `applyEq` erfaßt
  - `reflexivity`, `symmetry`, `transitivity`, `subst` decken den Rest ab

  Achtung! Es folgt nicht: "Gilt $u = v$, so ist $\vdash u{=}v$ beweisbar" (siehe Folie 29)
- **Kalkül kann auch definitorische Gleichheit verarbeiten**
  - Sind Konzepte durch definitorische Abkürzung erklärt, so ersetzt die Regel `unfold` die linke Seite der Definition durch die rechte

Vom λ-Kalkül zu echten Programmen

- **λ-Kalkül ist der Basismechanismus**
  - Die Assemblersprache funktionaler Programme
  - Nur die elementarsten Konstrukte sind vordefiniert
- **Programm- und Datenstrukturen werden codiert**
  - Berechnung auf λ-Ausdrücken muß Effekte auf Struktur simulieren
    (Analog zu konventionellen Computern, in denen alles als Bitmuster codiert wird)
- **Alle wichtigen Strukturen sind leicht codierbar**
  - Boolesche Operationen: T, F, $\mathsf{if}\ \boldsymbol{b}\ \mathsf{then}\ \boldsymbol{s}\ \mathsf{else}\ \boldsymbol{t}$
  - Tupel / Projektionen: $\langle \boldsymbol{s}, \boldsymbol{t} \rangle$, $\boldsymbol{p}.1$, $\boldsymbol{p}.2$
  - Injektion / Analyse: $\mathtt{inl}(a)$, $\mathtt{inr}(b)$, $\mathtt{case}\, x\, \mathtt{of}\, \mathtt{inl}(a) \to s \mid \mathtt{inr}(b) \to t$
  - Zahlen und arithmetische Operationen
  - Iteration oder Rekursion von Funktionen

Der λ-Kalkül kann alle berechenbaren Funktionen repräsentieren

Darstellung Boolescher Operatoren im λ-Kalkül

Wir brauchen zwei verschiedene Objekte und einen Test

T	$\equiv$	$\lambda x.\lambda y.x$	*Term für "true"*
F	$\equiv$	$\lambda x.\lambda y.y$	*Term für "false"*
$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$	$\equiv$	$b\,s\,t$	*Term für Konditional*

Beweis: Die Konditional(-darstellung) ist invers zu T und F

$\vdash \mathsf{if}\ \mathsf{T}\ \mathsf{then}\ s\ \mathsf{else}\ t = s$	BY unfold cond	$\vdash \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ s\ \mathsf{else}\ t = t$
$\vdash \mathsf{T}\,s\,t = s$	BY unfold T/F	$\vdash \mathsf{F}\,s\,t = t$
$\vdash (\lambda x.\lambda y.x)\,s\,t = s$	BY reduction	$\vdash (\lambda x.\lambda y.y)\,s\,t = t$
$\vdash (\lambda y.s)\,t = s$	BY reduction	$\vdash (\lambda y.y)\,t = t$
$\vdash s = s$	BY reflexivity	$\vdash t = t$

"Beweis" für $\mathsf{F} \neq \mathsf{T}$

Beweis zeigt "nur" die Absurdität von $\mathsf{F} = \mathsf{T}$

– Es ist nicht möglich, $\mathsf{F} \neq \mathsf{T}$ zu beweisen, aber wenn $\mathsf{F} = \mathsf{T}$ wäre, dann sind alle λ-Terme gleich

$\vdash \mathsf{F}=\mathsf{T} \Rightarrow \forall s,t:U.\ s = t$	BY impliesR THEN allR THEN allR
1. $\mathsf{F}=\mathsf{T}$, s:$\mathbb{U}$, t:$\mathbb{U} \vdash$ s = t	BY transitivity if F then s else t
1.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash$ s = if F then s else t	BY symmetry
1.1.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash$ if F then s else t = s	BY subst F=T
1.1.1.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash \mathsf{F} = \mathsf{T}$	BY axiom
1.1.1.2. $\mathsf{F}=\mathsf{T}, \ldots \vdash$ if T then s else t = s	BY unfold THEN unfold
1.1.1.2.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash (\lambda x.\lambda y.x)$ s t = s	BY reduction THEN reduction
1.1.1.2.1.1. $\ldots \vdash$ s = s	BY reflexivity
1.2. $\mathsf{F}=\mathsf{T}, \ldots \vdash$ if F then s else t = t	BY unfold THEN unfold
1.2.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash (\lambda x.\lambda y.y)$ s t = t	BY reduction THEN reduction
1.2.1.1. $\mathsf{F}=\mathsf{T}, \ldots \vdash$ t = t	BY reflexivity

– Beweis auch ohne Verwendung der subst Regel möglich, aber länger

Bildung und Analyse von Paaren

$$\langle u\,,v\rangle \equiv \lambda \mathtt{p}.\ \mathtt{p}\ u\ v$$

$$pair.1 \equiv pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x})$$

$$pair.2 \equiv pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})$$

$$\mathsf{match}\ pair\ \mathsf{with}\ \langle x\,,y\rangle \mapsto t \equiv pair\ (\lambda x.\lambda y.t)$$

(uniformer Analyseoperator, oft eleganter in Anwendung)

Der Analyseoperator ist invers zur Paarbildung

$$\begin{aligned}
& \mathsf{match}\ \langle u\,,v\rangle\ \mathsf{with}\ \langle x\,,y\rangle \mapsto t \\
\equiv\ & \langle u\,,v\rangle(\lambda x.\lambda y.t) \\
\equiv\ & (\lambda \mathtt{p}.\ \mathtt{p}\ u\ v)(\lambda x.\lambda y.t) \\
\longrightarrow\ & (\lambda x.\lambda y.t)\ u\ v \\
\longrightarrow\ & (\lambda y.t[u/x])\ v \\
\longrightarrow\ & t[u,v/x,y] \qquad \textit{(kurz für } t[u/x][v/y])
\end{aligned}$$

Verallgemeinerung boolescher Operatoren

$$\texttt{inl}(a) \equiv \lambda \texttt{x}.\lambda \texttt{y}.\texttt{x}\ a$$

$$\texttt{inr}(b) \equiv \lambda \texttt{x}.\lambda \texttt{y}.\texttt{y}\ b$$

$$\texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \to s \mid \texttt{inr}(y) \to t \equiv z\,(\lambda x.s)\,(\lambda y.t)$$

Fallanalyse ist invers zu Injektionen

$$\begin{aligned}
& \texttt{case}\ \texttt{inl}(a)\ \texttt{of}\ \texttt{inl}(x) \to s \mid \texttt{inr}(y) \to t \\
\equiv\ & (\texttt{inl}(a))\,(\lambda x.s)\,(\lambda y.t) \\
\equiv\ & (\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x}\ a)\,(\lambda x.s)\,(\lambda y.t) \\
\longrightarrow\ & (\lambda \texttt{y}.(\lambda x.s)\ a)\,(\lambda y.t) \\
\longrightarrow\ & (\lambda x.s)\ a) \\
\longrightarrow\ & s[a/x]
\end{aligned}$$

Analog: $\texttt{case}\ \texttt{inr}(b)\ \texttt{of}\ \texttt{inl}(x) \to s \mid \texttt{inr}(y) \to t \xrightarrow{*} t[b/x]$

Simulation von Zahlen und Arithmetik

- **Darstellung natürlicher Zahlen durch iterierte Terme**
  - Semantisch: wiederholte Anwendung von Funktionen
  - Repräsentiere die Zahl n durch den λ-Term $\lambda \mathtt{f}\,.\,\lambda \mathtt{x}\,.\,\underbrace{\mathtt{f}\,(\mathtt{f}\,..(\mathtt{f}\,\mathtt{x})..)}_{n\text{-}mal}$
  - Notation: $\overline{n} \equiv \lambda \mathtt{f}\,.\,\lambda \mathtt{x}\,.\,\mathtt{f}^n\,\mathtt{x}$

    (Markierung $\overline{n}$ hilft, eine Verwechselung von Zahlen und zugehörigen λ-Termen zu vermeiden)
  - Bezeichnung: **Church Numerals**
- **$f{:}\mathbb{N}^n \to \mathbb{N}$ λ-berechenbar:**
  - Es gibt einen λ-Term t mit der Eigenschaft

$$f(x_1,..,x_n) = m \;\Leftrightarrow\; t\;\overline{x_1}\,..\overline{x_n} = \overline{m}$$

- **Operationen müssen "Termvielfachheit" berechnen**
  - Simulation einer Funktion auf Darstellung von Zahlen muß Darstellung des Funktionsergebnisses liefern
  - z.B. muß add $\overline{m}$ $\overline{n}$ als Wert immer den Term $\overline{m+n}$ liefern

Arithmetische Operationen im λ-Kalkül

- **Nachfolgerfunktion:** $\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x})$
  - Der Wert von $\mathsf{s}\ \overline{n}$ ist der Term $\overline{n+1}$

$$
\begin{aligned}
\mathsf{s}\ \overline{n} \quad &\equiv \quad (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x}))\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}) \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ ((\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f}\ \mathtt{x}) \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ ((\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{x}) \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{f}^n\ \mathtt{x}) \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{n+1}\ \mathtt{x} \qquad \equiv \overline{n+1}
\end{aligned}
$$

- **Addition:** $\mathsf{add} \equiv \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m}\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x})$
- **Multiplikation:** $\mathsf{mul} \equiv \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m}\ (\mathtt{n}\ \mathtt{f})\ \mathtt{x}$
- **Test auf Null:** $\mathsf{zero} \equiv \lambda \mathtt{n}.\ \mathtt{n}\ (\lambda \mathtt{n}.\mathsf{F})\ \mathsf{T}$
- **Vorgängerfunktion:**

  $\mathsf{p} \equiv \lambda \mathtt{n}.\ (\mathtt{n}\ (\lambda \mathtt{f\_x}.\ \langle \mathsf{s}, \mathsf{match}\ \mathtt{f\_x}\ \mathsf{with}\ \langle \mathtt{f}, \mathtt{x} \rangle \mapsto \mathtt{f}\ \mathtt{x} \rangle)\ \langle \lambda \mathtt{z}.\overline{0},\ \overline{0} \rangle).2$

- **Einfache Rekursion:** $\mathsf{PRs}[b,h] \equiv \lambda \mathtt{n}.\ \mathtt{n}\ h\ b$
  - Für $f \equiv \mathsf{PRs}[b,h]$ gilt: $f\,\overline{0} = b$ und $f\,(s\,n) = h\,(f\,n)$

Korrektheit der Additionsfunktion

Beweise: add $\overline{m}$ $\overline{n}$ = $\overline{m+n}$

$\vdash$ add $\overline{m}$ $\overline{n}$ = $\overline{m+n}$	BY unfold
$\vdash$ (λm.λn.λf.λx. m f (n f x)) $\overline{m}$ $\overline{n}$ = $\overline{m+n}$	BY reduction
$\vdash$ (λn.λf.λx. $\overline{m}$ f (n f x)) $\overline{n}$ = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. $\overline{m}$ f ($\overline{n}$ f x) = $\overline{m+n}$	BY unfold
$\vdash$ λf.λx. (λf.λx. f^m x) f ($\overline{n}$ f x) = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. (λx. f^m x) ($\overline{n}$ f x) = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. f^m ($\overline{n}$ f x) = $\overline{m+n}$	BY unfold
$\vdash$ λf.λx. f^m ((λf.λx. f^n x) f x) = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. f^m ((λx. f^n x) x) = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. f^m (f^n x) = $\overline{m+n}$	BY reduction
$\vdash$ λf.λx. f^{m+n} x = $\overline{m+n}$	BY unfold
$\vdash$ λf.λx. f^{m+n} x = λf.λx. f^{m+n} x	BY reflexivity

Keine Angabe der Unterzieladresse im Beweisbaum, da Beweis linear ist

Beweis ist “generisch”. Konkreter Beweis nur mit konkreter Instanz für m und n

Darstellung von Listen

- **Erweiterung des Konzepts natürlicher Zahlen**
  - Wiederholte Anwendung einer Funktion auf Listenelemente
  - Repräsentiere $a_1..a_n$ durch den Term $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ a_1\ (\mathtt{f}..(\mathtt{f}\ a_n\ \mathtt{x})..)$
- **Grundkonzepte: Leere Liste, Anhängen, Rekursion**

$$\begin{aligned} [] &\equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x} \\ t::list &\equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}) \\ \mathsf{list\_ind}\,[b,h] &\equiv \lambda \mathtt{list}.\ \mathtt{list}\ h\ b \end{aligned}$$

- **Rekursionsgleichungen für** $f \equiv \mathsf{list\_ind}\,[b,h]$

$\vdash f\ [] = b$	BY unfold
$\vdash (\lambda \mathtt{list}.\ \mathtt{list}\ h\ b)\ [] = b$	BY reduce
$\vdash []\ h\ b = b$	BY unfold
$\vdash (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ h\ b = b$	BY reduce THEN reduce THEN reflexivity

$\vdash f\ (t::list) = h\ t\ (f\ list)$	BY unfold THEN reduce
$\vdash t::list\ h\ b = h\ t\ (f\ list)$	BY unfold
$\vdash (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}))\ h\ b = h\ t\ (f\ list)$	BY reduce THEN reduce
$\vdash h\ t\ (list\ h\ b) = h\ t\ (f\ list)$	BY symmetry THEN unfold
$\vdash h\ t\ ((\lambda \mathtt{list}.\ \mathtt{list}\ h\ b)\ list) = h\ t\ (list\ h\ b)$	BY reduce THEN reflexivity

Programmierung rekursiver Funktionen

- **Funktion, die durch Gleichung $f(x)=t[f,x]$ definiert ist**
  - d.h. im Programmkörper t darf f sich selbst und x aufrufen
- **Darstellung: Anwendung eines Fixpunktkombinators auf t**
  - Fixpunktkombinator: λ-Term R mit Eigenschaft $R\,t = t\,(R\,t)$ für alle t
  - Rekursive Funktion f ist implementierbar durch $f \equiv R\ (\lambda f\,.\,\lambda x\,.\,t)$, denn es gilt $f\,x = (R\,(\lambda f.\lambda x.t))\,x = (\lambda f.\lambda x.t)\ f\,x \xrightarrow{*} t[f,x]$
  - Programmiernotation für $R(\lambda f\,.\,\lambda x\,.\,t)$ ist "function $f(x) = t$" Notation "function $f(x,y) = t$" steht für $R(\lambda f\,.\,\lambda x\,.\,\lambda y\,.\,t)$
- **Bekanntester Fixpunktkombinator ist der Y-Kombinator**
  - $\mathsf{Y} \equiv \lambda\mathtt{f}\,.\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))$

```
⊢ Y t = t (Y t)                                          BY unfold THEN reduction
⊢ (λx. t (x x)) (λx. t (x x)) = t (Y t)                  BY reduction
⊢ t (λx. t (x x)) (λx. t (x x)) = t (Y t)                BY symmetry
⊢ t (Y t) = t (λx. t (x x)) (λx. t (x x))                BY applyEq
1.⊢ t = t                                                BY reflexivity
2.⊢ Y t = (λx. t (x x)) (λx. t (x x))                    BY unfold THEN reduction
2.1.⊢ (λx. t (x x)) (λx. t (x x)) = (λx. t (x x)) (λx. t (x x))  BY reflexivity
```

Beispiele rekursiv programmierter Funktionen

- **Fakultätsfunktion:** *Es ist* $0! = 1$ *und* $n! = n * (n-1)!$ *für* $n>0$
  - `fak` $\equiv$ `function fak(n) = if zero(n) then` $\overline{1}$ `else mul n (fak(p n))`
    $\equiv$ `Y ( λfak.λn. if zero(n) then` $\overline{1}$ `else mul n (fak(p n)) )`
- **Subtraktion:** $n-m = n$, *falls* $m=0$, *sonst* $(n-(m-1))-1$
  - `sub` $\equiv$ `function sub(n,m) = if zero(m) then n else p(sub n (p m))`
- **Test** $n<m$**:** *Es ist* $n<m$ *genau dann, wenn* $n+1-m = 0$
  - `less` $\equiv$ `λn.λm. zero(sub (s n) m)`
- **Division:** *Suche das erste* z *mit* $n<(z+1) * m$). *Starte Suche bei 0*
  - `div` $\equiv$ `λn.λm.(function search(z) =`
    `if less n (mul z m) then p z else search(s z))` $\overline{0}$
- **Ackermannfunktion:**
  - `A` $\equiv$ `function A(n,x) =`
    `if zero(x) then` $\overline{1}$
    `else if zero(n) then if zero(p x) then` $\overline{2}$ `else add x` $\overline{2}$
    `else A (p n) (A n (p x))`

Ausdruckskraft des λ-Kalküls

Alle μ-rekursiven Funktionen sind λ-berechenbar

- **Nachfolgerfunktion s:** $\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x})$
- **Projektionsfunktionen pr_m^n:** $\mathsf{pr}_m^n \equiv \lambda \mathtt{x}_1 . . \lambda \mathtt{x}_n .\ \mathtt{x}_m$
- **Konstantenfunktion c_m^n:** $\mathsf{c}_m^n \equiv \lambda \mathtt{x}_1 . . \lambda \mathtt{x}_n .\ \overline{\mathtt{m}}$
- **Komposition $f \circ (g_1, .., g_n)$:**

  $\circ \equiv \lambda \mathtt{f} . \lambda \mathtt{g}_1 . . \lambda \mathtt{g}_n .\ \lambda \mathtt{x} .\ \mathtt{f}\ (\mathtt{g}_1 \mathtt{x}) . . (\mathtt{g}_n \mathtt{x})$
- **Primitive Rekursion $Pr[f, g]$:**

  $\mathsf{PR} \equiv \lambda \mathtt{f} . \lambda \mathtt{g} .$ function $\mathtt{h}(\mathtt{x}) = \lambda \mathtt{y} .$ if zero $\mathtt{y}$ then $\mathtt{f}\ \mathtt{x}$ else $\mathtt{g}\ \mathtt{x}\ (\mathtt{p}\ \mathtt{y})\ (\mathtt{h}\ \mathtt{x}\ (\mathtt{p}\ \mathtt{y}))$
- **Minimierung $\mu[f]$:**

  $\mathsf{Mu} \equiv \lambda \mathtt{f} . \lambda \mathtt{x} .$ (function $\mathtt{min}(\mathtt{y}) =$ if zero$(\mathtt{f}\ \mathtt{x}\ \mathtt{y})$ then $\mathtt{y}$ else $\mathtt{min}\ (\mathtt{s}\ \mathtt{y}))\ \overline{0}$

Konsequenzen der grossen Ausdruckskraft

Ergebnisse der Berechenbarkeitstheorie übertragen sich

- **Keine nichttriviale extensionale Eigenschaft von λ-Termen kann automatisch geprüft werden** (Satz von Rice)
  - Terminiert die Berechnung einer Funktion f bei Eingabe x?
  - Ist die durch einen λ-Term f beschriebene Funktion total
  - Gehört ein Wert y zum Wertebereich einer Funktion f?
  - Berechnet die Funktion f bei Eingabe von x den Wert y
  - Verhalten sich zwei λ-berechenbare Funktionen gleich?

    Man kann nicht einmal für alle gleichen λ-Terme einen Gleichheitsbeweis finden

    Das Hauptproblem ist dabei die Gleichheit nichtterminierender λ-Terme

- **Beweis kann direkt im λ-Kalkül geführt werden**
  - Umweg über Berechenbarkeitstheorie mit Church'scher These, Gödelnummern, utm- oder smn-Theorem ist nicht erforderlich

Terminierung von λ-Termen ist unentscheidbar

- **Beweis stützt sich auf wenige Erkenntnisse**
  - $\lambda x.x$ terminiert mit Normalform $\lambda x.x$
  - $\bot = \mathsf{Y}(\lambda x.x)$ terminiert nicht, da $\mathsf{Y}(\lambda x.x) = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$
  - Die Boole'schen Wahrheitswerte sind verschieden: $\mathsf{T} \neq \mathsf{F}$
- **Einfacher Widerspruchsbeweis**
  - Wir nehmen an, Terminierung von λ-Termen sei entscheidbar

    d.h. es gibt einen λ-Term h mit $h(t) = \begin{cases} \mathsf{T} & \text{falls } t \text{ terminiert} \\ \mathsf{F} & \text{sonst} \end{cases}$
  - Definiere $d = \lambda x.\, \mathsf{if}\ h(x)\ \mathsf{then}\ \bot\ \mathsf{else}\ \lambda x.x.$
  - Dann $h(\mathsf{Y}d) = h(d(\mathsf{Y}\,d)) = h(\mathsf{if}\ h(\mathsf{Y}\,d)\ \mathsf{then}\ \bot\ \mathsf{else}\ \lambda x.x)$

    also $h(\mathsf{Y}\,d) = \begin{cases} h(\bot) = \mathsf{F}, & \text{falls } h(\mathsf{Y}d) = \mathsf{T} \\ h(\lambda x.x) = \mathsf{T}, & \text{falls } h(\mathsf{Y}d) = \mathsf{F} \end{cases}$
  - Dies ist ein Widerspruch, also ist das Halteproblem nicht entscheidbar

λ-Kalkül – Grenzen

- **Vielseitige, ausdrucksstarke Sprache**
  - Klares Berechenbarkeitsmodell
  - Wenige Vorgaben für formales Schließen
  - Mengentheoretische Semantik extrem kompliziert (domain theory)
- **Unkontrolliert: Kein Schließen über Datentypen möglich**
  - Wann gehört ein Objekt zu einem bestimmten Datentyp?
  - Welche Struktur hat der Datentyp eines Objekts?
  - Welche Eigenschaften hat ein Programm?
- **Erweiterung von Semantik / Inferenzsystem nötig**
  - Beschränke Freiheit der λ-Terme durch Präzisierung ihrer Eigenschaften
  - Wähle Datentypen als Beschreibung von Programmeigenschaften

Typentheorie