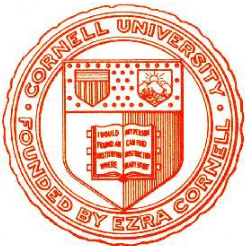
Automatisierte Logik und Programmierung

Einheit 6



Die einfache Typentheorie

1. Syntax und Semantik
2. Eigenschaften typisierbarer Terme
3. Refinement Kalkül für die Typentheorie
4. Die Curry-Howard Isomorphie
5. Type Checking und Typ-Inferenz

Wozu Typsysteme?

- **Ungetypte Programme sind unkontrollierbar**
  - Art der Ein- und Ausgaben ist unklar: eine eingegebene Zahl kann als Boolescher Wert, String, Array, Liste, etc. verarbeitet werden
  - Verhalten und gewünschte Eigenschaften sind nicht spezifiziert insbesondere kann Terminierung nicht gesichert werden
  - Selbstanwendung ist unbegrenzt möglich (aber was bedeutet `add add mul`?)
- **Typsysteme liefern Kontrolle über Programme**
  - Typsysteme begrenzen das erlaubte Verhaltens von Programmen durch Angabe überprüfbarer Grundeigenschaften von Ein-/Ausgaben
  - Komplexere Typen ermöglichen Formulierung detaillierter Eigenschaften
- **Typsysteme koppeln Termstruktur an Typstruktur**
  - Terme erhalten eine Bedeutung, die über das Operationale hinausgeht
  - Ermöglicht präzise Definition der Semantik einer Programmiersprache
  - Ermöglicht formale Beweise von Programmeigenschaften
    Im λ-Kalkül konnte man nur die Gleichheit von Termen beweisen

Wozu Typsysteme (II)?

- **Moderne Programmiersprachen haben Typsysteme**
  – Beschreibung der Grundstruktur von Objekten
  – Realisiert als Zuordnung eines Typs zu Programmtermen im Kontext
  – Ermöglicht statische Kontrolle elementarer Programmeigenschaften
  Viele Fehler in Programmen lassen sich als Typfehler identifizieren
- **Es gibt zwei statisch einsetzbare Kontrollmechanismen**
  – Typechecking: Überprüfung ob gegebener Typ zu einem Term passt
  – Typinferenz: Herleitung eines zum Term zugehörigen Typs
  – Beide basieren auf einem "Typzugehörigkeitskalkül"
- **Typkalkül für λ-Kalkül ist das Fundament**
  – Alle Programm- / Datenstrukturen sind durch λ-Terme beschreibbar
  – Alle Prinzipien von Typsystemen basieren auf Erkenntnissen,
  die mit Typkalkülen für λ-Terme gewonnen werden

Typsysteme vereinigen Logik und Programmierung

- **Zwei Sichtweisen des gleichen formalen Konzepts**
  - Programmierung: Terme sind Programme, Typen ihre Eigenschaften
  - Logik: Typen sind Formeln und Terme ihre Evidenzen
  - In beiden Fällen geht es um Typzugehörigkeit
- **Auch formal ist die Korrespendenz ist sehr eng**
  - Logische Aussagen korrespondieren mit Datentyp der Evidenzen
    λ-Terme sind Evidenz für logische Formeln (§2, Folie 15)
  - Typentheorie macht diesen Zusammenhang explizit
- **Zielrichtung ist invers**
  - Typinferenz: Bestimme Datentyp eines (Programm-)terms
  - Witness finding: Finde (Evidenz-)term zu einem logischen Typ
  - Beides läßt sich in Typzugehörigkeitskalkülen beschreiben
- **Typentheorie ist historisch unabhängig entstanden**
  - Zusammenhang wurde erst später entdeckt (Folie 32)

WAS SIND EIGENTLICH DATENTYPEN?

- **Programmterme besitzen strukturelle Eigenschaften**
  - `1,2,3`,... sind natürliche Zahlen
  - `x+4, y-z, 4*i`,... sind Ausdrücke über Zahlen
  - `(4,5), (7,8), (3-2,4*6)`,... sind Paare von Zahlen
  - `[1;4;7;8;5]` ist eine Liste von Zahlen
  - λ`x. 4*x` ist eine Funktion auf Zahlen

  Datentypen sind syntaktische Repräsentationen dieser Eigenschaften

- **Datentypen besitzen selbst eine Struktur**
  - $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$, String ... sind einfach strukturierte Typen
  - Datentypen können zusammengesetzt werden
    - λ`x. 4*x` besitzt den Typ $\mathbb{N}\to\mathbb{N}$
    - `(4,5)` ist vom Typ $\mathbb{N}\times\mathbb{N}$
    - `[1;4;7;8;5]` gehört zum Typ $\mathbb{N}$ `list`
  - Datentypen können Spezifikationen von Programmeigenschaften sein
    - $\{f{:}\mathbb{N}\to\mathbb{N} \mid (\forall x)\ (f(x)^2 \leq x < (f(x)+1)^2)\}$ spezifiziert $\lambda x. \lfloor\sqrt{x}\rfloor$

Formale Charakteristika eines Datentyps

Syntaktische Repräsentation von Termklassen

- **Direkte Beschreibung elementarer Datentypen**
  - Festlegung eines Namens für den Typ $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{B}$ ...
  - Beschreibung der Basiselemente `0,1,2,...,0,1,-1,2,...,T,F`
  - Beschreibung von Operatoren auf Elementen $-x$, $x+y$, $x-y$, $x*y$, ...
- **Konstruktoren für strukturierte Datentypen**
  - Typkonstruktoren: $T \rightarrow T'$, $T \times T'$, $T+T'$, T `list`,...
  - Konstruktion kanonischer Elemente: $\lambda x.t$, (a,b), $\mathtt{inl}(a)$, $a_1::l$, ...
  - Nichtkanonische Operatoren auf Elementen: $f(a)$, $p.1$, ... $hd(l)$, ...
- **Grundbestandteile der beschreibenden Syntax**
  - Typ-Ausdruck + kanonische Terme + nichtkanonische Terme
- **Alle Konzepte basieren auf Funktionstyp $T \rightarrow T'$**
  - Naheliegende Erweiterung des λ-Kalküls
  - Produkte, Summen, Abhängigkeiten, etc. kommen später hinzu

Typsysteme: Beschreibungsformen

- **Getypter λ-Kalkül**
  - Typen sind Teil der Syntax von λ-Termen: $\lambda f^{S\to T}.\lambda x^S.\,f^{S\to T}\,x^S$
  - Datentyp aller Teilausdrücke leicht zu ermitteln
  - Untypisierbare Terme sind nicht formulierbar
- **Typentheorie**
  - Terme und Typen sind unabhängige Konzepte
  - Typen werden Termen zugeordnet: $\lambda f.\lambda x.f\;x \in (S\to T)\to S\to T$

    Typisierung ist losgelöst von der Formulierung des Terms
  - Mehr Freiheit in der Formulierung von λ-Termen
  - Größerer Aufwand bei (nachträglicher) Bestimmung des Typs

Kalkül zum Schließen über Typzugehörigkeit

Einfache Typentheorie – Syntax

Minimal nötige Typkonstrukte für den λ-Kalkül

- **Erlaubte Symbole**
  - Variablen $x, y, z, x_0, y_0,, \ldots$
  - Typsymbole $S, T, S_0, T_0,, \ldots$
  - Strukturelle Symbole $\rightarrow$, λ, Punkte und Klammern
- **Typ-Ausdrücke (Typen)**
  - Typsymbole sind (atomare) Typen
  - Sind S und T Typen, dann auch $S{\rightarrow}T$ und (T)
- **Objekt-Ausdrücke (λ-Terme)**
  - Variablen sind Terme
  - Ist x Variable, t und f Terme, dann sind $\lambda x\,.\,t$, $f\;t$ und (t) Terme
- **Prioritätskonventionen sparen Klammern**
  - Applikation bindet stärker als λ-Abstraktion $\lambda x\,.\,f\;t \;\hat{=}\; \lambda x\,.\,(f\;t)$
  - Applikation ist links-assoziativ: $f\;t_1\;t_2 \;\hat{=}\; (f\;t_1)\;t_2$
  - $\rightarrow$ ist rechtsassoziativ: $T_1{\rightarrow}T_2{\rightarrow}T_3 \;\hat{=}\; T_1{\rightarrow}(T_2{\rightarrow}T_3)$

Einfache Typentheorie – Semantik

- **Zuordnung von Typen zu Termen**
  - $t \in T \;\equiv\; t$ ist ein Element des Datentyps T

    t beschreibt ein Objekt in der durch T beschriebenen Klasse
  - Formale Definition folgt induktivem syntaktischem Aufbau
  - Denotationelle Semantik wird gekoppelt an operationale Semantik
- **Zuordnung liefert Einschränkung "erlaubter" Terme**
  - $S \rightarrow T$ entspricht Klasse der Funktionen von S nach T
  - Typisierbare λ-Ausdrücke müssen (sinnvolle) Funktionen darstellen

    Keine unkontrollierte Rekursion oder Selbstanwendung

    Minimale Begrenzung der Ausdruckskraft des λ-Kalküls

Typzuordnung an konkrete Terme

- **Typzuordnung an Termvariablen**
  - Termvariablen sind Platzhalter für unbekannte Objekte
  - x kann zu jedem T gehören, wenn x Variable und T Typsymbol
  - Feste Zuordnung nur, wenn Annahmen über Typ von x vorliegen
- **Typzuordnung an λ-Abstraktionen**
  - $\lambda x.t$ muß zu einem Funktionenraum $S \to T$ gehören
  - Dabei muß t zu T gehören wenn x in S ist
- **Typzuordnung an Applikationen**
  - $f\ t$ gehört zum Bildtyp T der Funktion f, wenn der Typ S von t der Argumenttyp $f \in S \to T$ ist

Typisierung von $\overbrace{\lambda \mathtt{f}.\lambda \mathtt{x}.\overbrace{\underbrace{\underbrace{\mathtt{f}}_{S\to T}\ \underbrace{\mathtt{x}}_{S}}_{}}^{T}}^{(S\to T)\to(S\to T)}$ mit $\underbrace{\lambda \mathtt{x}.\,\mathtt{f}\ \mathtt{x}}_{S\to T}$ für beliebige S und T

Typzugehörigkeit: wichtige Erkenntnisse

- **Nur geschlossene Terme sind vollständig typisierbar**
  - Typ offener Terme benötigt Annahmen zu Typen der freien Variablen
- **Nicht jeder (geschlossene) λ-Term ist typisierbar**
  - Sonst gäbe es eine einfache denotationelle Semantik des λ-Kalküls
  - Beispiel: $\lambda \mathtt{x}.\ \underbrace{\mathtt{x}}_{S\to T}\ \underbrace{\mathtt{x}}_{S}$ für beliebige S und T
  - Die Gleichung $S = S\to T$ hat keine “natürliche” Lösung
- **Der Typ eines λ-Terms ist nicht eindeutig**
  - Naheliegend: $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x} \in (S\to T)\to(S\to T)$
  - Auch möglich $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x} \in (S\to(T\to S))\to(S\to(T\to S))$

    $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x} \in (S\to S)\to(S\to S)$
  - $(S\to T)\to(S\to T)$ ist die allgemeinste Form (prinzipielles Typschema)

Semantik der Typentheorie präzisiert

- **Urteile definieren Typzugehörigkeitsrelation $t \in T$**
  - $x \in T$ wenn x Termvariable, T Typvariable und $x \in T$ bekannt
  - $\lambda x.t \in S \to T$ falls $t \in T$ aus $x \in S$ folgt
  - $f\ t \in T$ falls $f \in S \to T$ und $t \in S$ für einen Typ S gelten
  - $(t) \in T$ falls $t \in T$ gilt
  - $t \in (T)$ falls $t \in T$ gilt

  Ein Term t heißt typisierbar, falls $t \in T$ für einen Typ T gilt

  Das prinzipielles Typschema von t ist der allgemeinste Typ T mit $t \in T$

- **λ-Kalkül definiert semantische Gleichheit von Termen**
  - $t = u$, falls es einen Term v gibt mit $t \xrightarrow{*} v$ und $u \xrightarrow{*} v$
  - Reduktion erhält Typzugehörigkeit: $(t \in T \wedge t \xrightarrow{*} t') \Rightarrow t' \in T$ (Folie 12)

    Semantisch gleiche typisierbare Terme gehören zum gleichen Typ
  - Umkehrung gilt nicht: $(t' \in T \wedge t \xrightarrow{*} t') \not\Rightarrow t \in T$

    $(\lambda f.\lambda y.y)(\lambda x.x\,x)$ nicht typisierbar, da $(\lambda x.x\,x)$ nicht typisierbar,

    aber $(\lambda f.\lambda y.y)(\lambda x.x\,x) \xrightarrow{*} \lambda y.y \in T \to T$

Typzugehörigkeit harmoniert mit Reduktion

Sind t und t' typisierbar und semantisch gleich, so gilt $t \in T \;\Leftrightarrow\; t' \in T$

1. Reduktion erhält den Typ: aus $t \xrightarrow{\beta} t'$ folgt $t \in T \;\Leftrightarrow\; t' \in T$

Beweis durch Induktion über die Termstruktur von t (Skizze)

– t Variable: t ist in Normalform, also ist t' identisch zu t $\surd$

– $t \equiv \lambda x\,.\,u$: Dann $t' \equiv \lambda x\,.\,u'$ mit $u \xrightarrow{\beta} u'$.

Aus $t \in T$ folgt $T \equiv S_1 \to S_2$ mit $u \in S_2$ wenn $x \in S_1$.

Per Induktionsannahme gilt $u' \in S_2$ wenn $x \in S_1$, also $t' \in T$

Gleiches Argument für $t \in T$ wenn $t' \in T$ $\surd$

– $t \equiv (\lambda x\,.\,u)\; v$ und $t' \equiv u[v/x]$: Aus $t \in T$ folgt $\lambda x\,.\,u \in S \to T$ und $v \in S$,

also $u \in T$, wenn $x \in S$. Damit $u[v/x] \in T$.

Aus $t' \in T$ folgt $v \in S$ und $u \in T$ wenn $x \in S$. Damit $(\lambda x\,.\,u)\; v \in T$. $\surd$

– $t \equiv f\; u$: Dann $t' \equiv f\; u'$ und $u \xrightarrow{\beta} u'$ oder $t' \equiv f'\; u$ und $f \xrightarrow{\beta} f'$.

Gleiches Argument wie zuvor. $\surd$

2. Aus $t \xrightarrow{*} t'$ folgt $t \in T \;\Leftrightarrow\; t' \in T$ (Induktion über Reduktionslänge)

3. t und t' sind semantisch gleich, wenn $t \xrightarrow{*} u$ und $t' \xrightarrow{*} u$

Eigenschaften typisierbarer Terme

- **Schwache Normalisierbarkeit**
  - Hat jeder Term eine Normalform (d.h. sind alle Funktionen total)?
  - Führt mindestens eine Reduktionsstrategie immer zu einem Wert?
- **Starke Normalisierbarkeit**
  - Terminiert jede beliebige Reduktionsfolge?
  - Damit könnte jede Reduktionsstrategie gewählt werden
- **Konfluenz (Church-Rosser Eigenschaft)**
  - Kann man verschiedene Reduktionsfolgen wieder zusammenführen?
  - Damit wäre die Normalform eines Terms eindeutig
- **Entscheidbarkeit**
  - Ist Typisierbarkeit oder Gleichheit von Termen entscheidbar?
  - Damit würde Korrektheit / Äquivalenz von Programmen testbar
- **Ausdruckskraft**
  - Gibt es wichtige Funktionen, die nicht typisierbar sind?

Erkenntnisse gelten auch für Erweiterungen (siehe Folie 33)

Schwache Normalisierbarkeit

Hat jeder typisierbare λ-Term eine Normalform?

- **Tiefe $d(T)$ eines Typausdrucks T**
  - $d(T) = 0$, falls $T \in \mathcal{T}$, $d(S \to T) = 1 + \max(d(S), d(T))$
  - Tiefe des Redex $(\lambda x\,.\,t)\;u \;\equiv$ Tiefe des Typs von $\lambda x\,.\,t$.
  - Tiefe eines Terms $t \;\equiv$ maximale Tiefe der Redizes von t
- **Rightmost-Maxdepth Strategie:**
  - Reduziere das am weitesten rechts stehende Redex maximaler Tiefe

Beispiel: `trice(trice(λx.x))` mit `trice` $\equiv$ `λf.λx.f(f(f x))`

Ergebnis `λx. x` Keine Redices

RIGHTMOST-MAXDEPTH STRATEGIE

Die Rightmost-Maxdepth-Strategie terminiert
auf allen typisierbaren λ-Termen

- Lemma: Die Strategie verringert Anzahl der Redizes maximaler Tiefe
  - Sei $r{=}(\lambda x.u)\,v$ das rightmost-maxdepth Redex von t
  - r wird reduziert zu dem Term $u[v/x]$ ohne Redizes maximaler Tiefe
  - Redizes von t außerhalb von r bleiben unverändert $\surd$

- Lemma: Hat t maximal m Redizes der maximalen Tiefe d
  dann hat t eine Normalform t' mit $t \xrightarrow{n} t'$ für ein $n \in \mathbb{N}$
  - Zeige durch Doppelinduktion über d und m mit obigem Lemma,
    daß die Rightmost-Maxdepth-Strategie immer terminiert $\surd$

Jeder typisierbare λ-Term ist normalisierbar

Starke Normalisierbarkeit

Führt jede Reduktionsfolge zu einer Normalform?

- **t stark normalisierbar (SN):**
  - Jede in t beginnende Reduktionsfolge terminiert
- **Nachweis extrem aufwendig**
  - Jede Reduktionsfolge müsste analysiert werden
  - Induktionsbeweis zeigt Verschärfung von starker Normalisierbarkeit

    Jeder typisierbare λ-Term ist "berechenbar"
- **Wichtige Hilfskonzepte**
  - **Berechenbare typisierte Terme**
    - $t \in T$ (T Typvariable) ist berechenbar, falls t stark normalisierbar
    - $t \in S \to T$ berechenbar, falls $t\ s$ berechenbar für berechenbare $s \in S$
  - Mehr als nur SN: jede Anwendung von t ist berechenbar
  - **Neutrale Terme**
    - t ist neutral, wenn t nicht die Gestalt $\lambda x\,.\,u$ hat

Nachweis der Starken Normalisierbarkeit

Zeige durch Induktion über die Struktur des Typs T von t und t'

1. t berechenbar $\Rightarrow$ t stark normalisierbar
2. t berechenbar, $t \xrightarrow{\beta} t'$ $\Rightarrow$ t' berechenbar
3. t neutral, $t \xrightarrow{\beta} t'$ impliziert t' berechenbar $\Rightarrow$ t berechenbar
4. t neutral, in Normalform $\Rightarrow$ t berechenbar

Zeige durch Doppel-Induktion über Anzahl von Reduktionsschritten

5. $t[s/x]$ berechenbar für berechenbare s $\Rightarrow$ $\lambda x\,.\,t$ berechenbar

Zeige durch Induktion über die Struktur von t=$t[x_1 \ldots x_n]$

6. Sind $x_1...x_n$ freie Variablen in t, b_i berechenbare Terme vom Typ der x_i, dann ist $t[b_1...b_n/x_1...x_n]$ berechenbar
7. Aus 6. folgt: **Jeder typisierbare λ-Term ist berechenbar**
   – Sind $x_1..x_n$ freie Variablen von t, so gilt $t = t[x_1...x_n/x_1...x_n]$
   – Alle x_i sind neutral und in Normalform, also berechenbar $\checkmark$

Jeder typisierbare λ-Term ist stark normalisierbar

STARKE NORMALISIERBARKEIT: BEWEIS VON 1.–4.

1. t berechenbar $\Rightarrow$ t stark normalisierbar
2. t berechenbar, $t \xrightarrow{\beta} t'$ $\Rightarrow$ t' berechenbar
3. t neutral, $t \xrightarrow{\beta} t'$ impliziert t' berechenbar $\Rightarrow$ t berechenbar
4. t neutral, in Normalform $\Rightarrow$ t berechenbar

Simultane Induktion über Struktur des Typs T von t und t'

4. folgt jeweils aus 3., da Terme in Normalform nicht reduzierbar sind

- **T atomar:**

1. t berechenbar $\Leftrightarrow$ t stark normalisierbar (per Definition) $\checkmark$
2. t berechenbar, $t \xrightarrow{\beta} t'$, dann t' SN, also t' berechenbar $\checkmark$
   (Jede Reduktionsfolge $t', t'_1, t'_2, \ldots$ terminiert, da $t, t', t'_1, t'_2, \ldots$ terminiert)
3. t neutral, $t \xrightarrow{\beta} t'$ impliziert t' berechenbar, dann t SN und berechenbar $\checkmark$
   (Jede Reduktionsfolge $t, t', t'', t''', \ldots$ terminiert, da $t', t'', t''', \ldots$ terminiert)

Starke Normalisierbarkeit: Beweis von 1.–4. (II)

- **Sei $T = T_1 \to T_2$, Behauptung gelte für Terme in T_1 und T_2**

  1. Sei $t \in T$ berechenbar, $x \in T_1$ Variable, $t \xrightarrow{\beta} t_1 \xrightarrow{\beta} t_2 \xrightarrow{\beta} \ldots$ Reduktionsfolge
     $\mapsto$ x neutral, in Normalform, also berechenbar (Induktionsannahme 4.)
     $\mapsto$ $t\,x \in T_2$ berechenbar (per Definition), also stark normalisierbar (Induktionsannahme 1.)
     $\mapsto$ Reduktionsfolge tx, t_1x, t_2x, … terminiert, also auch t, t_1, t_2, …
     $\mapsto$ t stark normalisierbar ✓

  2. Sei t berechenbar, es gelte $t \xrightarrow{\beta} t'$. Sei $s \in T_1$ berechenbar
     $\mapsto$ $t\,s \in T_2$ berechenbar und $ts \xrightarrow{\beta} t's$, also $t'\,s \in T_2$ berechenbar (Induktionsannahme 2.),
     $\mapsto$ t' berechenbar (Definition) ✓

  3. Es gelte t neutral, $t \xrightarrow{\beta} t'$ impliziert t' berechenbar. Sei $s \in T_1$ berechenbar
     $\mapsto$ s SN (Induktionsannahme 1). Wir zeigen $ts \xrightarrow{\beta} t''$ impliziert t'' berechenbar
     durch Induktion über maximale Anzahl n der Reduktionsschritte von s
     · n=0: s ist in Normalform, also $t'' = t'\,s$ mit $t \xrightarrow{\beta} t'$. Dann t', t'' berechenbar
     · n+1: t neutral, also $t\,s$ kein Redex
       · Falls $t'' = t'\,s$ mit $t \xrightarrow{\beta} t'$, dann t'' berechenbar
       · Falls $t'' = t\,s'$ mit $s \xrightarrow{\beta} s'$, dann s' SN mit maximal n Schritten und berechenbar
         $t\,s' \xrightarrow{\beta} t'''$ impliziert t''' berechenbar (innere Annahme) und $t\,s' \in T_2$ neutral
         also $t'' \equiv t\,s'$ berechenbar (Induktionsannahme 3.)
     $\mapsto$ $t\,s \in T_2$ neutral, also berechenbar (Induktionsannahme 3.)
     $\mapsto$ t berechenbar (Definition) ✓

Starke Normalisierbarkeit: Beweis von 5.

t typisierbar, $t[s/x]$ berechenbar für berechenbare $s \;\Rightarrow\; \lambda x\,.\,t$ berechenbar

Sei $t[s/x]$ berechenbar für alle berechenbare s

– Zu zeigen ist $(\lambda x.t)\,s$ berechenbar für berechenbare s
  bzw., da $(\lambda x\,.\,t)\,s$ neutral: $(\lambda x\,.\,t)\,s \xrightarrow{\beta} t'' \;\Rightarrow\; t''$ berechenbar (nach 3)

– t ist berechenbar, denn t=$t[x/x]$ und jede Variable x ist berechenbar nach (4)

– Also t und s stark normalisierbar nach (1)

Beweis durch Doppelinduktion über Summe der Reduktionssschritte von t und s

· Falls $t'' = t[s/x]$, dann t'' berechenbar (↦ Argument für Basisfall) $\surd$

· Falls $t'' = (\lambda x.t')\,s$ mit $t \xrightarrow{\beta} t'$
  Dann t' berechenbar (2) und SN (1) mit weniger Reduktionsschritten
  Also t''=$(\lambda x.t')\,s \;\xrightarrow{\beta} t''' \;\Rightarrow\; t'''$ berechenbar (Induktionsannahme für t')
  Also t'' berechenbar nach (3) (↦ Äußerer Induktionsschritt für t) $\surd$

· Falls $t'' = (\lambda x.t)\,s'$ mit $s \xrightarrow{\beta} s'$.
  Dann s' berechenbar (2) und SN (1) mit weniger Reduktionsschritten.
  Also t''=$(\lambda x.t)\,s' \;\xrightarrow{\beta} t''' \;\Rightarrow\; t'''$ berechenbar (Induktionsannahme für s')
  Also t'' berechenbar nach (3) (↦ Innere Induktionsschritte für s) $\surd$

Starke Normalisierbarkeit: Beweis von 6.

t typisierbar, $x_1 \ldots x_n$ (alle) freie Variablen von t,
b_i berechenbar und vom Typ der x_i. Dann $t[b_1..b_n \,/\, x_1..x_n]$ berechenbar

Induktion über die Struktur von t=$t[x_1..x_n]$

- Falls $t = x_i$, dann $t[b_1..b_n \,/\, x_1..x_n] = b_i$ berechenbar √
- Falls $t = \lambda x.u$, dann $t[b_1..b_n \,/\, x_1..x_n] = \lambda x\,.\,u[b_1..b_n \,/\, x_1..x_n]$

  und $u[b_1..b_n, b \,/\, x_1..x_n, x]$ berechenbar für alle b (Induktionsannahme)

  Also $t[b_1..b_n \,/\, x_1..x_n]$ berechenbar nach (5) √
- Falls $t = f\ u$, dann

  $t[b_1..b_n \,/\, x_1..x_n] = f[b_1..b_n \,/\, x_1..x_n]\ \ u[b_1..b_n \,/\, x_1..x_n]$

  und $f[b_1..b_n \,/\, x_1..x_n]$ berechenbar

  und $u[b_1..b_n \,/\, x_1..x_n]$ berechenbar (Induktionsannahme)

  Also $t[b_1..b_n \,/\, x_1..x_n]$ berechenbar (Definition für 'f berechenbar') √

Starke Normalisierbarkeit: Vorteile

- **Jede Reduktionsstrategie kann angewandt werden**
  - Terminierung ist immer gesichert
  - Es gibt effizientere ("riskantere") Strategien als Rightmost-Maxdepth
- **Rightmost (Call-by-value):**

  $\texttt{trice(trice(}\lambda\texttt{x.x))}$
  $\xrightarrow{8} \lambda\texttt{x.x}$

- **Rightmost-maxdepth:**

  $\texttt{trice(trice(}\lambda\texttt{x.x))}$
  $\xrightarrow{14} \lambda\texttt{x.x}$

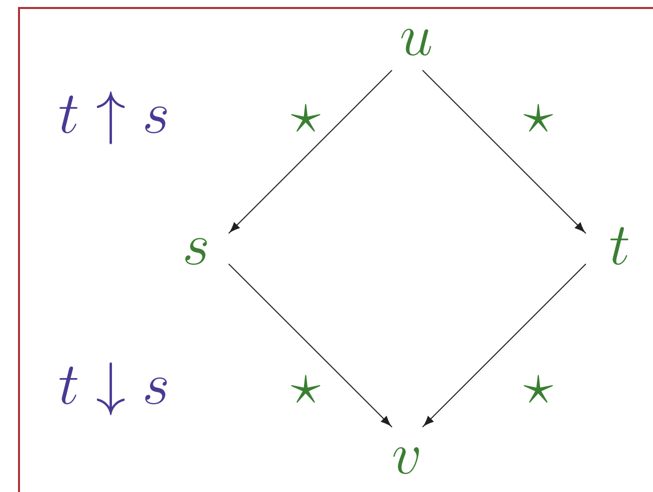
- **Leftmost (Call-by-name):**

  $\texttt{trice(trice(}\lambda\texttt{x.x))}$
  $\xrightarrow{19} \lambda\texttt{x. x}$

Konfluenz einer Reduktionsrelation $\xrightarrow{r}$

Ist die Normalform eines Terms eindeutig?

- **Iteration einer Reduktionsrelation** $\xrightarrow{r}$
  - $t \xrightarrow{n} s$: s ergibt sich aus t durch n Reduktionsschritte
    $t \xrightarrow{0} s$, falls t=s, $t \xrightarrow{n+1} s$, falls es ein u gibt mit $t \xrightarrow{r} u$ und $u \xrightarrow{n} s$
  - $t \xrightarrow{*} s$: es gibt ein $n \in \mathbb{N}$ mit $t \xrightarrow{n} s$
  - $t \xrightarrow{+} s$: es gibt ein n>0 mit $t \xrightarrow{n} s$



- **$t{\uparrow}s$:** es gibt u mit $u \xrightarrow{*} t$ und $u \xrightarrow{*} s$
  $t{\downarrow}s$: es gibt u mit $t \xrightarrow{*} v$ und $s \xrightarrow{*} v$
- **Konfluenz:** aus $t{\uparrow}s$ folgt immer $t{\downarrow}s$
  Lokale Konfluenz: aus $u \xrightarrow{r} t$, $u \xrightarrow{r} s$ folgt $t{\downarrow}s$

Diamond Lemma

$\xrightarrow{r}$ stark normalisierbar, lokal konfluent $\Rightarrow$ $\xrightarrow{r}$ konfluent

- **Für jeden Term u gibt es ein n, so daß jede in u beginnende Reduktionsfolge in maximal n Schritten terminiert**
  - Stark normalisierbar: jede Reduktionsfolge von u terminiert
  - Endliche Verzweigung von $\xrightarrow{r}$: es gibt eine maximale Schrittzahl
- **Für alle u folgt $t{\downarrow}s$ aus $u \xrightarrow{*} t$ und $u \xrightarrow{*} s$**

  Induktion über maximale Zahl n der Reduktionsschritte

  $n{=}0$: Es folgt $u{=}t{=}s$, also $t{\downarrow}s$ trivialerweise $\surd$

  $n{+}1$: Falls $u \xrightarrow{0} t$ oder $u \xrightarrow{0} s$, dann $u{=}t$ oder $u{=}s$ $\surd$

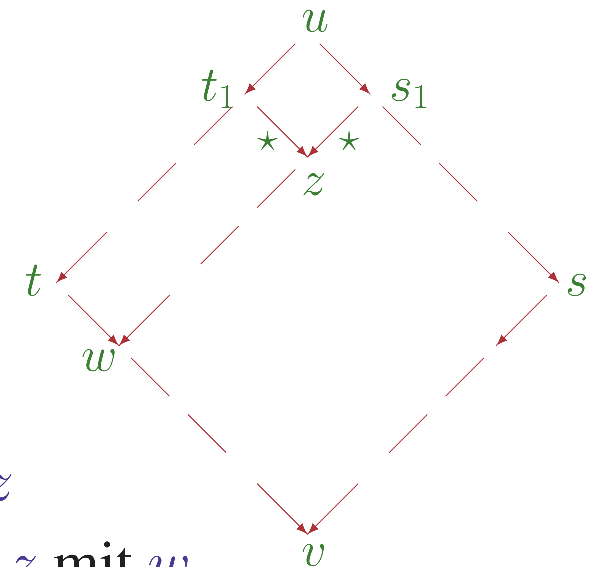
  Ansonsten $u \xrightarrow{r} t_1 \xrightarrow{*} t$ und $u \xrightarrow{r} s_1 \xrightarrow{*} s$

  Lokale Konfluenz: $t_1{\downarrow}s_1$ mit einem Term z

  Induktionsannahme für t_1: $t_1 \xrightarrow{*} t$, $t_1 \xrightarrow{*} z$ also $t{\downarrow}z$ mit w

  Induktionsannahme für s_1: $s_1 \xrightarrow{*} z \xrightarrow{*} w$, $s_1 \xrightarrow{*} s$ also $w{\downarrow}s$ mit v

  Insgesamt $t \xrightarrow{*} w \xrightarrow{*} v$ und $s \xrightarrow{*} v$, also $\boldsymbol{t{\downarrow}s}$ $\surd$



Church-Rosser Theorem

In der einfachen Typentheorie ist $\stackrel{\beta}{\longrightarrow}$ konfluent

- **$\stackrel{\beta}{\longrightarrow}$ ist lokal konfluent**

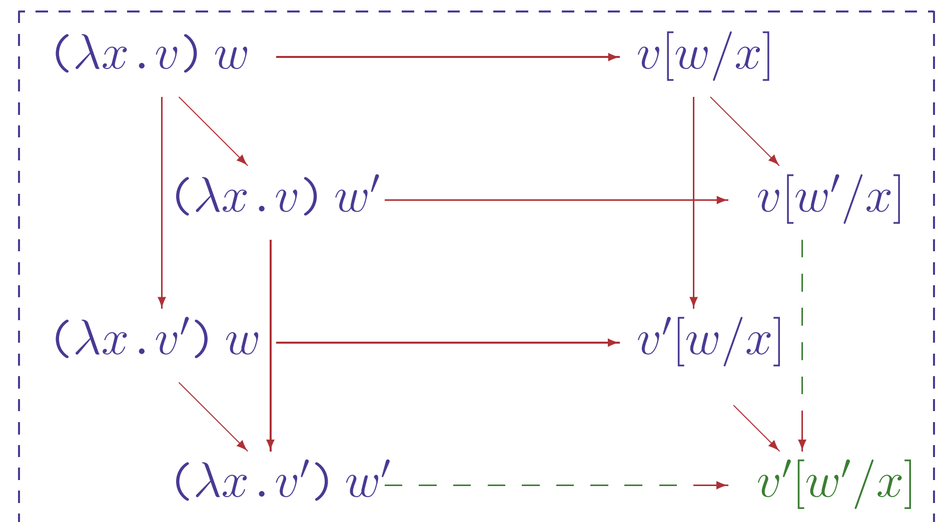
  Es gelte $u \stackrel{\beta}{\longrightarrow} t$, $u \stackrel{\beta}{\longrightarrow} s$ und $s{\neq}t$

  Betrachte Teilterm von u der Gestalt $(\lambda x\,.\,v)\ w$ oder $v\ w$, so daß

  – s entsteht durch Reduktion von v, w oder des äußeren β-Redex

  – t entsteht durch eine andere Reduktion des Teilterms

  Reduktionskette kann gemäß Diagramm zusammengeführt werden



- **Diamond Lemma + starke Normalsierbarkeit $\mapsto$ Konfluenz**

Typisierbare λ-Terme haben eindeutige Normalformen

Refinement Kalkül für die Typentheorie

- **Umsetzung der semantischen Urteile als Inferenzregeln**
  - Urteile sind schematische Regeln zur Definition von Semantik
  - Inferenzregeln sind syntaktische Vorschriften für Beweisführung
  - Ähnlichkeit erleichtert Nachweis von Korrektheit und Vollständigkeit
- **Schließen über Gleichheit und Berechnung**
  - Regeln zur Auswertung von Termen wie im λ-Kalkül
- **Schließen über Typzugehörigkeit**
  - Dekompositionsregeln zur Zuordnung von Typen zu Termen
- **Schließen über Struktur von Datentypen**
  - Regeln zur strukturellen Analyse von (Typ-)Ausdrücken

Integration von Typzugehörigkeit und Gleichheit

- **Gleichheit hängt eigentlich vom Typ ab**
  - 4 und 7 sind verschieden als Zahlen aber gleich in $\mathbb{Z}_3$
  - (2,4) und (1,2) sind verschieden als Zahlenpaare aber gleich in $\mathbb{Q}$
  - In der einfachen Typentheorie haben semantisch gleiche Terme allerdings immer denselben Typ
- **Kombiniere s=t und $t \in T$ zur Relation s=$t \in T$**
  - Prädikat mit fester Bedeutung: "s und t sind im Typ T gleich"
  - s=$t \in T$ bedingt neben der Gleichheit s=t auch $s \in T$ und $t \in T$
  - Typzugehörigkeit $t \in T$ wird als Kurzform für t=$t \in T$ betrachtet
  - Inferenzregeln für Typzugehörigkeit werden als Spezialfall der entsprechenden Gleichheitsregeln angesehen
- **Allgemeine Reflexivitätsregel wird ungültig**
  - t=$t \in T$ gilt nur, wenn auch $t \in T$ gezeigt werden kann

Erweiterungen des logischen Refinement Kalküls

- **Sequenzen enthalten Variablen- und Typ-Deklarationen**
  - $x:S$ (Term-Variablendeklaration), oder $T:\mathbb{U}$ (Typdeklaration)
  - $\mathbb{U}$ ist festes Symbol für das Universum aller Typen
  - Konzept der Deklaration in Sequenzen bekommt zusätzlichen Sinn
- **Konklusion kann Typisierungen enthalten**
  - Typisierte Gleichheit: $s\texttt{=}t \in T$ (Kurzform $t \in T$ für $t\texttt{=}t \in T$)
- **Definition der Sequenz wird verfeinert: $H_1, .., H_n \vdash C$**
  - $H_i = x_1{:}A_1$ Deklaration, C (Formel mit) Typisierung(en)
  - $H_1, .., H_n$ ist rein, wenn jede Variable genau einmal deklariert ist und jede Typvariable einer Variablendeklarationen zuvor deklariert wurde
  - $H \vdash t \in T$ ist Initialsequenz für Typisierungsbeweise, wenn H rein ist und genau die in T vorkommenden Typvariablen deklariert
- **Definition von Regel, Beweis, Theorem etc. unverändert**

Refinement-Regeln der einfachen Typentheorie

- **Regel für β-Reduktion**
  – Regel des λ-Kalküls wird erweitert um Typ der Terme in Konklusion

$$\begin{array}{l} H \vdash (\lambda x.u)\, s = t \in T \\ \quad H \vdash u[s/x] = t \in T \end{array} \qquad \texttt{reduction}$$

- **Kongruenzregel für Applikation**
  – Direkte Umsetzung des Urteils erweitert Regel des λ-Kalküls um Typen

$$\begin{array}{l} H \vdash f\, t = g\, u \in T \\ \quad H \vdash f = g \in S \to T \\ \quad H \vdash t = u \in S \end{array} \qquad \texttt{applyEq}\ S$$

  – Argumenttyp S muß als Parameter der Regel angegeben werden

- **Kongruenzregeln für Abstraktion**

$$\begin{array}{l} H \vdash \lambda x.t = \lambda y.u \in S \to T \\ \quad H, x':S \vdash t[x'/x] = u[x'/y] \in T \end{array} \qquad \texttt{lambdaEq}$$

(x' neue Variable, nicht frei in H)

  – Teilziel muß Typdeklaration $x':S$ der λ-Variablen enthalten

Gleichheits-Regeln der einfachen Typentheorie

- **Reflexivitätsregel benötigt Typdeklaration**

$H, x{:}T, H' \vdash x = x \in T$ `declaration`

- **Symmetrie, Transitivität, Substitution**

$H \vdash t = u \in T$
  $H \vdash u = t \in T$ `symmetry`

$H \vdash t = u \in T$
  $H \vdash t = s \in T$
  $H \vdash s = u \in T$ `transitivity` s

$H \vdash C[t/x]$
  $H \vdash t = u \in S$
  $H \vdash C[u/x]$ `subst` $t{=}u \in S$

Typ der ersetzten Terme muß als Parameter der Regel angegeben werden

REFINEMENTBEWEIS FÜR $(\lambda f.\lambda x.\,f\,x)(\lambda z.z) \in T \to T$

Initialsequenz muß Typsymbol T deklarieren

$T{:}\mathbb{U} \vdash (\lambda f.\lambda x.\,f\,x)(\lambda z.z) \in T \to T$	BY applyI $T \to T$
1 $T{:}\mathbb{U} \vdash \lambda f.\lambda x.\,f\,x \in (T \to T) \to (T \to T)$	BY lambdaEq
1.1 $T{:}\mathbb{U},\, f{:}T \to T \vdash \lambda x.\,f\,x \in T \to T$	BY lambdaEq
1.1.1 $T{:}\mathbb{U},\, f{:}T \to T,\, x{:}T \vdash f\,x \in T$	BY applyEq T
1.1.1.1 $T{:}\mathbb{U},\, f{:}T \to T,\, x{:}T \vdash f \in T \to T$	BY declaration
1.1.1.2. $T{:}\mathbb{U},\, f{:}T \to T,\, x{:}T \vdash x \in T$	BY declaration
2. $T{:}\mathbb{U} \vdash \lambda z.z \in T \to T$	BY lambdaEq
2.1 $T{:}\mathbb{U},\, z{:}T \vdash z \in T$	BY declaration

Typisierungsregeln vs. logische Regeln

lambdaEq	$H \vdash \lambda x.t \in S \rightarrow T$ $\quad H, x'{:}S \vdash t[x'/x] \in T$	$H \vdash A \Rightarrow B \quad$ ev = $\lambda a.b$ $\quad H, a{:}A \vdash B \quad$ ev = b	impliesR
applyEq S	$H \vdash f\, t \in T$ $\quad H \vdash f \in S \rightarrow T$ $\quad H \vdash t \in S$	$H \vdash B \quad$ ev = $f(a)$ $\quad H \vdash A \Rightarrow B \quad$ ev = f $\quad H \vdash A \quad$ ev = a	modus ponens A

Originalregel `impliesL` *ist Variante von* `modus ponens`, *die auf Hypothesen arbeitet und $A \Rightarrow B$ explizit voraussetzt*

$H, f{:}A \Rightarrow B, H' \vdash C \quad$ ev = $c[f(a)/b]$
$\quad H, f{:}A \Rightarrow B, H' \vdash A \quad$ ev = a
$\quad H, b{:}B, H' \vdash C \quad$ ev = c

Zusammenhang wurde bekannt als **Curry-Howard Isomorphie**

Typ	Formel
Variable vom Typ S	Annahme (von Evidenz für Formel) S
Term vom Typ T (freie Variablen aus S_i)	Beweis von T (Annahmen S_i)
Typechecking	Beweisführung
(Regel der) λ-Abstraktion	$\Rightarrow$-Einführung (rechts)
(Regel der) Applikation	$\Rightarrow$-Analyse (links)

Curry-Howard Isomorphie lässt sich fortsetzen

Erweiterte Typentheorie mit Produkte und Summen

- **Zusätzliche Typ- und Objekt-Ausdrücke**
  - Sind S und T Typen, dann auch $S{\times}T$, $S{+}T$
  - Sind x, y, z Variablen, a, b, s, t, u Terme, dann sind auch (a,b), $t.1$, $t.2$, $\mathtt{inl}(a)$, $\mathtt{inr}(b)$, und $\mathtt{case}\ u\ \mathtt{of}\ \mathtt{inl}(x) \rightarrow s \mid \mathtt{inr}(y) \rightarrow t$ Terme
- **Zusätzliche Urteile der Typzugehörigkeitsrelation $t \in T$**
  - $(a,b) \in S{\times}T$ falls $a \in S$ und $b \in T$
  - $t.1 \in S$ und $t.2 \in T$ falls $t \in S{\times}T$
  - $\mathtt{inl}(a) \in S{+}T$ und $\mathtt{inr}(b) \in S{+}T$ falls $a \in S$ bzw. $b \in T$
  - $\mathtt{case}\ u\ \mathtt{of}\ \mathtt{inl}(x) \rightarrow s \mid \mathtt{inr}(y) \rightarrow t \in T'$ falls $u \in S{+}T$ und $s \in T'$ aus $x \in S$ sowie $t \in T'$ aus $y \in T$ folgt
- **Zusätzliche Reduktion und Inferenzregeln entsprechend**
  - Regeln für Produkttyp entsprechen denen der Konjunktion
  - Regeln für Summentyp entsprechen denen der Disjunktion

Aussagenlogik kann als Spezialfall der Typentheorie realisiert werden

Typ-Inferenz und -Überprüfung

- **In der einfachen Typentheorie sind die Typen naheliegend**
  - Der Typ eines Ausdrucks läßt sich leicht schematisch bestimmen
- **Typinferenz: Berechne den Typ eines beliebigen Terms**
  - Vollautomatisch für einfache Typstrukturen (ML, Haskell, Java, ...)
  - Unentscheidbar für komplexe Datentypen (Logische Spezifikationen)
- **Analyse von $t \in T$**
  - Weise t einen unbekannten Typ T zu
  - Verfeinere T anhand der Regeln der Typzugehörigkeit
  - Ergibt prinzipielles Typschema oder Nachweis der Nichttypisierbarkeit
- **Typisierung von λ`f`.λ`x. f x`**

$$\lambda \underbrace{\texttt{f}}_{X_1}.\lambda \underbrace{\texttt{x}}_{X_3}.\ \underbrace{\texttt{f}}_{X_1}\underbrace{\texttt{x}}_{X_3} \in (X_3\to X_4)\to(X_3\to X_4)$$

- **Typisierung von** $\lambda \underbrace{\texttt{x}}_{X_1}.\ \underbrace{\texttt{x}}_{X_1}\ \underbrace{\texttt{x}}_{X_1} \in X_1\to X_2$

$X_1 = X_1\to X_2$

nicht typisierbar

Hindley-Milner Type-Checking Algorithmus

- **Eingabe**: geschlossener λ-Term t

  Ausgabe: prinzipielles Typschema von t oder Fehlermeldung

  Interne Daten: Umgebung *Env* deklarierter Typzugehörigkeiten
  Globale Substitution σ von Typvariablen durch Typen

- **Start**: Setze σ := [] und rufe TYPE-OF([], t) auf

- **Algorithmus** TYPE-OF(*Env*, t):

  Falls t=x (Variable): suche Deklaration x:T in *Env* und gebe T aus

  Falls t=$f\ u$: Setze S_1:=TYPE-OF(*Env*, f), S_2:=TYPE-OF(*Env*, u)
  Wähle neues X_{i+1}, unifiziere $\sigma(S_1)$ mit $S_2 \rightarrow X_{i+1}$.
  Setze $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation
  Ausgabe: $\sigma(X_{i+1})$ (Fehlermeldung, wenn Unifikation fehlschlägt)

  Falls t=$\lambda x\,.\,u$: Wähle neues X_{i+1}

  $$S_1 := \begin{cases} \text{TYPE-OF}(Env \cdot [x{:}X_{i+1}], u) & \text{falls } x \text{ nicht in } Env \\ \text{TYPE-OF}(Env \cdot [x'{:}X_{i+1}], u[x'/x]) & \text{sonst } (x' \text{ neue Variable}) \end{cases}$$

  Ausgabe: $\sigma(X_{i+1}) \rightarrow S_1$

TYPISIERUNG VON $\lambda f.\lambda x.\, f(f(f\,x))$

Env	*Aktueller Term t*	*Substitution σ*	UNIFY	*Typ T*
	`λf.λx.f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 \,/\, X_3, X_2, X_1, X_0]$		$(X_4 \to X_4) \to X_4 \to X_4$
`f:`X_0	`λx.f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 \,/\, X_3, X_2, X_1, X_0]$		$X_4 \to X_4$
`f:`X_0, `x:`X_1	`f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 \,/\, X_3, X_2, X_1, X_0]$	$X_3 \to X_3 = X_3 \to X_4$	X_4
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`f(f x)`	$[X_3, X_3, X_3 \to X_3 \,/\, X_2, X_1, X_0]$	$X_1 \to X_2 = X_2 \to X_3$	X_3
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`f x`	$[X_1 \to X_2 \,/\, X_0]$	$X_0 = X_1 \to X_2$	X_2
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`x`			X_1

Entscheidbarkeit

Welche Eigenschaften typisierbarer Terme sind entscheidbar?

- **Typisierbarkeit**
  - Hindley-Milner Algorithmus
- **Halteproblem und Totalität von Funktionen**
  - Trivial wegen starker Normalisierbarkeit
- **Gleichheit**
  - Teste Typisierbarkeit, normalisiere im Erfolgsfall, prüfe Kongruenz
- **Korrektheit und Äquivalenz von Programmen**
  - Die Tests $f(s)=t$ und $f=g$ sind spezielle Gleichheitsprobleme
- **Programmeigenschaften, wenn als Typ codierbar**
  - Teste $t \in T$ mit Hindley-Milner Algorithmus

Wie wirkt sich das auf die Ausdruckskraft aus?

AUSDRUCKSKRAFT

Welche Funktionen können durch typisierbare λ-Terme berechnet werden?

- **Church-Numerals sind typisierbar**
  - $\overline{n} \;\equiv\; \lambda f.\lambda x.\; f^n x \;\in\; (X\to X)\to X\to X \;\equiv\; \mathbb{N}$
  - $s \;\equiv\; \lambda n.\lambda f.\lambda x.\; f\;(n\;f\;x) \;\in\; \mathbb{N}\to\mathbb{N}$
  - $add \;\equiv\; \lambda m.\lambda n.\lambda f.\lambda x.\; m\;f\;(n\;f\;x) \;\in\; \mathbb{N}\to\mathbb{N}\to\mathbb{N}$
  - $mul \;\equiv\; \lambda m.\lambda n.\lambda f.\lambda x.\; m\;(n\;f)\;x \;\in\; \mathbb{N}\to\mathbb{N}\to\mathbb{N}$

- **(Simulation der) Paarbildung ist typisierbar**
  - $(s,t) \;\equiv\; \lambda p.\; p\,s\,t \;\in\; (\boldsymbol{S}\to\boldsymbol{T}\to Z)\to Z \;\equiv\; \boldsymbol{S}\times\boldsymbol{T}$
  - $(s,t).1 \;\equiv\; (s,t)\;(\lambda x.\lambda y.x) \;\in\; S$
  - $(s,t).2 \;\equiv\; (s,t)\;(\lambda x.\lambda y.y) \;\in\; T$

Ausdruckskraft typisierter Terme hat Grenzen

- **Typisierung boolescher Operatoren problematisch**
  - T $\equiv$ `λx.λy.x` $\in X \to Y \to X$
  - F $\equiv$ `λx.λy.y` $\in X \to Y \to Y$
  - if b then s else t $\equiv$ $b\,s\,t \in Z$ für $s \in \mathtt{X}$, $t \in \mathtt{Y}$, $b \in \mathbb{B} \equiv X \to Y \to Z$
  - Aber T $\in \mathbb{B}$ und F $\in \mathbb{B}$ verlangt $X{=}Y{=}Z$ und somit $\mathbb{B} \equiv X \to X \to X$
    Dies schränkt die Wahl möglicher Werte für s und t ein,
    besonders wenn ein Programm zwei Konditionale enthält
- **Church Numerals brauchen polymorphes $\mathbb{N}$**
  - exp $\equiv$ `λm.λn.λf.λx. n m f x` $\in X \to (X \to Y \to Z \to T) \to Y \to Z \to T$
    - · aber `m,n` $\in \mathbb{N}$ verlangt $X \to Y \to Z \to T = X$
  - PRs[$base$,h] $\equiv$ `λn. n` h $base$ $\in (X \to Y \to Z) \to Z$
    - · aber $h \in X{=}\mathbb{N} \to \mathbb{N}$ und $base \in Y{=}\mathbb{N}$ verlangt $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} = \mathbb{N}$

Simple Typstruktur zu schwach für praktische Anwendungen