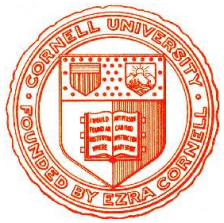
Theoretische Informatik II

Einheit 5.3



Funktionale & Logische Programme

1. Der λ-Kalkül
2. Arithmetische Repräsentierbarkeit
3. Die Churchsche These

Der λ-Kalkül

Grundlage funktionaler Programmiersprachen (Lisp, ML, Haskell,..)

- **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:

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

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

     $\texttt{f(4)} \;\hat{=}\; \texttt{(}\lambda\texttt{x.\,2*x+3)(4)}$ **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**
  - Der λ-Kalkül ist eine Turingmächtige funktionale Programmiersprache

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 $\quad \lambda x\,.\,f\ t \ \hat{=}\ \lambda x\,.\,(f\ t)$
  - Applikation ist links-assoziativ: $\quad f\ t_1\ t_2 \ \hat{=}\ (f\ t_1)\ t_2$
- **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

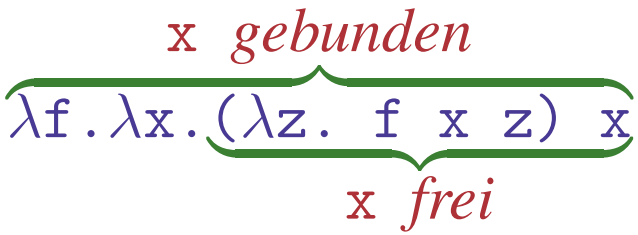
- **Interpretation als "normale" Funktionen ist kompliziert**
  - Erklärung benötigt Theorie vollständiger Halbordnungen (CPOs)
- **Semantik von λ-Termen wird operational erklärt**
  - Entspricht Methodik beim Rechnen auf Zahlen: $4{*}(3{+}2) = 4{*}5 = 20$
  - λ-Terme werden ausgewertet bis irreduzibler 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 genaue Definition eines Substitutionskonzepts auf λ-Termen

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



λ-Kalkül – Berechnung durch Auswertung

- **Ersetze Funktionsparameter durch -argumente**
  - **Substitution** $t[b/x]$: ersetze freie Vorkommen von x in t durch b

    Sonderfälle: $\lfloor\lambda x\,.\,t\rfloor[b/x] = \lambda x\,.\,t$ *x ist nicht frei in t*

    $\lfloor\lambda x\,.\,t\rfloor[b/y] = \lfloor\lambda z\,.\,t[z/x]\rfloor[b/y]$ *$y{\neq}x$ frei in t, x frei in b, z neu*

    d.h. $\lfloor\lambda \mathtt{f}\,.\,\mathtt{f\ x}\rfloor[\mathtt{f/x}] = \lambda \mathtt{g}.\ \mathtt{g\ f}$ *(λf . f f wäre ungewollte Selbstanwendung)*
  - **Reduktion**: $(\lambda x\,.\,t)(b) \xrightarrow{\beta} t[b/x]$
- **Substitution und Reduktion am Beispiel**

$$(\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})(\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})$$

$$= (\lambda \mathtt{n}.(\lambda \mathtt{f}.(\lambda \mathtt{x}.((\mathtt{n\ f})(\mathtt{f\ x})))))(\lambda \mathtt{f}.(\lambda \mathtt{x}.\mathtt{x}))$$ *(Reduktion ersetzt* n*)*

$$\longrightarrow \lambda \mathtt{f}.(\lambda \mathtt{x}.((\lambda \mathtt{f}.(\lambda \mathtt{x}.\mathtt{x}))\ \mathtt{f})(\mathtt{f\ x}))$$

$$\longrightarrow \lambda \mathtt{f}.(\lambda \mathtt{x}.((\lambda \mathtt{x}.\mathtt{x})(\mathtt{f\ x})))$$

$$\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{f\ x}$$

$$\Downarrow$$

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

Vom λ-Kalkül zu echten Programmen

- **λ-Kalkül ist der Basismechanismus**
  - Die Assemblersprache funktionaler Programme
  - Spezialhardware (Lisp-Maschinen) kann λ-Terme direkt auswerten
- **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)
- **Die wichtigsten Strukturen sind leicht codierbar**
  - Boolesche Operationen: T, F, $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$
  - Tupel / Projektionen: (s,t), $p.1$, $p.2$, $\mathsf{match}\ p\ \mathsf{with}\ (x,y) \mapsto t$
  - 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	$\equiv$	$\lambda \mathtt{x}\,.\,\lambda \mathtt{y}\,.\,\mathtt{x}$	*Term für "true"*
F	$\equiv$	$\lambda \mathtt{x}\,.\,\lambda \mathtt{y}\,.\,\mathtt{y}$	*Term für "false"*
$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$	$\equiv$	$b\,s\,t$	*Term für Konditional*

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

$\mathsf{if}\ \mathsf{T}\ \mathsf{then}\ s\ \mathsf{else}\ t$	$\mathsf{if}\ \mathsf{F}\ \mathsf{then}\ s\ \mathsf{else}\ t$
$\equiv\ \mathsf{T}\,s\,t$	$\equiv\ \mathsf{F}\,s\,t$
$\equiv\ (\lambda \mathtt{x}\,.\,\lambda \mathtt{y}\,.\,\mathtt{x})\,s\,t$	$\equiv\ (\lambda \mathtt{x}\,.\,\lambda \mathtt{y}\,.\,\mathtt{y})\,s\,t$
$\longrightarrow\ (\lambda \mathtt{y}\,.\,s)\,t$	$\longrightarrow\ (\lambda \mathtt{y}\,.\,\mathtt{y})\,t$
$\longrightarrow\ s$	$\longrightarrow\ t$

Zeige: wenn $\mathsf{F} = \mathsf{T}$ wäre, dann sind alle λ-Terme gleich (Übung)

Bildung und Analyse von Paaren

$$
\begin{array}{lcl}
(u\,,v) & \equiv & \lambda \mathtt{p}.\ \mathtt{p}\ u\ v \\
pair.1 & \equiv & pair\ \ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}) \\
pair.2 & \equiv & pair\ \ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}) \\
\mathsf{match}\ pair\ \mathsf{with}\ (x\,,y) \mapsto t & \equiv & pair\ \ (\lambda x.\lambda y.t)
\end{array}
$$

(uniformer Analyseoperator, oft eleganter in Anwendung)

Der Analyseoperator ist invers zur Paarbildung

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

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 $\mathsf{add}\ \overline{m}\ \overline{n}$ 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 $\overline{n+1}$

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

- **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:**

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

- **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 $f\ (s\ n) = h\ (f\ n)$

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 $\boldsymbol{f\,x} = (R\,(\lambda f.\lambda x.t))\,x = (\lambda f.\lambda x.t)\,f\,x \overset{*}{\longrightarrow} \boldsymbol{t[f,x]}$
  - 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**
  - $\mathsf{Y} \equiv \lambda\mathtt{f}\,.\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))$

Es gilt $\mathsf{Y}\ t \equiv (\lambda\mathtt{f}\,.\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x})))\ t$

$\longrightarrow (\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x}))$

$\longrightarrow t\ ((\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x})))$

$t\ (\mathsf{Y}\ t) \equiv t\ ((\lambda\mathtt{f}\,.\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))\,(\lambda\mathtt{x}\,.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x})))\ t)$

$\longrightarrow t\ ((\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}\,.\,t\,(\mathtt{x}\,\mathtt{x})))$

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 (λ`fak.`λ`n. 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` $\equiv$ λ`n.`λ`m. zero(sub (s n) m)`
- **Division:** *Suche das erste* z *mit* $n<(z+1) * m$. *Starte Suche bei 0*
  - `div` $\equiv$ λ`n.`λ`m.(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))`

Der λ-Kalkül ist Turing-mächtig

Alle μ-rekursiven Funktionen sind λ-berechenbar

- **Nachfolgerfunktion s:** $\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$
- **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 $\mathtt{min}\ (\mathtt{s}\ \mathtt{y}))\ \overline{0}$

Das Halteproblem

Terminierung von λ-Termen ist unentscheidbar

- **Beweis stützt sich auf wenige Erkenntnisse**
  - $\lambda x.x$ terminiert mit Wert $\lambda x.x$
  - $\bot = \mathsf{Y}(\lambda x.x)$ 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

Mehr zum Halteproblem und ähnlichen Unlösbarkeiten in Einheit 5.5

Rechtfertigung logischer Programmiersprachen

- **Spezifikation von Funktionen in logischem Kalkül**
  - Formeln repräsentieren Ein-/Ausgabeverhalten von Funktionen
  - Repräsentation muß eindeutig sein (nur eine Ausgabe pro Eingabe)
  - Eindeutigkeit muß ausschließlich aus logischen Axiomen beweisbar sein
- **Zentraler Begriff: Gültigkeit in einer Theorie**
  - Logische **Theorie T** gegeben durch formale Sprache und Axiome
  - Formel F ist gültig in T ($\models_T F$), wenn F logisch aus den Axiomen folgt
- **Berechenbarkeitsbegriff: $f{:}\mathbb{N}^k \to \mathbb{N}$ repräsentierbar in T**
  - Es gibt eine Formel F mit $\models_T F(\overline{i_1}, .., \overline{i_k}, \overline{j})$ g.d.w. $f(i_1, .., i_k) = j$, d.h. in der Theorie T ist beweisbar, ob f einen bestimmten Wert annimmt
  - $\overline{n}$ ist ein Term der formalen Sprache, der die Zahl n codiert

Die arithmetische Theorie $\mathcal{Q}$

- **Formale Sprache**
  - Sprache der Prädikatenlogik (mit Gleichheit)
  - Konstantensymbol $\overline{0}$
  - Einstelliges Funktionssymbol s
  - Zweistellige Funktionssymbole $+$ und $*$
- **Semantik: Logik + 7 Axiome** (ohne Induktion!)

Q_1: $\forall x, y.\ \mathsf{s}(x) = \mathsf{s}(y) \Rightarrow x = y$

Q_2: $\forall x.\ \mathsf{s}(x) \neq \overline{0}$

Q_3: $\forall x.\ x \neq \overline{0} \Rightarrow \exists y. x = \mathsf{s}(y)$

Q_4: $\forall x.\ x + \overline{0} = x$

Q_5: $\forall x, y.\ x + \mathsf{s}(y) = \mathsf{s}(x + y)$

Q_6: $\forall x.\ x * \overline{0} = \overline{0}$

Q_7: $\forall x, y. x * \mathsf{s}(y) = (x * y) + x$

- **Axiome gelten auch für Nichtstandardzahlen**
  - Es sind auch andere Interpretationen der Symbole s, $+$, $*$ möglich
    Definiere Operationen s, $+$, $*$ auf $\mathbb{N} \cup \{\infty, \infty'\}$
    Kommutativität, Assoziativität müssen auf $\mathbb{N} \cup \{\infty, \infty'\}$ nicht gelten

Dennoch kann man alle berechenbaren Funktionen in $\mathcal{Q}$ repräsentieren

Repräsentierbarkeit in $\mathcal{Q}$

- **$f{:}\mathbb{N}^k \to \mathbb{N}$ arithmetisch repräsentierbar**
  - f ist repräsentierbar in $\mathcal{Q}$, wobei $n \in \mathbb{N}$ codiert als $\overline{n} \equiv \underbrace{\mathsf{s}(\ldots(\mathsf{s}(\overline{0}))\ldots)}_{n-mal}$
- **Addition ist arithmetisch repräsentierbar**
  - Bestimme 3-stellige Formel ADD mit $i{+}j{=}k$ gdw. $\models_{\mathcal{Q}} \mathtt{ADD}(\overline{i},\overline{j},\overline{k})$

    Einfach, da + Teil der Sprache ist: $\mathtt{ADD}(x_1,x_2,y) \equiv y{=}x_1{+}x_2$

Korrektheit der Repräsentation

Zeige: für alle $i,j,k \in \mathbb{N}$ mit $i{+}j{=}k$ gilt $\models_{\mathcal{Q}} \overline{k}{=}\overline{i}{+}\overline{j}$ $(\hat{=}\ \mathtt{ADD}(\overline{i},\overline{j},\overline{k}))$

(Beweis für $i{+}j{\neq}k$ impliziert $\models_{\mathcal{Q}} \overline{k}{\neq}\overline{i}{+}\overline{j}$ ist analog)

Sei i beliebig, aber fest. Wir führen den Beweis durch Induktion über j:

– Für $j = 0$ folgt $i{=}k$, also $\overline{i}{=}\overline{k}$ und über Axiom Q_4: $\models_{\mathcal{Q}} \overline{i}{+}\overline{0}{=}\overline{i}$

– Es gelte $\models_{\mathcal{Q}} \overline{n}{=}\overline{i}{+}\overline{j}$ für alle n und $j{=}m \in \mathbb{N}$ mit $i{+}j{=}n$ und es gelte $i{+}j{=}k$.

Dann gilt $k = i{+}m{+}1 = n{+}1$ und $\overline{k}{=}\mathsf{s}(\overline{n})$.

Mit Axiom Q_5 folgt $\models_{\mathcal{Q}} \overline{i}{+}\overline{j} = \overline{i}{+}\mathsf{s}(\overline{m}) = \mathsf{s}(\overline{i}{+}\overline{m}) = \mathsf{s}(\overline{n}) = \overline{k}$

Wichtige repräsentierbare Funktionen

- **Multiplikation** $\qquad \mathtt{MUL}(x_1, x_2, y) \;\equiv\; y\mathtt{=}x_1\mathtt{*}x_2$

- **Vergleich** $\leq$ *(Hilfsprädikat für Funktionsbeschreibungen)* $\qquad \mathtt{LE}(x, y) \;\equiv\; \exists \mathtt{z}\,.\, x\mathtt{+z=}y$

  $<$ $\qquad \mathtt{LT}(x, y) \;\equiv\; \mathtt{LE}(\mathtt{s}(x), y)$

- **Subtraktion** $\qquad \mathtt{SUB}(x_1, x_2, y) \;\equiv\; x_1\mathtt{=}x_2\mathtt{+}y \;\vee\; (\mathtt{LE}(x_1, x_2) \wedge y\mathtt{=}\overline{\mathtt{0}})$

- **Division** $\qquad \mathtt{DIV}(x_1, x_2, q) \;\equiv\; \exists \mathtt{r}\,.\, \mathtt{LT(r,}x_2\mathtt{)} \wedge x_2\mathtt{*}q\mathtt{+r=}x_1$

  Divisionsrest/Modulo

  $\mathtt{MOD}(x_1, x_2, r) \;\equiv\; \mathtt{LT}(r\mathtt{,}x_2\mathtt{)} \;\wedge\; \exists \mathtt{q}\,.\, x_2\mathtt{*q+}r\mathtt{=}x_1$

- **Teilbarkeit** *(Prädikat)* $\qquad \mathtt{DIVIDES}(x_1, x_2) \;\equiv\; \exists \mathtt{q}\,.\, x_1\mathtt{*q=}x_2$

  Primzahleigenschaft *(Prädikat)*

  $\mathtt{PRIME}(x) \;\equiv\; \mathtt{LT(}\overline{\mathtt{1}}\mathtt{,}x\mathtt{)} \wedge \forall \mathtt{y}\,.\,(\mathtt{LT(y,}x\mathtt{)} \wedge \mathtt{LT(}\overline{\mathtt{1}}\mathtt{,y))} \Rightarrow \neg\,\mathtt{DIVIDES(y,}x\mathtt{)}$

Repräsentierbarkeit in $\mathcal{Q}$ ist Turing-mächtig

– Alle μ-rekursiven Funktionen sind repräsentierbar in $\mathcal{Q}$ $\qquad \mapsto$ Anhang

Weitere Modelle für Berechenbarkeit

- **Abakus**
  - Erweiterung des mechanischen Abakus: beliebig viele Stangen / Kugeln
  - Zwei Operationen: Kugel hinzunehmen / Kugel wegnehmen
- **Registermaschinen**
  - Direkter Zugriff auf endliche Zahl von Registern
  - Register enthalten (unbegrenzte) natürliche Zahlen
  - Befehle entsprechen elementarem Assembler
- **Mini-PASCAL / mini-JAVA**
  - Basisversion einer imperativen höheren Programmiersprache
  - Arithmetische Operationen, Fallunterscheidung, Schleifen
  - Operationale Semantik erklärt Bedeutung der Befehle
- **Markov-Algorithmen**
  - Wie Typ-0 Grammatiken, aber mit fester Strategie für Regelanwendung
  - Verarbeitet Eingabeworte, statt mit einem Startsymbol zu beginnen

Alle Modelle sind ebenfalls Turing-mächtig

Die Churchsche These

- **Alle Berechenbarkeitsmodelle sind äquivalent**
  - Keines kann mehr berechnen als Turingmaschinen
  - Es ist keine Funktion bekannt, die man intuitiv als berechenbar ansieht, aber nicht mit einer Turingmaschine berechnen kann

- **These von Alonzo Church: Die Klasse der Turing-berechenbaren Funktionen ist identisch mit der Klasse der intuitiv berechenbaren Funktionen**
  - **Unbeweisbar**e, aber wahrscheinlich richtige Behauptung
  - **Arbeitshypothese** für theoretische Argumente, die es ermöglicht, in Beweisen intuitiv formulierte Programme anstelle von konkreten Turingmaschinen zu verwenden

Berechenbarkeitsmodelle im Rückblick

- **Es gibt viele äquivalente Modelle**
  - Maschinenbasierte Modelle: Turingmaschinen, Registermaschinen, ...
  - Programmiersprachenbasierte Modelle: Mini-PASCAL, Mini-Java, ...
  - Abstrakte mathematische Beschreibung: rekursive Funktionen
  - Funktionale Programmierung: λ-Kalkül
  - Logische Programmierung: Arithmetische Repräsentierbarkeit
- **Alle Berechenbarkeitsmodelle sind i.w. äquivalent**
  - These: Alle berechenbaren Funktionen sind Turing-berechenbar (oder rekursiv, λ-berechenbar, arithmetisch repräsentierbar, ...)
  - Die Theorie des Berechenbaren hängt nicht vom konkreten Modell ab, sondern basiert auf allgemeinen Eigenschaften, die alle Modelle (implizit) gemeinsam haben

ANHANG

Repräsentierbarkeit in $\mathcal{Q}$ ist Turing-mächtig (I)

Definiere Min-rekursive Funktionen

Definition rekursiver Funktionen ohne primitive Rekursion

- $\mathcal{R}_{min}$: **Menge der min-rekursiven Funktionen**
  - Addition, Nachfolger, Projektions- oder Konstantenfunktion sowie
  - Alle Funktionen, die aus min-rekursiven Funktionen durch Komposition oder Minimierung entstehen

  Wichtiger Sonderfall für Vergleiche mit anderen Modellen

- $\mathcal{R}_{min} \subseteq \mathcal{R}$: min-rekursive Funktionen sind μ-rekursiv
  - Offensichtlich, da Addititon μ-rekursiv ist

- $\mathcal{R} \subseteq \mathcal{R}_{min}$: μ-rekursive Funktionen sind min-rekursiv
  - Beschreibe Abarbeitung des Stacks einer primitiven Rekursion
  - Suche nach erstem erzeugten Stack der Länge 1 (Details aufwendig)

Repräsentierbarkeit in $\mathcal{Q}$ ist Turing-mächtig (II)

Alle min-rekursiven Funktionen sind repräsentierbar

- **Nachfolgerfunktion s:** $\mathtt{S}(x,y) \equiv y{=}\mathtt{s}(x)$
- **Projektionsfunktionen pr^n_m:** $\mathtt{PR}^n_m(x_1,..,x_n,y) \equiv y{=}x_m$
- **Konstantenfunktion c^n_m:** $\mathtt{C}^n_m(x_1,..,x_n,y) \equiv y{=}\overline{m}$
- **Addition add:** $\mathtt{ADD}(x_1,x_2,y) \equiv y{=}x_1{+}x_2$
- **Komposition $f \circ (g_1,..,g_n)$:**

$$H(\vec{x},z) \equiv \exists y_1,..,y_n.(G_1(\vec{x},y_1) \wedge .. \wedge G_n(\vec{x},y_n) \wedge F(y_1,..,y_n,z))$$

H repräsentiert $f \circ (g_1,..,g_n)$, wenn $F, G_1, ..G_n$ Repräsentationen von $f, g_1, .., g_n$

- **Minimierung $\mu[f]$:**

$$H(\vec{x},y) \equiv \forall \mathtt{w}.\ \mathtt{LE(w,}y\mathtt{)} \Rightarrow [F(\vec{x},y,\overline{0}) \Leftrightarrow \mathtt{w}{=}y]$$

H repräsentiert $\mu[f]$, wenn F Repräsentation von f