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:

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

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

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

     $(\lambda\mathtt{x}.\ \mathtt{2*x+3})\mathtt{(4)} \xrightarrow{\beta} \mathtt{2*4+3} \xrightarrow{*} \mathtt{11}$ **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 — $\lambda x\,.\,f\,t \;\hat{=}\; \lambda x\,.\,(f\,t)$
  - Applikation ist links-assoziativ: — $f\ t_1\,t_2 \;\hat{=}\; (f\ t_1)\,t_2$

- **Beispiele für λ-Terme** (mit impliziten und ‘kosmetischen’ Klammern)
  - `x` — Symbole sind immer Variablen
  - λ`f.(`λ`x.(f(x)))` — Anwendung einer Funktion
  - λ`f.(`λ`g.(`λ`x.(((f g) (g x)))))` — Funktionen höherer Ordnung
  - `x(x)` — Selbstanwendung
  - `(`λ`x.x(x)) (`λ`x.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 f und t bleiben frei
(t)	gebundene Vorkommen von x bleiben gebunden

$$\overbrace{\texttt{}\lambda\texttt{f.}\lambda\texttt{x.}\underbrace{\texttt{(}\lambda\texttt{z. f x z) x}}_{\texttt{x}\ \textit{frei}}}^{\texttt{x}\ \textit{gebunden}}$$

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

- **Substitution und Reduktion am Beispiel**

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

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

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

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

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

$$\Downarrow$$

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

Reduktion ist nicht deterministisch

Naheliegende Reduktionsreihenfolge

$$
\begin{array}{ll}
 & \texttt{(}\lambda\texttt{f.}\lambda\texttt{x. f x (f f)) (}\lambda\texttt{x.}\lambda\texttt{y. y)} \\
\longrightarrow & \lambda\texttt{x. (}\lambda\texttt{x.}\lambda\texttt{y. y) x ((}\lambda\texttt{x.}\lambda\texttt{y. y)(}\lambda\texttt{x.}\lambda\texttt{y. y))} \\
\longrightarrow & \lambda\texttt{x. (}\lambda\texttt{y. y) ((}\lambda\texttt{x.}\lambda\texttt{y. y)(}\lambda\texttt{x.}\lambda\texttt{y. y))} \\
\longrightarrow & \lambda\texttt{x. ((}\lambda\texttt{x.}\lambda\texttt{y. y)(}\lambda\texttt{x.}\lambda\texttt{y. y)} \\
\longrightarrow & \lambda\texttt{x. }\lambda\texttt{y. y}
\end{array}
$$

Alternative Reduktionsreihenfolgen sind möglich

$$
\begin{array}{ll}
 & \texttt{(}\lambda\texttt{f.}\lambda\texttt{x. f x (f f)) (}\lambda\texttt{x.}\lambda\texttt{y. y)} \\
\longrightarrow & \lambda\texttt{x. (}\lambda\texttt{x.}\lambda\texttt{y. y) x ((}\lambda\texttt{x.}\lambda\texttt{y. y)(}\lambda\texttt{x.}\lambda\texttt{y. y))} \\
\longrightarrow & \lambda\texttt{x. (}\lambda\texttt{x.}\lambda\texttt{y. y) x (}\lambda\texttt{y. y)} \\
\longrightarrow & \lambda\texttt{x. (}\lambda\texttt{y. y)(}\lambda\texttt{y. y)} \\
\longrightarrow & \lambda\texttt{x. }\lambda\texttt{y. y}
\end{array}
$$

Bekommt man immer das gleiche Ergebnis?

Wichtige Eigenschaften von λ-Termen

- **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: `(λx. x x) (λx. 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

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.y)\ ((\lambda x.\,x\,x\,x)\ (\lambda x.\,x\,x\,x))\ (\lambda x.\,x) \\
\longrightarrow & (\lambda y.y)\ (\lambda x.\,x) \\
\longrightarrow & \lambda x.\,x
\end{array}
$$

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

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.y)\ ((\lambda x.x\,x\,x)\ (\lambda x.x\,x\,x))\ (\lambda x.x) \\
\stackrel{*}{\longrightarrow} & (\lambda x.\lambda y.y)((\lambda x.x\,x\,x)\ (\lambda x.x\,x\,x)\ (\lambda x.x\,x\,x))\ (\lambda x.x) \\
\stackrel{*}{\longrightarrow} & (\lambda x.\lambda y.y)((\lambda x.x\,x\,x)(\lambda x.x\,x\,x)(\lambda x.x\,x\,x)\ (\lambda x.x\,x\,x))\ (\lambda x.x) \\
 & \qquad\qquad\vdots \qquad\qquad\qquad \vdots
\end{array}
$$

- **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**
  - $\boldsymbol{t = u}$: 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

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, $\mathsf{if}\ \boldsymbol{b}\ \mathsf{then}\ \boldsymbol{s}\ \mathsf{else}\ \boldsymbol{t}$
  - Tupel / Projektionen: $(\boldsymbol{s},\boldsymbol{t})$, $\mathsf{fst}(\boldsymbol{p})$, $\mathsf{snd}(\boldsymbol{p})$, $\mathsf{match}\ \boldsymbol{p}\ \mathsf{with}\ (\boldsymbol{x},\boldsymbol{y}) \mapsto \boldsymbol{t}$
  - Zahlen und arithmetische Operationen
  - Iteration oder Rekursion von Funktionen
- **Codierung verwendet definitorische Erweiterungen**
  - Neue Sprachkonstrukte sind definitorische Abkürzung für existierende λ-Terme, ggf. mit Parametern (ähnlich zu Macros)
    z.B. doppelte Funktionsanwendung $\mathtt{twice}(f) \equiv \lambda\mathtt{x}.\ f(f(\mathtt{x}))$

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

$$\begin{aligned} & \mathsf{if}\ \mathsf{T}\ \mathsf{then}\ s\ \mathsf{else}\ t \\ \equiv\ & \mathsf{T}\,s\,t \\ \equiv\ & (\lambda x.\lambda y.x)\,s\,t \\ \longrightarrow\ & (\lambda y.s)\,t \\ \longrightarrow\ & s \end{aligned}$$

$$\begin{aligned} & \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ s\ \mathsf{else}\ t \\ \equiv\ & \mathsf{F}\,s\,t \\ \equiv\ & (\lambda x.\lambda y.y)\,s\,t \\ \longrightarrow\ & (\lambda y.y)\,t \\ \longrightarrow\ & t \end{aligned}$$

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

Bildung und Analyse von Paaren

$$(u,v) \equiv \lambda \texttt{op}.\ \texttt{op}\ u\ v$$

$\mathsf{fst}(p)$ *(auch $p.1$ oder p_1)* $\equiv p\ (\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x})$

$\mathsf{snd}(p)$ *(auch $p.2$ oder p_2)* $\equiv p\ (\lambda \texttt{x}.\lambda \texttt{y}.\texttt{y})$

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

(uniformer Analyseoperator, oft eleganter in Anwendung)

Der Analyseoperator ist invers zur Paarbildung

$$\begin{aligned}
& \mathsf{match}\ (u,v)\ \mathsf{with}\ (x,y) \mapsto t \\
\equiv\ & (u,v)(\lambda x.\lambda y.t) \\
\equiv\ & (\lambda \texttt{op}.\ \texttt{op}\ u\ v)(\lambda x.\lambda y.t) \\
\longrightarrow\ & (\lambda x.\lambda y.t)\ u\ v \\
\longrightarrow\ & (\lambda y.t[u/x])\ v \\
\longrightarrow\ & t[u,v/x,y]
\end{aligned}$$

(kurz für $t[u/x][v/y]$)

Simulation von Zahlen und Arithmetik

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

    (Markierung $\overline{n}$ hilft, eine Verwechselung von Zahlen und zugehörigen λ-Termen zu vermeiden)
  - Bezeichnung: **Church Numerals**

- **f:$\mathbb{N}^n \rightarrow \mathbb{N}$ 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 $\texttt{add}\;\overline{m}\;\overline{n}$ 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 $\mathsf{s}\ \overline{\boldsymbol{n}}$ ist der Term $\overline{\boldsymbol{n}\mathsf{+1}}$

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

- **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:**

  $\mathsf{p} \equiv \lambda\mathtt{n}.\ \mathsf{snd}((\mathtt{n}\ (\lambda\mathtt{fx}.\ (\mathsf{s},\ \mathsf{match}\ \mathtt{fx}\ \mathsf{with}\ (\mathtt{f},\mathtt{x}) \mapsto \mathtt{f}\ \ \mathtt{x}))\ (\lambda\mathtt{z}.\overline{0},\ \overline{0})))$
- **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 des Programms für die Addition

- **Zeige:** add $\overline{m}$ $\overline{n}$ **reduziert zu** $\overline{m+n}$

$$
\begin{array}{lcl}
\mathsf{add}\ \overline{m}\ \overline{n} & \equiv & (\lambda\mathtt{m}.\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)})\ \overline{m}\ \overline{n} \\
& \longrightarrow & (\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \overline{m}\ \mathtt{f\ (n\ f\ x)})\ \overline{n} \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \overline{m}\ \mathtt{f}\ (\overline{n}\ \mathtt{f\ x}) \\
& \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ \mathtt{f}\ (\overline{n}\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ (\overline{n}\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^m\,(\overline{n}\ \mathtt{f\ x}) \\
& \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^m\,((\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^m\,((\lambda\mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{x}) \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^m\,(\mathtt{f}^n\,\mathtt{x}) \\
& \longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^{m+n}\,\mathtt{x} \qquad\qquad \equiv \quad \overline{m+n}
\end{array}
$$

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

    z.B. Fakultätsfunktion `n! = if n=0 then 1 else n*(n-1)!`

- **Darstellung: Anwendung eines Fixpunktkombinators auf t**
  - Fixpunktkombinator: λ-Term R, der jede rekursive Funktion f mit der Eigenschaft $f(x)=t[f,x]$ aus dem Funktionskörper t und den Variablenbindungen f und x konstruieren kann

    d.h. der λ-Term $F \equiv R(\lambda f.\lambda x.t)$ erfüllt für alle x die Gleichung

    $$R(\lambda f.\lambda x.t)\,x \equiv F\,x = t[F,x] \equiv t[R(\lambda f.\lambda x.t),x]$$

    Eine noch allgemeinere Forderung ist: $\boldsymbol{R\,u=u\,(R\,u)}$ für jeden λ-Term u
  - Programmier notation für $R(\lambda f.\lambda x.t)$ ist "function $f(x)=t$"

    d.h. function $f(x)=t$ ist definitorische Abkürzung für den Term $R(\lambda f.\lambda x.t)$

    Analog ist "function $f(x,y)=t$" Abkürzung für $R(\lambda f.\lambda x.\lambda y.t)$

Gibt es solche Fixpunktkombinatoren und wie programmiert man mit ihnen?

Fixpunktkombinatoren gibt es tatsächlich

- **Bekanntester Fixpunktkombinator ist der Y-Kombinator**
  – $\mathsf{Y} \equiv \lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x}))\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x}))$

  Es gilt
  $$\begin{aligned} \mathsf{Y}\ t &\equiv (\lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x}))\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x})))\ t \\ &\longrightarrow (\lambda\mathtt{x}.\,t\,(\mathtt{x\,x}))\ (\lambda\mathtt{x}.\,t\,(\mathtt{x\,x})) \\ &\longrightarrow t\ ((\lambda\mathtt{x}.\,t\,(\mathtt{x\,x}))\ (\lambda\mathtt{x}.\,t\,(\mathtt{x\,x}))) \\ t\ (\mathsf{Y}\ t) &\equiv t\ ((\lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x}))\,(\lambda\mathtt{x}.\,\mathtt{f}\,(\mathtt{x\,x})))\ t) \\ &\longrightarrow t\ ((\lambda\mathtt{x}.\,t\,(\mathtt{x\,x}))\ (\lambda\mathtt{x}.\,t\,(\mathtt{x\,x}))) \end{aligned}$$

  – Es gilt also $\mathsf{Y}\,t = t\,(\mathsf{Y}\,t)$ für jeden beliebigen Term t

  Das scheint widersinnig, wenn t z.B. die Nachfolgerfunktion ist

- **Es gibt auch andere Fixpunktkombinatoren**
  – $\boldsymbol{T} \equiv (\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y}\,(\mathtt{x\,x\,y}))\,(\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y}\,(\mathtt{x\,x\,y}))$

  hier gilt sogar $\boldsymbol{T}\,t \xrightarrow{*} t\,(\boldsymbol{T}\,t)$ für jeden beliebigen Term t

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))

Der λ-Kalkül ist Turing-mächtig

Alle μ-rekursiven Funktionen sind λ-berechenbar

- **Nachfolgerfunktion s:** $\quad \mathtt{s} \equiv \lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n\ f\ (f\ x)}$

- **Projektionsfunktionen pr_m^n:** $\quad \mathsf{pr}_m^n \equiv \lambda\mathtt{x}_1..\lambda\mathtt{x}_n.\ \mathtt{x}_m$

- **Konstantenfunktion c_m^n:** $\quad \mathsf{c}_m^n \equiv \lambda\mathtt{x}_1..\lambda\mathtt{x}_n.\ \overline{\mathtt{m}}$

- **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 $Pr[f, g]$:**

  $\mathsf{PR} \equiv \lambda\mathtt{f}.\lambda\mathtt{g}.\ \mathsf{function}\ \mathtt{h(x)} = \lambda\mathtt{y}.\mathsf{if\ zero}\ \mathtt{y}\ \mathsf{then}\ \mathtt{f\ x}\ \mathsf{else}\ \mathtt{g\ x\ (p\ y)\ (h\ x\ (p\ y))}$

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

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

Das Halteproblem

Terminierung von λ-Termen ist unentscheidbar

- **Beweis stützt sich auf wenige Erkenntnisse**
  - $\lambda x.x$ terminiert mit Wert $\lambda x.x$
  - $\perp = \mathsf{Y}(\lambda x.x)$ 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 $d = \lambda x.\, \mathsf{if}\ h(x)\ \mathsf{then}\ \perp\ \mathsf{else}\ \lambda x.x.$
  - Dann $h(\mathsf{Y}d) = h(d(\mathsf{Y}\, d)) = h(\mathsf{if}\ h(\mathsf{Y}\, d)\ \mathsf{then}\ \perp\ \mathsf{else}\ \lambda x.x)$

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

Mehr zum Halteproblem und ähnlichen Unlösbarkeiten in Einheit 5.5

Arithmetische Repräsentierbarkeit

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 ($\models_T \boldsymbol{F}$), wenn F logisch aus den Axiomen folgt
- **Berechenbarkeitsbegriff: $f{:}\mathbb{N}^k \to \mathbb{N}$ repräsentierbar in T**
  - Es gibt eine Formel F mit $\models_T \boldsymbol{F}(\overline{i_1}, .., \overline{i_k}, \overline{j})$ g.d.w. $\boldsymbol{f}(i_1, .., i_k) = \boldsymbol{j}$,
    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 $\mathcal{Q}$

- $f{:}\mathbb{N}^k \rightarrow \mathbb{N}$ **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** $\quad \texttt{MUL}(x_1,x_2,y) \;\equiv\; y\texttt{=}x_1\texttt{*}x_2$
- **Vergleich** $\leq$ *(Hilfsprädikat für Funktionsbeschreibungen)* $\quad \texttt{LE}(x,y) \;\equiv\; \exists\texttt{z}.\,x\texttt{+z=}y$

  $<$ $\quad \texttt{LT}(x,y) \;\equiv\; \texttt{LE}(\texttt{s}(x),y)$
- **Subtraktion** $\quad \texttt{SUB}(x_1,x_2,y) \;\equiv\; x_1\texttt{=}x_2\texttt{+}y \;\vee\; (\texttt{LE}(x_1,x_2) \wedge y\texttt{=}\overline{\texttt{0}})$
- **Division** $\quad \texttt{DIV}(x_1,x_2,q) \;\equiv\; \exists\texttt{r}.\,\texttt{LT}(\texttt{r},x_2) \wedge x_2\texttt{*}q\texttt{+r}=x_1$

  Divisionsrest/Modulo

  $\texttt{MOD}(x_1,x_2,r) \;\equiv\; \texttt{LT}(r,x_2) \;\wedge\; \exists\texttt{q}.\,x_2\texttt{*q+}r=x_1$
- **Teilbarkeit** *(Prädikat)* $\quad \texttt{DIVIDES}(x_1,x_2) \;\equiv\; \exists\texttt{q}.\,x_1\texttt{*q=}x_2$

  Primzahleigenschaft *(Prädikat)*

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

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

– Alle μ-rekursiven Funktionen sind repräsentierbar in $\mathcal{Q}$ $\quad \mapsto$ 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 Additition 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:** $\quad \mathtt{S}(x,y) \equiv y\mathtt{=}\mathtt{s}(x)$

- **Projektionsfunktionen pr_m^n:** $\quad \mathtt{PR}_m^n(x_1,..,x_n,y) \equiv y\mathtt{=}x_m$

- **Konstantenfunktion c_m^n:** $\quad \mathtt{C}_m^n(x_1,..,x_n,y) \equiv y\mathtt{=}\overline{m}$

- **Addition add:** $\quad \mathtt{ADD}(x_1,x_2,y) \equiv y\mathtt{=}x_1\mathtt{+}x_2$

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

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

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