

Theoretische Informatik II

Einheit 4.4

Funktionale & Logische Programme



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

Grundlage funktionaler Programmiersprachen

- **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:

$$f \hat{=} \lambda x. 2*x+3$$

λ -Abstraktion

Name der Funktion irrelevant für Beschreibung des Verhaltens

2. Anwendung der Funktion:

$$f(4) \hat{=} (\lambda x. 2*x+3)(4)$$

Applikation

3. Auswertung der Funktion:

$$(\lambda x. 2*x+3)(4) \xrightarrow{\beta} 2*4+3 \xrightarrow{*} 11$$

Reduktion

● Einfache Programmiersprache: λ -Terme

– Variablen x

– $\lambda x . t$, 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

● Prioritäten und Kurzschreibweisen

– Applikation bindet stärker als λ -Abstraktion

– Applikation ist links-assoziativ:

$$f t_1 t_2 \hat{=} (f t_1) t_2$$

– Notation $f(t_1, \dots, t_n)$ steht für iterierte Applikation $f t_1 \dots t_n$

● Beispiele für λ -Terme

– x

Symbole sind immer Variablen

– $\lambda f . \lambda x . f(x)$

– $\lambda f . \lambda g . \lambda x . f g (g x)$

Funktionen höherer Ordnung

– $x(x)$

Selbstanwendung

λ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

- Ersetze Funktionsparameter durch -argumente

- **Reduktion** $(\lambda x.t) (b) \xrightarrow{\beta} t[b/x]$

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

- Substitution und Reduktion am Beispiel

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

SUBSTITUTION PRÄZISIERT

● Vorkommen von Variablen in λ -Termen

- 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_i bleiben frei
- (t) gebundene Vorkommen von x bleiben gebunden

$$\lambda f . \lambda x . \underbrace{(\lambda z . f \ x \ z)}_{x \text{ frei}} \ x$$

x gebunden

● Substitution $u[t/x]$ in λ -Termen

$x[t/x]$	$= t$	$x[t/y]$	$= x$	$(y \neq x)$
$\lambda x . u[t/x]$	$= \lambda x . u$			
$\lambda x . u[t/y]$	$= \llbracket \lambda z . u[z/x] \rrbracket [t/y]$ *	$\lambda x . u[t/y]$	$= \lambda x . \llbracket u[t/y] \rrbracket$ **	
$f u \sigma$	$= f \sigma u \sigma$	$(u) \sigma$	$= (u \sigma)$	

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

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

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**
 - Nicht anders als in konventionellen Computern
 - Datenstrukturen werden als Bitketten codiert
 - Programmstrukturen werden in Sprungbefehle übersetzt
- **Die wichtigsten Strukturen sind leicht codierbar**
 - Boolesche Operationen: **T**, **F**, **if b then s else t**
 - Tupel / Projektionen: **(s, t)** , **$pair.1$** , **$pair.2$** , **let $(x, y) = pair$ in t**
 - Zahlen und arithmetische Operationen
 - Iteration oder Rekursion von Funktionen

DARSTELLUNG BOOLESCHER OPERATOREN IM λ -KALKÜL

Zwei verschiedene Objekte und ein Konditional

$$\mathbf{T} \equiv \lambda x. \lambda y. x$$

$$\mathbf{F} \equiv \lambda x. \lambda y. y$$

$$\text{if } b \text{ then } s \text{ else } t \equiv b s t$$

Konditional ist invers zu T und F

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

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

BILDUNG UND ANALYSE VON PAAREN

$$\begin{aligned}(s, t) &\equiv \lambda p. p s t \\ \text{pair}.1 &\equiv \text{pair } (\lambda x. \lambda y. x) \\ \text{pair}.2 &\equiv \text{pair } (\lambda x. \lambda y. y) \\ \text{let } (x, y) = \text{pair in } t &\equiv \text{pair } (\lambda x. \lambda y. t)\end{aligned}$$

Analyseoperator ist invers zur Paarbildung

$$\begin{aligned}&\text{let } (x, y) = (u, v) \text{ in } t \\ &\equiv (u, v) (\lambda x. \lambda y. t) \\ &\equiv (\lambda p. p u v) (\lambda x. \lambda y. t) \\ &\longrightarrow (\lambda x. \lambda y. t) u v \\ &\longrightarrow (\lambda y. t[u/x]) v \\ &\longrightarrow t[u, v/x, y]\end{aligned}$$

OPERATIONEN AUF NATÜRLICHEN ZAHLEN

- **Darstellung von Zahlen durch iterierte Terme**

- Semantisch: wiederholte Anwendung von Funktionen
- Repräsentiere die Zahl n durch den Term $\lambda f . \lambda x . \underbrace{f (f \dots (f x) \dots)}_{n\text{-mal}}$
- Notation: $\bar{n} \equiv \lambda f . \lambda x . f^n x$
- Bezeichnung: **Church Numerals**

- **$f: \mathbb{N}^n \rightarrow \mathbb{N}$ λ -berechenbar:**

- Es gibt einen λ -Term t mit der Eigenschaft

$$f(x_1, \dots, x_n) = m \Leftrightarrow t \bar{x}_1 \dots \bar{x}_n = \bar{m}$$

- **Operationen müssen Termvielfachheit berechnen**

- z.B. $\text{add } \bar{m} \bar{n}$ muß als Wert immer den Term $\overline{m+n}$ ergeben

PROGRAMMIERUNG IM λ -KALKÜL

- **Nachfolgerfunktion:** $s \equiv \lambda n. \lambda f. \lambda x. n f (f x)$

– Zeige: Der Wert von $s \bar{n}$ ist der Term $\overline{n+1}$

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

- **Addition:** $add \equiv \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$

- **Multiplikation:** $mul \equiv \lambda m. \lambda n. \lambda f. \lambda x. m (n f) x$

- **Test auf Null:** $zero \equiv \lambda n. n (\lambda n. F) T$

- **Vorgängerfunktion:**

$$p \equiv \lambda n. (n (\lambda f x. (s, \text{let } (f, x) = f x \text{ in } f x)) (\lambda z. \bar{0}, \bar{0})).2$$

KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

$$\begin{aligned} \text{add } \overline{m} \ \overline{n} &\equiv (\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)) \ \overline{m} \ \overline{n} \\ &\longrightarrow (\lambda n. \lambda f. \lambda x. \overline{m} \ f \ (n \ f \ x)) \ \overline{n} \\ &\longrightarrow \lambda f. \lambda x. \overline{m} \ f \ (\overline{n} \ f \ x) \\ &\equiv \lambda f. \lambda x. (\lambda f. \lambda x. f^m \ x) \ f \ (\overline{n} \ f \ x) \\ &\longrightarrow \lambda f. \lambda x. (\lambda x. f^m \ x) \ (\overline{n} \ f \ x) \\ &\longrightarrow \lambda f. \lambda x. f^m \ (\overline{n} \ f \ x) \\ &\equiv \lambda f. \lambda x. f^m \ ((\lambda f. \lambda x. f^n \ x) \ f \ x) \\ &\longrightarrow \lambda f. \lambda x. f^m \ ((\lambda x. f^n \ x) \ x) \\ &\longrightarrow \lambda f. \lambda x. f^m \ (f^n \ x) \\ &\longrightarrow \lambda f. \lambda x. f^{m+n} \ x \qquad \equiv \overline{m+n} \end{aligned}$$

Anwendung sogenannter Fixpunktkombinatoren

● Fixpunktkombinator

- λ -Term R mit der Eigenschaft: $R t = t (R t)$ für beliebige Terme t
- Definiert man eine Funktion f durch $F \equiv R (\lambda f. \lambda x. t[f, x])$, wobei im Programmkörper t sowohl f als auch x vorkommen können dann gilt $F x = (\lambda f. \lambda x. t[f, x]) F x \xrightarrow{*} t[F, x]$
- Kurzschreibweise: $F \equiv \text{letrec } f(x) = t \quad (\equiv R(\lambda f. \lambda x. t))$

● Y-Kombinator: $Y \equiv \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$

- Bekanntester Fixpunktkombinator

$$\begin{aligned}
 Y t &\equiv (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) t \\
 &\longrightarrow (\lambda x. t (x x)) (\lambda x. t (x x)) \\
 &\longrightarrow t ((\lambda x. t (x x)) (\lambda x. t (x x)))
 \end{aligned}$$

$$\begin{aligned}
 t (Y t) &\equiv t ((\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) t) \\
 &\longrightarrow t ((\lambda x. t (x x)) (\lambda x. t (x x)))
 \end{aligned}$$

DER λ -KALKÜL IST TURING-MÄCHTIG

Alle μ -rekursiven Funktionen sind λ -berechenbar

- **Nachfolgerfunktion s :** $s \equiv \lambda n. \lambda f. \lambda x. n f (f x)$
- **Projektionsfunktionen pr_m^n :** $pr_m^n \equiv \lambda x_1. \dots \lambda x_n. x_m$
- **Konstantenfunktion c_m^n :** $c_m^n \equiv \lambda x_1. \dots \lambda x_n. \bar{m}$
- **Komposition $f \circ (g_1, \dots, g_n)$:**
 $\circ \equiv \lambda f. \lambda g_1. \dots \lambda g_n. \lambda x. f (g_1 x) \dots (g_n x)$
- **Primitive Rekursion $Pr[f, g]$:**
 $PR \equiv \lambda f. \lambda g.$
 $\text{letrec } h(x) = \lambda y. \text{if zero } y \text{ then } f x \text{ else } g x (p y) (h x (p y))$
- **Minimierung $\mu[f]$:**
 $Mu \equiv \lambda f. \lambda x. (\text{letrec } \text{min}(y) = \text{if zero}(f x y) \text{ then } y \text{ else } \text{min}(s y)) \bar{0}$

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** , wenn sie logisch aus den Axiomen folgt
 - Kurzschreibweise $\models_T F$
- **Eine einfache arithmetische Theorie reicht aus**
 - 7 Axiome für Null $\bar{0}$, Nachfolger s , Addition $+$ und Multiplikation $*$
 - Axiome lassen auch andere Interpretationen dieser Symbole zu (!!)

REPRÄSENTIERBARKEIT IN NUMERISCHER THEORIE

- **Numerische Theorie T**

- Formale Sprache von T enthält mindestens $\bar{0}$ und s
- Interpretation (Bedeutung) der Symbole nicht eindeutig festgelegt
- Axiome geben Randbedingungen an mögliche Interpretationen

- **Numeral \bar{n}**

- Term der formalen Sprache, der die Zahl n codiert: $\bar{n} \equiv \underbrace{s(\dots(s(\bar{0}))\dots)}_{n\text{-mal}}$

- **$f : \mathbb{N}^k \rightarrow \mathbb{N}$ repräsentierbar in T**

Es gibt eine $k+1$ -stellige Formel F , so daß für alle $(i_1, \dots, i_k) \in \mathbb{N}^k$, $j \in \mathbb{N}$ gilt:

$$f(i_1, \dots, i_k) = j \text{ impliziert } \models_T F(\bar{i}_1, \dots, \bar{i}_k, \bar{j})$$

$$f(i_1, \dots, i_k) \neq j \text{ impliziert } \models_T \neg F(\bar{i}_1, \dots, \bar{i}_k, \bar{j})$$

In der Theorie T ist beweisbar, ob f bei einer konkreten Eingabe einen bestimmten Wert annimmt

DIE ARITHMETISCHE THEORIE \mathcal{Q}

● Formale Sprache

- Sprache der Prädikatenlogik (mit Gleichheit)
- Konstantensymbol $\bar{0}$
- Einstelliges Funktionssymbol \mathbf{s}
- Zweistellige Funktionssymbole $+$ und $*$

● Semantik: Logik + 7 Axiome (ohne Induktion!)

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

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

$$Q_2: \forall x. \mathbf{s}(x) \neq \bar{0}$$

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

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

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

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

● Axiome gelten auch für Nichtstandardzahlen

- Definiere Operationen \mathbf{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

BEISPIELE REPRÄSENTIERBARER FUNKTIONEN

● Addition

- Bestimme eine 3-stellige Formel **ADD** mit $i+j=k$ gdw. $\models_{\mathcal{Q}} \text{ADD}(\bar{i}, \bar{j}, \bar{k})$
- Einfach, da Addition vordefiniert: $\text{ADD}(x_1, x_2, y) \equiv y=x_1+x_2$

● Korrektheit der Repräsentation

- Zeige: für alle $i, j, k \in \mathbb{N}$ mit $i+j=k$ gilt $\models_{\mathcal{Q}} \bar{k}=\bar{i}+\bar{j}$ ($\hat{=} \text{ADD}(\bar{i}, \bar{j}, \bar{k})$)
und für alle $i, j, k \in \mathbb{N}$ mit $i+j \neq k$ gilt $\models_{\mathcal{Q}} \bar{k} \neq \bar{i}+\bar{j}$

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

- Für $j = 0$ folgt $i=k$, also $\bar{i}=\bar{k}$ und über Axiom Q_4 : $\models_{\mathcal{Q}} \bar{i}+\bar{0}=\bar{i}$
- Es gelte $\models_{\mathcal{Q}} \bar{n}=\bar{i}+\bar{j}$ für alle n und $j=m \in \mathbb{N}$ mit $i+j=n$
- Es sei $j=m+1$, also $\bar{j}=\mathbf{s}(\bar{m})$ und es gelte $i+j=k$.

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

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

Der Beweis für $i+j \neq k$ impliziert $\models_{\mathcal{Q}} \bar{k} \neq \bar{i}+\bar{j}$ ist analog

BEISPIELE REPRÄSENTIERBARER FUNKTIONEN (II)

- **Multiplikation** $MUL(x_1, x_2, y) \equiv y = x_1 * x_2$
- **Vergleich** \leq (Prädikat) $LE(x, y) \equiv \exists z. x + z = y$
 $<$ $LT(x, y) \equiv LE(s(x), y)$
- **Subtraktion** $SUB(x_1, x_2, y) \equiv x_1 = x_2 + y \vee (LE(x_1, x_2) \wedge y = \bar{0})$
- **Division** (Prädikat) $DIV(x_1, x_2, y) \equiv \exists z. LT(z, x_2) \wedge x_2 * y + z = x_1$
- **Divisionsrest/Modulo**
 $MOD(x_1, x_2, y) \equiv LT(y, x_2) \wedge \exists z. x_2 * z + y = x_1$
- **Teilbarkeit** (Prädikat) $DIVIDES(x_1, x_2) \equiv \exists z. x_1 * z = x_2$
- **Primzahleigenschaft** (Prädikat)
 $PRIME(x) \equiv \forall y. (LT(y, x) \wedge LT(\bar{1}, y)) \Rightarrow \neg DIVIDES(y, x)$

Mehr Beispiele in den Übungen

REPRÄSENTIERBARKEIT IN \mathcal{Q} IST TURING-MÄCHTIG

Alle min-rekursiven Funktionen sind repräsentierbar

- **Nachfolgerfunktion s :** $S(x, y) \equiv y=s(x)$
- **Projektionsfunktionen pr_m^n :** $PR_m^n(x_1, \dots, x_n, y) \equiv y=x_m$
- **Konstantenfunktion c_m^n :** $C_m^n(x_1, \dots, x_n, y) \equiv y=\bar{m}$
- **Addition add :** $ADD(x_1, x_2, y) \equiv y=x_1+x_2$
- **Komposition $f \circ (g_1, \dots, g_n)$:**
$$H(\vec{x}, z) \equiv \exists y_1, \dots, y_n. (G_1(\vec{x}, y_1) \wedge \dots \wedge G_n(\vec{x}, y_n) \wedge F(y_1, \dots, y_n, z))$$

H repräsentiert $f \circ (g_1, \dots, g_n)$, wenn F, G_1, \dots, G_n Repräsentationen von f, g_1, \dots, g_n
- **Minimierung $\mu[f]$:**
$$H(\vec{x}, y) \equiv \forall w. LE(w, y) \Rightarrow [F(\vec{x}, y, \bar{0}) \Leftrightarrow w=y]$$

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

WEITERE MODELLE FÜR BERECHENBARKEIT

● **Abakus**

- Erweiterung des mechanischen Abakus: beliebig viele Stangen und 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**

- 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 ansehen würde, aber nicht mit einer Turingmaschine berechnen kann
- **These von A. Church: Die Klasse der Turing-berechenbaren Funktionen ist identisch mit der Klasse der intuitiv berechenbaren Funktionen**
 - **Unbeweisbar**, aber wahrscheinlich richtige Behauptung
 - **Arbeitshypothese** für theoretische Argumente
erlaubt, intuitiv formulierte Programme in Beweisen zu verwenden

- **Es gibt viele äquivalente Modelle**

- Maschinen-basierte Modelle: Turingmaschinen, Registermaschinen, ...
- Programmiersprachen-basierte 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