

Theoretische Informatik II

Einheit 5.3

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 (ohne Auswertung):

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

Applikation

3. **Auswertung** einer Funktionsanwendung (tatsächliches Ausrechnen):

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

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

– $\lambda f . \lambda x . f(x)$

Anwendung einer Funktion

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

Funktionen höherer Ordnung

– $x(x)$

Selbstanwendung

- **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

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

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

- **Substitution und Reduktion am Beispiel**

$(\lambda n . \lambda f . \lambda x . n \ f \ (f \ x)) \ (\lambda f . \lambda x . x)$

→ $\lambda f . \lambda x . (\lambda f . \lambda x . x) \ f \ (f \ x)$

→ $\lambda f . \lambda x . (\lambda x . x) \ (f \ x)$

→ $\lambda f . \lambda x . (f \ x)$

⇓

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

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*
 - 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

Zwei verschiedene Objekte und ein Test

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

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

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

Konditional(-simulation) ist invers zu T und F

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

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

PAARE: DATENKAPSELUNG UND KOMPONENTENZUGRIFF

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

Analyseoperator ist invers zur Paarbildung

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

- **Darstellung natürlicher 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**

- Simulation einer Funktion auf Darstellung von Zahlen muß Darstellung des Funktionsergebnisses liefern
- z.B. **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:** $zero \equiv \lambda n. n (\lambda n. F) T$

- **Vorgängerkfunktion:**

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

- **Wende Fixpunktkombinator auf Funktionskörper an**

- **Fixpunktkombinator**: λ -Term R mit Eigenschaft $R t = t (R t)$ für alle t
- Definiert man eine Funktion durch $F \equiv R (\lambda f . \lambda x . t[f, x])$, wobei im Programmkörper t sowohl f als auch x vorkommen können, dann gilt

$$F x = (\lambda f . \lambda x . t[f, x]) F x \xrightarrow{*} t[F, x]$$

- Übliche Notation für den Term $R(\lambda f . \lambda x . t)$ ist **letrec** $f(x) = t$

- **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** ($\models_T F$), wenn F logisch aus den Axiomen folgt
- **Berechenbarkeitsbegriff: $f: \mathbb{N}^k \rightarrow \mathbb{N}$ repräsentierbar in T**
 - Es gibt eine Formel F mit $f(i_1, \dots, i_k) = j$ g.d.w. $\models_T F(\bar{i}_1, \dots, \bar{i}_k, \bar{j})$,
d.h. in der Theorie T ist beweisbar, ob f einen bestimmten Wert annimmt
 - \bar{n} ist ein **Term** der formalen Sprache, der die **Zahl n** codiert

• Formale Sprache

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

• Semantik: Logik + 7 Axiome

(ohne Induktion!)

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

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

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

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

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

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

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

• Axiome gelten auch für Nichtstandardzahlen

- Es sind auch andere Interpretationen der Symbole \mathbf{s} , $+$, $*$ möglich

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

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 $\bar{n} \equiv \underbrace{s(\dots(s(\bar{0})))}_{n\text{-mal}}$

- **Addition ist arithmetisch repräsentierbar**

- Bestimme 3-stellige Formel ADD mit $i+j=k$ gdw. $\models_{\mathcal{Q}} \text{ADD}(\bar{i}, \bar{j}, \bar{k})$

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

Korrektheit der Repräsentation

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

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

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

WICHTIGE REPRÄSENTIERBARE FUNKTIONEN

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

Repräsentierbarkeit in \mathcal{Q} ist Turing-mächtig

– Alle μ -rekursiven Funktionen sind repräsentierbar in \mathcal{Q}

↪ 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**, aber wahrscheinlich richtige Behauptung
 - **Arbeitshypothese** für theoretische Argumente, die es ermöglicht, in Beweisen intuitiv formulierte Programme anstelle von konkreten Turingmaschinen zu verwenden

- **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

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 Addition 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 :** $S(x, y) \equiv y = S(x)$
- **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