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

- **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:

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

   Name der Funktion irrelevant für Beschreibung des Verhaltens

2. Anwendung der Funktion (ohne Auswertung):

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

3. Auswertung einer Funktionsanwendung (tatsächliches Ausrechnen):

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

λ-Kalkül – Syntax

- **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: $f\ t_1\ t_2 \mathrel{\hat{=}} (f\ t_1)\ t_2$
  - Notation $f\,(t_1, \ldots, t_n)$ steht für iterierte Applikation $f\ t_1\ \ldots\ t_n$
- **Beispiele für λ-Terme**
  - `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-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

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

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

- **Substitution und Reduktion am Beispiel**

```
        (λn.λf.λx. n f (f x)) (λf.λx.x)
⟶      λf.λx. (λf.λx.x) f (f x)
⟶      λf.λx. (λx.x) (f x)
⟶      λf.λx. (f x)
```

$$\Downarrow$$

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

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 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 Test

$$\begin{array}{lcl} \mathsf{T} & \equiv & \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x} \\ \mathsf{F} & \equiv & \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y} \\ \text{if } b \text{ then } s \text{ else } t & \equiv & b\,s\,t \end{array}$$

Konditional(-simulation) ist invers zu T und F

$$\begin{array}{lcl} & & \text{if } \mathsf{T} \text{ then } s \text{ else } t \\ & \equiv & \mathsf{T}\,s\,t \\ & \equiv & (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x})\;s\,t \\ & \longrightarrow & (\lambda \mathtt{y}.s)\;t \\ & \longrightarrow & s \end{array}$$

$$\begin{array}{lcl} & & \text{if } \mathsf{F} \text{ then } s \text{ else } t \\ & \equiv & \mathsf{F}\,s\,t \\ & \equiv & (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})\;s\,t \\ & \longrightarrow & (\lambda \mathtt{y}.\mathtt{y})\;t \\ & \longrightarrow & t \end{array}$$

Paare: Datenkapselung und Komponentenzugriff

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

Analyseoperator ist invers zur Paarbildung

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

Simulation arithmetischer Operationen

- **Darstellung natürlicher Zahlen durch iterierte Terme**
  - Semantisch: wiederholte Anwendung von Funktionen
  - Repräsentiere die Zahl n durch den Term $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \underbrace{\mathtt{f\ (f\ ..\ (f\ x)\ ..)}}_{n\text{-}mal}$
  - Notation: $\overline{n} \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ \mathtt{x}$
  - Bezeichnung: **Church Numerals**
- **$f{:}\mathbb{N}^n \longrightarrow \mathbb{N}$ 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. add $\overline{m}$ $\overline{n}$ muß als Wert immer den Term $\overline{m+n}$ ergeben

Programmierung im λ-Kalkül

- **Nachfolgerfunktion:** $\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$
  - Zeige: Der Wert von $\mathsf{s}\ \overline{n}$ ist der Term $\overline{n+1}$

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

- **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:**

$$\mathsf{p} \equiv \lambda \mathtt{n}.\ (\mathtt{n}\ (\lambda \mathtt{fx}.\ (\mathsf{s},\ \mathsf{let}\ (\mathtt{f},\mathtt{x}) = \mathtt{fx}\ \mathsf{in}\ \mathtt{f}\ \mathtt{x}))\ (\lambda \mathtt{z}.\overline{0},\ \overline{0}))\ .2$$

Programmierung rekursiver Funktionen

- **Wende Fixpunktkombinator auf Funktionskörper an**
  - Fixpunktkombinator: λ-Term R mit Eigenschaft $R\,t = t\,(R\,t)$ für alle t
  - Definiert man 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

    $$\boldsymbol{F\,x} = (\lambda f.\lambda x.t[f,x])\;F\;x \xrightarrow{*} \boldsymbol{t[F,x]}$$
  - Kurzschreibweise: $F \equiv \text{letrec}\;f\,(x) = t\;\;(\equiv R\,(\lambda f.\lambda x.t))$
- **Y-Kombinator:** $\mathsf{Y} \equiv \lambda \mathtt{f}.\,(\lambda \mathtt{x}.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))\,(\lambda \mathtt{x}.\,\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))$
  - Bekanntester Fixpunktkombinator

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

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

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^n_m:** $\mathsf{pr}^n_m \equiv \lambda \mathtt{x}_1..\lambda \mathtt{x}_n.\ \mathtt{x}_m$
- **Konstantenfunktion c^n_m:** $\mathsf{c}^n_m \equiv \lambda \mathtt{x}_1..\lambda \mathtt{x}_n.\ \overline{\mathtt{m}}$
- **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 $Pr[f, g]$:**

  $\mathsf{PR} \equiv \lambda \mathtt{f}.\lambda \mathtt{g}.$ letrec $\mathtt{h}(\mathtt{x}) = \lambda \mathtt{y}.$ if zero $\mathtt{y}$ then $\mathtt{f}\ \mathtt{x}$ else $\mathtt{g}\ \mathtt{x}\ (\mathsf{p}\ \mathtt{y})\ (\mathtt{h}\ \mathtt{x}\ (\mathsf{p}\ \mathtt{y}))$
- **Minimierung $\mu[f]$:**

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

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

- $f{:}\mathbb{N}^k \longrightarrow \mathbb{N}$ **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 $\models_{\mathcal{Q}} \overline{k}{=}\overline{i}{+}\overline{j}$ $\quad (\hat{=}\ \mathtt{ADD}\,(\overline{i},\overline{j},\overline{k}))$

  (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 $i{+}j{=}n$
- Es sei $j{=}m{+}1$, also $\overline{j}{=}\mathsf{s}(\overline{m})$ 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** $\quad \mathtt{MUL}(x_1, x_2, y) \equiv y{=}x_1{*}x_2$
- **Vergleich** $\leq$ *(Hilfsprädikat für Funktionsbeschreibungen)* $\quad \mathtt{LE}(x, y) \equiv \exists \mathtt{z}\,.\, x{+}\mathtt{z}{=}y$

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

  Divisionsrest/Modulo

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

  Primzahleigenschaft *(Prädikat)*

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

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

– Alle μ-rekursiven Funktionen sind repräsentierbar in $\mathcal{Q}$ $\mapsto$ 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 Addititition 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 $\mu[f]$:**

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

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