

# Theoretische Informatik II

## Einheit 5.3

### Funktionale & Logische Programme



1. Der  $\lambda$ -Kalkül
2. Arithmetische Repräsentierbarkeit
3. Die Churchsche These

## 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

## 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:

$$f(x) = 2*x+3$$

## 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:

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

## 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:

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

**Abstraktion** von x

Name der Funktion irrelevant für Beschreibung des Verhaltens

## 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:

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

**$\lambda$ -Abstraktion**

Name der Funktion irrelevant für Beschreibung des Verhaltens

## 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:

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

**$\lambda$ -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**

## 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:

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

**$\lambda$ -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**



## 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:

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

**$\lambda$ -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**

- **Alle anderen Konstrukte können simuliert werden**

- Der  $\lambda$ -Kalkül ist eine Turingmächtige funktionale Programmiersprache

- **Einfache Programmiersprache:  $\lambda$ -Terme**

- Variablen  $x, y, z, x_0, y_0, \dots$

- $\lambda x . t$ , wobei  $x$  Variable und  $t$   $\lambda$ -Term

Vorkommen von  $x$  in  $t$  werden **gebunden**

- $f t$ , wobei  $t$  und  $f$   $\lambda$ -Terme

- $(t)$ , wobei  $t$   $\lambda$ -Term

**$\lambda$ -Abstraktion**

**Applikation**

- **Einfache Programmiersprache:  $\lambda$ -Terme**

- Variablen  $x, y, z, x_0, y_0, \dots$

- $\lambda x . t$ , wobei  $x$  Variable und  $t$   $\lambda$ -Term

- Vorkommen von  $x$  in  $t$  werden **gebunden**

- $f t$ , wobei  $t$  und  $f$   $\lambda$ -Terme

- $(t)$ , wobei  $t$   $\lambda$ -Term

**$\lambda$ -Abstraktion**

**Applikation**

- **Prioritätskonventionen sparen Klammern**

- Applikation bindet stärker als  $\lambda$ -Abstraktion

$$\lambda x . f t \hat{=} \lambda x . (f t)$$

- Applikation ist links-assoziativ:

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

## ● Einfache Programmiersprache: $\lambda$ -Terme

– Variablen  $x, y, z, x_0, y_0, \dots$

–  $\lambda x . t$ , wobei  $x$  Variable und  $t$   $\lambda$ -Term

Vorkommen von  $x$  in  $t$  werden **gebunden**

–  $f t$ , wobei  $t$  und  $f$   $\lambda$ -Terme

–  $(t)$ , wobei  $t$   $\lambda$ -Term

**$\lambda$ -Abstraktion**

**Applikation**

## ● Prioritätskonventionen sparen Klammern

– Applikation bindet stärker als  $\lambda$ -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 $\lambda$ -Terme

–  $x$

–  $\lambda f . \lambda x . f x$

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

–  $x x$

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

Symbole sind immer Variablen

Anwendung einer Funktion

Funktionen höherer Ordnung

Selbstanwendung

Fixpunkt

## ● Einfache Programmiersprache: $\lambda$ -Terme

– Variablen  $x, y, z, x_0, y_0, \dots$

–  $\lambda x . t$ , wobei  $x$  Variable und  $t$   $\lambda$ -Term

**$\lambda$ -Abstraktion**

Vorkommen von  $x$  in  $t$  werden **gebunden**

–  $f t$ , wobei  $t$  und  $f$   $\lambda$ -Terme

**Applikation**

–  $(t)$ , wobei  $t$   $\lambda$ -Term

## ● Prioritätskonventionen sparen Klammern

– Applikation bindet stärker als  $\lambda$ -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 $\lambda$ -Terme

(mit **impliziten** und ‘kosmetischen’ Klammern)

–  $x$

**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**

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

**Fixpunkt**

- **Interpretation als “normale” Funktionen ist kompliziert**
  - Erklärung benötigt Theorie vollständiger Halbordnungen (CPOs)
- **Semantik von  $\lambda$ -Termen wird operational erklärt**
  - Entspricht Methodik beim Rechnen auf Zahlen:  $4*(3+2) = 4*5 = 20$
  - $\lambda$ -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  $\lambda$ -Variablen  $x$  durch den  $\lambda$ -Term  $b$
  - Benötigt genaue Definition eines Substitutionskonzepts auf  $\lambda$ -Termen

# VORKOMMEN VON VARIABLEN IN $\lambda$ -TERMEN

- **Vorkommen der Variablen  $x$  im Term  $t$ , informal**
  - **Gebunden**:  $x$  erscheint im Bindungsbereich einer  $\lambda$ -Abstraktion  $\lambda x$
  - **Frei**:  $x$  kommt in  $t$  vor, ohne gebunden zu sein
  - $t$  heißt **geschlossen** falls  $t$  keine freien Variablen enthält
  - $t[x_1, \dots, x_n]$ : Term  $t$  hat mögliche freie Vorkommen von  $x_1, \dots, x_n$

- **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

$x$  *gebunden*

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

$x$  *frei*

- **Ersetze Funktionsparameter durch -argumente**

- **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*

d.h.  $[\lambda f . f x][f/x] = \lambda g . g f$  ( $\lambda f . f f$  wäre ungewollte Selbstanwendung)

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



- **Ersetze Funktionsparameter durch -argumente**

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

- **Ersetze Funktionsparameter durch -argumente**

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

$= (\lambda n . (\lambda f . (\lambda x . ((n \ f) (f \ x)))) (\lambda f . (\lambda x . x))$  *(mit impliziten Klammern)*

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

- **Ersetze Funktionsparameter durch -argumente**

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

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

- **Ersetze Funktionsparameter durch -argumente**

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

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

$\longrightarrow \llbracket \lambda f . (\lambda x . ((n \ f) (f \ x))) \rrbracket [(\lambda f . (\lambda x . x)) / n]$

- **Ersetze Funktionsparameter durch -argumente**

- **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*

d.h.  $[\lambda f . f \ x][f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

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

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

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

$\longrightarrow \lambda f . (\lambda x . (\llbracket \lambda x . x \rrbracket [f / f]) (f \ x))$

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

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



## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

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

- **Ersetze Funktionsparameter durch -argumente**

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

- **Substitution und Reduktion am Beispiel**

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

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

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

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

$\longrightarrow \lambda f . (\lambda x . (\llbracket x \rrbracket [f \ x / x]))$

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

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

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

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## ● Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## ● Substitution und Reduktion am Beispiel

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

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

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

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

$\longrightarrow \lambda f . \lambda x . f \ x$

# $\lambda$ -KALKÜL – BERECHNUNG DURCH AUSWERTUNG

## • Ersetze Funktionsparameter durch -argumente

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

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

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

d.h.  $\llbracket \lambda f . f \ x \rrbracket [f/x] = \lambda g . g \ f$  ( $\lambda f . f \ f$  wäre ungewollte Selbstanwendung)

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

## • Substitution und Reduktion am Beispiel

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

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

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

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

$\longrightarrow \lambda f . \lambda x . f \ x$



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

## Naheliegende Reduktionsreihenfolge

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

## Naheliegende Reduktionsreihenfolge

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

## Naheliegende Reduktionsreihenfolge

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

## Alternative Reduktionsreihenfolgen sind möglich

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

**Bekommt man immer das gleiche Ergebnis?**



- **Bedeutung von  $\lambda$ -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  $t$

# WICHTIGE EIGENSCHAFTEN VON $\lambda$ -TERMEN

- **Bedeutung von  $\lambda$ -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  $\lambda$ -Term eine Normalform?**

# WICHTIGE EIGENSCHAFTEN VON $\lambda$ -TERMEN

- **Bedeutung von  $\lambda$ -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  $\lambda$ -Term eine Normalform?**
  - Nein**:  $(\lambda x. x x) (\lambda x. x x)$  ist nicht normalisierbar
  - Terminierung von  $\lambda$ -Programmen ist nicht garantiert

# WICHTIGE EIGENSCHAFTEN VON $\lambda$ -TERMEN

- **Bedeutung von  $\lambda$ -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  $\lambda$ -Term eine Normalform?**

- **Nein**:  $(\lambda x. x x) (\lambda x. x x)$  ist nicht normalisierbar
- Terminierung von  $\lambda$ -Programmen ist nicht garantiert

- **Haben normalisierbare  $\lambda$ -Terme eindeutige Werte?**

Notwendig für Einsatz des  $\lambda$ -Kalküls als Programmiersprache

- Führt jede Reduktionsfolge zu einer Normalform?

Wenn nein, kann man eine Normalform systematisch finden?

- Ist die Normalform eines  $\lambda$ -Terms eindeutig?

# REDUKTION NORMALISIERBARER $\lambda$ -TERME

- **Führt jede Reduktionsfolge zu einer Normalform?**

## • Führt jede Reduktionsfolge zu einer Normalform?

**Nein:** Reduktionsfolgen normalisierbarer Terme müssen nicht terminieren

$$(\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\longrightarrow (\lambda y. y) (\lambda x. x)$$

$$\longrightarrow \lambda x. x$$

Eine “innermost” Strategie führt bei diesem Term nicht zur Normalform

$$(\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\xrightarrow{*} (\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\xrightarrow{*} (\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\vdots$$

$$\vdots$$

## ● Führt jede Reduktionsfolge zu einer Normalform?

**Nein:** Reduktionsfolgen normalisierbarer Terme müssen nicht terminieren

$$(\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\longrightarrow (\lambda y. y) (\lambda x. x)$$

$$\longrightarrow \lambda x. x$$

Eine “innermost” Strategie führt bei diesem Term nicht zur Normalform

$$(\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\xrightarrow{*} (\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\xrightarrow{*} (\lambda x. \lambda y. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) (\lambda x. x)$$

$$\vdots$$

$$\vdots$$

## ● 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



# 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  $\lambda$ -Termen ist extensional**

- $t = u$ : 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

# 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  $\lambda$ -Termen ist extensional**

- $t = u$ : 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  $\lambda$ -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  $\lambda$ -Terms sind **kongruent** und können (durch Konversionen) auf denselben Term reduziert werden
- Semantisch **gleiche** Terme haben dieselben Normalformen oder keine

# VOM $\lambda$ -KALKÜL ZU ECHTEN PROGRAMMEN

- **$\lambda$ -Kalkül ist der Basismechanismus**
  - Die *Assemblersprache* funktionaler Programme
  - Spezialhardware (Lisp-Maschinen) kann  $\lambda$ -Terme direkt auswerten
- **Programm- und Datenstrukturen werden codiert**
  - Berechnung auf  $\lambda$ -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:  $\top$ ,  $F$ , *if  $b$  then  $s$  else  $t$*
  - Tupel / Projektionen:  *$(s, t)$ ,  $\text{fst}(p)$ ,  $\text{snd}(p)$ ,  $\text{match } p \text{ with } (x, y) \mapsto t$*
  - Zahlen und arithmetische Operationen
  - Iteration oder Rekursion von Funktionen
- **Codierung verwendet definitorische Erweiterungen**
  - Neue Sprachkonstrukte sind *definitorische Abkürzung* für existierende  $\lambda$ -Terme, ggf. mit Parametern (ähnlich zu Macros)  
z.B. doppelte Funktionsanwendung  *$\text{twice}(f) \equiv \lambda x. f(f(x))$*

**Der  $\lambda$ -Kalkül kann alle berechenbaren Funktionen repräsentieren**

# DARSTELLUNG BOOLESCHER OPERATOREN IM $\lambda$ -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"*

$\text{if } b \text{ then } s \text{ else } t \equiv b s t$  *Term für Konditional*

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

# DARSTELLUNG BOOLESCHER OPERATOREN IM $\lambda$ -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"*

$\text{if } b \text{ then } s \text{ else } t \equiv b s t$  *Term für Konditional*

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

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

# DARSTELLUNG BOOLESCHER OPERATOREN IM $\lambda$ -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"*

**if b then s else t**  $\equiv b s t$  *Term für Konditional*

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

if T then *s* else *t*  
 $\equiv T s t$   
 $\equiv (\lambda x. \lambda y. x) s t$   
 $\longrightarrow (\lambda y. s) t$   
 $\longrightarrow s$

if F then *s* else *t*  
 $\equiv F s t$   
 $\equiv (\lambda x. \lambda y. y) s t$   
 $\longrightarrow (\lambda y. y) t$   
 $\longrightarrow t$

**Zeige: wenn  $F = T$  wäre, dann sind alle  $\lambda$ -Terme gleich** (Übung)

# BILDUNG UND ANALYSE VON PAAREN

$$\begin{aligned}(u, v) &\equiv \lambda \text{op. op } u v \\ \text{fst}(p) &\quad (\text{auch } p.1 \text{ oder } p_1) \quad \equiv p (\lambda x. \lambda y. x) \\ \text{snd}(p) &\quad (\text{auch } p.2 \text{ oder } p_2) \quad \equiv p (\lambda x. \lambda y. y) \\ \text{match } p \text{ with } (x, y) \mapsto t &\quad \equiv p (\lambda x. \lambda y. t) \\ &(\text{uniformer Analyseoperator, oft eleganter in Anwendung})\end{aligned}$$

## Der Analyseoperator ist invers zur Paarbildung

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

- **Darstellung natürlicher Zahlen durch iterierte Terme**

- Semantisch: wiederholte Anwendung von Funktionen
- Repräsentiere die **Zahl**  $n$  durch den  $\lambda$ -Term  $\lambda f . \lambda x . \underbrace{f (f \dots (f x) \dots)}_{n\text{-mal}}$
- Notation:  $\bar{n} \equiv \lambda f . \lambda x . f^n x$   
(Markierung  $\bar{n}$  hilft, eine Verwechslung von Zahlen und zugehörigen  $\lambda$ -Termen zu vermeiden)
- Bezeichnung: **Church Numerals**

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

- Es gibt einen  $\lambda$ -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. muß  $\text{add } \bar{m} \bar{n}$  als Wert immer den Term  $\overline{m+n}$  liefern



# ARITHMETISCHE OPERATIONEN IM $\lambda$ -KALKÜL

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

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

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

• **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:**

$$p \equiv \lambda n. snd((n (\lambda f x. (s, match\ fx\ with\ (f, x) \mapsto f\ x)) (\lambda z. \bar{0}, \bar{0})))$$

• **Einfache Rekursion:**  $PRs[b, h] \equiv \lambda n. n h b$

– Für  $f \equiv PRs[b, h]$  gilt:  $f \bar{0} = b$  und  $f (s n) = h (f n)$

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

$$\text{add } \bar{m} \ \bar{n} \equiv (\lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)) \ \bar{m} \ \bar{n}$$

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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



# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# KORREKTHEIT DES PROGRAMMS FÜR DIE ADDITION

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

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

# 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  $n! = \text{if } n=0 \text{ then } 1 \text{ else } n * (n-1)!$

# 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  $n! = \text{if } n=0 \text{ then } 1 \text{ else } n * (n-1)!$
- **Darstellung: Anwendung eines Fixpunktkombinators auf  $t$** 
  - **Fixpunktkombinator**:  $\lambda$ -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

# 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  $n! = \text{if } n=0 \text{ then } 1 \text{ else } n * (n-1)!$
- **Darstellung: Anwendung eines Fixpunktkombinators auf  $t$** 
  - **Fixpunktkombinator**:  $\lambda$ -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  $\lambda$ -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:  $R u = u (R u)$  für jeden  $\lambda$ -Term  $u$



# 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  $n! = \text{if } n=0 \text{ then } 1 \text{ else } n * (n-1)!$
  - **Darstellung: Anwendung eines Fixpunktkombinators auf  $t$** 
    - **Fixpunktkombinator**:  $\lambda$ -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  $\lambda$ -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:  $R u = u (R u)$  für jeden  $\lambda$ -Term  $u$
    - Programmier notation für  $R(\lambda f. \lambda x. t)$  ist “function  $f(x) = t$ ”
    - d.h. function  $f(x) = t$  ist definatorische 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****

- $Y \equiv \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$

Es gilt  $Y t \equiv (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) t$

$$\longrightarrow (\lambda x. t (x x)) (\lambda x. t (x x))$$

$$\longrightarrow t ((\lambda x. t (x x)) (\lambda x. t (x x)))$$

$$t (Y t) \equiv t ((\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) t)$$

$$\longrightarrow t ((\lambda x. t (x x)) (\lambda x. t (x x)))$$

- Es gilt also  $Y t = t (Y t)$  für jeden beliebigen Term  $t$

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

- **Es gibt auch andere Fixpunktkombinatoren**

- $T \equiv (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y))$

hier gilt sogar  $T t \xrightarrow{*} t (T t)$  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  $\bar{1}$  else mul n (fak(p n))  
 $\equiv Y (\lambda \text{fak} . \lambda n . \text{if zero}(n) \text{ then } \bar{1} \text{ else mul } n \text{ (fak}(p \ n)) )$
- **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**  $\equiv \lambda n . \lambda m . \text{zero}(\text{sub } (s \ n) \ m)$
- **Division:** *Suche das erste  $z$  mit  $n < (z + 1) * m$ . Starte Suche bei 0*
  - **div**  $\equiv \lambda n . \lambda m . (\text{function search}(z) =$   
 $\text{if less } n \text{ (mul } z \ m) \text{ then } p \ z \text{ else search}(s \ z)) \bar{0}$
- **Ackermannfunktion:**
  - **A**  $\equiv$  function A(n, x) =  
if zero(x) then  $\bar{1}$   
else if zero(n) then if zero(p x) then  $\bar{2}$  else add x  $\bar{2}$   
else A (p n) (A n (p x))

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

**Alle  $\mu$ -rekursiven Funktionen sind  $\lambda$ -berechenbar**

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

**Alle  $\mu$ -rekursiven Funktionen sind  $\lambda$ -berechenbar**

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

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

## Alle $\mu$ -rekursiven Funktionen sind $\lambda$ -berechenbar

- **Nachfolgerfunktion  $s$ :**  $s \equiv \lambda n. \lambda f. \lambda x. n f (f x)$
- **Projektionsfunktionen  $pr_m^n$ :**  $pr_m^n \equiv \lambda x_1. \dots \lambda x_n. x_m$

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

## Alle $\mu$ -rekursiven Funktionen sind $\lambda$ -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  $c_m^n$ :**  $c_m^n \equiv \lambda x_1. \dots \lambda x_n. \bar{m}$

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

## Alle $\mu$ -rekursiven Funktionen sind $\lambda$ -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  $f \circ (g_1, \dots, g_n)$ :**
  - $\equiv \lambda f. \lambda g_1. \dots \lambda g_n. \lambda x. f (g_1 x) \dots (g_n x)$



# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

## Alle $\mu$ -rekursiven Funktionen sind $\lambda$ -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  $f \circ (g_1, \dots, g_n)$ :**
  - $\equiv \lambda f. \lambda g_1. \dots \lambda g_n. \lambda x. f (g_1 x) \dots (g_n x)$
- **Primitive Rekursion  $Pr[f, g]$ :**  
 $PR \equiv \lambda f. \lambda g. \text{function } h(x) = \lambda y. \text{if zero } y \text{ then } f x \text{ else } g x (p y) (h x (p y))$

# DER $\lambda$ -KALKÜL IST TURING-MÄCHTIG

## Alle $\mu$ -rekursiven Funktionen sind $\lambda$ -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  $f \circ (g_1, \dots, g_n)$ :**
  - $\equiv \lambda f. \lambda g_1. \dots \lambda g_n. \lambda x. f (g_1 x) \dots (g_n x)$
- **Primitive Rekursion  $Pr[f, g]$ :**  
 $PR \equiv \lambda f. \lambda g. \text{function } h(x) = \lambda y. \text{if zero } y \text{ then } f x \text{ else } g x (p y) (h x (p y))$
- **Minimierung  $\mu[f]$ :**  
 $Mu \equiv \lambda f. \lambda x. (\text{function } \min(y) = \text{if zero}(f x y) \text{ then } y \text{ else } \min(sy)) \bar{0}$

## Terminierung von $\lambda$ -Termen ist unentscheidbar

- **Beweis stützt sich auf wenige Erkenntnisse**
  - $\lambda x.x$  terminiert mit Wert  $\lambda x.x$
  - $\perp = Y(\lambda x.x)$  terminiert nicht, da  $Y(\lambda x.x) = (\lambda x.x x) (\lambda x.x x)$
  - Die Boole'schen Wahrheitswerte sind verschieden:  $T \neq F$

## Terminierung von $\lambda$ -Termen ist unentscheidbar

- **Beweis stützt sich auf wenige Erkenntnisse**

- $\lambda x.x$  terminiert mit Wert  $\lambda x.x$
- $\perp = Y(\lambda x.x)$  terminiert nicht, da  $Y(\lambda x.x) = (\lambda x.x x) (\lambda x.x x)$
- Die Boole'schen Wahrheitswerte sind verschieden:  $\top \neq \text{F}$

- **Einfacher Widerspruchsbeweis**

- Wir nehmen an, Terminierung von  $\lambda$ -Termen sei entscheidbar

d.h. es gibt einen  $\lambda$ -Term  $h$  mit  $h(t) = \begin{cases} \top & \text{falls } t \text{ terminiert} \\ \text{F} & \text{sonst} \end{cases}$

- Definiere  $d = \lambda x. \text{if } h(x) \text{ then } \perp \text{ else } \lambda x.x.$

- Dann  $h(Yd) = h(d(Yd)) = h(\text{if } h(Yd) \text{ then } \perp \text{ else } \lambda x.x)$

also  $h(Yd) = \begin{cases} h(\perp) = \text{F}, & \text{falls } h(Yd) = \top \\ h(\lambda x.x) = \top, & \text{falls } h(Yd) = \text{F} \end{cases}$

- 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

## 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

## 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

z.B. Repräsentation der Subtraktion  $sub : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $sub(x_1, x_2) = x_1 \dot{-} x_2$

- Auf den ganzen Zahlen ist  $x_1 - x_2 = y \Leftrightarrow x_1 = x_2 + y$
- Auf natürlichen Zahlen gilt zusätzlich  $y = 0$ , falls  $x_1 \leq x_2$
- Damit folgt  $x_1 \dot{-} x_2 = y \Leftrightarrow x_1 = x_2 + y \vee x_1 \leq x_2 \wedge y = 0$

Die Formel  $x_1 = x_2 + y \vee x_1 \leq x_2 \wedge y = 0$  wäre eine logische Repräsentation des Verhaltens der Funktion  $sub$

und muß nur noch in einem formalen Kalkül formuliert werden

## 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



## 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  $\models_T F(\bar{i}_1, \dots, \bar{i}_k, \bar{j})$  g.d.w.  $f(i_1, \dots, i_k) = j$ ,  
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  $s$
- Zweistellige Funktionssymbole  $+$  und  $*$

## • Semantik: Logik + 7 Axiome

(ohne Induktion!)

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

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

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

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

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

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

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

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

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

- **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:

$$\text{ADD}(x_1, x_2, y) \equiv y = x_1 + x_2$$

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

- **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<sub>4</sub>:  $\models_{\mathcal{Q}} \bar{i} + \bar{0} = \bar{i}$

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

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

Mit Axiom Q<sub>5</sub> folgt  $\models_{\mathcal{Q}} \bar{i} + \bar{j} = \bar{i} + s(\bar{m}) = s(\bar{i} + \bar{m}) = s(\bar{n}) = \bar{k}$

# 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**  $DIV(x_1, x_2, q) \equiv \exists r. LT(r, x_2) \wedge x_2 * q + r = x_1$   
**Divisionsrest/Modulo**  $MOD(x_1, x_2, r) \equiv LT(r, x_2) \wedge \exists q. x_2 * q + r = x_1$
- **Teilbarkeit** (Prädikat)  $DIVIDES(x_1, x_2) \equiv \exists q. x_1 * q = x_2$   
**Primzahleigenschaft** (Prädikat)  $PRIME(x) \equiv LT(\bar{1}, x) \wedge \forall y. (LT(y, x) \wedge LT(\bar{1}, y)) \Rightarrow \neg DIVIDES(y, x)$

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

– Alle  $\mu$ -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:  $\lambda$ -Kalkül
- Logische Programmierung: Arithmetische Repräsentierbarkeit

- **Alle Berechenbarkeitsmodelle sind i.w. äquivalent**

- **These:** Alle berechenbaren Funktionen sind Turing-berechenbar (oder rekursiv,  $\lambda$ -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
- $\mathcal{R}_{min} \subseteq \mathcal{R}$ : min-rekursive Funktionen sind  $\mu$ -rekursiv
  - Offensichtlich, da Addition  $\mu$ -rekursiv ist
- $\mathcal{R} \subseteq \mathcal{R}_{min}$ :  $\mu$ -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  $f$