

# PROGRAMMIERUNG

## ÜBUNG 13: $H_0$ – EIN EINFACHER KERN VON HASKELL

---

Eric Kunze

`eric.kunze@tu-dresden.de`

1. Funktionale Programmierung
  - 1.1 Einführung in Haskell: Listen
  - 1.2 Algebraische Datentypen
  - 1.3 Funktionen höherer Ordnung
  - 1.4 Typpolymorphie & Unifikation
  - 1.5 Beweis von Programmeigenschaften
  - 1.6  $\lambda$ -Kalkül
2. Logikprogrammierung
3. Implementierung einer imperativen Programmiersprache
  - 3.1 Implementierung von  $C_0$
  - 3.2 Implementierung von  $C_1$
4. Verifikation von Programmeigenschaften
5.  **$H_0$  – ein einfacher Kern von Haskell**

# **$H_0$ – ein einfacher Kern von Haskell**

---

▶ **Ziel:** verstehe den Zusammenhang  $H_0 \leftrightarrow AM_0 \leftrightarrow C_0$

▶ H<sub>0</sub>: *tail recursive* Funktionen — rechte Seite enthält

▶ keinen Funktionsaufruf

$$f\ x1\ x2 = x1 * x2$$

▶ einen Funktionsaufruf an der äußersten Stelle (nicht verschachtelt)

$$f\ x1\ x2 = f\ (x1-1)\ (x2 * x2)$$

▶ eine Fallunterscheidung, deren Zweige wie oben aufgebaut sind

$$f\ x1\ x2 = \text{if} \\ \text{then } \underline{\hspace{2cm}} \\ \text{else } \underline{\hspace{2cm}}$$

**Erinnerung:** Abstrakte Maschine AM<sub>0</sub>

▶ Ein- und Ausgabeband

▶ Datenkeller

▶ Hauptspeicher

▶ Befehlszähler

# $H_0 \leftrightarrow AM_0$

$H_0$  ist klein genug, dass es auf der  $AM_0$  laufen kann:

- ▶ Befehle bleiben die gleichen
- ▶ baumstrukturierte Adressen beginnen mit Funktionsbezeichner (z.B.  $f.1.3$ )

**Übersetzung von rechten Seiten**  $\dots = exp$ :

- ▶ Übersetze  $exp$

$f \ x1 \ x2 = exp$

- ▶ STORE 1 (ja - immer die 1)

- ▶ WRITE 1

- ▶ JMP 0

$f \ (x1-1) \ (x2*x2) \ x3$

**Übersetzung von Funktionsaufrufen**  $\dots = f \ x1 \ x2 \ x3$ :

- ▶ LOAD  $x1$ ; LOAD  $x2$ ; LOAD  $x3$

- ▶ STORE  $x3$ ; STORE  $x2$ ; STORE  $x1$  (umgekehrte Reihenfolge!)

- ▶ JMP  $f$

$H_0$  (funktional) und  $C_0$  (imperativ) sind gleich stark – wir können Programme jeweils ineinander äquivalent übersetzen!

Standardisierung:

- ▶ keine Konstanten
- ▶ Es gibt  $m$  Variablen  $x_1, \dots, x_m$  ( $m \geq 1$ )
- ▶ Wir lesen  $k$  Variablen  $x_1, \dots, x_k$  ein ( $0 \leq k \leq m$ )
- ▶ Es gibt genau eine Schreibanweisung direkt vor `return`

- ▶ jedes Statement (in  $C_0$ ) erhält einen *Ablaufpunkt*
- ▶ jeder Ablaufpunkt  $i$  wird durch eine Funktion  $f_i$  (in  $H_0$ ) repräsentiert, die *alle* Programmvariablen als Argumente hat
- ▶ Funktionswerte beschreiben Veränderungen im Programmablauf

(einfaches) **Beispiel:**

- ▶ zwei Variablen  $x_1$  und  $x_2$
- ▶ betrachte Zuweisung  $x_2 = x_1 * x_1$  in  $C_0$
- ▶ Übersetzung zu  $f_1 \ x_1 \ x_2 = f_{11} \ x_1 \ (x_1 * x_1)$

Ein  $H_0$ -Programm kann in  $C_0$  mittels einer while-Schleife dargestellt werden. Dazu verwenden wir drei Hilfsvariablen:

- ▶ `flag` steuert den Ablauf der `while`-Schleife, d.h. wenn das  $H_0$ -Programm terminiert, wird `flag` falsch
- ▶ `function` steuert in einer geschachtelten `if-then-else`-Anweisung, welche Funktion ausgeführt wird
- ▶ `result` speichert den Rückgabewert der Funktion

# Übungsblatt 13

## *Aufgabe 1*

---

# AUFGABE 1 – TEIL (A)

$$f(4) = -1 + 2 - 3 + 4 = 2$$

$$f: \mathbb{N} \rightarrow \mathbb{N} \quad \text{mit} \quad f(n) = \sum_{i=1}^n (-1)^i \cdot i$$

module main where

$f$  :: Int  $\rightarrow$  Int  $\rightarrow$  Int

$f$  x1 x2 = if x1 == 0  
then x2

else if x1 `mod` 2 == 0

then  $f$  (x1-1) (x2+x1)

else  $f$  (x1-1) (x2-x1)

main = do x1  $\leftarrow$  readLn  
print ( $f$  x1 0)

$$\begin{aligned} f(n) &= (-1)^n \cdot n + \sum_{i=1}^{n-1} (-1)^i \cdot i \\ &= \underbrace{(-1)^n \cdot n}_{\oplus} \oplus f(n-1) \end{aligned}$$

## AUFGABE 1 – TEIL (A)

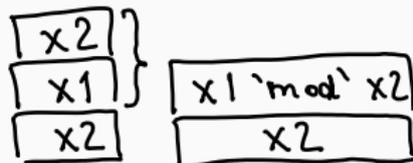
$$f: \mathbb{N} \rightarrow \mathbb{N} \quad \text{mit} \quad f(n) = \sum_{i=1}^n (-1)^i \cdot i$$

```
1 module Main where
2
3 --      i      sum
4 f :: Int -> Int -> Int
5 f x1 x2 = if x1 == 0
6           then x2
7           else if x1 `mod` 2 == 0
8                 then f (x1 - 1) (x2 + x1)
9                 else f (x1 - 1) (x2 - x1)
10
11 main = do x1 <- readLn
12           print (f x1 0)
```

**Gegeben:**

```
1 f :: Int -> Int -> Int
```

```
2 f x1 x2 = if x2 == 0
3           then x1
4           else f x2 (x1 'mod' x2)
```

**Gesucht:** äquivalentes  $AM_0$ -Programm

```
f:   LOAD 2 ; LIT 0 ; EQ ; JMC f.3;
      LOAD 1 ; STORE 1 ; WRITE 1 ; JMP 0;
f.3: LOAD 2; LOAD 1; LOAD 2; MOD;
      STORE 2; STORE 1; JMP f;
```

**Gegeben:**

```

1 f :: Int -> Int -> Int
2 f x1 x2 = if x2 == 0
3           then x1
4           else f x2 (x1 `mod` x2)
main = do x1 ← readLn → READ 1;

```

**Gesucht:** äquivalentes AM<sub>0</sub>-Programm

```

f:      LOAD 2; LIT 0; EQ; JMC f.3;
        LOAD 1; STORE 1; WRITE 1; JMP 0;
f.3:   LOAD 2; LOAD 1; LOAD 2; MOD;
        STORE 2; STORE 1; JMP f;

```

### Zusatzaufgabe 1 (AGS 17.13 a \*)

Eine Folge  $e_i$  ( $i \geq 1$ ) von ganzen Zahlen soll wie folgt konstruiert werden:

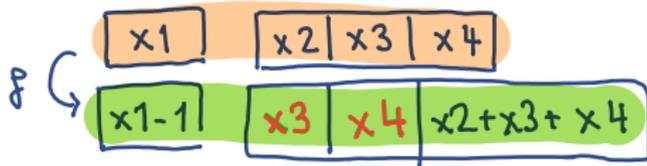
- Das erste Glied der Folge sei 1.
- Das zweite Glied der Folge sei 2.
- Das dritte Glied soll von der Eingabe gelesen werden.
- Ab dem vierten Glied der Folge soll gelten: Jedes Folgeglied ist gleich der Summe der drei Vorgängerglieder.

$$1, 2, x, 1+2+x, 2+x+1+2+x, \dots$$

$$\underline{1, 2, 3, 6, 11, 20, \dots}$$

Geben Sie ein  $H_0$ -Programm P an, welches das  $n$ -te Folgeelement, also  $e_n$ , dieser Folge berechnet und ausgibt.

Bsp:  $e_4 = 6$



	x1	x2	x3	x4
4	4	1	2	3
3	3	2	3	6
2	2	3	6	11
1	1	6	11	20
-				

module main where

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$f \ x1 \ x2 \ x3 \ x4 = \text{if } x1 == 1$

then  $x2$

else  $f \ (x1-1) \ x3 \ x4 \ (x2+x3+x4)$

main = do  $x1 \leftarrow \text{readLn}$

$x2 \leftarrow \text{readLn}$

print  $(f \ x1 \ 1 \ 2 \ x2)$

## Zusatzaufgabe 2 (AGS 17.14 a \*)

Transformieren Sie das folgende  $H_0$ -Programm mittels der Funktion *trans* in ein  $AM_0$ -Programm mit linearen Adressen. Sie brauchen dabei keine Zwischenschritte anzugeben.

```
1 module Main where
2
3 fac :: Int -> Int -> Int
4 fac x1 x2 = if x1 > 0 then fac (x1 - 1) (x1 * x2) else x2
5
6 main = do x1 <- readLn
7           print (fac x1 1)
```

```
      READ 1;
      LOAD 1; LIT 1;
      STORE 2; STORE 1; JMP fac; } main
fac:  LOAD 1; LIT 0; GT; JMC fac.3; } Bed.
      LOAD 1; LIT 1; SUB;
      LOAD 1; LOAD 2; MUL;
      STORE 2; STORE 1; JMP fac; } then
fac.3: LOAD 2; STORE 1; WRITE 1; JMP 0; } else
```