

PROGRAMMIERUNG

ÜBUNG 13: H_0 – EIN EINFACHER KERN VON HASKELL

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 11. Juli 2022

letzte Änderung:
11.07.2022, 09:44

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 λ -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_0 : *tail recursive* Funktionen — rechte Seite enthält
 - ▷ keinen Funktionsaufruf
 - ▷ einen Funktionsaufruf an der äußersten Stelle (nicht verschachtelt)
 - ▷ eine Fallunterscheidung, deren Zweige wie oben aufgebaut sind

Erinnerung: Abstrakte Maschine AM_0

- ▶ 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 $\dots = \text{exp}$:

- ▶ Übersetze *exp*
- ▶ STORE 1 (ja – immer die 1)
- ▶ WRITE 1
- ▶ JMP 0

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 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 deterministischen rechten Seiten $\dots = exp$:

- | | |
|-----------------------------------|---------------------------------|
| ▶ Übersetze exp | $rhstrans(e, a) := exptrans(e)$ |
| ▶ STORE 1 (ja – immer die 1) | STORE 1; |
| ▶ WRITE 1 | WRITE 1; |
| ▶ JMP 0 | JMP 0; |

Übersetzung von Funktionsaufrufen ... = f exp₁ exp₂ exp₃

- ▶ Übersetze exp₁, exp₂, exp₃ in richtiger Reihenfolge
- ▶ STORE x₃; STORE x₂; STORE x₁ in umgekehrter Reihenfolge
- ▶ Funktionsaufruf mit JMP f

formal: $rhstrans(f\ e_1\ \dots\ e_n, a) := exptrans(e_1)\ \dots\ exptrans(e_n)$

STORE n; ... STORE 1;

JMP f;

Übersetzung von Verzweigungen: *Änderung der Sprungmarken!*

- ▶ Bedingung übersetzen
- ▶ JMC basis.3;
- ▶ then-Zweig übersetzen mit interner Basisadresse .1
- ▶ basis.3: else-Zweig übersetzen mit interner Basisadresse .2

formal: $rhstrans(if\ be\ then\ r_1\ else\ r_2, a) := bexptrans(be)$

JMC a.3;

$rhstrans(r_1, a.1)$

$a.3 : rhstrans(r_2, a.2)$

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)

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

```
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;

```