PROGRAMMIERUNG

ÜBUNG 13: H_0 – EIN EINFACHER KERN VON HASKELL

Eric Kunze
eric.kunze@tu-dresden.de

INHALT

H_0 – ein einfacher Kern von Haskell

H_0

- **Ziel:** verstehe den Zusammenhang $\boxed{H_0 \leftrightarrow AM_0} \leftrightarrow C_0$
- H_0: *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 = if
    then ___
    else ___

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 ... = exp:

f x1 x2 = exp

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

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

Übersetzung von Funktionsaufrufen ... = `f x1 x2 x3`:

- `LOAD x1`; `LOAD x2`; `LOAD x3`
- `STORE x3`; `STORE x2`; `STORE x1` (umgekehrte Reihenfolge!)
- `JMP f`

$H_0 \leftrightarrow C_0$

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

Standardisierung:

- keine Konstanten
- Es gibt m Variablen `x1, ..., xm` ($m \geq 1$)
- Wir lesen k Variablen `x1, ..., xk` ein ($0 \leq k \leq m$)
- Es gibt genau eine Schreibanweisung direkt vor `return`

$C_0 \rightarrow H_0$

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

(einfaches) **Beispiel:**

- zwei Variablen `x1` und `x2`
- betrachte Zuweisung `x2 = x1 * x1` in C_0
- Übersetzung zu `f1 x1 x2 = f11 x1 (x1 * x1)`

$H_0 \to C_0$

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)

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

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

$f(n) = (-1)^n \cdot n + \sum_{i=1}^{n-1} (-1)^i \cdot i$

$= (-1)^n \cdot n + f(n-1)$

```haskell
module main where

         i       sum
f :: Int -> Int -> 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 <- readLn
          print (f x1 0)
```

AUFGABE 1 – TEIL (A)

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

```haskell
module Main where

--      i       sum
f :: Int -> Int -> 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 <- readLn
          print (f x1 0)
```

Gegeben:

```
f :: Int -> Int -> Int
f x1 x2 = if x2 == 0
          then x1
          else f x2 (x1 `mod` x2)
```

x2
x1
x2

x1 `mod` x2
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;

AUFGABE 1 – TEIL (B) $H_0 \rightarrow AM_0$

Gegeben:

```
f :: Int -> Int -> Int
f x1 x2 = if x2 == 0
          then x1
          else f x2 (x1 `mod` x2)
```

main = do x1 ← readLn → READ 1;

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;
```

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.

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

1, 2, x, 1+2+x, 2+x+1+2+x, ...

$\underbrace{1, 2, 3}$, 6, 11, 20, ...

$\boxed{x1}$ $\boxed{x2 \mid x3 \mid x4}$

$\xrightarrow{f}$

$\boxed{x1-1}$ $\boxed{x3 \mid x4 \mid x2+x3+x4}$

Bsp.: $e_4 = 6$

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

```
module main where
    f :: Int -> Int -> Int -> Int -> Int
    f x1 x2 x3 x4 = if x1 == 1
                    then x2
                    else f (x1-1) x3 x4 (x2+x3+x4)

main = do x1 <- readLn
          x2 <- readLn
          print (f x1 1 2 x2)
```

Zusatzaufgabe 2 (AGS 17.14 a $\star$)

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.

```
module Main where

fac :: Int -> Int -> Int
fac x1 x2 = if x1 > 0 then fac (x1 - 1) (x1 * x2) else x2

main = do x1 <- readLn
          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