TECHNISCHE UNIVERSITÄT DRESDEN

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

INHALT

H_0 – ein einfacher Kern von Haskell

H_0

- **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 `... = exp`:

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

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

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

$H_0 \leftrightarrow AM_0$

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

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

$H_0 \leftrightarrow AM_0$

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 deterministischen rechten Seiten `...` `=` *exp*:

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

$$rhstrans(e, a) := exptrans(e)$$

```
STORE 1;
WRITE 1;
JMP 0;
```

Übersetzung von Funktionsaufrufen `... = f` exp_1 exp_2 exp_3

- Übersetze exp_1, exp_2, exp_3 in richtiger Reihenfolge
- `STORE x3; STORE x2; STORE x1` in umgekehrter Reihenfolge
- Funktionsaufruf mit `JMP f`

formal: $rhstrans(f\ e_1\ \ldots\ e_n, a) \coloneqq exptrans(e_1)\ \ldots\ exptrans(e_n)$

```
STORE n; ... STORE 1;
JMP f;
```

Übersetzung von Funktionsaufrufen `...` `=` `f` exp_1 exp_2 exp_3

- Übersetze exp_1, exp_2, exp_3 in richtiger Reihenfolge
- `STORE x3; STORE x2; STORE x1` in umgekehrter Reihenfolge
- Funktionsaufruf mit `JMP f`

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

`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: $rhstrans(\texttt{if}\ be\ \texttt{then}\ r_1\ \texttt{else}\ r_2, a) := bexptrans(be)$

`JMC a.3;`

$rhstrans(r_1, a.1)$

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

$H_0 \leftrightarrow C_0$

H_0 (funktional) und C_0 (imperativ) sind 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`

$C_0 \to H_0$

- 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)`

$H_0 \rightarrow C_0$

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)

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

AUFGABE 1 – TEIL (A)

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

```
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

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;
```