Übungsblatt 13

Übung zur Vorlesung "Programmierung" im Sommersemester 2019

Lösungsvorschläge – Eric Kunze

Aufgabe 1. Teilsprache C_1 und abstrakte Maschine AM_1

Hinweis. *Die baumstrukturierten Adressen ergeben sich immer nach dem Aufrufzeitpunkt und nicht nach dem tatsächlichen Vorkommen in der Übersetzung. Da wir nur lokale Referenzen betrachten, wird bei* `LOADI` *nur die Adresse angegeben und auf die Angabe* `lokal` *verzichtet.*

In den Ablaufprotokollen dürfen Zellen, deren Wert sich nicht ändert, auch leer gelassen werden (Achtung: ein Wechsel $0 \to \varepsilon$ *ist eine Änderung und muss eingetragen werden!).*

Lösung. (a) Symboltabelle:

$$lokal - tab_{\texttt{f}} = \Big[\texttt{x}/(\text{var}, \text{global}, 1), \texttt{h}/(\text{proc}, 1), \texttt{g}/(\text{proc}, 2), \texttt{f}/(\text{proc}, 3), \\ \texttt{c}(\text{var}, \text{lokal}, 1), \texttt{a}/(\text{var}, \text{lokal}, -3), \texttt{b}/(\text{var-ref}, -2)\Big]$$

Übersetzung:

```
        LOAD(global, 1);
        LIT 1;
        GT;
        JMC 3.1.1;
        LOAD(lokal,-2);
        PUSH
        CALL 2;
        JMP 3.1.3;
3.1.1:  LOAD(lokal,-3);
        PUSH;
        LOADA(global,1);
        PUSH;
        CALL 1;
3.1.3:  LOADI(-2);
        LIT 1;
        ADD;
        STORE(lokal,1);
```

(b) Das übliche Ablaufprotokoll:

BZ	DK	LZK	REF	Inp	Out
12	ε	0:3:0:7	3	5	ε
13	ε	5:3:0:7	3	ε	ε
14	7	5:3:0:7	3	ε	ε
15	ε	5:3:0:7:7	3	ε	ε
16	1	5:3:0:7:7	3	ε	ε
17	ε	5:3:0:7:7:1	3	ε	ε
4	ε	5:3:0:7:7:1:18:3	8	ε	ε
5	ε	5:3:0:7:7:1:18:3:0	8	ε	ε
6	7	5:3:0:7:7:1:18:3:0	8	ε	ε
7	5:7	5:3:0:7:7:1:18:3:0	8	ε	ε
8	12	5:3:0:7:7:1:18:3:0	8	ε	ε
9	ε	12:3:0:7:7:1:18:3:0	8	ε	ε
18	ε	12:3:0:7	3	ε	ε
19	ε	12:3:0:7	3	ε	12
3	ε	12	0	ε	12
0	ε	12	0	ε	12

Aufgabe 2. Haskell

Hinweis. *In der Klausur dürfen (sofern nicht anders gefordert) alle Funktionen der Standard-Bibliothek* `Prelude` *verwendet werden. Achtet bei den Programmieraufgaben immer auch auf die Zeit - man vergisst sich da leicht. Deswegen die Empfehlung: Programmieraufgaben immer zuletzt lösen.*

Lösung. Für die Berechnung des Durchschnitts könnte man schlichtweg die Summe aller Listeneinträge durch die Länge der Liste dividieren. Dann würde die Liste aber mehrfach durchlaufen werden, was in der Aufgabenstellung ausgeschlossen ist. Also nutzen wir wieder den Umweg über eine eigene Speicherhilfsfunktion, in der wir uns in den Argumenten jeweils die aktuelle Summe und die aktuelle Listenlänge merken und am Ende die Division der Parameter ausführen. Dies sieht dann wie folgt aus:

```
1 avg :: [Float] -> Float
2 avg [] = 0
3 avg xs = f 0 0 xs
4   where
5     f :: Float -> Int -> [Float] -> Float
6     f sum len []     = sum / len
7     f sum len (y:ys) = f (sum + y) (len+1) ys
```

Um eine Liste bezüglich eines Prädikates `p` zu partitionieren nutzen wir den günstigen Fall der Logik: es gibt nur zwei Alternativen und die eine ergibt sich als Negation der anderen. Somit können wir die beiden Komponenten des zu generierenden Paares explizit angeben: die zweite Komponente gleicht der ersten, wird jedoch um die Funktion `not` ergänzt. Für die genaue Partition können wir uns wahlweise die Listennotation wählen, oder wir greifen auf die Funktion `filter` zurück.

```
1  partition ::  (a -> Bool) -> [a] -> ([a], [a])
2  partition p xs
3    = ( [x | x <- xs , p x], [x | x <- xs , not(p x)])
```

```
1  partition ::  (a -> Bool) -> [a] -> ([a], [a])
2  partition p xs = (filter p xs , filter (not . p) xs)
```

Um die maximale Wiederholung zu finden, gehen wir die Liste von hinten vollständig durch. Die Funktion `f` generiert uns dabei immer ein Tripel `(x, cxs, mxs)`, wobei in `cxs` die aktuell längste Wiederholung gespeichert wird und wenn diese größer als `mxs` (die maximale Wiederholung) ist, wird auch dieses geändert. In `x` steht schließlich das zuletzt gelesene Element.

```
1  maxrep :: [Int] -> Int
2  maxrep xs = let (_,_,m) = f xs in m
3    where
4      f :: [Int] -> (Int, Int, Int)
5      f []  = (0,0,0)
6      f [x] = (x,1,1)
7      f (x:xs) = let (x', cxs, mxs) = f xs
8                     c = if x == x' then cxs + 1
9                         else 1
10                    m = max c mxs
11                in (x,c,m)
```

Alternative Lösung: Wir erinnern uns an Aufgabenblatt 2 (Aufgabe 1), wo wir aufeinanderfolgende gleiche Zeichen einer Zeichenkette mit der Funktion `pack` zusammengefügt haben. Dieses Verfahren wenden wir schließlich auch hier an (nur für `Int`) und lesen anschließend die maximale Listenlänge aus.

```
1  maxrep' :: [Int] -> Int
2  maxrep' = maximum . length' . pack
3
4  length' :: [[Int]] -> [Int]
5  length' = map length
6
7  pack :: [Int] -> [[Int]]
8  pack [] = []
9  pack (x:xs) = collect x (x:xs) : next x xs
```

```
10
11 collect :: Int -> [Int] -> [Int]
12 collect _ []= []
13 collect y (x:xs)
14   | y == x = x : collect y xs
15   | otherwise = []
16
17 next :: Int -> [Int] -> [[Int]]
18 next _ [] = []
19 next y (x:xs)
20   | y == x = next y xs
21   | otherwise = pack (x:xs)
```

Aufgabe 3. Unifikation

Wir betrachten das Rangalphabet $\Sigma = \left\{\sigma^{(2)}, \gamma^{(1)}, \alpha^{(0)}\right\}$ und die beiden Terme

$$t_1 = (\sigma(\alpha), \sigma(\gamma(\alpha), \sigma(x_2, x_3)))$$
$$t_2 = \sigma(\alpha, \sigma(x_1, \sigma(x_2, \sigma(x_2, x_1))))$$

Lösung.

$$\left\{\begin{pmatrix}\sigma(\alpha,\sigma(\gamma(\alpha)\,,\sigma(x_2\,,x_3\quad\quad\quad)))\\ \sigma(\alpha,\sigma(x_1\quad,\sigma(x_2\,,\sigma(x_2,x_1)\,)))\end{pmatrix}\right\} \overset{\text{Dek.}}{\Longrightarrow} \left\{\begin{pmatrix}\alpha\\ \alpha\end{pmatrix}, \begin{pmatrix}\sigma(\gamma(\alpha)\,,\sigma(x_2\,,x_3\quad\quad\quad))\\ \sigma(x_1\quad,\sigma(x_2\,,\sigma(x_2,x_1)\,))\end{pmatrix}\right\}$$
$$\overset{2\,*\,\text{Dek.}}{\Longrightarrow} \left\{\begin{pmatrix}\gamma(\alpha)\\ x_1\end{pmatrix}, \begin{pmatrix}\sigma(x_2\,,x_3\quad\quad)\\ \sigma(x_2\,,\sigma(x_2,x_1)\,)\end{pmatrix}\right\}$$
$$\overset{\text{Dek.}}{\Longrightarrow} \left\{\begin{pmatrix}\gamma(\alpha)\\ x_1\end{pmatrix}, \begin{pmatrix}x_2\\ x_2\end{pmatrix}, \begin{pmatrix}x_3\\ \sigma(x_2,x_1)\end{pmatrix}\right\}$$
$$\overset{\text{El.}}{\Longrightarrow} \left\{\begin{pmatrix}\gamma(\alpha)\\ x_1\end{pmatrix}, \begin{pmatrix}x_3\\ \sigma(x_2,x_1)\end{pmatrix}\right\}$$
$$\overset{\text{Vert.}}{\Longrightarrow} \left\{\begin{pmatrix}x_1\\ \gamma(\alpha)\end{pmatrix}, \begin{pmatrix}x_3\\ \sigma(x_2,x_1)\end{pmatrix}\right\}$$
$$\overset{\text{Subst.}}{\Longrightarrow} \left\{\begin{pmatrix}x_1\\ \gamma(\alpha)\end{pmatrix}, \begin{pmatrix}x_3\\ \sigma(x_2,\gamma(\alpha))\end{pmatrix}\right\}$$

Damit ergibt sich der allgemeinste Unifikator zu

$$x_1 \mapsto \gamma(\alpha) \qquad x_2 \mapsto x_2 \qquad x_3 \mapsto \sigma(x_2, \gamma(\alpha))$$

Aufgabe 4. Strukturelle Induktion

Hinweis. *Wir beweisen hier die Korrektheit von Aussagen bezüglich eines Programms. Entsprechend korrekt und genau müssen wir auch arbeiten. Dazu gehören die vollständige Quantifizierung aller vorkommenden Variablen und das befolgen des Induktionsprinzips. Insbesondere ist es wichtig zwischen bereits gezeigter, angenommener und noch zu zeigender Aussage zu unterscheiden.*

Man beachte beispielsweise in der folgenden Aufgabe, dass wir über die Listen `xs` *induzieren, jedoch auch immer die Parameter* `a` *und* `b` *vollständig mit Typ angeben (das entspricht der vollständigen Quantifizierung). Dabei sind* `a` *und* `b` *stets beliebig gewählt, sodass die zu zeigende Aussage unabhängig von der wirklichen Ausprägung von* `a` *und* `b` *gilt.*

Lösung. Wir wollen zeigen, dass für jede Liste `ys :: [Int]`

```
mul b (mul a ys) = mul (b * a) ys
```

für alle `a,b :: Int` gilt, d.h. das auch die definierte skalare Listenmultiplikation assoziativ ist.

Induktionsanfang Sei `ys = [ ]` und seien `a,b :: Int` beliebig. Es gilt

$$\texttt{mul b (mul a [ ])} \overset{(2)}{=} \texttt{mul b [ ]} \overset{(2)}{=} \texttt{[ ]} \overset{(2)}{=} \texttt{mul (b*a) [ ]}$$

Induktionsvoraussetzung Sei `xs :: [Int]`, sodass für alle `a,b :: Int` gilt:

```
mul b (mul a xs) = mul (b*a) xs
```

Induktionsschritt Für jedes `x :: Int` und jedes beliebige `a,b :: Int` folgt dann

$$\begin{aligned}
\texttt{mul b (mul a (x:xs))} &\overset{(3)}{=} \texttt{mul b ((a*x) : mul a xs)} \\
&\overset{(3)}{=} \texttt{(b*a*x) : mul b (mul a xs)} \\
&\overset{(\mathrm{IV})}{=} \texttt{(b*a*x) : mul (b*a) xs} \\
&\overset{(3)}{=} \texttt{mul (b*a) (x:xs)}
\end{aligned}$$

Aufgabe 5. λ-Kalkül

Hinweis. *Bei der Berechnung der Normalform ist immer auf die implizite Linksassoziativität zu achten! Bei den Auswertungen mit Fixpunktkombinator ist stets eine Nebenrechnung zu führen, in der die Wirkunsgweise des Fixpunktkombinators gezeigt wird. Diese ist immer gleich.*

Lösung. (a) Wir berechnen die Normalform des λ-Terms $\lambda f x.x(f(fx))(\lambda y.xy)$.

$$\begin{aligned}
\lambda f \underbrace{x.x(f(fx))}_{GV=\{x\}})(\underbrace{\lambda y.xy}_{\{x\}}) &\Rightarrow_\alpha (\lambda f u.u(f(fu)))(\lambda y.xy) \\
&\Rightarrow_\beta (\lambda u.u((\lambda y.xy)((\lambda y.\ \underbrace{xy}_{GV=\emptyset}\ )\ \underbrace{u}_{FV=\{u\}}\ ))) \\
&\Rightarrow_\beta (\lambda u.u((\lambda y.\ \underbrace{xy}_{GV=\emptyset}\ )(\ \underbrace{xu}_{FV=\{x,u\}}\ ))) \\
&\Rightarrow_\beta (\lambda u.u(x(xu)))
\end{aligned}$$

(b) Die Haskell-Funktion setzt die Fallunterscheidungen in den λ-Termen mittels Pattern Matching um.

```
1 f :: Int -> Int -> Int
2 f x 0 = 2 * x
3 f x y =
4   | y `mod` 2 == 0 = f x (y-1)
5   | otherwise      = f (x*2) (y-1)
```

(c) Wir zeigen zuerst wieder die Wirkunsgweise des Fixpunktkombinators.

$$\begin{aligned}
\langle \mathrm{Y}\rangle\langle \mathrm{F}\rangle = &\ \Big(\lambda z.\ (\lambda u.z(uu))\ (\lambda u.z(uu))\Big)\ \langle \mathrm{F}\rangle \\
\Rightarrow^\beta &\ \Big(\lambda u.\langle \mathrm{F}\rangle(uu)\Big)\ \Big(\lambda u.\langle \mathrm{F}\rangle(uu)\Big) \qquad =: \langle \mathrm{Y_F}\rangle \\
\Rightarrow^\beta &\ \langle \mathrm{F}\rangle\langle \mathrm{Y_F}\rangle
\end{aligned}$$

Nun können wir die Funktion durch Berechnung der Normalform auswerten:

$$\begin{aligned}
\langle \mathrm{Y}\rangle\langle \mathrm{F}\rangle\langle 2\rangle\langle 1\rangle &\Rightarrow^*\langle \mathrm{Y}\rangle\langle \mathrm{Y_F}\rangle\langle 2\rangle\langle 1\rangle \\
&\Rightarrow^*\langle \mathrm{ite}\rangle\Big(\underbrace{\langle \mathrm{iszero}\rangle\langle 1\rangle}_{\Rightarrow^*\langle \mathrm{false}\rangle}\Big)\Big(\ldots\Big) \\
&\quad\Big(\langle \mathrm{ite}\rangle\Big(\underbrace{\langle \mathrm{iszero}\rangle(\langle \mathrm{mod}\rangle\langle 1\rangle\langle 2\rangle)}_{\Rightarrow^*\langle \mathrm{false}\rangle}\Big)\Big(\ldots\Big)\Big(\underbrace{\langle \mathrm{Y_F}\rangle}_{\Rightarrow^*\langle \mathrm{F}\rangle\langle \mathrm{Y_F}\rangle}\ \underbrace{(\langle \mathrm{mult}\rangle\langle 2\rangle\langle 2\rangle)}_{\Rightarrow^*\langle 4\rangle}\underbrace{(\langle \mathrm{pred}\rangle\langle 1\rangle)}_{\Rightarrow^*\langle 0\rangle}\Big)\Big) \\
&\Rightarrow^*\langle \mathrm{F}\rangle\langle \mathrm{Y_F}\rangle\langle 4\rangle\langle 0\rangle \\
&\Rightarrow^*\langle \mathrm{ite}\rangle\Big(\underbrace{\langle \mathrm{iszero}\rangle\langle 0\rangle}_{\Rightarrow^*\langle \mathrm{true}\rangle}\Big)\Big(\underbrace{\langle \mathrm{mult}\rangle\langle 4\rangle\langle 2\rangle}_{\Rightarrow^*\langle 8\rangle}\Big)\Big(\ldots\Big) \\
&\Rightarrow^*\langle 8\rangle
\end{aligned}$$

Aufgabe 6. Prolog

Hinweis. *Fehlerkorrektur: Die Aufgabe entspricht der Aufgabe 13.13 in der Aufgabensammlung.*

Das angegebene und in der Vorlesung wie Übung praktizierte Schema ist stets beizubehalten. Dazu wird links immer der jeweilige Unifikator angegeben, in dieser Zeile direkt die Einsetzung und Ableitung ausgeführt und am Ende die verwendete Zeile angegeben.

Lösung. Wir geben zwei verschiedene Lösungswege an, die beide als SLD-Refutationen mit einem empty-goal enden.

Alternative 1:

```
                          ?-   insert(<t1>, <t2>, X).
  {X = tree(a, LT1, RT1)} ?-   insert(tree(b, nil, nil), <t2>, LT1),
                               insert(tree(v, nil, nil), <t2>, RT1).   % 6
{LT1 = tree(b, LT2, RT2)} ?-   insert(nil, <t2>, LT2),
                               insert(nil, <t2>, RT2),
                               insert(tree(v,nil,nil), <t2>, RT1).     % 6
   {LT2 = nil, RT2 = nil} ?-*  insert(tree(v, nil, nil), <t2>, RT1).   % 4
            {RT1 = <t2>}  ?-   istree(<t2>).                           % 5
                          ?-*  istree(nil), istree(nil), istree(nil).  % 2
                          ?-*  .                                       % 4
```

Somit ist `X = tree(a, tree(b, nil, nil), <t2>)`.

Alternative 2:

```
                          ?-   insert(<t1>, <t2>, X).
  {X = tree(a, LT1, RT1)} ?-   insert(tree(b, nil, nil), <t2>, LT1),
                               insert(tree(v, nil, nil), <t2>, RT1).   % 6
{LT1 = tree(b, LT2, RT2)} ?-   insert(nil, <t2>, LT2),
                               insert(nil, <t2>, RT2),
                               insert(tree(v,nil,nil), <t2>, RT1).     % 6
   {LT2 = nil, RT2 = nil} ?-*  insert(tree(v, nil, nil), <t2>, RT1).   % 4
{RT1 = tree(v, LT3, RT3)} ?-   insert(nil, <t2>, LT3),
                               insert(nil, <t2>, RT3).                 % 6
   {RT3 = nil, LT3 = nil} ?-*  .                                       % 4
```

Somit ist `X = tree(a, tree(b, nil, nil), tree(v, nil, nil))`.