13. Übung (16.–20. Juli 2018)

# Programmierung

*Lernraum Programmierung am Sa., 28.07.2018, 13:00–15:00 Uhr, APB/E023*

### Aufgabe 1 (AGS 17.25 b)

```
1  module Main where
2
3  h :: Int -> Int -> Int
4  h x1 x2 = if x2 == x1  then 30  else x2
5
6  g :: Int -> Int -> Int
7  g x1 x2 = if 10 <= x2  then g (x1 - x2) (x2 - 1)
8                         else h (x1 + x2) 10
9
10 main = do x1 <- readLn
11           print (g (3 + x1) 5)
```

Füllen Sie die mit `/*A*/` bis `/*F*/` markierten Lücken in der folgenden Übersetzung des obigen $H_0$-Programms in ein äquivalentes $C_0$-Programm:

```
1  #include <stdio.h>
2
3  int main() {
4    int x1, x2, function = 2, flag, result;
5    /*A*/
6    while (flag == 1)
7      if (function == 1)
8        if (/*B*/) { /*C*/ } else { /*D*/ }
9      else if (/*E*/)
10       /*F*/
11   printf("%d", result);
12   return 0;
13 }
```

## Wiederholung

### Aufgabe 2 (AGS 12.1.50 ⋆)

(a) Sei `xs` eine Liste. Eine Teilliste von `xs` ist eine Liste, die durch Löschen von beliebig vielen Elementen aus `xs` entsteht. Die Teillisten von `[1, 2, 3]` sind beispielsweise `[]`, `[1]`, `[2]`, `[3]`, `[1, 2]`, `[1, 3]`, `[2, 3]` und `[1, 2, 3]`.

Schreiben Sie in Haskell die Funktion `isSubseqOf :: Eq a => [a] -> [a] -> Bool`, die prüft, ob die erste übergebene Liste eine Teilliste der zweiten ist.

(b) Gegeben seien folgende Funktionen.

```
1 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
2 zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
3 zipWith _ _         _       = []
4
5 h :: [Int] -> [Int] -> [Int]
6 h xs ys = zipWith (\ x y -> 2 * x + y) xs ys
```

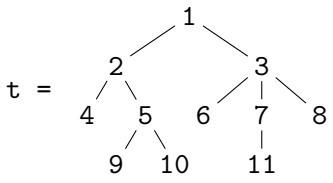
Berechnen Sie schrittweise `h [1, 2] [3, 4, 5]`. Sie dürfen dabei sinnvolle Abkürzungen einführen.

(c) Gegeben sei folgender algebraischer Datentyp für beliebig verzweigende Bäume.

```
data Tree a = Node a [Tree a]
```

Ein Pfad eines Baumes ist die Liste aller Knotenbeschriftungen auf dem Weg von der Wurzel des Baumes zu einem Knoten. (Daraus folgt, dass jeder Pfad mindestens ein Element enthält, nämlich die Beschriftung des Wurzelknotens.)

Implementieren Sie die Funktion `isPathOf :: Eq a => [a] -> Tree a -> Bool`, welche prüft, ob ein Baum einen gegebenen Pfad beinhaltet. Zum Beispiel:




```
isPathOf [] t = False
isPathOf [1, 2] t = True
isPathOf [1, 2, 9] t = False
isPathOf [1, 2, 5, 9] t = True
```

**Aufgabe 3 (AGS 12.2.14 ⋆)**

(a) Gegeben seien die Terme

$$t_1 = \delta(\alpha, \sigma(x_1, \alpha), \sigma(x_2, x_3)) \text{ und}$$
$$t_2 = \delta(\alpha, \sigma(x_1, x_2), \sigma(x_2, \gamma(x_2)))$$

über dem Rangalphabet $\Sigma = \{\delta^{(3)}, \sigma^{(2)}, \gamma^{(1)}, \alpha^{(0)}\}$. Wenden Sie den Unifikationsalgorithmus auf die Terme $t_1$ und $t_2$ an. Wenden Sie bei jedem Umformungsschritt nur eine Regelsorte an und geben Sie diese jeweils an. Geben Sie anschließend den von Ihnen bestimmten allgemeinsten Unifikator an.

(b) Gegeben seien die Haskell-Typterme

$$t_1 = \texttt{(a, [a])}, \quad t_2 = \texttt{(Int, [Double])} \quad \text{und} \quad t_3 = \texttt{(b, c)}.$$

Kreuzen Sie die richtige Antwort an und ergänzen Sie ggf. einen allgemeinsten Unifikator:

$t_1$ und $t_2$ sind
☐ nicht unifizierbar
☐ unifizierbar mit

`a` ↦

$t_1$ und $t_3$ sind
☐ nicht unifizierbar
☐ unifizierbar mit

`a` ↦

`b` ↦

`c` ↦

$t_2$ und $t_3$ sind
☐ nicht unifizierbar
☐ unifizierbar mit

`b` ↦

`c` ↦

**Aufgabe 4 (AGS 12.4.33 ⋆)**

(a) Berechnen Sie die Normalform des $\lambda$-Terms $(\lambda fx.x(f(fx)))(\lambda y.xy)$, indem Sie ihn *schrittweise* reduzieren.

(b) Gegeben sei der $\lambda$-Term $\langle F \rangle$:

$$\Big(\lambda fxy.\langle \text{ite} \rangle \ (\langle \text{iszero} \rangle\, y) \ (\langle \text{mult} \rangle\, x\, \langle 2 \rangle) \ \Big(\langle \text{ite} \rangle \ (\langle \text{iszero} \rangle\, (\langle \text{mod} \rangle\, y\, \langle 2 \rangle)) \ (f\, x\, (\langle \text{pred} \rangle\, y)) \ (f\, (\langle \text{mult} \rangle\, x\, \langle 2 \rangle)\, (\langle \text{pred} \rangle\, y))\Big)\Big)$$

Geben Sie eine Haskell-Funktion `f` an, so dass $\texttt{f} = \langle Y \rangle \langle F \rangle$ gilt!

(c) Betrachten Sie den $\lambda$-Term $\langle F \rangle$ aus Teilaufgabe 4 (b). Berechnen Sie die Normalform des Terms $\langle Y \rangle \langle F \rangle \langle 2 \rangle \langle 1 \rangle$. Schreiben Sie für jeden Aufruf von $\langle F \rangle$ jeweils **zwei Zeilen**: eine in der Sie die Werte der Parameter des Aufrufs protokollieren, und eine in der Sie ihre Auswertung skizzieren.

Stellen Sie zuerst die Wirkungsweise des Fixpunktkombinators $\langle Y \rangle$ in einer Nebenrechnung dar. Führen Sie zweckmäßige Abkürzungen der $\lambda$-Terme ein. Die spitzen Klammern $\langle \cdots \rangle$ dürfen Sie weglassen. Sie können die in der AGS gegebenen Terme und Beziehungen benutzen.

**Aufgabe 5**

```
1  nat(0).
2  nat(s(X)) :- nat(X).
3
4  a(0, 0, 0).
5  a(X, 0, X) :- nat(x).
6  a(0, Y, Y) :- nat(Y).
7  a(s(X), s(Y), s(s(Z)))
       :- a(X, Y, Z).
8
9  b(X, 0, 0) :- nat(X).
10 b(X, s(Y), Z)
       :- b(X, Y, M), a(M, X, Z).
11
12 c(X, 0, s(0)) :- nat(X).
13 c(X, s(Y), Z)
       :- c(X, Y, Z1),b(Z1, X, Z).
```

(a) Im Folgenden verwenden wir die Abkürzung `<n>` für den Term, der die natürliche Zahl `n` darstellt. Geben Sie eine SDL-Refutation für `?- c(<2>, <2>, <4>).` an.

(b) Was berechnen die drei Relationen `a`, `b` und `c`?

**Aufgabe 6 (AGS 15.23 ⋆)**

(a) Gegeben sei folgendes Fragment eines $C_1$-Programms mit den Funktionen `g` und `f`:

```
while (*a > b)  if (b != 0)  g(a, &b);  else *a = b;
```

Übersetzen Sie dieses Fragment mittels *sttrans* in $AM_1$-Code mit baumstrukturierten Adressen. Sie müssen keine Zwischenschritte angeben. Nehmen Sie dabei an, dass die `while`-Anweisung das zweite Statement in `f` ist, und es sei

$$tab_{\text{f+lDecl}} = [\texttt{g}/(\text{proc}, 1),\ \texttt{f}/(\text{proc}, 2),\ \texttt{b}/(\text{var}, \text{global}, 2),\ \texttt{a}/(\text{var-ref}, -2),\ \texttt{y}/(\text{var}, \text{lokal}, 1)].$$

(b) Erstellen Sie ein Ablaufprotokoll der $AM_1$ für das unten angegebene Programm, bis die $AM_1$ terminiert. Die Startkonfiguration sei $(8, \varepsilon, 2:3:0:1:2:4:29:3, 8, \varepsilon, \varepsilon)$.

```
 1: INIT 1;            11: STOREI(-2);       21: READ(global,1);
 2: CALL 20;           12: LOAD(lokal,-3);   22: LIT 1;
 3: JMP 0;             13: LIT 1;            23: STORE(lokal,1);
 4: LOAD(lokal,-3);    14: ADD;              24: LIT 2;
 5: LOAD(global,1);    15: PUSH;             25: PUSH;
 6: LE;                16: LOAD(lokal,-2);   26: LOADA(lokal,1);
 7: JMC 19;            17: PUSH;             27: PUSH;
 8: LOADI(-2);         18: CALL 4;           28: CALL 4;
 9: LOAD(lokal,-3);    19: RET 2;            29: WRITE(lokal,1);
10: MUL;               20: INIT 1;           30: JMP 0;
```

**Aufgabe 7 (AGS 17.14 ⋆)**

(a) Transformieren Sie die folgende Funktion `h` eines $H_0$-Programms in ein $\mathrm{AM}_0$-Programm mit baumstrukturierten Adressen.

```
h x1 = if x1 == 0 then h 1 else x1
```

(b) Folgendes $H_0$-Programm sei gegeben:

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

Vervollständigen Sie die Angaben `/*A*/` bis `/*F*/` in der folgenden Übersetzung des $H_0$-Programms, so dass ein äquivalentes $\mathrm{C}_0$-Programm entsteht:

```
#include <stdio.h>

int main() {
  int x1, x2, function, flag, result;
  /*A*/
  flag = 1;
  function = 1;
  while (flag == 1)
    if (/*B*/)
      if (x1 < x2) { /*C*/ function = 2; } else { /*D*/ }
    else if (/*E*/) { /*F*/ }
  printf("%d", result);
  return 0;
}
```