Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik

1. Juli 2014

10. Musterlösung zur Vorlesung

Programmierung und Modellierung

A10-1 ***Redex*** Identifizieren Sie alle Redexe in den folgenden Programmausdrücken, und geben Sie an, welche davon innerste und äußerste Redexe sind.

a) `(1 + 2) * (4 / 5)`

LÖSUNGSVORSCHLAG:

Es gibt zwei Redexe `1 + 2` und `4 / 5`. Die Multiplikation ist kein Redex: wir können die Multiplikation jetzt noch nicht ausführen, da die dazu benötigten Argumente noch nicht vorliegen.

Damit sind beide Redexe sowohl innerste als auch äußerste Redexe - es gibt hier ja keinerlei Verschachtelung der Redexe.

b) `snd (1 + (2 + 3), 4 + 5)`
wobei die Funktion `snd :: (a,b) -> b` üblicherweise definiert ist durch `\(x,y) -> y`

LÖSUNGSVORSCHLAG:

Hier haben wir drei Redex: Die beiden innersten Redexe `2 + 3` und `4 + 5` sind enthalten im äußersten Redex `snd (1 + (2 + 3), 4 + 5)`, also der Anwendung eines Argumentes auf die Funktion `snd`. Die Funktionsanwendung können wir immer reduzieren, denn wir müssen dabei ja nur die Argumente im definierenden Rumpf der Funktion substituieren.

c) `(\x -> (1 * 2) + x) (3+4)`

LÖSUNGSVORSCHLAG:

Nach unserer Definition gibt es keinen Redex unter einem Lambda, d.h. `1*2` ist kein Redex.

`3+4` ist ein innerster Redex; dieser ist enthalten im äußerster Redex `(\x -> (1 * 2) + x) (3+4)`, der Funktionsanwendung.

d) `(\f -> f (1 * 2)) (\x -> 3+4)`

LÖSUNGSVORSCHLAG:

Wir reduzieren nicht unter einem Lambda, d.h. nach einem `->` brauchen wir nicht nach Redexen suchen.

Es gibt hier nur einen Redex, den kompletten Ausdruck, welcher erneut eine Funktionsanwendung ist. Dieser ist innerster und äußerster zugleich.

A10-2 ***Auswertestrategie*** Werten Sie den Programmausdruck entweder mit der Auswertestrategie Call-By-Name oder Call-By-Value aus. Welche Auswertestrategie haben Sie jeweils gewählt und warum?

a) `(\(x,y) -> y) (1 + (2 + 3), 4 + 5)`

LÖSUNGSVORSCHLAG:

Wir demonstrieren hier beide Varianten. Auch schreiben wir ganz ausführlich als Extraschritte mit Gleichheitszeichen die Ausfaltung der Funktionsdefinition und die Substitution der Funktionsargumente hin.

Call-By-Name

$$\begin{aligned} &\texttt{snd}\ (1 + (2 + 3), 4 + 5) = (\backslash(x, y)\ \texttt{->}\ y)\ (1 + (2 + 3), 4 + 5) \\ &\rightsquigarrow y[(1 + (2 + 3))/x, (4 + 5)/y] = 4 + 5 \rightsquigarrow 9 \end{aligned}$$

Insgesamt also 2 Evaluationsschritte.

Call-By-Value

$$\begin{aligned} &\texttt{snd}\ (1 + (2 + 3), 4 + 5) \rightsquigarrow \texttt{snd}\ (1 + 5, 4 + 5) \rightsquigarrow \texttt{snd}\ (6, 4 + 5) \\ &\rightsquigarrow \texttt{snd}\ (6, 9) = (\backslash(x, y)\ \texttt{->}\ y)\ (6, 9) \rightsquigarrow y[6/x, 9/y] = 9 \end{aligned}$$

Insgesamt also 4 Evaluationsschritte.

b) `(\x -> x + x) ((\y -> y * y) (1+1))`

LÖSUNGSVORSCHLAG:

Call-By-Name:

$$(\backslash x\ \texttt{->}\ x+x)((\backslash y\ \texttt{->}\ y*y)(1+1)) \leadsto ((\backslash y\ \texttt{->}\ y*y)(1+1)) + ((\backslash y\ \texttt{->}\ y*y)(1+1))$$
$$\leadsto ((1+1)*(1+1)) + ((\backslash y\ \texttt{->}\ y*y)(1+1)) \leadsto ((2)*(1+1)) + ((\backslash y\ \texttt{->}\ y*y)(1+1))$$
$$\leadsto (2*2) + ((\backslash y\ \texttt{->}\ y*y)(1+1)) \leadsto 4 + ((\backslash y\ \texttt{->}\ y*y)(1+1)) \leadsto 4 + ((1+1)*(1+1))$$
$$\leadsto 4 + (2*(1+1)) \leadsto 4 + (2*2) \leadsto 4+4 \leadsto 8$$

Insgesamt 10 Schritte.

Call-By-Value:

$$(\backslash x\ \texttt{->}\ x+x)((\backslash y\ \texttt{->}\ y*y)(1+1)) \leadsto (\backslash x\ \texttt{->}\ x+x)((\backslash y\ \texttt{->}\ y*y)2)$$
$$\leadsto (\backslash x\ \texttt{->}\ x+x)(2*2) \leadsto (\backslash x\ \texttt{->}\ x+x)4 \leadsto 4+4 \leadsto 8$$

Insgesamt 5 Schritte.

A10-3 ***Verzögerte Auswertung*** Betrachten Sie das folgende Haskell Programm:

```
ps = iterate (+1) 0
qs = iterate (*2) 1
rs = iterate (^2) 2

iterate f x = x : iterate f (f x)

foo (_:x:xs) ys@(y:_) zs
  | x  > y    = x
  | otherwise = foo ys zs xs
```

Aufgrund der verzögerten Auswertestrategie von Haskell wird z.B. die Liste `ps` anfangs nur als ein Verweis auf den Code `iterate (1+) 0` abgespeichert. Erst sobald auf das erste Element dieser Liste zugegriffen wird, wird zeigt `ps` auf die Liste `0: iterate (+1) (0+1)`. Wird später dann auch noch das zweite Element benötigt, so zeigt der Bezeichner `ps` nun auf die Liste `0:1: iterate (+1) (1+1)` im Speicher.

Bis zu welchem Element werden die Listen `ps`, `qs`, `rs` im Speicher ausgewertet, wenn nur der Aufruf `foo ps qs rs` ausgeführt wird?

LÖSUNGSVORSCHLAG:

Zur Veranschaulichung werten wir die Listen erst einmal aus, damit wir leichter darüber reden können:

```
> take 4 $ iterate (+1) 0
[0,1,2,3,4]
> take 4 $ iterate (*2) 1
[1,2,4,8,16]
> take 4 $ iterate (^2) 2
[2,4,16,256,65536]
```

Zuerst wird das zweite Elemente von `ps` und das erste Element von `qs`, um den Vergleich zu überprüfen. Wegen $1 \not> 1$ kommt es zum rekursiven Aufruf. Jetzt wird das zweite Element von `qs` und das erste Element von `rs` benötigt. Wegen $2 \not> 2$ kommt es erneut zu einem rekursiven Aufruf. Verglichen wird nun das zweite Element von `rs` mit dem dritten Element von `xs`. Da $4 > 2$ gilt, ist die Auswertung nur beendet. Die Liste `ps` wurde also bis zum dritten Element ausgewertet. Die Listen `qs` und `rs` nur bis zum jeweils zweiten Element.

A10-4 ***Faule Fibonacci Zahlen*** Definieren Sie die Liste aller Fibonacci-Zahlen in Haskell, also `fibs :: [Integer]`. Achten Sie dabei auch auf Effizienz!

Zur Erinnerung: Die Liste aller Fibonacci Zahlen beginnt mit 0 und 1. Die i-te Fibonacci-Zahl ist immer die Summe ihrer beiden Vorgänger.

LÖSUNGSVORSCHLAG:

Zuerst eine herkömmliche Lösung zu Fuss:

```
fibs = 0 : 1 : gen_fibs fibs
  where gen_fibs (h1:(h2:t))   = h1 + h2 : gen_fibs (h2:t)
```

Mit einem `@`-Pattern kann man die Speichernutzung etwas verbessern:

```
fibs = 0 : 1 : gen_fibs fibs
  where gen_fibs (h1:t@(h2:_)) = h1 + h2 : gen_fibs t
```

Eine besonders elegante Lösung ermöglicht die Funktion `zipWith`:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Bemerkung: Bei einer Rekursion wird die Eingabe Schritt-für-Schritt verkleinert und die Berechnung auf jeweils einfachere Teile zurückgeführt. Hier jedoch werden die größeren/späteren Teile der Datenstruktur mithilfe der kleineren Teile aufgebaut. Solche Programme nennt man daher auch *co-rekursiv*, da die Rekursion praktisch anders herum verläuft.

H10-1 ***Redex II*** **(2 Punkte)** (Abgabeformat: Text oder PDF)
Identifizieren Sie alle Redexe in den folgenden Programmausdrücken, und geben Sie an, welche davon innerste und äußerste Redexe sind.

a) `(\(x,y,z) -> (0+1,y*z)) (2,3+4,(\u->5) 6)`

LÖSUNGSVORSCHLAG:

Der äußerste Redex ist die Funktionsanwendung des Tripels auf die Funktion `\(x,y,z)->...`. Es gibt zwei innerste Redexe `3+4` und `(\u->5) 6`. Da per Definition unter einem Lambda nicht reduzieren dürfen, gibt es keine weiteren Redexe hier.

b) `(\g z -> (\f x -> f (f x)) (\y -> 1+2) 3)`

LÖSUNGSVORSCHLAG:

Der Ausdruck beginnt mit einem Lambda, die Klammer am Anfang endet erst am Schluss des gesamten Ausdrucks. Da wir unter einem Lambda nicht reduzieren, enthält dieser Ausdruck keinen Redex.

H10-2 ***Auswertestrategie II*** **(3 Punkte)** (Abgabeformat: Text oder PDF)
Werten Sie den Programmausdruck entweder mit der Auswertestrategie Call-By-Name oder Call-By-Value aus:

```
(\f x -> f ( f x )) ((\y z -> y+y) (2*2)) (3+4)
```

Welche Auswertestrategie haben Sie gewählt und warum?

LÖSUNGSVORSCHLAG:
Wir betrachten beide Varianten.

Call-By-Name:

$$
\begin{aligned}
&(\backslash f\ x\ \texttt{->}\ f(fx))\ ((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (3+4)\\
&\rightsquigarrow (\backslash x\ \texttt{->}\ ((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ x))\ (3+4)\\
&\rightsquigarrow ((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (3+4))\\
&\rightsquigarrow (\backslash z\ \texttt{->}\ (2*2)+(2*2))\ (((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (3+4))\\
&\rightsquigarrow (2*2)+(2*2) \rightsquigarrow 4+(2*2) \rightsquigarrow 4+4 \rightsquigarrow 8
\end{aligned}
$$

Insgesamt 6 Schritte.

Call-By-Value:

$$
\begin{aligned}
&(\backslash f\ x\ \texttt{->}\ f(fx))\ ((\backslash y\ z\ \texttt{->}\ y+y)\ (2*2))\ (3+4)\\
\leadsto\ &(\backslash f\ x\ \texttt{->}\ f(fx))\ ((\backslash y\ z\ \texttt{->}\ y+y)\ 4)\ (3+4)\\
\leadsto\ &(\backslash f\ x\ \texttt{->}\ f(fx))\ (\backslash z\ \texttt{->}\ 4+4)\ (3+4)\\
\leadsto\ &(\backslash f\ x\ \texttt{->}\ f(fx))\ (\backslash z\ \texttt{->}\ 4+4)\ 7\\
\leadsto\ &(\backslash x\ \texttt{->}\ (\backslash z\ \texttt{->}\ 4+4)((\backslash z\ \texttt{->}\ 4+4)x))\ 7\\
\leadsto\ &(\backslash z\ \texttt{->}\ 4+4)((\backslash z\ \texttt{->}\ 4+4)7))\\
\leadsto\ &4+4 \leadsto 8
\end{aligned}
$$

Insgesamt 7 Schritte.

Nicht vergessen: In beiden Varianten dürfen wir nicht unter einem Lambda reduzieren, daher bleibt z.B. die $4+4$ in der Call-By-Value Variante bis zum Schluss stehen!

H10-3 *Fixpunktkombinator* (3 Punkte) (`.hs`-Datei als Lösung abgeben)
Ein Wert `x0` heißt Fixpunkt einer Funktion `f`, wenn die Gleichung `f x0 = x0` gilt. Der Fixpunktkombinator `fix` berechnet Fixpunkte von Funktionen:

```
fix f = f (fix f)
```

Es sei `x1 = fix foo`, dann gilt offenbar `foo x1 = x1`. Wer es nicht glaubt, kann es durch Einsetzen der definierenden Gleichungen leicht nachrechnen:

```
foo x1 = foo (fix foo) = fix foo = x1
```

Implementieren Sie eine Funktion zur Berechnung der n-ten Fibonaccizahl mithilfe des Fixpunktkombinators, also ohne Rekursion und ohne Funktionen der Standardbibliothek!

Hinweis: Der Trick besteht darin, eine Funktion zu definieren, welche zwei Argumente bekommt: das erste Argument soll die partielle Zielfunktion darstellen ("Fibonacci für Argumente bis zu einer gewissen Größe"), das zweite Argument ist das tatsächliche Argument für die Zielfunktion. Die Funktion soll die partielle Zielfunktion dann um einen Schritt erweitern. Den Rest erledigt der Fixpunktkombinator.

LÖSUNGSVORSCHLAG:

Unsere unvollständige Funktion beherrscht nur die Basisfälle und die Berechnung eines Schrittes:

```
fib = fix fib'
  where
    fib' _ 0 = 0
    fib' _ 1 = 1
    fib' f n = (f (n-1)) + (f (n-2))
```

Durch wiederholte Anwendung dieser Funktion erhalten wir die gewünschte Funktion als Fixpunkt.

Hinweis: Themenvorschläge für die Wiederholungsvorlesung der kommenden Woche bitte im Forum die-informatiker.net sammeln.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 8.07.2014, 11:00 Uhr mit UniworX abgegeben werden.

Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.