Institut für Informatik der Ludwig-Maximilians-Universität München
Prof. Martin Hofmann, PhD
Dr. Steffen Jost

SoSe 2015
21.07.2015

Lösung zur Klausur zur Vorlesung
Programmierung und Modellierung

Die Bearbeitungszeit beträgt 120 min. Hilfsmittel sind nicht erlaubt, auch der Besitz ausgeschalteter elektronischer Geräte wird als Betrug gewertet. Schreiben Sie Ihren vollständigen Namen und Ihre Matrikelnummer deutlich lesbar auf dieses Deckblatt, sowie Ihren Namen in die Kopfzeile auf **jedem Blatt** der Klausurangabe! Geben Sie alle Blätter ab, lassen Sie diese zusammengeheftet! Verwenden Sie nur dokumentenechte Stifte und weder die Farben rot noch grün.

Kontrollieren Sie, ob Sie alle Aufgabenblätter erhalten haben. Aufgabenstellungen befinden sich auf den Seiten 1–11. Sie dürfen die Rückseiten für Nebenrechnungen nutzen. Falls Sie die Rückseiten für Antworten nutzen, so markieren Sie klar, was zu welcher Aufgabe gehört und schreiben Sie in der entsprechenden Aufgabe, wo alle Teile Ihrer Antwort zu finden sind. Streichen Sie alles durch, was nicht korrigiert werden soll!

Lesen Sie die Aufgabenstellungen vollständig durch, bevor Sie mit der Bearbeitung der Aufgaben beginnen. Es gibt 8 unterschiedlich gewichtete Aufgaben zu insgesamt 50 Punkten. Mit 25 Punkten haben Sie sicherlich bestanden. Die Teilaufgaben können unabhängig voneinander bearbeitet werden. Gemäß Studienordung gilt für alle (Teil-)Aufgaben mit Antworten zum Ankreuzen (Multiplechoice), dass jede falsche Antwort genau so viel Minuspunkte bringt, wie eine richtige Antwort Pluspunkte bringt. Die Minuspunkte werden aber nicht auf andere Aufgaben übertragen, d.h. im schlimmsten Fall bekommt man 0 Punkte auf die jeweilige Aufgabe. Eine unbeantwortete Frage zählt 0 Punkte.

Mit Ihrer Unterschrift bestätigen Sie, dass Sie zu Beginn der Klausur in ausreichend guter gesundheitlicher Verfassung sind, und diese Klausurprüfung verbindlich annehmen.

Nachname:

Vorname:

Matrikelnummer:

Studiengang:

Hiermit erkläre ich die Richtigkeit der obigen Angaben:

(Unterschrift)

Aufgabe	1	2	3	4	5	6	7	8	$\sum$	Bonus	Note
Punkte	7	6	5	7	6	6	5	8	**50**		
erreicht											

Lösung Aufgabe 1 (Auswertung): (7 Punkte)

a) Rechnen Sie aus, zu welchem Wert die folgenden Haskell-Ausdrücke vollständig auswerten. Geben Sie nur das Endergebnis an, etwaige Nebenrechnungen bitte deutlich abtrennen:

(i) `forM [1..3] print`

`[(),(),()]` wie in A11-3b geübt und in H11-3 versprochen! ;)

(ii) `let foo = "bar" in if True && False then "foo" else foo`

“`bar`“ Aufgaben sind alle analog zu A1-2 & A10-1

(iii) `[(x,y,z) | let y = 2, x<-[1..2], z<-[x..3], odd (x+y) ]`
Hinweis: `odd` gibt nur dann `True` zurück, wenn das Argument ungerade ist.

`[(1,2,1),(1,2,2),(1,2,3)]`

(iv) `(\x-> (\y -> y:x)) [] ()`

`[()]`

b) Werten Sie folgenden Ausdruck in Einzelschritten vollständig aus. Unterstreichen Sie dabei jeweils den reduzierten Redex. Verwenden Sie die Auswertestrategie Call-By-Name um die volle Punktzahl zu erreichen!

`(\x->(\y->y x x)) (3+4) (\a->(\b->a*b))` $\leadsto$

LÖSUNG: Funktionsanwendung ist implizit links geklammert, d.h. auszuwerten ist `((\x->(\y->(y x) x)) (3+4)) (\a->(\b->a*b))` Fehlerhafte Klammerung ist hier oft der häufigste Fehler. Analog zu A1-3, A11-1, H11-1, A13-1, etc.

Call-By-Name

$$\begin{aligned}
&\underline{((\backslash x \texttt{ -> } (\backslash y \texttt{ -> } (y\ x)\ x))\ (3+4))}\ (\backslash a \texttt{ -> } (\backslash b \texttt{ -> } a*b))\\
\leadsto\ &\underline{(\backslash y \texttt{ -> } (y\ (3+4))\ (3+4))\ (\backslash a \texttt{ -> } (\backslash b \texttt{ -> } a*b))}\\
\leadsto\ &\underline{((\backslash a \texttt{ -> } (\backslash b \texttt{ -> } a*b))\ (3+4))}\ (3+4)\\
\leadsto\ &\underline{(\backslash b \texttt{ -> } (3+4)*b)\ (3+4)} \leadsto \underline{(3+4)}*(3+4) \leadsto 7*\underline{(3+4)} \leadsto \underline{7*7} \leadsto 49
\end{aligned}$$

c) Nennen Sie einen Vorteil von Call-By-Value gegenüber Call-By-Name:

LÖSUNG: Vermeidet doppelte Berechnungen bei doppelter Verwendung eines Arguments – wie in der vorherigen Teilaufgabe.

d) Welche Auswertestrategie benutzt die Haskell-Implementierung `ghc`:

Lazy Evaluation (auch akzeptiert: Call-By-Need) [Skriptwissen]

Lösung Aufgabe 2 (Induktion mit Listen): (6 Punkte)

Wir betrachten folgende Funktionsdefinitionen:

```
pone :: [Integer] -> [Integer]
pone  []      = [1]                          -- (PN)
pone (68:xs) = 69 : xs                       -- (PS)
pone ( x:xs) =  x : (pone xs)                -- (PC)

sum :: [Integer] -> Integer
sum  []      = 0                             -- (SN)
sum (y:ys)   = y + (sum ys)                  -- (SC)
```

Beweisen Sie mit Induktion über die Länge, dass für eine beliebige Liste z gilt:

`sum (pone z)` $=$ `1 + (sum z)`

Hinweise: Formen Sie beide Seiten der geforderten Gleichung in Einzelschritten in den exakt gleichen Term um. Begründen Sie jeden Umformungsschritt durch Angabe des Kürzels der verwendeten Gleichung! Für Funktionsdefinitionen also `(PN)`, `(PS)`, `(PC)`, `(SN)` oder `(SC)`; für die Induktionshypothese `(IH)`; und bei Verwendung der üblichen Rechenregel für Plus (also z.B. Assoziativität, Kommutativität, etc.) das Kürzel `(+)`.

Beispiel: $1 + (\texttt{sum}\ [2,3]) =_{(SC)} 1 + (2 + (\texttt{sum}\ [3])) =_{(+)} (1+2) + (\texttt{sum}\ [3])$

LÖSUNG: *Diese Aufgabe ist sehr ähnlich zu A4-3 bzw. A13-2:* `pone` *entspricht* `insert` *und* `sum` *entspricht* $|\cdot|$ *(A4-3) bzw.* `length` *(A13-2). Die Struktur des Beweises bleibt aber völlig identisch, obwohl* `sum xs` *schon etwas ganz anderes ausrechnet als* `length xs`*. Offenbar führte auswendiges Wissen ohne Verstand hier zu dem häufigsten Fehler* `sum (h : t) =(??)= 1 + sum t` *anstatt der richtigen Gleichung* `sum (h : t) =(SC)= h + sum t`*!*

Der Beweis wird mit Induktion über die Länge der Liste `z` geführt.

Induktionsanfang für eine Liste der Länge 0: Es gilt also `z = []`.

`sum (pone []) =(PN)= sum [1] =(SC)= 1 + (sum [])`

Damit haben wir die linke Seite der geforderten Gleichung direkt in die rechte Seite überführt. Den Schritt `=(SN)= 1 + 0 = 1` benötigen wir also gar nicht mehr.

Induktionsschritt für Liste der Länge $n > 0$: Da die Liste `z` in diesem Fall nicht leer ist, können wir dem Kopf und Rumpf von `z` eigene, frische Namen geben: Es sei `z = (h:t)`. Damit ist `t` eine Liste mit einer Länge kleiner als n. Somit dürfen wir als Induktionshypothese `(IH)` die Gleichung `sum (pone t) = 1 + (sum t)`.

Wir müssen die Gleichung `sum (pone (h:t)) = 1 + sum (h:t)` beweisen. Hier ist nun eine Fallunterscheidung notwendig, je nachdem, ob $h = 68$ gilt:

Fall $h \neq 68$: Wir rechnen:

```
sum (pone (h:t)) =(PC)= sum (h : (pone t)) =(SC)= h + (sum (pone t))
=(IH)= h + (1 + (sum t)) =(+)= 1 + (h + (sum t)) =(SC)= 1 + sum (h:t)
```

Wer Schwierigkeiten mit dem letzten Umformungsschritt hat, sollte diesen einfach von rechts-nach-links lesen.

Fall $h = 68$: `sum (pone (68:t)) =(PS)= sum (69 : t) =(SC)= 69 + (sum t)`
`=(+)= 68 + (1 + (sum t)) =(+)= 1 + (68 + (sum t)) =(SC)= 1 + sum (68:t)`

Hier ist keine Induktion notwendig – es gibt ja in diesem Fall auch keinen rekursiven Aufruf!

Lösung Aufgabe 3 (Abstiegsfunktion): (5 Punkte)

Wir wollen mithilfe einer geeigneten Abstiegsfunktion zeigen, dass die folgende rekursive Funktion immer terminiert:

```
data Tree a = Empty | Node a (Tree a) (Tree a)

breadthForest :: [Tree a] -> [a]
breadthForest [] = []
breadthForest ( Empty       : frst) = breadthForest frst
breadthForest ((Node x l r) : frst) = x : breadthForest (frst ++ [l,r])
```

a) Zeigen Sie, dass die Funktion m' : `[Tree a]` $\rightarrow \mathbb{N}$, welche die Anzahl aller in der Eingabe enthalten `Empty`-Konstruktoren zurückgibt, noch keine geeignete Abstiegsfunktion für den Terminationsbeweis von `breadthForest` ist. *Beispiel:* m'(`[Node 1 Empty Empty, Empty]`) $= 3$

LÖSUNG: Wir wählen als Argument das vorgegebene `[Node 1 Empty Empty,Empty]` (viele andere Werte sind auch möglich). Damit sind wir im dritten Fall der Funktionsgleichung. Es findet dort ein rekursiver Aufruf mit dem Argument `[Empty] ++ [Empty, Empty]` statt.

Wenn m' eine Abstiegsfunktion wäre, dann müsste m'(`[Node 1 Empty Empty, Empty]`) $>$ m'(`[Empty,Empty, Empty]`) gelten.

Dies gilt aber nicht, denn in beiden Fällen sind jeweils drei `Empty`-Konstruktoren in der Eingabe enthalten, also: m'(`[Node 1 Empty Empty, Empty]`) $= 3$ und m'(`[Empty, Empty, Empty]`) $=$ 3, aber $3 > 3$ gilt eben nicht.

b) Finden Sie eine geeignete Abstiegsfunktion und beweisen Sie damit, dass `breadthForest` immer terminiert.
Hinweise: Auf die Bedingungen Auf und Def verzichten wir hier. Weiterhin dürfen Sie annehmen, dass `l ++ r` die gleichen Elemente enthält wie die Einzellisten `l` und `r` zusammen.

LÖSUNG: Wir wählen als Abstiegsfunktion m : `[Tree a]` $\rightarrow \mathbb{N}$ die Summe aller in der Eingabe enthalten `Empty`-Konstruktoren und `Node`-Konstruktoren.

Wir überprüfen alle rekursiven Aufrufe:

2. Fall: Es gilt m(`Empty` : `frst`) $= 1 + m$(`frst`), und damit folgt direkt m(`Empty` : `frst`) $>$ m(`frst`) wie benötigt.

3. Fall: Es gilt offensichtlich m(`[Node x l r]`) $= 1 + m$(`[l]`) $+ m$(`[r]`) und damit auch m(`(Node x l r):frst`) $= 1 + m$(`[l]`) $+ m$(`[r]`) $+ m$(`frst`) $> m$(`frst ++ [l,t]`) wie benötigt.

Diese Aufgabe ist also tatsächlich einfacher als die sehr ähnliche H6-1! Anstatt irgendeiner abstrakten `fubar`-Funktion, unter der man sich nichts vorstellen kann, haben wir eine sehr konkrete Funktion (Folie 7-14) verwendet. Doch einige haben offenbar nicht verstanden, worum es bei der Abstiegsfunktion wirklich geht und haben dann hilflos mit Betrag und Maximum-Funktion herumhantiert.

Lösung Aufgabe 4 (Datenstrukturen): (7 Punkte)

Gegeben sind folgende Datentypdeklaration:

```
data Geld = EUR Double | CHF Double | GBP Double deriving (Show)
data Fruchtkorb a = Apfel a | Birne (Int,a) | Banane Double
                  | Mischkiste (Fruchtkorb a) Int (Fruchtkorb a) deriving (Show)
```

a) Schreiben Sie eine Funktion `toEuro :: Geld -> Geld`, welche einen Geldbetrag in einer der drei Währungen in Euro umrechnet. Für uns gilt 1 GBP = 1.44 EUR und 1 CHF = 0.96 EUR.

LÖSUNG: Hier muss man nur wissen wie Pattern-Matching geht; und dass man in Haskell Konstruktoren *vor* die Argumente schreibt: Anstatt `f x = x EUR` muss es `f x = EUR x` heissen! Den ersten Fall darf man hier nicht vergessen!

```
toEuro :: Geld -> Geld
toEuro e@(EUR _) = e                 --  oder:  toEuro (EUR e) = EUR e
toEuro (CHF f)   = EUR $ f * 0.96
toEuro (GBP p)   = EUR $ p * 1.44
```

b) Schreiben Sie eine Funktion, welche alle `a`-Werte eines Fruchtkorb mit einer gegebenen Funktion in `b`-Werte verwandelt, also `convert :: (a -> b) -> (Fruchtkorb a -> Fruchtkorb b)`. Alles andere eines Fruchtkorb-Wertes soll dabei unverändert bleiben!

Beispiel:
```
> convert toEuro (Mischkiste (Apfel (CHF 2.99)) 7 (Birne (3, GBP 0.99)))
Mischkiste (Apfel (EUR 2.8704)) 7 (Birne (3, EUR 1.4256))
```

LÖSUNG: Gesucht ist also lediglich die Funktoren-Instanz, also die Definition von `fmap` für diesen Typ, wie in A6-3b/c(!), H5-2, A10-3 & A13-5 geübt.

```
convert :: (a -> b) -> Fruchtkorb a -> Fruchtkorb b
convert f (Apfel x)            = Apfel $ f x
convert f (Birne (i,x))        = Birne (i,f x)
-- convert f b@(Banane _)      = b -- Typfehler! (nicht als Fehler gewertet)
convert f (Banane b)           = Banane b
convert f (Mischkiste xs i ys) = Mischkiste (convert f xs) i (convert f ys)
```

c) Machen Sie den Typ `Fruchtkorb` zu einer Instanz der Typklasse `Functor`. Sie dürfen dazu Ihre eigenen Definition zu dieser Aufgabe wiederverwerten (ggf. auch unvollständige/fehlerhafte).

LÖSUNG: Wer hat erkannt, dass die vorangegangene Teilaufgabe einen Funktor beschreibt (der geringe Platz deutete darauf hin) und weiß noch, wie Instanzdeklarationen funktionieren?

```
instance Functor Fruchtkorb where
  fmap = convert
```

d) In der Vorlesung wurden AVL-Bäume und Rot-Schwarz-Bäume als verbesserte Suchbäume behandelt. Was war das Problem mit den naiven Suchbäumen gewesen?

LÖSUNG: Ein Suchbaum kann unblanciert sein / zu einer Liste degenerieren, so dass im schlimmsten Fall $O(n)$ Zugriffe zum Finden eines Elementes notwendig sind. [Skriptwissen]

Lösung Aufgabe 5 (Funktionen höherer Ordnung): (6 Punkte)

a) Implementieren Sie die Funktion `myAny :: (a -> Bool) -> [a] -> Bool`, welche testet, ob es mindestens ein Element in der Liste gibt, welches das Prädikat erfüllt. Für die volle Punktzahl müssen Sie direkte Rekursion verwenden und dürfen keine Funktionen höherer Ordnung aus der Standardbibliothek verwenden.

Beispiele:

```
> myAny even [1,3,5,7]
False
> myAny even [3,5,8,13,21,34,55]
True
```

LÖSUNG: Ähnlich zu A5-1 und H5-1:

```
myAny :: (a -> Bool) -> [a] -> Bool
myAny p []    = False
myAny p (h:t) = p h || myAny p t
```

b) Schreiben Sie eine Funktion `getFirst :: (a -> Bool) -> [a] -> Maybe a`, welche das erste Element einer Liste zurückgibt, welches das Prädikat erfüllt, falls es so eins gibt. Sie dürfen alle Funktionen aus der Standardbibliothek einsetzen. Um die volle Punktzahl zu erhalten, dürfen Sie keine direkte Rekursion verwenden! Verwenden Sie stattdessen Funktionen höherer Ordnung wie z.B. `foldr :: (a -> b -> b) -> b -> [a] -> b`

Beispiele:

```
> getFirst odd [40,20,10,5,16,8,4,2,1]
Just 5
> getFirst odd [2,4,8,16,32,64]
Nothing
```

LÖSUNG: Einfache Lösung mit direkter Rekursion:

```
getFirst :: (a -> Bool) -> [a] -> Maybe a
getFirst p [] = Nothing
getFirst p (h : t)
  | p h = Just h
  | otherwise = getFirst p t
```

Hier eine der alternative Lösungsmöglichkeiten mit `foldr`, siehe neben A5-1/H5-1 auch den Code aus der Fragestunde:

```
getFirst :: (a -> Bool) -> [a] -> Maybe a
getFirst p = foldr helper Nothing
  where helper x acc
          | p x       = Just x
          | otherwise = acc
```

c) Gegeben sind weiterhin folgende Deklarationen:

```
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : iterate f (f x)
cs = iterate (\n -> 3*n+1) 1
hasEvenCs = myAny even cs
```

(i) Gegeben Sie die ersten 3 Elemente von `cs` an: `[1,4,13,..])`

(ii) Die Liste `cs` hat keine endliche Länge. Betrachten Sie noch mal Ihre Lösung `myAny` aus Aufgabenteil a: Ja oder Nein: Wertet `hasEvenCs` mit Ihrem Code zu einem Wert aus oder nicht? Begründen Sie in jedem Fall Ihre Antwort durch 1–2 Sätze:

LÖSUNG: *Die Lösung hängt natürlich vom angegebenen Code ab; in unserem Fall gilt:* Ja! `hasEvenCs` wertet zu `True` aus. Aufgrund der verzögerten Auswertung wird das zweite Argument von `||` nicht mehr ausgwertet, wenn das erste bereits `True` ergibt. Damit findet kein rekursiver Aufruf mehr statt.

Wer dagegen so etwas hier zur a) abgab:

```
myAny2 :: (a -> Bool) -> [a] -> Bool
myAny2 p []    = False
myAny2 p (h:t) = myAny p t || p h

myAny3 :: (a -> Bool) -> [a] -> Bool
myAny3 p []    = False
myAny3 p (h:t) | myAny p t = True
               | otherwise = p h
```

...muss schreiben: Nein! Der Aufruf von `myAny` wertet die Liste zuerst bis zum Ende aus, bevor ein Ergebnis berechnet wird. Bei einer List ohne Ende terminiert dies nicht.

Lösung Aufgabe 6 (Semantik): **(6 Punkte)**

a) Geben Sie das Symbol an, mit dem in der denotationellen Semantik eine nicht-terminierende Berechnung beschrieben wird:

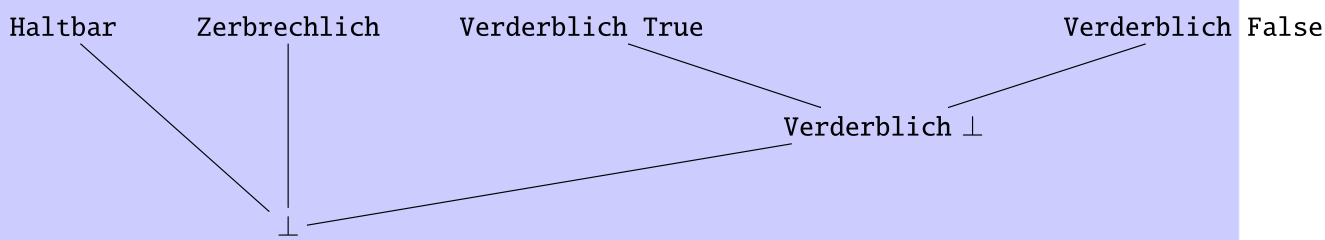
$\perp$

b) Zeichnen Sie ein Hasse Diagramm für die vollständige Halbordnung (dcpo), welche den nachfolgenden Datentyp im Sinn der denotationellen Semantik modelliert:

```
data Produktstatus = Haltbar | Zerbrechlich | Verderblich Bool
```

Hasse-Diagramm:

LÖSUNG: Wir zeichnen entsprechend:



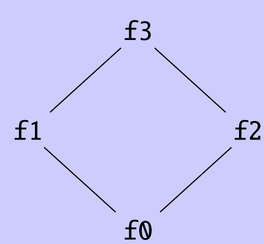
Analog zur A12-1. Erstaunlicherweise haben hier alle das $\perp$ Symbol eingezeichnet – insbesondere auch diejenigen, welche die erste Teilaufgabe nicht lösen konnten?!?

c) Betrachten Sie folgende Funktionsdefinitionen:

```
f0 x = undefined
f1 x = if even x then 2*x else undefined
f2 x = if odd  x then x+x else undefined
f3 x = x+x
```

Betrachten Sie die partielle Ordnung $\big(\{\texttt{f0}, \texttt{f1}, \texttt{f2}, \texttt{f3}\}, \sqsubseteq\big)$. Es gilt z.B. $f0 \sqsubseteq f3$. Was gilt noch? Beantworten Sie die Frage durch Zeichnen des Hasse-Diagramms zu dieser partiellen Ordnung:

LÖSUNG: Hier ist Verständnis ähnlich wie in A12-2 gefragt:



d) Betrachten Sie folgende Definitionen partieller Funktionen:

```
f0 n = undefined
f1 n = if n<0 then 27 else f0 (n-1)
f2 n = if n<0 then 27 else f1 (n-1)
f3 n = if n<0 then 27 else f2 (n-1)
f4 n = if n<0 then 27 else f3 (n-1)
-- usw.
```

Geben Sie Haskell-Code für das Supremum $\sup\big(\texttt{f0} \sqsubseteq \texttt{f1} \sqsubseteq \texttt{f2} \sqsubseteq \texttt{f3} \sqsubseteq \cdots\big)$ an:

`supremum_fn n = 27` oder auch `const 27` siehe auch A12-3 & A12-2

Lösung Aufgabe 7 (Par-Monade): **(5 Punkte)**

LÖSUNG: *Interessanterweise haben sich mit dieser Aufgabe einige über die Bestehensgrenze gerettet, welche sonst wenig Punkte erzielt haben! Viele andere dagegen haben diese Aufgabe gar nicht erst versucht (und sogar vorzeitig abgegeben), obwohl man doch auch ohne Kenntnis der Par-Monade mit Wissen über die DO-Notation und der gegebenen Typsignaturen hier schon viel beantworten kann!?!*

Gegeben ist folgendes Programm:

```
import Control.Monad.Par

foopar = runPar $ do
  let x = "I"
  [s1,s2,s3,s4,s5,s6] <- mapM (\_ -> new) [1..6]
  fork $ put s1 $ worker1 x
  fork $ put s2 $ worker2 x
  r1 <- get s1
  r2 <- get s2
  fork $ put s3 $ worker3 x
  r0 <- get s1
  r7 <- get s2
  fork $ put s4 $ worker4 x
  r4 <- get s4
  r3 <- get s3
  fork $ put s5 $ worker5 r0 r3
  r5 <- get s5
  fork $ put s6 $ worker6 r2 r4
  r6 <- get s6
  return $ (r5,r6)

worker1 x   = "1"++x
worker2 x   = "2"++x
worker3 x   = x++"3"
worker4 x   = x++"4"
worker5 x y = x++"5"++y
worker6 x y = x++"6"++y
```

LÖSUNG: **c)** Es reicht, ein paar Zeilen zu vertauschen, so dass `[[1,2,3,4],[5,6]]` `worker`1–4 parallel und danach die davon abhängigen `worker`5–6 parallel ausgeführt werden.

```
import Control.Monad.Par

foopar = runPar $ do
  let x = "I"
  [s1,s2,s3,s4,s5,s6] <- mapM (\_ -> new) [1
  fork $ put s1 $ worker1 x
  fork $ put s2 $ worker2 x
  fork $ put s3 $ worker3 x
  fork $ put s4 $ worker4 x
  r0 <- get s1   -- r0=="1I"
  r1 <- get s1   -- r1=="1I"
  r3 <- get s3   -- r3=="I3"
  fork $ put s5 $ worker5 r0 r3 --> auch 3
  r2 <- get s2   -- r2=="2I"    --  Zeilen
  r7 <- get s2   -- r7=="2I"    --  später
  r4 <- get s4   -- r4=="I4"    --< okay
  fork $ put s6 $ worker6 r2 r4
  r5 <- get s5   -- r5=="1I5I3"
  r6 <- get s6   -- r6=="2I6I4"
  return $ (r5,r6)
```

Zur Erinnerung:

```
runPar :: Par a -> a
new    :: Par (IVar a)
put    :: NFData a => IVar a -> a -> Par ()
fork   :: Par () -> Par ()
get    :: IVar a -> Par a
```

a) Zu welchem Wert wertet `foopar` aus? (“1I5I3“,“2I6I4“)

b) Angenommen die `worker`N-Funktionen wären zeitaufwendige Berechnungen, welche `worker`N-Funktionen werden parallel zueinander berechnet?

LÖSUNG: `[[1,2],[3,4],[5],[6]]`: Zuerst werden `worker1` und `worker2` parallel ausgeführt, dann `worker3` und `worker4` parallel; danach `worker5` und erst danach `worker6` alleine.

Hier muss man also noch wissen, dass `fork` einen parallelen Thread startet und das `get` die Ausführung anhält und auf das Ergebnis eines parallelen Threads wartet. Da zum Beispiel `get s5` auf das Ergebnis von `worker5` wartet, wird `worker6` also erst ganz zum Schluß alleine gestartet.

c) Beschleunigen Sie das oben angegebene Programm, so dass möglichst viele `worker`#-Funktionen parallel ausgeführt werden. Schreiben Sie Ihre Änderungen einfach oben rechts daneben.
Hinweis: Natürlich soll das Programm nach wie vor alle `worker`#-Funktionen aufrufen und ansonsten das gleiche Ergebnis liefern.

Lösung Aufgabe 8 (Typen): **(8 Punkte)**

LÖSUNG: Die Aufgabe sollte niemanden überrascht haben (Blatt 8, A7-3, A13-8, etc.).

`undefined` stellte hier einige vor ein Problem, doch eigentlich konnte man aus jeglichen Dateivorlagen und Folien her wissen, dass `undefined` jeden gewünschten Typ annehmen kann; dann folgt der Rest sofort aus der Typsignatur des Listenkonstruktors: `(:) :: a -> [a] -> [a]`

a) Geben Sie jeweils den allgemeinsten Typ des gegebenen Haskell-Ausdrucks an, inklusive etwaiger Typklassen Einschränkungen. Bitte nur das Ergebnis hinschreiben. Nebenrechnungen bitte deutlich abtrennen.

(i) `(head "tail", 3<2, (Just False))`

`(Char,Bool,Maybe Bool)`

(ii) `(():():undefined)`

`[()]`

(iii) `let foo x y z = if x>z then show y else [] in foo`

`(Ord a, Show b) => a -> b -> a -> String`

b) Geben Sie den allgemeinsten Unifikator für folgende Typgleichungen an:

$\{\alpha \to \texttt{Int} = \beta \to \gamma, \beta = \gamma \to \delta\}$ $[(\texttt{Int} \to \delta)/\alpha,\ (\texttt{Int} \to \delta)/\beta, \texttt{Int}/\gamma]$

c) Beweisen Sie folgendes Typurteil unter Verwendung der zur Erinnerung auf Seite 11 angegeben Typregeln in einer der beiden in der Vorlesung behandelten Notationen (Herleitungsbaum oder lineare Schreibweise).
Hinweis: Wenn Sie die Herleitung auf der Rückseite dieses Blattes zeichnen, dann brauchen Sie nicht ständig umblättern, um die Typregeln dabei zu sehen!

$\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ \backslash z\ \texttt{->}\ 4\ \texttt{else}\ (\backslash y\ \texttt{->}\ f)\ x :: \texttt{Char} \to \texttt{Int}$ wobei $\Gamma = \{x{::}\texttt{Bool}, \texttt{f}{::}\texttt{Char} \to \texttt{Int}\}$

LÖSUNG:

$$\dfrac{\dfrac{}{\Gamma \vdash x{::}\texttt{Bool}}(\textsc{Var}) \quad \dfrac{\dfrac{}{\Gamma, z{::}\texttt{Char} \vdash 4{::}\texttt{Int}}(\textsc{Int})}{\Gamma \vdash \backslash z\ \texttt{->}\ 4 :: \texttt{Char} \to \texttt{Int}}(\textsc{Abs}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma, y{::}\texttt{Bool} \vdash f{::}\texttt{Char} \to \texttt{Int}}(\textsc{Var})}{\Gamma \vdash \backslash y\ \texttt{->}\ f{::}\texttt{Bool} \to (\texttt{Char} \to \texttt{Int})}(\textsc{Abs}) \quad \dfrac{}{\Gamma \vdash x{::}\texttt{Bool}}(\textsc{Var})}{\Gamma \vdash (\backslash y\ \texttt{->}\ f)\ x :: \texttt{Char} \to \texttt{Int}}(\textsc{App})}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ \backslash z\ \texttt{->}\ 4\ \texttt{else}\ (\backslash y\ \texttt{->}\ f)\ x :: \texttt{Char} \to \texttt{Int}}(\textsc{Cond})$$

Typregeln:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)} \quad (\textsc{Var})$$

Alternative Schreibweise für Var:

$$\frac{x{::}A \in \Gamma}{\Gamma \vdash x :: A} \quad (\textsc{Var})$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \texttt{Int}} \quad (\textsc{Int})$$

$$\frac{c \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash c :: \texttt{Bool}} \quad (\textsc{Bool})$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad (\textsc{App})$$

$$\frac{\Gamma, x{::}A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad (\textsc{Abs})$$

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A} \quad (\textsc{Cond})$$