FP 2009-2010, Tussentoets

Dec 4, 9.00-12.00

The exam consists of 6 multiple choice questions (1 point each) and two open questions (2 points each). Put the right answer to the multiple choice questions in the corresponding box on the answer sheet.

1. In the context of the functions

$$\begin{array}{lll} \mathit{until}\ p\ f\ a & \mid p\ a & = a \\ & \mid \mathit{True} & = \mathit{until}\ p\ f\ (f\ a) \\ \mathit{iterate}\ f\ x & & = x : \mathit{iterate}\ f\ (f\ x) \end{array}$$

   which of the following holds?

   (a) $\mathit{until}\ p\ f.\mathit{until}\ p\ f$ can safely be replaced by $\mathit{until}\ p\ f$
   (b) $\mathit{iterate}\ g.\mathit{iterate}\ f$ can safely be replaced by $\mathit{iterate}\ (g.f)$
   (c) $\mathit{foldl}\ (-)\ 0$ can safely be replaced by $\mathit{foldr}\ (-)\ 0$
   (d) $\mathit{until}\ p\ f.\mathit{until}\ q\ f$ can safely be replaced by $\mathit{until}\ q\ f$ if $p \Rightarrow q$ holds

   Het goede antwoord is (a). Het antwoord (b) is op grond van de types al incorrect. Immers de expressie *iterate* f levert een lijst op, dus de eerste g neemt een lijst als argument. In de tweede expressie neemt g het resultaat van f, dus een element van zo'n lijst. (c) is onjuist omdat de operator $(-)$ niet commutatief is. Had er i.p.v. $(-)$ een $(+)$ gestaan dan was e.e.a wel goed geweest. Voor het laatste geval neem voor p bijvoorbeeld >5 en voor q het predicaat >2. Dan levert de expressie $\mathit{until}\ (>5)\ (+1).\mathit{until}\ (>2)\ (+1)\$\ 0$ de waarde 6 op en de expressie $\mathit{until}\ (>2)\ (+1)\ \$\ 0$ de waarde 3.

2. What is the result of $\mathit{foldr}\ ((+).(2*))\ 3\ [3,3]$?

   (a) 9
   (b) 15
   (c) 27
   (d) 30

   We schrijven de expressie iets anders om te zien wat er precies gebeurt:

$$\begin{array}{ll} \mathit{foldr}\ ((+).(2*)) & 3\ [3,3] \\ \mathit{foldr}\ (\lambda x\ r \rightarrow ((+).(2*))\ x\ r) & 3\ [3,3] \\ \mathit{foldr}\ (\lambda x\ r \rightarrow (+)\ (2*x)\ \ r) & 3\ [3,3] \\ \mathit{foldr}\ (\lambda x\ r \rightarrow (2*x) +\ \ r) & 3\ [3,3] \end{array}$$

   De operator vermenigvuldigt dus kennelijk elk element uit de lijst en telt deze waarden allemaal bij de beginwaarde 3 op. Het resultaat is dus 15

3. What is the type of *map map*?

   (a) $[[a \to b]] \to [[a] \to [b]]$
   (b) $[a \to b] \to [[a] \to [b]]$
   (c) $[a \to b] \to [[[a]] \to [b]]$
   (d) the expression *map map* is not well-typed

   types van de functie en het argument op:

$$map :: (a \to b) \to [a] \to [b]$$
$$map :: (c \to d) \to [c] \to [d].$$

Let op: om verwarring te voorkomen kiezen we verschillende typevariabelen voor de functie (a en b) en het argument (c en d).

De functie *map* verwacht twee argumenten, maar wordt hier slechts toegepast op een enkel argument; het resultaat is een functie die nog een lijstargument verwacht:

$$map :: \underbrace{(a \to b)}_{\text{argumenttype}} \to \underbrace{[a] \to [b]}_{\text{resultaattype}}.$$

De functie *map* is polymorf in het domein en codomein van haar functieargument (vandaar de typevariabelen a en b in het argumenttype) en wordt hier toegepast op een argument van type $(c \to d) \to [c] \to [d]$. Aanpassen van het type gaat dus als volgt:

$$\underbrace{a \to b}_{\text{argumenttype van } map} \equiv \underbrace{(c \to d) \to [c] \to [d]}_{\text{type van het argument}}$$

$$\Rightarrow \begin{cases} a \equiv c \to d \\ b \equiv [c] \to [d]. \end{cases}$$

De eerste gelijkheid ($a \equiv c \to d$) maakt dat we alle voorkomens van de typevariabele a in het type van de funtie *map* vervangen door $c \to d$; vanwege de tweede gelijkheid ($b \equiv [c] \to [d]$) vervangen we alle voorkomens van de typevariabele b door $[c] \to [d]$:

$$map :: \underbrace{(\overset{(c \to d)}{a} \to \overset{[c] \to [d]}{b})}_{\text{argumenttype}} \to \underbrace{[\overset{c \to d}{a}] \to [\overset{[c] \to [d]}{b}]}_{\text{resultaattype}}.$$

Deze vervangingen resulteren in

$$map :: \underbrace{((c \to d) \to [c] \to [d])}_{\text{type van het argument}} \to \underbrace{[c \to d] \to [[c] \to [d]]}_{\text{resultaattype}}.$$

Voor *map map* vinden we dus:

$$map\ map :: [c \to d] \to [[c] \to [d]],$$

ofwel

$$map\ map :: [a \to b] \to [[a] \to [b]],$$

4. Which of the following is true?

   (a) Only a value with the name *main* can have a value of type *IO* ()

   (b) Expressions of type *IO a* can only occur in a **do**-construct.

   (c) The function *main* in the module *Main* must have type *IO* ().

   (d) The function *main* cannot call itself recursively.

   De expressie *putChar* `'a'` heeft ook type *IO* (). Expressie van type *IO* () duiden gewone waarden aan en kunnen dus overal voor komen. Hun effect komt pas tot uiting als de waade op een of andere manier onderdeel wordt van de waarde van type *IO* () die aan de waarde *main* gebonden wordt. Dat mag zelfs meerdere keren zijn. Het antwoord (c) is het goede antwoord. Het volgende is een perfect Haskell rpogramma:

   $$\begin{aligned} main = \mathbf{do}\ & c \leftarrow getChar \\ & \mathbf{if}\ c \equiv \texttt{'c'}\ \mathbf{then}\ main\ \mathbf{else}\ return\ () \end{aligned}$$

5. Someone who wants to program without list comprehensions, translates the right-hand side of the following definition into comprehension-free code. Which of the alternatives is the correct one?

   $$segs\ xs = [\,] : [\,t \mid i \leftarrow inits\ xs, t \leftarrow tails\ i, not\ (null\ t)]$$

   (a) $$\begin{aligned} & [\,] : concat\ (map\ f\ (inits\ xs)) \\ & \quad \mathbf{where}\ f\ i = concat\ (map\ g\ (tails\ i)) \\ & \qquad\qquad \mathbf{where}\ g\ t = \mathbf{if}\ not\ (null\ t)\ \mathbf{then}\ [\,]\ \mathbf{else}\ [t] \end{aligned}$$

   (b) $$\begin{aligned} & [\,] : concat\ (map\ f\ (inits\ xs)) \\ & \quad \mathbf{where}\ f\ i = concat\ (map\ g\ (tails\ i)) \\ & \qquad\qquad \mathbf{where}\ g\ t = \mathbf{if}\ not\ (null\ t)\ \mathbf{then}\ [t]\ \mathbf{else}\ [\,] \end{aligned}$$

   (c) $$\begin{aligned} & [\,] : concat\ (map\ f.map\ g)\ (inits\ xs) \\ & \quad \mathbf{where}\ f\ i = tails\ i \\ & \qquad\qquad g\ t = \mathbf{if}\ null\ t\ \mathbf{then}\ [\,]\ \mathbf{else}\ [t] \end{aligned}$$

   (d) $$\begin{aligned} & [\,] : filter\ (not.null).concat\ (map\ f.map\ g)\ (inits\ xs) \\ & \quad \mathbf{where}\ f\ i = tails\ i \\ & \qquad\qquad g\ t = [t] \end{aligned}$$

   (b), zie slides van college

6. The function $intersperse :: a \rightarrow [a] \rightarrow [a]$ puts its first argument between all the elements of a non-empty list. Thus *intersperse* `'a'` `"xyz"` results in `"xayaz"`. Which definition is correct?

   (a) $intersperse\ a = \lambda as \rightarrow tail\ [(a : e) \mid e \leftarrow as]$

   (b) $intersperse\ a = foldr\ (\lambda e\ r \rightarrow (e : a : r))\ [\,]$

   (c) $intersperse\ a = tail.concat.map\ (\lambda x \rightarrow [a, x])$

   (d) $intersperse\ a = foldl\ (\lambda r\ e \rightarrow (a : e : r))\ [\,]$

   (c); probeer zelf de andere "oplossingen"

7. Write a function $reverseOdds :: [Int] \rightarrow [Int]$, with the properties that:

- the even numbers keep their position
- the sublist formed by the odd numbers is in reverse order

Thus *reverseOdds* $[1, 2, 3, 4, 6, 7, 8, 9]$ results in $[9, 2, 7, 4, 6, 3, 8, 1]$.

De eenvoudigste manier is om de oneven elementen uit de lijst te filteren en dan de lijst te copiëren terwijl je de oneven elementen vervangt:

```
reversOdds l = replaceOdds l (reverse (filter odd l))
  where replaceOdds l [] = l
        replaceOdds (x : xs) (o : os)
                     | even x = x : replaceOdds' xs (o : os)
                     | odd x  = o : replaceOdds' xs os
        replaceOdds [] odds = odds
```

Let op: het eerste alternatief voor *replaceOdds'* is echt nodig, want het tweede alternatief slaagt alleen als de lijst van oneven getallen niet leeg is, zelfs als neem de branch bij de guard *even x*. Later zullen we zien dat we door $\sim(o : os)$ te schrijven deze pattern matching kunnen uitstellen totdat echt een van de componenten voor de verdere berekening nodig is.

Een nadeel van deze oplossing is dat de test op even/oneven zijn voor elk element twee keer wordt uitgevoerd. Nu is deze test hier niet zo duur, maar je kan afvragen of dit echt nodig is. We kunnen ook de berekeningen in elkaar schuiven. We berekenen op de heenweg, dus terwijl we de oorsponkelijk lijst recursief afbreken in een accumulerende parameter de lijst van oneven elementen, aan het eind keren we die om, en leveren de resultaatlijst en de nog te gebuiken elementen op. Aan het eind is de lijst van onvene elementen leeg en nemen we de eerste component van het resultaat.

```
reversOdds2 l = fst (replaceOdds l [])
  where replaceOdds (x : xs) odds
              | even x = let (r, os) = replaceOdds xs odds
                         in  (x : r, os)
              | odd x  = let (r, o : os) = replaceOdds xs (x : odds)
                         in  (o : r, os)
        replaceOdds [] odds = ([], reverse odds)
```

We hebben dus de lijst met oneven elementen in de volgorde opgebouwd is die omgekeerd is aan waarin ze in de lijst staan; omdat de lijst ook weer omgekeerd wordt afgebroekn moeten we hem aan het eind omdraaien. Ook dat kan voorkomen worden door deze lijst eerst helemaal als resultaat op te leveren en vervolgens te gebruiken. We schijven eerst de functie *filterreverse* die de gefilterde lijst in omgekeerde volgorde oplevert:

```
filterreverse p [] r              = r
filterreverse p (x : xs) r | p x = filterreverse p xs (x : r)
   | True = filterreverse p xs r

reverseOdds3 l = let (result, reverseodds) = filterreversereplace l reverseodds []
  filterreversereplace []        _ acc = ([], acc)
  filterreversereplace (x : xs) rodds acc
     | even x = let (r, ror) = filterreversereplace xs rodds acc
                in (x : r, ror)
```

$$
\begin{aligned}
&\mid \mathit{odd}\ x \quad = \mathbf{let}\ (r, \mathit{ror}) = \mathit{filterreversereplace}\ \mathit{xs}\ (\mathit{tail}\ \mathit{rodds})\ (x : \mathit{acc}) \\
&\qquad\qquad\quad\ \ \mathbf{in}\ (\mathit{head}\ \mathit{rodds} : r, \mathit{ror}) \\
&\mathbf{in}\ \mathit{result}
\end{aligned}
$$

8. Write a program which defines the infinite list of prime numbers, using the sieve of Erathostenes.

$$\mathit{primes} = \mathit{zeef}\ [2\,..] \ \mathbf{where}\ \mathit{zeef}\ (x : \mathit{xs}) = x : \mathit{zeef}\ (\mathit{filter}\ ((\not\equiv 0).(`\mathit{mod}`x))\ \mathit{xs})$$

Wednesday, 19.00: Haskell User Group Meeting, `http://haskell.org/haskellwiki/Dutch_HUG`