totaal: 4 pagina's

Tentamen Functioneel Programmeren

7 november 2003, 9.00–12.00 uur

Schrijf met blauwe of zwarte pen; *niet* met potlood en *niet* met rode pen. Voorzie alle bladen van je naam. Nummer de bladen en vermeld op het eerste blad het totale aantal.
Houd je Haskell-programma's zo kort en helder mogelijk, mede door, waar dat handig is, gebruik te maken van standaardfuncties uit het cursusboek, in het bijzonder uit het gedeelte over lijsten. Geef als je Haskell-programma (toch) wat ingewikkelder is, een heldere toelichting om de correctheid ervan voldoende aannemelijk te maken.

Opgave 1.

(i) Bepaal de (eventueel polymorfe) typen van de volgende vijf expressies. Geef telkens een beknopte argumentatie.

```
map ord
map filter
foldr (++) []
map . map
filter even [ x*x | x <- [2,4,7]]
```

Hierbij is gegeven, dat `ord` en `even` respectievelijk de typen `Char -> Int` en `Int -> Bool` hebben.

(ii) Definiëer in Haskell de functies `compl` en `compr` door:

```
compl f g h = (f . g) . h
compr f g h = f . (g . h)
```

Bepaal de typen van `compl` en `compr` en toon aan dat `compl` = `compr` .

Opgave 2. Een *bag* is een verzameling waarin elk element een positief natuurlijk getal als multipliciteit heeft. In deze opgave representeren we bags als lijsten van verschillende paren, waarvan de eerste component een object is uit de bag en de tweede component de multipliciteit ervan. (Hierbij wordt geëist dat verschillende paren een verschillende eerste component hebben.) Voorbeeld: `[('a',4),('b',2),('g',1)]` en `[('b',2),('a',4),(g',1)]` zijn representaties van de bag bestaande uit `'a'`, `'b'`, `'g'`, waarbij de respectievelijke multipliciteiten zijn: 4, 2, 1.

(i) Geef een Haskell-definitie van een functie

```
elements ::  Eq a => [(a,Int)] -> [a]
```

die een representatie van een bag omzet in een lijst van alle verschillende objecten uit die bag.

(ii) Geef een Haskell-definitie van een functie

```
multip ::  Eq a => a -> [(a,Int)] -> Int
```

met de volgende eigenschap: als `b` een representatie is van een eindige bag en `x` een object van het passende type, dan is `multip x b` de multipliciteit van `x` in `b`. Als `x` niet voorkomt in `b`, dan is deze multipliciteit per definitie gelijk aan 0.

Voorbeelden: `multip 'a' [('a',4),('b',2),('g',1)]` is gelijk aan 4 en
`multip 'c' [('a',4),('b',2),('g',1)]` is gelijk aan 0.

(iii) Geef een Haskell-definitie van een functie

```
addmultip ::  Eq a => a -> Int -> [(a,Int)] -> [(a,Int)]
```

met de volgende eigenschap: als `b` een eindige bag representeert en `x` een object is (van het passende type) en `n` een positief natuurlijk getal is, dan is `addmultip x n b` een representatie van de bag die uit `b` verkregen wordt door verhoging van de multipliciteit van `x` met `n`. (Speciaal randgeval: `x` komt nog niet voor in de bag; dan wordt `x` toegevoegd aan de bag, en wel met multipliciteit `n`.)

Opgave 3.

(i) Geef een Haskell-definitie van een functie

```
voegtoe ::  a -> [a] -> [[a]]
```

die voor een object `x` en een eindige lijst `xs` van objecten van hetzelfde type als `x` oplevert: een lijst van alle lijsten die uit `xs` te verkrijgen zijn door `x` ergens toe te voegen aan `xs`.
Voorbeeld: `voegtoe 3 [2,4]` is de lijst met als verschillende elementen (al dan niet in deze volgorde): `[3,2,4]`, `[2,3,4]` en `[2,4,3]`.

Zet de definitie van `voegtoe` recursief op door gelijkheden te geven voor `voegtoe x[]` en voor `voegtoe x (y:ys)`, maar vermijd — via handig gebruik van standaardfunctie(s) — verdere (uiterlijke) recursie. (D.w.z.: verdere recursie mag hooguit "achter de schermen" plaatsvinden, en wel in definities van de standaardfunctie(s).)

(ii) Geef met behulp van `voegtoe` een Haskell-definitie van een functie

```
perms ::  [a] -> [[a]]
```

die voor elke lijst oplevert: een lijst van alle permutaties van die lijst.

Zet de definitie van `perms` recursief door gelijkheden te geven voor `perms []` en voor `perms (x:xs)`, maar vermijd — via gebruik van `voegtoe` en wederom via handig gebruik van standaardfunctie(s) — verdere (uiterlijke) recursie (naast deze uiterlijke recursie en de uiterlijke recursie die er al was voor `voegtoe`).

Merk bij de constructie van de gelijkheid voor `perms (x:xs)` op dat de permutaties van `(x:xs)` precies die lijsten zijn die te verkrijgen zijn door een permutatie van `xs` te nemen en daar ergens `x` aan toe te voegen.

Opgave 4. Bewijs inductief dat voor alle eindige lijsten `xs` geldt (onder passende aannamen over de typen van `p` en `q`):

```
filter p (filter q xs) = filter (\ x -> (p x && q x)) xs
```

Opgave 5. (Aangaande **FEM**.) Zij Γ een willekeurige context. Onder *herschrijfrijen* verstaan we in deze opgave steeds: *herschrijfrijen ten aanzien van* Γ.

(i) Zij A een FEM-term en zij n een natuurlijk getal. Beschouw het FEM-programma (Γ, A). Wanneer zeggen we dat dit programma *waarde n heeft*?

(ii) Uit de theorie is het volgende bekend:

Als A, B_1, B_2 FEM-termen zijn met de eigenschap dat er vanuit A herschrijfrijen zijn naar B_1 en B_2, dan bestaat er een FEM-term C met de eigenschap dat er vanuit B_1 en B_2 herschrijfrijen zijn naar C.

Leid hieruit af dat elk FEM-programma hooguit één getal als waarde heeft.

(iii) Zij Γ nu in het bijzonder de volgende context:

```
( u x y z = x z ( y z), k x y = x, i x = x)
```

Zij A de FEM-term `(u k i 5)`.

(a) Laat zien dat het FEM-programma (Γ, A) een getalwaarde heeft. Geef die getalwaarde expliciet.

(b) Laat tevens zien dat het FEM-programma (Γ, A) typeerbaar is (in de zin van het FEM-dictaat).

Opgave 6. Definiëer als volgt een verzameling Ster van *simpele termen*; dit zijn speciale ASCII-strings (zonder spaties en newline-characters).

```
<ster> ::= 0 | x | y | (s<ster>) | (p<ster>) | (t<ster><ster><ster>)
```

(Eigenlijk moeten hierin als volgt aanhalingstekens toegevoegd worden:

```
'o', 'x', 'y', '(', ')', 's', 'p', 't' .
```

Maar zo, als boven, is het leesbaarder.)

Simpele termen vatten we op als speciale FEM-termen, en wel als volgt. Voeg (ter vermijding van syntactische dubbelzinnigheden) binnen simpele termen spaties toe na `s`, `p` en `t`, alsmede rond `0`, `x` en `y`. (We gaan hier uit van de conventies uit het FEM-dictaat betreffende de weergave van FEM-termen als ASCII-strings, inclusief de conventie m.b.t. het eventueel weglaten van haakjes. Na passende toevoeging van haakjes kunnen bepaalde spaties eventueel weer weggelaten worden.)

Voorbeeld 1. De simpele term `(tx0y)` identificeren we uiteindelijk met de FEM-term `(((tx)0)y)`.

Observatie 1. Als we natuurlijke getallen m en n toekennen aan de variabelen `x` en `y`, dan heeft elke simpele term A een specifiek natuurlijk getal als waarde. Lees A namelijk als FEM-term en beschouw het FEM-programma $(\Gamma_{m,n}, A)$, waarin $\Gamma_{m,n}$ bestaat uit parameterloze declaraties van `x` en `y` met als respectievelijke rechterleden: de bij m en n behorende constanten $\overline{m}$ en $\overline{n}$.

Voorbeeld 2. Als we de waarden 2 en 3 toekennen aan respectievelijk `x` en `y`, dan heeft de simpele term `(tx0(sy))` waarde 4.

Observatie 2. Zodra waarden toegekend zijn aan `x` en `y`, is de corresponderende waarde van een simpele termen A rechttoe rechtaan met recursie naar de opbouw van A te definiëren.

In de rest van deze opgave gaat het om de vraag hoe in Haskell handig definities op te zetten zijn van functies
`testSter :: String -> Bool` en `waardeSter :: String -> Int -> Int -> Int`
met de volgende eigenschappen: `testSter` test of een string een simpele term is en `waardeSter` bepaalt de waarde van een simpele term, afhankelijk van gegeven waarden (binnen $\mathbb{N}$) van de variabelen `x`, respectievelijk `y`.

(i) Geef een voor dit doel geschikte definitie van een recursief algebraïsch type `TypSter`, corresponderend met Ster.

Geef tevenshet corresponderende type van een parser `parserSter` voor Ster.

(De Haskell-definitie van `parserSter` wordt hier niet gevraagd.)

(ii) In een uiteindelijke definitie van `waardeSter` zal gebruik gemaakt worden van een passend gedefiniëerde hulpfunctie

```
eval ::  TypSter -> Int -> Int -> Int
```

Als `b` van type `TypSter` een term A representeert, dan moet `(eval b)` een implementatie zijn van de functie zijn die de waarde van A oplevert afhankelijk van gegeven waarden van de variabelen `x` en `y`.

Geef een een expliciete (recursieve) Haskell-defintie van `eval`. (Het gaat hier om een implementatie van de in in "Observatie 2" bedoelde recursie.)

Slotopmerking. `testSter` en `waardeSter` zijn te definiëren met behulp van `parserSter` en `eval`, maar verdere details worden hier niet gevraagd.