Gequoteerde zitting Haskell: Symbolic Differentiation

NAAM: RICHTING:

Enkele praktische afspraken

- Je krijgt twee uur om deze opdracht **individueel** op te lossen.
- Je raadpleegt enkel de Haskell slides (eventueel in afgedrukte vorm met handgeschreven nota's) en oefening die via Toledo voor dit vak ter beschikking gesteld zijn. Je mag ook de manuals vermeld op Toledo en eventueel Hoogle raadplegen.
- In de map **1516_Gequoteerde/Haskell_Donderdag** op Toledo vind je de bestanden `SymbolicDifferentiation.hs` en `SymbolicDifferentiationTest.hs`. Ook de indienmodule staat daar.
  - **Download en open** het bestand `SymbolicDifferentiation.hs`, hierin staat reeds een template voor je oplossing.
  - Als eerste vul je bovenaan je naam, studentennummer en richting in.

    ```
    -- Jan Jansen
    -- r0123456
    -- master cw
    ```

  - Bovenaan in het bestand worden reeds een aantal functies geïmporteerd die je waarschijnlijk nodig zal hebben. Zoek hun werking op indien je ze nog niet kent. Je mag ook **extra functies en types importeren**!
  - Voor elke opdracht zijn een aantal functies reeds gedefinieerd met een bijbehorende typesignatuur. Deze typesignaturen mogen niet gewijzigd worden. Vervang telkens `undefined` met jouw implementatie. Je mag argumenten voor het gelijkheidsteken toevoegen of verwijderen. Het is natuurlijk ook altijd toegestaan om extra (hulp)functies te schrijven.
  - Je kan je oplossing testen m.b.v. `SymbolicDifferentiationTest.hs`. Dit doe je door in de map waarin de twee `.hs` bestanden staan het volgende commando uit te voeren:

    ```
    runhaskell SuggestionsTest.hs
    ```

    N.B. dat alle testen slagen betekent niet per se dat je programma helemaal correct is of dat je het maximum van de punten verdient.
  - Na twee uur of wanneer je klaar bent, dien je het bestand `SymbolicDifferentiation.hs` in via Toledo.

Symbolic Differentiation

Voor veel functies bestaan er eenvoudige regels waarmee de afgeleide van die functie berekend kan worden. Deze regels gebruik je dan ook impliciet wanneer je met de hand afgeleides bepaalt. Bijvoorbeeld, de afgeleide van de veelterm $x^2 + x + 1$ berekenen we als volgt:

$$\begin{aligned}
\frac{d(x^2+x+1)}{dx} &= \frac{d(x^2)}{dx} + \frac{dx}{dx} + \frac{d1}{dx} && \text{(de afgeleide van een som is de som van de afgeleides)} \\
&= \frac{d(x^2)}{dx} + 1 + 0 && \text{(de afgeleide van } x \text{ is 1, de afgeleide van 1 is 0)} \\
&= 2x + 1 + 0 && \text{(de afgeleide van } x^n \text{ is } nx^{n-1}\text{)} \\
&= 2x + 1
\end{aligned}$$

In deze opgave laten we *de computer* op dezelfde manier afgeleides berekenen.

Deel 1: Data representatie

We beperken ons tot reële functies die bestaan uit constanten, een variabele (die meerdere keren kan voorkomen), optelling, vermenigvuldiging, machtsverheffing met een constante exponent en de natuurlijke logaritme $\ln(\cdot)$.

Deze functies worden voorgesteld door het datatype `Function`:

```
data Function
  = Const Double
    -- ^ constant
  | X
    -- ^ variable
  | Function :+: Function
    -- ^ addition
  | Function :*: Function
    -- ^ multiplication
  | Function :^: Double
    -- ^ power with constant exponent
  | Ln Function
    -- ^ natural logarithm
  deriving (Show, Eq)
```

Merk op dat deze definitie datatype operator-constructors gebruikt (`:+:`, `:*:`, `:^:`). Op deze constructoren kan je net zoals bij gewone constructoren pattern-matchen:

```
isPlus :: Function -> Bool
isPlus (_ :+: _) = True
isPlus _         = False
```

Operation	Domain	Result given $x = a$	Derivative
`Const c`	$x \in \mathbb{R}$	c	0
`X`	$x \in \mathbb{R}$	a	1
`f :+: g`	$x \in \mathbb{R}$	$f(a) + g(a)$	$\frac{d}{dx}f(x) + \frac{d}{dx}g(x)$
`f :*: g`	$x \in \mathbb{R}$	$f(a)g(a)$	$\left(\frac{d}{dx}f(x)\right)g(x) + f(x)\frac{d}{dx}g(x)$
`f :^: c`	$\begin{cases} x \in \mathbb{R}_0 & \text{if } c \leq 0 \\ x \in \mathbb{R} & \text{if } c > 0 \end{cases}$	$(f(a))^c$	$c(f(x))^{c-1}\frac{d}{dx}f(x)$
`Ln f`	$x \in \mathbb{R}_0^+$	$\ln(f(a))$	$\frac{1}{f(x)} \cdot \frac{d}{dx}f(x)$

Tabel 1: The domains, results, and derivatives of the operations.

De functie `isPlus` geeft `True` als zijn argument een optelling (`:+:`) is, en anders `False`, en doet dit door te pattern matchen op zijn argument.

Opdracht 1 Vervolledig de definitie van de `Num`-instance voor `Function`. Vervang de `undefined`s zodat het mogelijk wordt om volgende expressies te schrijven:

```
ghci> X + 1
X :+: Const 1.0
ghci> (-X) * (Ln 1)
(Const (-1.0) :*: X) :*: Ln (Const 1.0)
```

Opmerking *Als deze opgave niet lukt, laat de* `undefined`*s dan staan, en pas goed op wanneer je de voorbeelden schrijft, want het gebruik van* `+` *of* `*` *in plaats van* `:+:` *of* `:*:` *leidt tot fouten!*

Deel 2: Functie evaluatie

Een waarde `f` van het type `Function` kunnen we evalueren door elke `X` in `f` te vervangen door een gegeven $a \in \mathbb{R}$ en voor de andere operaties het vanzelfsprekende te doen (dus bijvoorbeeld voor `f :+: g` een optelling te gebruiken, zie Tabel 1).

Merk op dat sommige functies niet voor alle reële getallen gedefiniëerd zijn. In het bijzonder is de machtsverheffing (x^r) niet gedefiniëerd als $r \leq 0$ en $x = 0$, en de natuurlijke logaritme $\ln(x)$ is alleen gedefiniëerd voor $x > 0$. Tabel 1 geeft het domein en verwacht resultaat voor iedere operatie (waar we met $\mathbb{R}_0$ het domein $\mathbb{R}\backslash\{0\}$ bedoelen, en met $\mathbb{R}_0^+$ alle strikt positieve reële getallen).

Opdracht 2 Vul de functie `evaluate :: Function -> Double -> Maybe Double` verder aan zodat `evaluate f a` het resultaat geeft in overeenstemming met Tabel 1. Als a buiten het domein valt geef je `Nothing`, anders `Just`. Maak gebruik van de `Maybe`-monad en `do`-notatie. ***Hint:*** *`:info Double` geeft een lijst van alle typeclassen die door* `Double` *worden geïnstantieerd. Gebruik de methoden in deze typeclasses om* `evaluate` *te implementeren.* Enkele voorbeelden:

```
ghci> evaluate X 1
Just 1.0
```

```
ghci> evaluate (1 :+: (2 :*: X):^:2) 1
Just 5.0
ghci> evaluate (Ln (X:^:2 :+: (-4))) 2.0
Nothing
ghci> evaluate (Ln (X :+: 1)) 0
Just 0.0
```

Deel 3: Afgeleiden

Nu is het de bedoeling om de regels voor het berekenen van afgeleiden toe te passen op de `Function`-structuur. De afleidingsregels voor de ondersteunde operaties staan in de laatste kolom van Tabel 1. Let bij de laatste twee regels op voor het gebruik van de kettingregel.

Opdracht 3 Vul de functie `derivative :: Function -> Function` verder aan, zodat `derivative f` een nieuwe `Function` is, die de afgeleide van `f` voorstelt.

Enkele voorbeelden:

```
ghci> derivative (X :+: 1)
Const 1.0 :+: Const 0.0
ghci> derivative (X:^:2 :*: 1)
((Const 2.0 :*: X :^: 1.0) :*: Const 1.0) :*: Const 1.0 :+: X :^: 2.0 :*: Const 0.0
ghci> derivative (Ln (X:^:2))
(X :^: 2.0) :^: (-1.0) :*: ((Const 2.0 :*: X :^: 1.0) :*: Const 1.0)
ghci> evaluate (derivative (Ln (X:^:2)))) 2
Just 1.0
```

Deel 4: Pretty Printing

De `Show`-instance van `Function` geeft uitvoer die weinig aantrekkelijk is. We definiëren daarom een functie `pretty` die veel mooiere strings aflevert.

Opdracht 4 Vul de functie `pretty' :: Int -> Function -> String` verder aan voor `Const c`, `X`, `:+:`, `:*:`, `:^:` en `Ln`. Schrijf deze constructoren als `c`, `x`, `+`, `*`,`^` en `ln`. Zet steeds spaties naast `+` en `*`, maar niet naast `^` of `ln`

De functie `pretty'` heeft het type `Int -> Function -> String`. Het eerste argument van het type `Int` is de prioriteit van de operator waarin het tweede argument (van het type `Function`) voorkomt. De prioriteiten van de verschillende operatoren staan in Tabel 2. Plaats alleen haakjes rond een functie wanneer dit nodig is om de functie correct weer te geven. Met andere woorden, plaats alleen haakjes rond een functie wanneer deze voorkomt als de operand van een operator met hogere prioriteit.[1]

[1] Merk op dat `Ln` de hoogste prioriteit heeft. Hierrond zullen we dus nooit haakjes plaatsten, zodat `pretty (Ln (Ln 1)) == "lnln(1.0)"`

Operator	priority
`Ln`	4
`Const`, `X`	3
`:^:`	2
`:*:`	1
`:+:`	0

Tabel 2: The operators and their priorities.

Bijvoorbeeld, in de volgende uitdrukking plaatsen we haakjes rond de (1 + 2) omdat de functie anders als 1 + (2 * 3) geïnterpreteerd wordt ((1 + 2) is hier een operand van de * operator):

```
ghci> pretty ((1 :+: 2) :*: 3)
"(1.0 + 2.0) * 3.0"
```

Maar we doen dit niet voor de volgende functie:

```
ghci> pretty ((1 :*: 2) :*: 3)
"1.0 * 2.0 * 3.0"
```

Nog een voorbeeld:

```
ghci> pretty ((Ln (1 :+: X:^:2)) :*: X)
"ln(1.0 + x^2.0) * x"
```

Meer specifiek voor `pretty'` is het gewenste gedrag dus:

```
ghci> pretty' 3 (Const 1)
"1.0"
ghci> pretty' 4 (Const 1)
"(1.0)"
```

Deel 5: User Interface

Nu schrijven we een commandline programma `evaluateIO` dat een `Function` als argument krijgt en de volgende stappen uitvoert:

1. Het programma `pretty`-print de functie die het als argument kreeg.
2. Het programma vraagt de gebruiker om een getal in te voeren.
3. Het programma evalueert de functie op dit getal en print het resultaat.
4. Het programma `pretty`-print de afgeleide van de functie.
5. Het programma vraagt opnieuw om een getal in te voeren.
6. Het programma evalueert de afgeleide en print het resultaat.

Het volgende is een voorbeeld van de uitvoer:

```
ghci> evaluateIO (X:^:2)
Please enter a value for x to evaluate the function f(x) on, where:
  f(x) = x^2.0
x = 2                                    <--- "2" is user-input
  f(x) = Just 4.0
The derivative of this function is:
  df/dx = 2.0 * x^1.0 * 1.0
Enter a value for x to evaluate df/dx on:
x = 2                                    <--- "2" is user-input
  df/dx | (x = 2.0) = Just 4.0
```

Opdracht 5 Implementeer het programma `evaluateIO :: Function -> IO ()` zoals hierboven beschreven is. Indien opdracht 4 (`pretty`) nog niet af, is gebruik dan `show`. Indien opdracht 3 (`derivative`) nog niet af is, hergebruik dan de functie in plaats van de afgeleide.