Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting gregor.snelting@kit.edu
Prof. Dr. Ralf Reussner reussner@kit.edu

Programmierparadigmen – WS 2021/22

https://pp.ipd.kit.edu/lehre/WS202122/paradigmen/uebung

Blatt 4: Datentypen, Typklassen

Abgabe: 19.11.2021, 14:00
Besprechung: 22.11. – 23.11.2021

Reichen Sie Ihre Abgabe bis zum 19.11.2021 um 14:00 in unserer Praktomat-Instanz unter https://praktomat.cs.kit.edu/pp_2021_WS ein.

1 Typen und Typklassen in Haskell

Geben Sie für folgende Haskell-Funktionen die allgemeinstmöglichen Typen (falls sie typisierbar sind) einschließlich etwaiger Typklasseneinschränkungen an. Begründen Sie jeweils kurz, warum diese Einschränkungen nötig sind[1].

```
1 fun1 xs = (xs == [])
2 fun2 f a = foldr f "a"
3 fun3 f a xs c = foldl f a xs c
4 fun4 f xs = map f xs xs
5 fun5 a b c = (maximum [a..b], 3 * c)
6 fun6 x y = succ (toEnum (last [fromEnum x..fromEnum y]))
7 fun7 x = if show x /= [] then x else error
```

Beispiellösung:
`fun1 :: (Eq t) => [t] -> Bool`
Listen-Vergleich `xs==[]` ist definiert über den Vergleich `==` vom Elementtyp `t` der Listen. Daher muss `t` in Typklasse `Eq` sein.

`fun2 :: (s -> [Char] -> [Char]) -> t -> [s] -> [Char]`
`foldr` nimmt Operator, Initialwert und Eingabeliste entgegen. Daher ist `foldr f "a"` eine Unterversorgung, `fun2` nimmt damit als drittes Argument die Eingabeliste (Typ `[s]`) entgegen. Der Initialwert ist vom Typ `[Char]`, damit auch der Rückgabewert. Der Operator `f` muss solche Werte mit Listenelementen vom Typ `s` wieder zu einem Wert vom Typ `[Char]` verknüpfen: `(s -> [Char] -> [Char])`.
Das zweite Argument von `fun2` wird nirgends verwendet (`a` $\neq$ `"a"`), daher ist dessen Typ `t` beliebig.
(`String` statt `[Char]` ist natürlich genauso richtig.)
(Seit GHC 7.10 ist `foldr` von Listen auf Instanzen der Typklasse `Foldable` generalisiert worden; damit wäre `fun2 :: Foldable f => (s -> [Char] -> [Char]) -> t -> f s -> [Char]`.)

`fun3 :: ((t -> s) -> b -> (t -> s)) -> (t -> s) -> [b] -> t -> s`
`foldl` wird mit Operator, Initialwert und Eingabeliste (Typ `[b]`) versorgt, dann auf `c` angewendet! Daher: `foldl f a xs` und Initialwert `a` müssen Funktionstyp `t -> s` haben. Dementsprechend muss der Operator solche Funktionen zusammen mit Listenelement wieder zu Funktionen verknüpfen:

[1] Der GHC kann `fun7` nur mit der Spracherweiterung `FlexibleContexts` inferieren.

`((t -> s) -> b -> (t -> s))`. `fun3` nimmt als viertes Argument also ein `t` entgegen und gibt ein `s` zurück.
(Analog zu `fun2` ist seit GHC 7.10 der allgemeinste Typ
`fun3 :: Foldable f => ((t -> s) -> b -> (t -> s)) -> (t -> s) -> f b -> t -> s`.)

`fun4` nicht typisierbar
`map` erwartet Funktion und Liste, gibt Liste zurück, daher muss `(map f xs)` einen Listen-Typ haben. Andererseits wird `(map f xs)` auf `xs` *angewendet*, muss als einen Funktionstyp haben! Damit ist `fun4` nicht typisierbar.

`fun5 :: (Enum s, Ord s, Num t) => s -> s -> t -> (s, t)`
`[a..b]` erfordert, dass `a` und `b` vom gleichem, aufzählbarem Typ `s` sind: `Enum s`.
`maximum` erfordert, dass Listenelemente geordnet sind: `(Ord s)`.
`fun5` gibt ein Tupel zurück, von `c` wird lediglich gefordert, dass für seinen Typ `t` Multiplikation definiert ist: `Num t`.

`fun6 :: (Enum s, Enum t, Enum u) => s -> t -> u`
`toEnum, fromEnum` konvertiert zwischen Werten aufzählbarer Typen und deren Index in der Aufzählung. Daher: `x, y` von möglicherweise unterschiedlichen Typen `s, t` mit `Enum s, Enum t`. `last` angewandt auf eine Liste vom Typ `[Int]` ergibt einen `Int`, damit ist `toEnum ...` und auch `succ (toEnum ...)` von beliebigem aufzählbaren Typ `u` mit `Enum u`.
Bei einem Aufruf, z.B. `fun6 3 71`, wird Ergebnistyp (und damit: Ergebniswert) durch Typkontext bestimmt.

Beispiel:

```
(fun6 3 71):"ello"  ⇒*  Hello
(fun6 3 71)+0       ⇒*  72.
```

`fun7 :: (Show ([Char] -> a)) => ([Char] -> a) -> ([Char] -> a)`
`error` (im **`else`**) hat Typ `[Char] -> a`, damit ist der Typ von `x` und des gesamten **`if`**–**`then`**–**`else`**-Ausdrucks von dieser Gestalt. Aufruf `show x` erzwingt `Show ([Char] -> a)`.

2 Abstrakte Syntaxbäume

Geben Sie Ihre Lösung als Modul `Ast` ab.

1. Definieren Sie einen Datentyp `Exp t` für arithmetische Ausdrücke. Ein Ausdruck ist dabei entweder eine Variable, eine Integer-Konstante oder die Summe zweier arithmetischer Ausdrücke. Der Typ der vorkommenden Variablennamen soll dabei nicht spezifiziert sein, verwenden Sie dafür den Typparameter `t`.

   Beispiellösung:

   ```
   data Exp t
     = Var t
     | Const Integer
     | Add (Exp t) (Exp t)
   ```

Im Weiteren sei der Typ **`type`** `Env t = t -> Integer` gegeben. Ein Element von diesem Typ stellt eine *Variablenumgebung* dar, also eine Funktion, welche jeder Variablen einen Integer-Wert zuweist.

2. Definieren Sie eine Funktion `eval :: Env t -> Exp t -> Integer`, welche eine Variablenumgebung und einen Ausdruck nimmt, und diesen komplett auswertet.

   Beispiellösung:

```
eval :: Env a -> Exp a -> Integer
eval env (Var v) = env v
eval env (Const c) = c
eval env (Add e1 e2) = (eval env e1) + (eval env e2)
```

3. Erweitern Sie den Datentyp `Exp t` sowie die Auswertungsfunktion `eval` um die boole'schen Operationen `Less`, `And` und `Not`, sowie um einen ternären If-Then-Else-Operator. Nehmen Sie für die boole'schen Operationen C-Semantik an. D.h. False entspricht dem Wert 0, jeder andere Wert ist True. Sollte eine boole'sche Operation das Ergebnis True generieren, verwenden Sie als Rückgabewert die 1.

   Beispiellösung:

```
data Exp t
  = Var t
  | Const Integer
  | Add (Exp t) (Exp t)
  | Less (Exp t) (Exp t)
  | And (Exp t) (Exp t)
  | Not (Exp t)
  | If (Exp t) (Exp t) (Exp t)

eval :: Env a -> Exp a -> Integer
eval env (Var v) = env v
eval env (Const c) = c
eval env (Add e1 e2) = (eval env e1) + (eval env e2)
eval env (Less e1 e2)
  | eval env e1 < eval env e2 = 1
  | otherwise = 0
eval env (And e1 e2)
  | eval env e1 == 0 || eval env e2 == 0 = 0
  | otherwise = 1
eval env (Not e1)
  | eval env e1 == 0 = 1
  | otherwise = 0
eval env (If b t e)
  | eval env b == 0 = eval env e
  | otherwise = eval env t
```

4. Geben Sie eine Instanziierung der Typklasse `Show` für Ihren Datentyp an, die einen Ausdruck in Infix-Notation darstellt. Die Addition zweier Konstanten 3 und 4 soll beispielsweise als `(3 + 4)` dargestellt werden. Für die Konvertierung in eine Zeichenkette in der Typklasse `Show` ist die Funktion `show :: a -> String` verantwortlich, die Sie implementieren müssen. Achten Sie auf ausreichende Klammerung der Ausdrücke, da wir für die Operationen keine Präzedenzen definiert haben.

   Beispiellösung:

```
instance Show a => Show (Exp a) where
  show (Var a) = show a
  show (Const i) = show i
```

```
show (Add e1 e2) = "(" ++ show e1 ++ "␣+␣" ++ show e2 ++ ")"
show (Less e1 e2) = "(" ++ show e1 ++ "␣<␣" ++ show e2 ++ ")"
show (And e1 e2) = "(" ++ show e1 ++ "␣&␣" ++ show e2 ++ ")"
show (Not e) = "!(" ++ show e ++ ")"
show (If b t e) = "(" ++ show b ++ "␣?␣" ++ show t ++ "␣:␣" ++ show e ++ ")"
```

3 Haskell: Ropes [alte Klausuraufgabe, 18 Punkte]

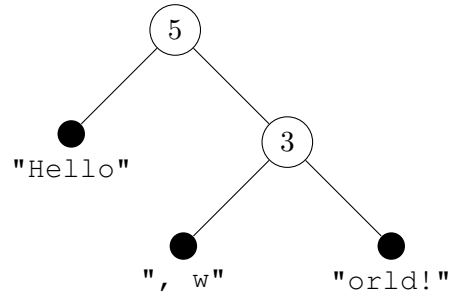
Geben Sie Ihre Lösungen als Modul `Ropes` ab.

Zur Manipulation langer Strings greifen Texteditoren häufig auf *Ropes* zurück. Ein Rope ist ein binärer Baum, der eine Liste (z.B. einen String) repräsentiert. In den Blättern ist von links nach rechts eine Zerlegung der Liste gespeichert. Innere Knoten sind hingegen mit der Länge der vom linken Teilbaum dargestellten Teilliste, ihrem *Gewicht*, annotiert.

Nachfolgend sehen Sie eine Datentypdefinition für Ropes in Haskell (*In der Klausur ohne Typparameter, sondern nur für Strings*). Rechts daneben ist eine von mehreren gültigen Darstellungen des Strings `"Hello, world!"` als `Rope Char` und ihr zugehöriger Haskell-Ausdruck abgebildet.

```
data Rope a
  = Leaf [a]
  | Inner (Rope a) Int (Rope a)
```

```
Inner (Leaf "Hello") 5
      (Inner (Leaf ", w") 3 (Leaf "orld!"))
```



1. Implementieren Sie die Funktion [5 Punkte]

   ```
   ropeLength :: Rope a -> Int
   ```

   die die Länge der durch das Rope dargestellten Zeichenkette berechnet. Nutzen Sie das Gewicht innerer Knoten, um möglichst wenige Knoten zu besuchen.

   Beispiellösung:

   ```
   ropeLength :: Rope a -> Int
   ropeLength (Leaf s)       = length s
   ropeLength (Inner _ w r) = w + ropeLength r
   ```

2. Implementieren Sie die Funktion [2 Punkte]

   ```
   ropeConcat :: Rope a -> Rope a -> Rope a
   ```

   die die übergebenen Ropes verkettet. Benutzen Sie `ropeLength` zur Berechnung des Gewichts.

   Beispiellösung:

   ```
   ropeConcat :: Rope a -> Rope a -> Rope a
   ropeConcat l r = Inner l (ropeLength l) r
   ```

3. Implementieren Sie die Funktion [11 Punkte]

```
ropeSplitAt :: Int -> Rope a -> (Rope a, Rope a)
```

`ropeSplitAt i r` zerlegt das Rope `r` der Länge n, das die Liste $[a_0, ..., a_{n-1}]$ darstellt, an Index `i` in zwei Teilropes: Das erste Teilrope stellt die Liste $[a_0, ..., a_{i-1}]$ dar, das zweite Teilrope $[a_i, ..., a_{n-1}]$. Für Indexwerte außerhalb des Intervalls $[0, n]$ ist die Funktion unspezifiert.

Verwenden Sie das Gewicht innerer Knoten, um die Spaltposition zu finden. Die Listenfunktionen `drop, take :: Int -> [a] -> [a]` sind hilfreich, um die Listen in den Blattknoten zu zerlegen.

Beispiel[2]:

```
> let (l, r) = ropeSplitAt 6 (fromList "Hello, world!")
> toList l
"Hello,"
> toList r
" world!"
```

Beispiellösung:

```
ropeSplitAt :: Int -> Rope a -> (Rope a, Rope a)
ropeSplitAt i (Leaf s) = (Leaf (take i s), Leaf (drop i s))
ropeSplitAt i (Inner l w r)
  | i < w
  = let (ll, lr) = ropeSplitAt i l
    in (ll, ropeConcat lr r)
  | i > w
  = let (rl, rr) = ropeSplitAt (i-w) r
    in (ropeConcat l rl, rr)
  | otherwise
  = (l, r)
```

4. Implementieren Sie die Funktion *Nicht Teil der Klausur* [3 Punkte]

```
ropeInsert :: Int -> Rope a -> Rope a -> Rope a
```

`ropeInsert i a b` fügt Rope `a` in Rope `b` an Index `i` ein. Für Indexwerte außerhalb des Intervalls $[0, \texttt{ropeLength b}]$ ist die Funktion unspezifiert.

Hinweis: Verwenden Sie von Ihnen schon definierte Funktionen!

Beispiel:

```
> toList (ropeInsert 6 (fromList " cruel")
                       (fromList "Hello, world!"))
"Hello, cruel world!"
```

Beispiellösung:

```
ropeInsert :: Int -> Rope a -> Rope a -> Rope a
ropeInsert i s rope = l `ropeConcat` s `ropeConcat` r
  where
    (l, r) = ropeSplitAt i rope
```

[2] Dieses Beispiel verwendet die Funktionen `fromList :: [a] -> Rope a` und `toList :: Rope a -> [a]`, um zwischen Strings und zugehörigen Ropes zu konvertieren. Diese dürfen Sie nicht in Ihrer Lösung verwenden. *Für das Übungsblatt dürfen Sie sie aber gerne selbst definieren, um die Beispiele und andere Eingaben auszuprobieren.*

5. Implementieren Sie die Funktion *Nicht Teil der Klausur* [3 Punkte]

   `ropeDelete :: Int -> Int -> Rope a -> Rope a`

   `ropeDelete i j rope` löscht die Teilliste $[a_i, ..., a_{j-1}]$ aus `rope`, so dass das resultierende Rope die Liste $[a_0, ..., a_{i-1}, a_j, ..., a_{n-1}]$ darstellt. Falls nicht $0 \leq i \leq j \leq n$ gilt, ist die Funktion unspezifiert.

   Hinweis: Verwenden Sie von Ihnen schon definierte Funktionen!

   Beispiel:

```
> toList (ropeDelete 1 3 (fromList "0123"))
"03"
```

   Beispiellösung:

```
ropeDelete i j r = ropeConcat a d
  where
    (a, _) = ropeSplitAt i r
    (_, d) = ropeSplitAt j r
```

6. Implementieren Sie mit Ropes einen trivialen, aber *Nicht Teil der Klausur* [1337 Bonuspunkte] theoretisch effizienten Texteditor!
   Der Editor soll mit einem leeren Dokument beginnen und pro Eingabezeile einen der folgenden Befehle akzeptieren und auf das aktuelle Dokument ausführen (Zeilennummern sind dabei 1-basiert und inklusiv zu verstehen):

   `i` *line text*: Füge *text* als neue Zeile mit der Zeilennumer *line* ein

   `d` *start stop*: Lösche Zeilen *start* bis *stop*

   `c` *start stop text*: Ersetze Zeilen *start* bis *stop* mit *text*

   Nach jedem Befehl soll der Editor das gesamte aktuelle Dokument ausgeben.

   Beispiel:

```
$ runhaskell SampleSolution.hs
i 1 hello
hello

i 2 world
hello
world

i 2 cruel
hello
cruel
world

c 1 2 goodbye
goodbye
world
```

   Hinweise: Stellen Sie ein Dokument als `Rope String` dar, wobei jeder `String` einer Dokumentzeile entspricht. Damit können wir effizient eine oder mehrere Zeilen einfügen, ersetzen oder löschen.

   Implementieren Sie Ihren Editor als Funktion `ed :: [String] -> [String]`, welche eine Liste von Eingeabe-Strings lazy in eine Liste von Ausgabestrings übersetzt. Eine passend `main`-Funktion können Sie dann als

```
main = interact (unlines . ed . lines)
```

definieren, welches, ebenfalls lazy, die Zeilen aus der Eingabe liest, mit `ed` verarbeitet und die Ergebniszeilen ausgibt. Dank Laziness können hierbei Zeilen ausgegeben werden noch bevor alle Eingabezeilen eingelesen wurden. (Diese `main`-Funktion unter `ghci` auszuführen kann zu Problemen führen, mit `runhaskell` wie im Beispiel oben sollte es aber funktionieren.)

Die Parameter jedes Befehls können Sie mit `words :: String -> [String]` zerlegen und mit der gegenteiligen Funktion `unwords :: [String] -> String` wieder zusammenfügen. Zeilennummern können Sie mittels `read :: Read a => String -> a`, der Umkehrung von `show`, zu einem `Integer` parsen.

Sonderzusatzaufgabe: Definieren Sie den Befehl `u` zum (ggf. mehrfachen) Rückgängigmachen der letzten Änderung(en)! Sie sollten dazu statt einem einzelnen Rope nun eine Liste (Stack) von Ropes speichern, um sich alte Dokumentzustände zu merken. Hier können wir insbesondere von der Eigenschaft von Ropes (sowie allen anderen Haskell-Datenstrukturen) Gebrauch machen, dass Transformationen ein Rope nicht destruktiv verändern, sondern immer ein neues Rope zurückgeben.

Beispiellösung:

```
fromList :: [a] -> Rope a
fromList as = Leaf as

toList :: Rope a -> [a]
toList (Leaf as) = as
toList (Inner r1 _ r2) = toList r1 ++ toList r2

ed :: [String] -> [String]
ed = go (fromList [])
  where
    go _ [] = []
    go r (cmd:cmds) =
      let r' = run (words cmd) r in
      toList r' ++ "" : go r' cmds
    -- dass hier aufeinanderfolgende Leerzeichen verloren gehen, ist
    -- natürlich ein Feature, kein Bug
    run ("i":l:ss) = ropeInsert (read l - 1) (fromList [unwords ss])
    run ["d", s, e] = ropeDelete (read s - 1) (read e)
    run ("c":s:e:ss) = run ("i":s:ss) . run ["d", s, e]
    run cmd = error "invalid command"

edUndo :: [String] -> [String]
edUndo = go [fromList []]
  where
    go _ [] = []
    go rs (cmd:cmds) =
      let (r:rs') = run (words cmd) rs in
      toList r ++ "" : go (r:rs') cmds
    run ("i":l:ss) (r:rs) =
      ropeInsert (read l - 1) (fromList [unwords ss]) r : r : rs
    run ["d", s, e] (r:rs) = ropeDelete (read s - 1) (read e) r : r : rs
    run ("c":s:e:ss) rs =
      let (r:_:rs) = run ("i":s:ss) (run ["d", s, e] rs) in
      r:rs
```

```
    run ["u"] (r:r':rs) = r':rs
    run _ _ = error "invalid command"

main = interact (unlines . edUndo . lines)
```