

Fusion

auch mit Kern

Christian Höner zu Siederdisen
`christian.hoener.zu.siederdisen@uni-jena.de`

Theoretische Bioinformatik, Bioinformatik Uni Jena

Jan 12th, 2023

Haskell Arrays: Index

Eine Alternative zu den Listen für Rush-hour

```
1  -- Zugriff auf Arrays
2  import Data.Array.IArray as A
3
4  --Index-Typen in Arrays
5  class Ix a where
6    range :: (a, a) -> [a]
7    index :: (a, a) -> a -> Int
8    inRange :: (a, a) -> a -> Bool
9
10 ix = ( (1,1), (6,6) )    -- Spielplan Dimension
11 rn = range ix           -- aktive Zellen
12 at = index ix (2,3)    -- direkter Index
13 ok = inRange ix (7,5)  -- Zelle ist ausserhalb
```

Haskell Arrays

```
1  -- Arrays haben diesen Constraint
2  type IArray :: (* -> * -> *) -> * -> Constraint
3
4  -- Gegeben Array 'a', mit Index Typ i, WertTyp e
5  -- Beziehe Wert and Position i
6  (!) :: (IArray a e, Ix i) => a i e -> i -> e
7
8  -- Array-Konstruktion
9  array :: (IArray a e, Ix i) => (i, i) -> [(i, e)] -> a i e
10
11 -- Neues Array, gegeben altes Array und Update
12 (//) :: (IArray a e, Ix i) => a i e -> [(i, e)] -> a i e
```

Beispiel

```
1 ix = ((1,1),(2,3))
2 test :: A.Array Dim2 Int
3 test = array ix [ (k,0) | k <- range ix ]
4           // [ ((2,3),9) ]
5
6
7
8 % array ((1,1),(2,3))
9 %   [ ((1,1),0), ((1,2),0), ((1,3),0)
10 %   , ((2,1),0), ((2,2),0), ((2,3),9) ]
```

Beispiel in C

```
1 int a = 0;
2 for (int i = 1; i < 10; i++)
3     for (int j = i; j < 10; i++)
4     {
5         a += i % 2 == 0 ? a : 0;
6     }
7 return a
```

Lösung in Haskell: Laufvariablen implizit

```
1 sum [i | i <- [1..10], j <- [i..10], even i]
```

So what?!

- Bei komplizierteren Algorithmen werden immer wieder Listenkonstruktoren (:) erschaffen und zerstört
- Dadurch sind die Algorithmen viel ($\geq 100\times$) langsamer

Dann schreiben wir es besser

Schritte in “Schleifen” (ab jetzt dann Streams)

Done signalisiert das die Schleife beendet ist

Yield berechnet das “momentane” Element und die nächste Schleifenvariable

Skip überspringt das “momentane” Element und gibt direkt die nächste Schleifenvariable

!x macht x *strikt*, “bang”

```
1 {-# LANGUAGE BangPatterns #-}  
2  
3 data Step a s = Yield a !s | Skip !s | Done
```

Code auf den Slides leicht vereinfacht. Es fehlen die strikten Datenstrukturen für Tupel und Maybe.

From Lists to Streams to Nothing at all¹

- ein `Stream` ist analog zu einer Liste und hält Werte vom Typ `a`
- jeder `Stream` ist definiert durch eine *Streamfunktion*
 $s \rightarrow \text{Step } a \ s$, die einen Wert und einen neuen Step-wert liefert
- Ausserdem wird der jeweils aktuelle Stepwert gespeichert

```
1 data Stream a s = Stream !(s -> Step a s) !s
2
3 go (Stream (\x -> Yield x (x+1)) 3) ==> Yield 3 4
```

¹Coutts, Leshchinskiy, Stewart, 2007

sum

```
1 sum :: Num a => Stream a s -> a
2 sum (Stream next s0) = go 0 s0
3   where
4     go !a !s = case next s of
5       Done -> a
6       Skip t -> go a t
7       Yield x t -> go (a+x) t
```

vergleiche

```
1 sum :: Num a => [a] -> a
2 sum xs = go 0 xs
3   where
4     go a [] = a
5     -- ???
6     go a (x:xs) = go (a+x) xs
```

enum

```
1 enum :: Int -> Int -> Stream Int Int
2 enumNo l h = Stream go l
3   where
4     go !k | k > h = Done
5           | otherwise = Yield k (k+1)
```

vergleiche

```
1 enum :: Int -> Int -> [Int]
2 enum l h = go l
3   where
4     go k | k > h = []
5           | otherwise = k : go (k+1)
```

Wir sind fertig!

```
1 summe l h = sum $ enum l h
2 == =>
3 summe 1 10 == 55
```

Oder?

```
1 summe l h = sum . rly $ enum l h
2   where rly (Stream f x) = Stream f (x+10)
```

Wir haben hier unsere Schleifenvariable x modifiziert und den Algorithmus kaputt gemacht!

Vorschläge wie man das verhindern könnte?

Ein Ausflug zu Quantoren: Universell

Betrachten wir eine Funktion:

$$f_1 :: X \rightarrow X$$

$$f_2 :: \forall X : X \rightarrow X$$

In beiden Fällen, implizit (f_1) und explizit (f_2) kann die Funktion auf jedes *vom Aufrufer* gewählte Argument benutzt werden. f muss "für alle" X nutzbar sein.

Existentiell

Betrachte nun

$$g_1 :: \exists X : X \rightarrow X$$

$$g_2 :: \exists X \Rightarrow \text{Show } X : X \rightarrow X$$

$$g_3 :: (\forall X \Rightarrow \text{Show } X) : X \rightarrow X$$

g_1 und g_2 beschränken den Raum der möglichen X , erlauben aber gleichzeitig das g_1 , g_2 nun wieder auf allen Typen funktionieren müssen, die diesen Constraint erfüllen (wie man in g_3 sieht).

Existentiell Quantifizierte Typen

Betrachte nun folgenden Typ:

```
1 data Stream a = forall s . Strm (s -> a) s
```

Dieser Typ hat folgende Konsequenzen:

- Wird ein Datenkonstruktor `Strm f s` gebaut, so können wir den Typ von `s` frei wählen
- `s` selbst ist “von aussen” nicht zugänglich, da `s` selbst nicht “links” vom `=` auftaucht!
- Alles was wir machen können ist die Funktion $s \rightarrow a$ auf `s` anwenden

```
1 case (strm :: Stream a)  
2   of Strm f s -> (f s :: a)
```

Man beachte den Typ `a`

Der korrekte Typ für Streams

```
1 {-# LANGUAGE BangPatterns #-}
2 {-# LANGUAGE ExistentialQuantification #-}
3
4 data Step a s = Yield a !s | Skip !s | Done
5
6 data Stream a = forall s . Stream !(s -> Step a s) !s
```

Beachte das s existentiell quantifiziert ist: $\exists s == (\forall s. \dots)$

sum

```
1 sum :: Num a => Stream a -> a
2 {-# Inline [0] sum #-}
3 sum (Stream next s0) = go 0 s0
4   where
5     go !a !s = case next s of
6       Done -> a
7       Skip !t -> go a t
8       Yield !x !t -> go (a+x) t
```

enum

```
1 enum :: Int -> Int -> Stream Int
2 {-# Inline [0] enum #-}
3 enum l h = Stream go l
4   where
5     go !k | k > h = Done
6           | otherwise = Yield k (k+1)
```

map

Vorführung . . .

filter

Skip in Aktion. Falls x in $\text{Yield } x \ t$ nicht das Prädikat f erfüllt, so wird x “ge-Skip-ed”.

```
1 filter :: (a -> Bool) -> Stream a -> Stream a
2 {-# Inline [0] filter #-}
3 filter f (Stream next s0) = Stream go s0
4   where
5     go !s = case next s of
6       Done -> Done
7       Skip !t -> Skip t
8       Yield !x !t | f x -> Yield x t
9                 | otherwise -> Skip t
```

concatMap

```
1 concatMap :: (a -> Stream b) -> Stream a -> Stream b
2 {-# Inline [0] concatMap #-}
3 concatMap f (Stream next0 s0) = Stream next (s0 :!: Nothing)
4   where
5     {-# INLINE next #-}
6     next (!s :!: Nothing) = case next0 s of
7       Done          -> Done
8       Skip    s'    -> Skip (s' :!: Nothing)
9       Yield x s'    -> Skip (s' :!: Just (f x))
10
11    next (!s :!: Just (Stream g t)) = case g t of
12      Done          -> Skip    (s :!: Nothing)
13      Skip    t'    -> Skip    (s :!: Just (Stream g t'))
14      Yield x t'    -> Yield x (s :!: Just (Stream g t'))
```

Bugs?

- Nur `enum` kann das `x` manipulieren, indem die Funktion `f` bereit stellt.
- Alle anderen Funktionen können nur `fx` schreiben, aber nicht `x` verändern!

```
1 nojoy l h = sum . rly $ enum l h
2   where rly (Stream f x) = Stream f (x+10)
3
4
5
6 % - No instance for (Num s) arising from a use of (+)
7 %   Possible fix:
8 %     add (Num s) to the context of data ctor Stream
9 % - In the second argument of Stream, namely (x + 10)
10 %   In the expression: Stream f (x + 10)
11 %   In an equation: rly (Stream f x) = Stream f (x + 10)
```

Call-pattern Specialization

Der Haskell-Compiler darf semantik-erhaltende Transformationen durchführen! Eine einfache ist "CallSpec".

```
1 data Maybe a = Just a | Nothing
2
3 f :: Maybe Int -> Int
4 f Nothing = 0
5 f (Just x) = x+1
6
7 z = f (Just 1)    ===    case Just 1 of
8                       Nothing -> 0
9                       Just x   -> x+1
```

Das kann man auch anders schreiben:

```
1 f_Nothing = 0
2 f_Just x = x+1
3
4 z = f (Just x) === f_Just x === x+1
```

case-of-case

```
1 data AB = A | B
2 data XY = X | Y
3
4 f :: AB -> XY
5 f ab = case ab of {A -> X; B -> Y}
6
7 g :: XY -> Int
8 g xy = case xy of {X -> 0; Y -> 1}
9
10 g(f x) = case
11           case x of
12             A -> X
13             B -> Y
14           of
15             X -> 0
16             Y -> 1
```

case-of-case

```
1 g(f x) = case
2           case x of
3             A -> X
4             B -> Y
5           of
6             X -> 0
7             Y -> 1
```

Ersetze die Fälle direkt:

```
1 g(f x) = case-of-case x of
2           A -> X -> 0
3           B -> Y -> 1
```

Vereinfache:

```
1 g(f x) = case x of
2           A -> 0
3           B -> 1
```

Beispiel-Code

Wir wollen `sum (map square (enum m n))` optimieren.
Handgeschrieben:

```
1 sumMapSquare n = go 0 1 where
2   go acc cur = if cur > n
3                 then acc
4                 else go (acc + square cur) (cur + 1)
```

Das ist zu fehleranfällig das selbst umzuschreiben. Stream fusion und case-of-case laufen lassen!

```
1 sumMapSquare m n = go 0 m where
2     next_enum from = case (from <= n) of -- enum "next"
3         True  -> Yield from (from+1)
4         False -> Done
5     next_map s = case next_enum s of -- map "next"
6         Done      -> Done
7         Yield x s' -> Yield (square x) s'
8     go acc s = case next_map s of -- sum "go"
9         Done      -> acc
10        Yield x s' -> go (acc+x) s'
```

- next-enum in next-map einsetzen

```
1 sumMapSquare m n = go 0 m where
2   next_map s =
3     case
4       case (s <= n) of
5         True  -> Yield s (s+1)
6         False -> Done
7     of
8       Done          -> Done
9       Yield x s'   -> Yield (square x) s'
10  go acc s = case next_map s of
11    Done          -> acc
12    Yield x s'   -> go (acc+x) s'
```

- case-of-case !

```
1 sumMapSquare m n = go 0 m where
2   next_map s = case (s <= n) of
3     True  -> Yield (square s) (s+1)
4     False -> Done
5   go acc s = case next_map s of
6     Done      -> acc
7     Yield x s' -> go (acc+x) s'
```

- next-map in go einsetzen

```
1 sumMapSquare m n = go 0 m where
2     go acc s =
3         case
4             case (s <= n) of
5                 True  -> Yield (square s) (s+1)
6                 False -> Done
7         of
8             Done          -> acc
9             Yield x s'   -> go (acc+x) s'
```

- case-of-case !

```
1 sumMapSquare m n = go 0 m where
2     go acc s = case (s <= n) of
3         True  -> go (acc + square s) (s+1)
4         False -> acc
```

```
sum (map square (enum m n))
```

```
1  mapsumsquare = \ i j ->
2    case i of { I# ii ->
3      case j of { I# jj ->
4        case $wmapsumsquare ii jj of z { __DEFAULT ->
5          I# z
6        }}}
7
8  $wmapsumsquare = \ ii jj ->
9    joinrec {
10      $wgo_s292 aa bb
11        = case ># bb jj of {
12          __DEFAULT ->
13            jump $wgo_s292
14              (+# aa (*# bb bb)) (+# bb 1#);
15          1# -> aa
16        }; } in
17    jump $wgo_s292 0# ii
```

Zusammenfassung

- Haskell hat mächtige, Codetransformationswerkzeuge
- Damit ist es möglich semantik gleiche Transformationen durchzuführen
- diese “inlinen” und vereinfachen (“case-of-case”) Code
- das erlaubt es uns komplexe Funktionen aus einfachen Funktionen zusammensetzen
- *ohne* das wir die Kosten für tatsächliche Funktionsaufrufe zahlen müssen
- diese Ideen sind im Paket `vector` implementiert
- `vector` liefert auch noch die Möglichkeit `lineare` Array-strukturen effizient zu manipulieren