Fusion

auch mit Kern

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Januar, 2024

Beispiel in C

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

Lösung in Haskell: Laufvariablen implizit

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

So what?!

- Bei komplizierteren Algorithmen werden immer wieder Listenkonstruktoren (:) erschaffen und zestört
- 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”

```haskell
{-# LANGUAGE BangPatterns #-}

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 all[1]

- 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

```haskell
data Stream a s = Stream !(s -> Step a s) !s

go (Stream (\x -> Yield x (x+1)) 3)    ==>    Yield 3 4
```

[1] Coutts, Leshchinskiy, Stewart, 2007

sum

```
sum :: Num a => Stream a s -> a
sum (Stream next s0) = go 0 s0
  where
    go !a !s = case next s of
      Done -> a
      Skip t -> go a t
      Yield x t -> go (a+x) t
```

vergleiche

```
sum :: Num a => [a] -> a
sum xs = go 0 xs
  where
    go a [] = a
    -- ???
    go a (x:xs) = go (a+x) xs
```

enum

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

vergleiche

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

Wir sind fertig!

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

Oder?

```haskell
summe l h = sum . rly $ enum l h
  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:

$$f_1 :: X \to X$$
$$f_2 :: \forall X : \quad X \to X$$

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

$$
\begin{aligned}
g_1 &:: \exists X : \quad X \to X \\
g_2 &:: \exists X \Rightarrow \text{Show } X : \quad X \to X \\
g_3 &:: (\forall X \Rightarrow \text{Show } X) : \quad X \to X
\end{aligned}
$$

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:

```
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 mir machen können ist die Funktion $s \to a$ auf s anwenden

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

Man beachte den Typ a

Der korrekte Typ für Streams

```haskell
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE ExistentialQuantification #-}

data Step a s = Yield a !s | Skip !s | Done

data Stream a = forall s . Stream !(s -> Step a s) !s
```

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

sum

```haskell
sum :: Num a => Stream a -> a
{-# Inline [0] sum #-}
sum (Stream next s0) = go 0 s0
  where
    go !a !s = case next s of
      Done -> a
      Skip !t -> go a t
      Yield !x !t -> go (a+x) t
```

enum

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

map

Vorführung . . .

filter

`Skip` in Aktion. Falls x in `Yield x t` nicht das Prädikat f erfüllt, so wird x “ge-Skip-ed”.

```haskell
filter :: (a -> Bool) -> Stream a -> Stream a
{-# Inline [0] filter #-}
filter f (Stream next s0) = Stream go s0
  where
    go !s = case next s of
      Done -> Done
      Skip !t -> Skip t
      Yield !x !t | f x -> Yield x t
        | otherwise -> Skip t
```

concatMap

```haskell
concatMap :: (a -> Stream b) -> Stream a -> Stream b
{-# Inline [0] concatMap #-}
concatMap f (Stream next0 s0) = Stream next (s0 :!: Nothing)
  where
    {-# INLINE next #-}
    next (!s :!: Nothing) = case next0 s of
      Done         -> Done
      Skip     s' -> Skip (s' :!: Nothing)
      Yield x s' -> Skip (s' :!: Just (f x))

    next (!s :!: Just (Stream g t)) = case g t of
      Done         -> Skip    (s :!: Nothing)
      Skip     t' -> Skip    (s :!: Just (Stream g t'))
      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!

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

% - No instance for (Num s) arising from a use of (+)
%   Possible fix:
%     add (Num s) to the context of data ctor Stream
% - In the second argument of Stream, namely (x + 10)
%   In the expression: Stream f (x + 10)
%   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".

```haskell
data Maybe a = Just a | Nothing

f :: Maybe Int -> Int
f Nothing = 0
f (Just x) = x+1

z = f (Just 1)    ===    case Just 1 of
                          Nothing -> 0
                          Just x   -> x+1
```

Das kann man auch anders schreiben:

```haskell
f_Nothing = 0
f_Just x = x+1

z = f (Just x) === f_Just x === x+1
```

case-of-case

```
data AB = A | B
data XY = X | Y

f :: AB -> XY
f ab = case ab of {A -> X; B -> Y}

g :: XY -> Int
g xy = case xy of {X -> 0; Y -> 1}

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

case-of-case

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

Ersetze die Fälle direkt:

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

Vereinfache:

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

Beispiel-Code

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

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

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

```haskell
sumMapSquare m n = go 0 m where
    next_enum from = case (from <= n) of  -- enum "next"
      True  -> Yield from (from+1)
      False -> Done
    next_map s = case next_enum s of      -- map "next"
      Done        -> Done
      Yield x s' -> Yield (square x) s'
    go acc s = case next_map s of         -- sum "go"
      Done        -> acc
      Yield x s' -> go (acc+x) s'
```

- next-enum in next-map einsetzen

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

- case-of-case !

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

- next-map in go einsetzen

```
sumMapSquare m n = go 0 m where
    go acc s =
      case
        case (s <= n) of
          True  -> Yield (square s) (s+1)
          False -> Done
      of
        Done        -> acc
        Yield x s’ -> go (acc+x) s’
```

- case-of-case !

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

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

```
mapsumsquare = \ i j ->
  case i of { I# ii ->
  case j of { I# jj ->
  case $wmapsumsquare ii jj of z { __DEFAULT ->
  I# z
}}}

$wmapsumsquare = \ ii jj ->
  joinrec {
    $wgo_s292 aa bb
      = case >## bb jj of {
          __DEFAULT ->
            jump $wgo_s292
              (+# aa (*# bb bb)) (+# bb 1#);
          1# -> aa
        }; } in
  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 zusammenzusetzen
- *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