MinHeight-Trees

Bessere Performance via *Equational Reasoning!*

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

Jan 26^{th}, 2023

Min-Height Bäume

Min-Height Bäume sind Blatt-gewichtete Bäume deren Pfadgewicht minimal ist.

- Gegeben eine Liste, (oBdA) $[1, \ldots, 100]$
- Finde Binärbaum folgender Konstruktion
- die Reihenfolge der Blätter ("fringe") entpricht der Reihenfolge der Liste
- Jedes Blatt hat das Gewicht der Annotation
- Jedes Pfadstück von der Wurzel zum Blatt erhöht das Gewicht um 1
- Finde Binärbaum mit minimalem Gewicht

Beispiel

Eingabe $[1, \ldots, 4]$

```
 1  data Tree a = Leaf a | Fork (Tree a) (Tree a)
 2
 3  type Forest a = [Tree a]
 4
 5      Fork                                Fork
 6      |   |                               |   |
 7  Fork    Fork                        Fork    4
 8  |   |   |   |                       |   |
 9  1   2   3   4                   Fork    3
10                                   |   |
11                                   1   2
```

im "Bild" sind die `(Leaf x)` durch `x` ersetzt, um das Bild einfach zu halten.

Kostenfunktion

Die Kostenfunktion ist am besten programmatisch dargestellt:

```
1  cost :: (Tree Int -> Int)
2  cost (Leaf x) = x
3  cost (Fork l r) = 1 + (cost l `max` cost r)
4
5  minCost :: Forest Int -> Tree Int
6  minCost = minimumBy (comparing cost)
```

Diese Funktion ist nicht “deterministisch” (mit Anführungsstrichen). Was kann hier gemeint sein?

Algorithmus

Zuerst wird der Wald aller Bäume konstruiert.

```
1  -- | Gegeben die Liste xs, generiere den Forest aller
2  -- moeglichen Baeume. Diese Definition ist rekursiv.
3
4  trees :: [Int] -> Forest Int
5  -- singleton liefert Forest mit einem Blatt.
6  trees [x] = [Leaf x]
7  -- Eine vielelementige Liste generiert alle Baeume auf
8  -- dem Tail. Fuer jeden solchen Baum wird der head links
9  -- in den linken spine an alle Positionen gesetzt.
10 trees (x:xs) = concatMap (prefixes x) (trees xs)
```

mein Bild, p42 an der Tafel

prefixes ist trickreich

```
1  -- | Konstruiere alle Prefixe mit @x@
2  -- , fuer einen gegebenen Baum.
3
4  prefixes :: Int -> Tree Int -> [Tree Int]
5  -- Ein Leaf wird rechtes Blatt,
6  -- um die fringe-property beizubehalten
7  prefixes x t@(Leaf y) = [Fork (Leaf x) t]
8  -- Ansonsten konstruiere den linken spine,
9  -- die Anzahl bestimmt sich durch den Pfad von der Wurzel
10 -- zum Kind ganz links in l. (siehe Tafelbild)
11 prefixes x t@(Fork l r) = Fork (Leaf x) t
12                          : [Fork l' r | l' <- prefixes x l]
```

Test

- `Inline [1]` verschiebt Haskells Inlining
- ignoriert das erstmal

```
1  -- | Gegeben irgendeine Liste, konstruiere alle Waelder
2  -- mit gleicher fringe, berechne deren Kosten
3  -- und gebe das lexikographisch kleinste Element mit
4  -- minimalen Kosten zurueck.
5
6  mincostTree :: [Int] -> Tree Int
7  {-# Inline [1] mincostTree #-}
8  mincostTree = minimumBy (comparing cost) . trees
```

Welche Laufzeit erwarten wir?

Exponentielle Laufzeit

- Aber gelehrte Menschen haben Linearzeitalgorithmen geschrieben?
- Wir konstruieren jetzt nach und nach einen neuen Algorithmus der (hoffentlich) schneller ist

```
1  -- @mincostTree [1..k]@ braucht
2  --   - k=13 =  2s
3  --   - k=14 = 10s
4  --   - k=15 = 33s
```

trees umschreiben

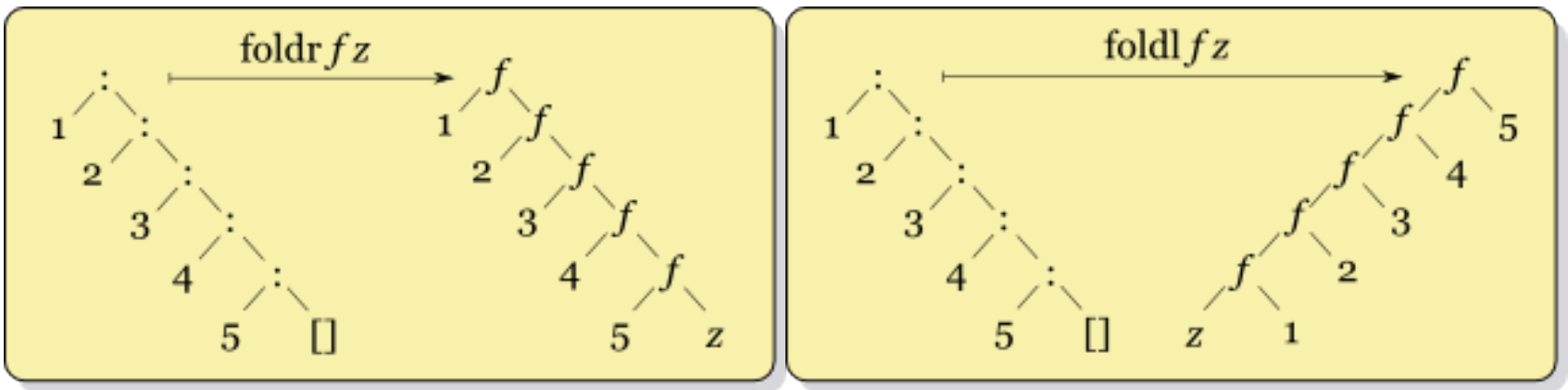
Für die nun folgende Aufgabe benötigen wir eine Hilfsfunktion

```
1  -- | Vergleiche @foldr :: (a->b->b) -> b -> [a] -> b
2  --
3  -- Hier haben wir @x1 `f` (x2 `f` (x3 `f` ... `f` g x))@
4
5  foldrn :: (a->b->b) -> (a->b) -> [a] -> b
6  -- Wenn [x] singulaer, dann @g x@
7  foldrn f g [x] = g x
8  -- Ansonsten @x `f` (foldrn f g xs)
9  foldrn f g (x:xs) = f x (foldrn f g xs)
10
11
12 trees = foldrn (concatMap . prefixes) ((:[]) . Leaf)
```

Nochmal umschreiben

```
 1  trees :: [Int] -> Forest Int
 2  trees = map rollup . forests
 3
 4  forests = :: [Int] -> [Forest Int]
 5  forests = foldrn (concatMap . prefixes) (wrap . wrap . Leaf)
 6
 7  prefixes :: Int -> Forest Int -> [Forest Int]
 8  prefixes x ts = [ Leaf x : rollup (take k ts) : drop k ts
 9                  | k <- [1..length ts] ]
10
11  rollup :: Forest Int -> Tree Int
12  rollup = foldl1 Fork
13
14  -- foldl1 Fork [1,2,3] == (1 `F` 2) `F` 3
```

Zur Erinnerung: foldr vs foldl



https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Fusion

- *Programmfusion* kombiniert einzelne Funktionen in semantik-erhaltender Weise
- Das Ziel kann sein weniger Speicher zu verbrauchen und / oder die Laufzeit eines Programms zu reduzieren
- Fusionsregeln können in Haskell auch explizit angegeben werden und der Compiler transformiert dann das eigene Programm

Es gilt:

```
1  h (foldrn f g xs) = foldrn f' g' xs
2  -- solange gilt:
3  h (g x) = g' x
4  h (f x y) = f' x (h y)
```

Das Problem ist, das *h* nicht *deterministisch* ist. Deshalb nutzen wir die schwächere Regel

```
1  h (foldrn f g xs)  ⇝  foldrn f' g' xs
2  -- solange gilt:
3  h (g x)  ⇝  g' x
4  h (f x y)  ⇝  f' x (h y)
```

minBy Fusion

```
1  minBy f = minimumBy (comparing f)
2
3  minBy cost . wrap = id
4
5  -- gegeben
6  minBy cost ts  ⇝  t
7   ⇒  minBy cost (concatMap (prefixes x ) ts)
8   ⇝  insert x t
9
10
11 -- damit bekommen wir
12 minBy cost (foldrn (concatMap . prefixes) (wrap . Leaf) xs)
13  ⇝
14 foldrn insert Leaf xs
```

insert fusion

Wir wollen nun das gilt `insert x t` produziert irgendeinen min-cost Baum in `prefixes x t`

```
1  -- spezifiziere das gelte:
2  minBy cost    prefixes x  ⇝  insert x
```

Dann folgt:

```
1  minBy cost ts  ⇝  t
2   ⇒  minBy cost (map (insert x) ts)  ⇝  insert x t
```

Gegeben das *t* minimal in *ts* ist, kann man *x* direkt in *t* nutzen, statt *x* in alle *ts* einzusetzen und dann nach dem min-cost Baum zu fragen!

```haskell
-- | Konstruiere einen Forest, wobei @x@ der erste Baum
-- wird (und nur aus einem Leaf besteht). Der
-- restliche Wald geht durch ein 'split'.

insert :: Int -> Forest Int -> Forest Int
insert x ts = Leaf x : split x ts

-- | 'split' wandelt den Forest so das der Baum @u@ oder
-- @x@ guenstiger sind als @v@, oder wird den Anfang
-- @u,v@ solange als neuen Fork bauen, bis das stimmt.

split :: Int -> Forest Int -> Forest Int
split x [u] = [u]
split x (u:v:ts) = if x `max` cost u < cost v then u:v:ts
                   else split x (Fork u v:ts)

-- | Neue Konstruktion in quadratischer Zeit

mincostTree2 = foldl1 Fork . foldrn insert (wrap . Leaf)
```

Es geht noch einen besser

Linearzeit-Algorithmus mittels *augmentation*

```
1  type Tree' = (Int, Tree Int)
2
3  -- | Nutze prime-Versionen,
4  -- die die Kosten explizit annotieren
5
6  mincostTree3 :: [Int] -> Tree Int
7  mincostTree3 = foldl1 Fork . map snd
8    . foldrn insert' (wrap . leaf')
```

Smart-Constructor

- Ein “smart constructor” ist eine Funktion die den tatsächlichen Konstruktor einpackt.
- Zum Beispiel kann man Fehlerbehandlungen einbauen
- oder, wie in diesem Fall, zusätzliche Information mitgeben

Jeder Konstruktor kommt mit einem Gewicht

```
1  -- | Augmentation mit Leaf: smart constructor
2
3  leaf' x = (x, Leaf x)
4
5  -- | Fork, mit Augmentation als smart constructor.
6
7  fork' (a,u) (b,v) = (1+ a `max` b, Fork u v)
```

Smart insert und split

```
1  -- | Insert, aber konstruiere die Leafs zusammen mit Kosten
2
3  insert' :: Int -> [Tree'] -> [Tree']
4  insert' x ts = leaf' x : split' x ts
5
6  -- | Split rechnet jetzt nicht mit Kosten,
7  -- sondern nimmt die Augmentation.
8
9  split' :: Int -> [Tree'] -> [Tree']
10 split' x [u] = [u]
11 split' x (u:v:ts) = if x `max` fst u < fst v then u : v : ts
12                     else split' x (fork' u v : ts)
```

Zusammenfassung

- Hier haben wir mittels Programmtransformation ein Programm in exponentieller Laufzeit verbessert
- die finale Variante mit Augmentation ist linear in der Laufzeit
- allerdings funktioniert das nur, da wir einen *greedy* Algorithmus gebaut haben
- solange wir uns nur für *ein* bestes Ergebnis, nicht für alle, interessieren, ist alles ok
- das bedeutet auch das wir hier die Regeln nicht umkehren können; die Semantik des Programms bleibt nur unter der greedy-Bedingung erhalten

RULES

Haskell-beispiele

```
1  {-# RULES
2  "map/map" forall f g xs. map f (map g xs) = map (f.g) xs
3
4  "map/append" forall f xs ys. map f (xs ++ ys)
5    = map f xs ++ map f ys
6
7  "map/map" [2]  forall f g xs. map f (map g xs)
8    = map (f.g) xs
9
10 "fold/build"
11   forall k z (g::forall b. (a->b->b) -> b -> b) .
12     foldr k z (build g) = g k z
13   #-}
```

Unsere Regel:

```
1  {-# RULES
2    "mincostTree" mincostTree = mincostTree2
3    #-}
```