

Dependently Typed Heaps

<https://github.com/brunjar/heap>

About Me



Lars Brünjes (PhD)

(Pure) Mathematician, Lead Software Architect

44 years old, based in Regensburg, Germany

Email: brunjar@gmail.com

Github: <https://github.com/brunjar>

Outline

- Motivation
- Leftist Heaps
- Proving Theorems in Haskell
- Dependently Typed Heaps
- Reflection
- Questions & Comments



Motivation

Motivation

- Types help catching errors at compile time.
- Some invariants cannot be expressed by “simple” types.
- Haskell steadily moves towards dependent types.
- I wanted to see whether it is possible to prove theorems in Haskell...
- ...and use this to encode some non-trivial invariants.
- Heaps seem to be a good example.



Leftist Heaps

Heap

We consider binary trees whose nodes carry some payload and a **priority** (a natural number):

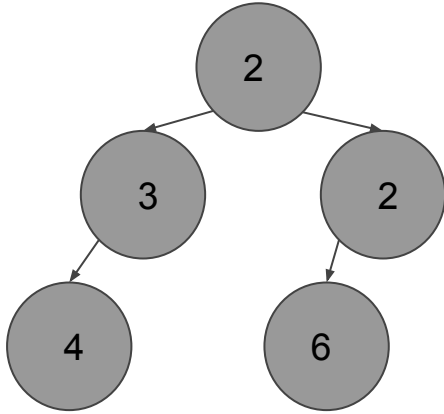
```
data Tree a = Empty | Node Natural a (Tree a) (Tree a)
```

Such a tree is a **heap** if it satisfies the **heap property**: The priority of a node is not bigger than the priority of any of its children.

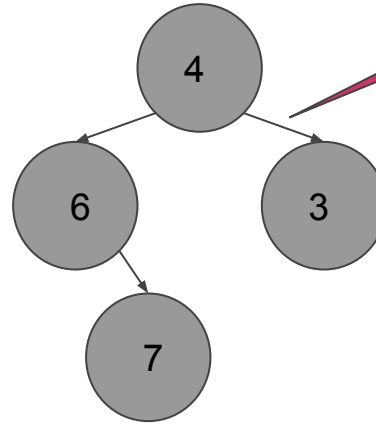
(So in a non-empty heap, the root has minimal priority.)



Heap



This is a heap.

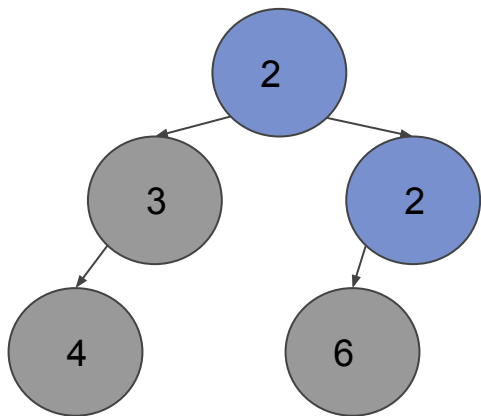


Heap property
violated!

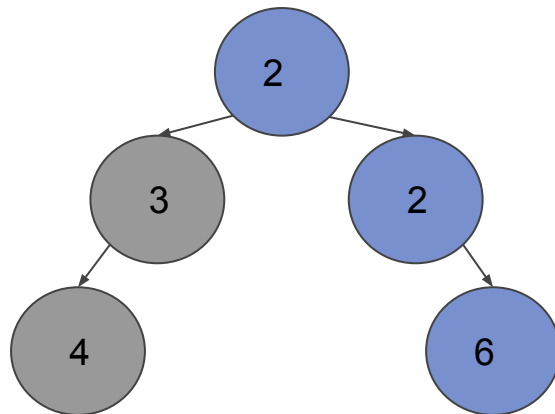
This is **not** a heap.

Rank

The **rank** of a heap is the length of its **right spine**.



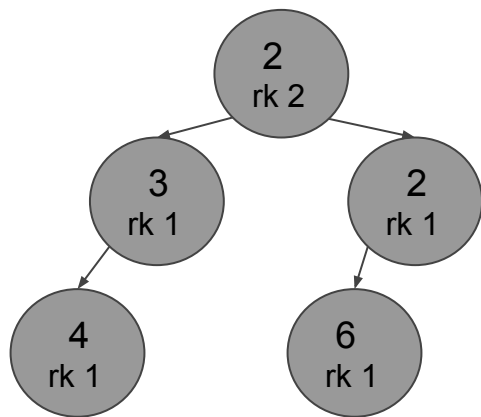
Rank 2



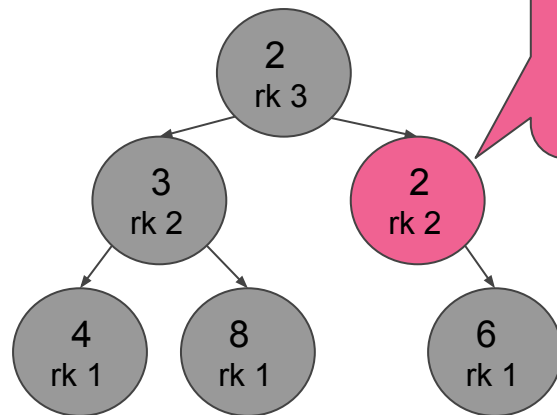
Rank 3

Leftist Heap

A heap is **leftist** if in each node, the rank of the left child is not smaller than the rank of the right child.



leftist



not leftist

Merging Leftist Heaps

```
data Heap a = Empty | Node !Natural !Natural a (Heap a) (Heap a) deriving (Show, Functor)

rank :: Heap a -> Natural
rank Empty          = 0
rank (Node _ r _ _ _) = r

priority :: Heap a -> Maybe Natural
priority Empty      = Nothing
priority (Node p _ _ _ _) = Just p

singleton :: Natural -> a -> Heap a
singleton p x = Node p 1 x Empty Empty

merge :: Heap a -> Heap a -> Heap a
merge Empty h'          = h'
merge h Empty          = h
merge h@(Node p _ x ys zs) h'@(Node q _ _ _ _)
  | q < p               = merge h' h
  | otherwise           =
    let h''@(Node _ r _ _ _) = merge zs h'
        r'                   = rank ys
    in if r <= r'
       then Node p (succ r) x ys h''
       else Node p (succ r') x h'' ys
```

Weakly typed heaps

- Neither heap property nor leftist property are enforced by the compiler.
- “Classical solution”: smart constructors, but those only catch errors at runtime.
- Algorithms like “merge” can easily be done wrong.
- Can we define leftist heaps in such a way that the **compiler** prevents us from constructing “illegal” heaps?



Proving Theorems in Haskell

Technical Tools

- Constraints to express statements
- reified statements (dictionaries) (see “constraints” library by Kmett on Hackage) 
- Singleton Types (see “singletons” library by Eisenberg & Stolarek on Hackage)

```
data Dict :: Constraint -> * where
```

Values of type `Dict p` capture a dictionary for a constraint of type `p`.

e.g.

```
Dict :: Dict (Eq Int)
```

captures a dictionary that proves we have an:

```
instance Eq 'Int
```

Pattern matching on the `Dict` constructor will bring this instance into scope.

Constructors

```
Dict :: a => Dict a
```

A Simple Example

```
data Peano = Z | S Peano deriving (Show, Read, Eq)

infix 4 ??

type family (m :: Peano) ?? (n :: Peano) :: Ordering where
  'Z    ?? 'Z    = 'EQ
  'Z    ?? _     = 'LT
  'S _  ?? 'Z    = 'GT
  'S m  ?? 'S n  = m ?? n

type (m :: Peano) < (n :: Peano) = (m ?? n) ~ 'LT

data SingPeano :: Peano -> * where

  SZ :: SingPeano 'Z
  SS :: SingPeano n -> SingPeano ('S n)

ltS :: SingPeano n -> Dict (n < 'S n)
ltS SZ = Dict
ltS (SS n) = ltS n
```

Dependently Typed Heaps

Dependently Typed Heaps

```
data Heap' nat (p :: Maybe nat) (r :: nat) a where

  Empty :: Heap' nat 'Nothing (Zero nat) a

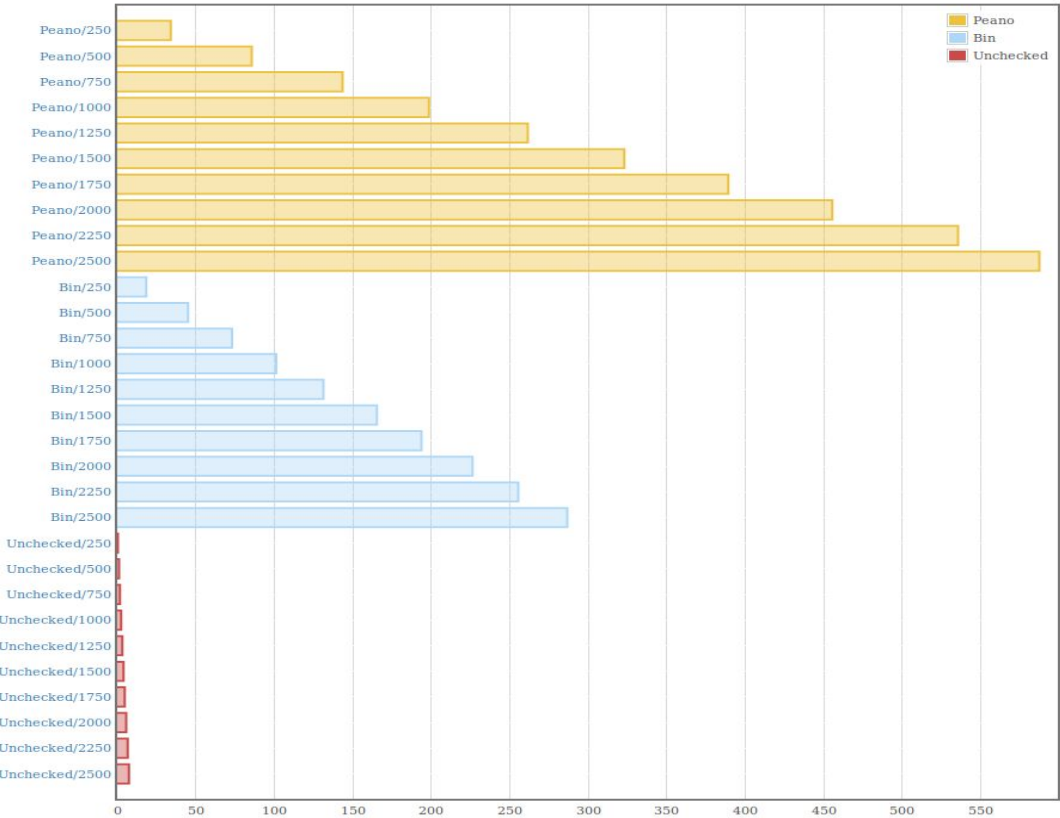
  Node :: ( (p    <=. p')
            , (p    <=. p'')
            , (r'' <=  r')
            )
        => !(Sing nat p)
        -> !(Sing nat (Succ nat r''))
        -> !a
        -> !(Heap' nat p' r' a)
        -> !(Heap' nat p'' r'' a)
        -> Heap' nat ('Just p) (Succ nat r'') a
```

Type-Safe Merging

```
merge :: Nat nat => Heap'' nat p a -> Heap'' nat q a -> Heap'' nat (Min' p q) a
merge (Heap'' Empty) h' = h'
merge h (Heap'' Empty) = h
merge h@(Heap'' (Node p _ x ys zs)) h'@(Heap'' (Node q _ _ _)) =
  alternative (ltGeqDec q p)
    (using (minSymm p q) $ merge h' h) $
  let h'' = merge (Heap'' zs) h'
  in case h'' of
    Heap'' Empty -> error "impossible branch"
    Heap'' h'''@(Node _ r _ _ _) ->
      using (minProd' p (priority zs) (Just' q)) $
        alternative (leqGtDec r $ rank ys)
          (Heap'' $ Node p (succ' r) x ys h''')
          (Heap'' $ Node p (succ' $ rank ys) x h''' ys)
```

Reflection

Benchmark



Type Safety

- Haskell's type system is powerful enough to encode non-trivial invariants on the type-level.
- Haskell functions can serve as “proofs” for statements about types.
- **Caution:** Compiler gives no termination guarantee, so would accept a non-terminating proof.





Questions & Comments