Dependently Typed Heaps

https://github.com/brunjlar/heap
About Me

Lars Brünjes (PhD)

(Pure) Mathematician, Lead Software Architect

based in Regensburg, Germany

Email: brunjal@gmail.com

Github: https://github.com/brunjlar
Agenda

- Motivation
- Leftist Heaps
- Proving Theorems in Haskell
- Dependently Typed Heaps
- Reflection on Results
- 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.

This is not a heap.

Heap property violated!
The **rank** of a heap is the length of its **right spine.**

Rank 2

Rank 3
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 Heap Diagram](image)

- **leftist**: The heap on the left side of the diagram is a leftist heap.
- **not leftist**: The heap on the right side of the diagram violates the leftist property.
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 (Node p _ x ys zs) (Node q __ __) = merge h' h
  | q < p
  | otherwise =
    let h''@ (Node _ r __ _) = merge zs h'
    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)
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)

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

```haskell
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 (legGtDec r $ rank ys)
          (Heap'' $ Node p (succ' r) x ys h'')
          (Heap'' $ Node p (succ' $ rank ys) x h'' ys)
```
Reflection on Results
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