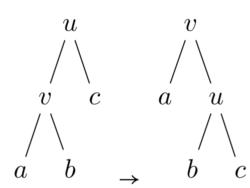
AVL Trees

D Oisín Kidney

July 28, 2018

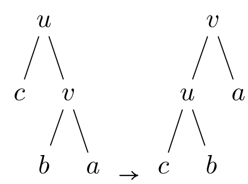
```
open import Relation.Binary
open import Relation.Binary.PropositionalEquality
open import Level using (Lift; lift; _⊔_)
open import Data.Nat as ℕ using (ℕ; suc; zero; pred)
open import Data.Product
open import Data.Unit renaming (⊤ to 1)
open import Data.Maybe
open import Function
open import Data.Bool
open import Data.Empty renaming (⊥ to 0)

module AVL
  {k r} (Key : Set k)
  {_<_ : Rel Key r}
  (isStrictTotalOrder : IsStrictTotalOrder _≡_ _<_)
  where

  open IsStrictTotalOrder isStrictTotalOrder


infix 5 [_]

data ⊤⊥ : Set k where
  ⊥ ⊤ : ⊤⊥
  [_] : (k : Key) → ⊤⊥

infix 4 _⊤⊥<_
_⊤⊥<_ : ⊤⊥ → ⊤⊥ → Set r
⊥     ⊤⊥< ⊥     = Lift r 0
⊥     ⊤⊥< ⊤     = Lift r 1
⊥     ⊤⊥< [ _ ] = Lift r 1
⊤     ⊤⊥< _     = Lift r 0
[ _ ] ⊤⊥< ⊥     = Lift r 0
[ _ ] ⊤⊥< ⊤     = Lift r 1
[ x ] ⊤⊥< [ y ] = x < y
```

```
infix 4 _<_<_

_<_<_ : ⊤⊥ → Key → ⊤⊥ → Set r
l < x < u = l ⊤⊥< [ x ] × [ x ] ⊤⊥< u


module Bounded where

  data ⟨_⊔_⟩≡_ : ℕ → ℕ → ℕ → Set where
    ◿ : ∀ {n} → ⟨ suc n ⊔     n ⟩≡ suc n
    ▽ : ∀ {n} → ⟨     n ⊔     n ⟩≡     n
    ◺ : ∀ {n} → ⟨     n ⊔ suc n ⟩≡ suc n

  ◺⇒◿ : ∀ {x y z} → ⟨ x ⊔ y ⟩≡ z → ⟨ z ⊔ x ⟩≡ z
  ◺⇒◿ ◿ = ▽
  ◺⇒◿ ▽ = ▽
  ◺⇒◿ ◺ = ◿

  ◿⇒◺ : ∀ {x y z} → ⟨ x ⊔ y ⟩≡ z → ⟨ y ⊔ z ⟩≡ z
  ◿⇒◺ ◿ = ◺
  ◿⇒◺ ▽ = ▽
  ◿⇒◺ ◺ = ▽


data Tree {v} (V : Key → Set v) (l u : ⊤⊥) : ℕ → Set (k ⊔ v ⊔ r) where
  leaf  : (l<u : l ⊤⊥< u) → Tree V l u 0
  node : ∀ {h lh rh}
           (k : Key)
           (v : V k)
           (bl : ⟨ lh ⊔ rh ⟩≡ h)
           (lk : Tree V l [ k ] lh)
           (ku : Tree V [ k ] u rh) →
           Tree V l u (suc h)

Inserted : ∀ {v} (V : Key → Set v) (l u : ⊤⊥) (n : ℕ) → Set (k ⊔ v ⊔ r)
Inserted V l u n = ∃[ inc ] (Tree V l u (if inc then suc n else n))

pattern same tr = false , tr
pattern chng tr = true  , tr
```




```
rotʳ : ∀ {lb ub rh v} {V : Key → Set v}
```

```
      → (k : Key)
      → V k
      → Tree V lb [ k ] (suc (suc rh))
      → Tree V [ k ] ub rh
      → Inserted V lb ub (suc (suc rh))
rotʳ u uc (node v vc ⨬ ta tb) tc = same (node v vc ⨪ ta (node u uc ⨪ tb tc))
rotʳ u uc (node v vc ⨪ ta tb) tc = chng (node v vc ⨫ ta (node u uc ⨬ tb tc))
rotʳ u uc (node v vc ⨫ ta (node w wc bw tb tc)) td =
  same (node w wc ⨪ (node v vc (⨫⇒⨬ bw) ta tb) (node u uc (⨬⇒⨫ bw) tc td))
```




```
rotˡ : ∀ {lb ub lh v} {V : Key → Set v}
      → (k : Key)
      → V k
      → Tree V lb [ k ] lh
      → Tree V [ k ] ub (suc (suc lh))
      → Inserted V lb ub (suc (suc lh))
rotˡ u uc tc (node v vc ⨫ tb ta) = same (node v vc ⨪ (node u uc ⨪ tc tb) ta)
rotˡ u uc tc (node v vc ⨪ tb ta) = chng (node v vc ⨬ (node u uc ⨫ tc tb) ta)
rotˡ u uc td (node v vc ⨬ (node w wc bw tc tb) ta) =
  same (node w wc ⨪ (node u uc (⨫⇒⨬ bw) td tc) (node v vc (⨬⇒⨫ bw) tb ta))
```

```
insert : ∀ {l u h v} {V : Key → Set v} (k : Key)
         → V k
         → (V k → V k → V k)
         → Tree V l u h
         → l < k < u
         → Inserted V l u h
insert v vc f (leaf l<u) (l , u) = chng (node v vc ⨪ (leaf l) (leaf u))
insert v vc f (node k kc bl tl tr) prf with compare v k
insert v vc f (node k kc bl tl tr) (l , _)
  | tri< a _ _ with insert v vc f tl (l , a)
... | same tl′ = same (node k kc bl tl′ tr)
... | chng tl′ with bl
...     | ⨬ = rotʳ k kc tl′ tr
...     | ⨪ = chng (node k kc ⨬ tl′ tr)
...     | ⨫ = same (node k kc ⨪ tl′ tr)
insert v vc f (node k kc bl tl tr) _
  | tri≈ _ refl _ = same (node k (f vc kc) bl tl tr)
insert v vc f (node k kc bl tl tr) (_ , u)
```

```
      | tri> _ _ c with insert v vc f tr (c , u)
    ... | same tr′ = same (node k kc bl tl tr′)
    ... | chng tr′ with bl
    ...     | ↖ = same (node k kc ∸ tl tr′)
    ...     | ∸ = chng (node k kc ↘ tl tr′)
    ...     | ↘ = rotˡ k kc tl tr′

  lookup : (k : Key)
         → ∀ {l u s v} {V : Key → Set v}
         → Tree V l u s
         → Maybe (V k)
  lookup k (leaf l<u) = nothing
  lookup k (node v vc _ tl tr) with compare k v
  ... | tri< _ _ _ = lookup k tl
  ... | tri≈ _ refl _ = just vc
  ... | tri> _ _ _ = lookup k tr

module DependantMap where
  data Map {v} (V : Key → Set v) : Set (k ⊔ v ⊔ r) where
    tree : ∀ {h} → Bounded.Tree V ⊥ ⊤ h → Map V

  insertWith : ∀ {v} {V : Key → Set v} (k : Key)
             → V k
             → (V k → V k → V k)
             → Map V
             → Map V
  insertWith k v f (tree tr) =
    tree (proj₂ (Bounded.insert k v f tr (lift tt , lift tt)))

  insert : ∀ {v} {V : Key → Set v} (k : Key) → V k → Map V → Map V
  insert k v = insertWith k v const

  lookup : (k : Key) → ∀ {v} {V : Key → Set v} → Map V → Maybe (V k)
  lookup k (tree tr) = Bounded.lookup k tr

module Map where
  data Map {v} (V : Set v) : Set (k ⊔ v ⊔ r) where
    tree : ∀ {h} → Bounded.Tree (const V) ⊥ ⊤ h → Map V

  insertWith : ∀ {v} {V : Set v} (k : Key) → V → (V → V → V) → Map V → Map V
  insertWith k v f (tree tr) =
    tree (proj₂ (Bounded.insert k v f tr (lift tt , lift tt)))

  insert : ∀ {v} {V : Set v} (k : Key) → V → Map V → Map V
  insert k v = insertWith k v const

  lookup : (k : Key) → ∀ {v} {V : Set v} → Map V → Maybe V
```

```
  lookup k (tree tr) = Bounded.lookup k tr

module Sets where
  data ⟨Set⟩ : Set (k ⊔ r) where
    tree : ∀ {h} → Bounded.Tree (const 1) ⊥ ⊤ h → ⟨Set⟩

  insert : Key → ⟨Set⟩ → ⟨Set⟩
  insert k (tree tr) =
    tree (proj₂ (Bounded.insert k tt const tr (lift tt , lift tt)))

  member : Key → ⟨Set⟩ → Bool
  member k (tree tr) = is-just (Bounded.lookup k tr)
```