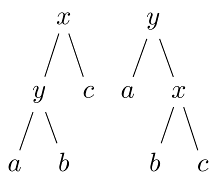
# AVL Trees

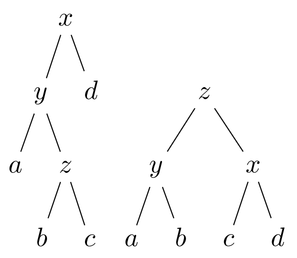
D Oisín Kidney

July 28, 2018

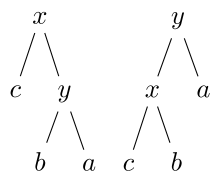
```
open import Relation.Binary
open import Relation.Binary.PropositionalEquality
open import Level using (Lift; lift; _⊔_; lower)
open import Data.Nat as ℕ using (ℕ; suc; zero; pred)
open import Data.Product
open import Data.Unit renaming (⊤ to 1)
open import Data.Maybe
open import Function
open import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

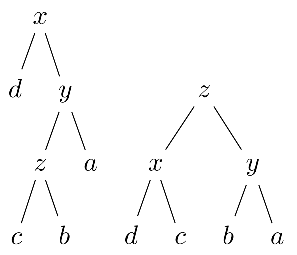
```
x≮⊥ : ∀ {x} → x ⊤⊥< ⊥ → Lift r 0
x≮⊥ {⊥} = lift ∘ lower
x≮⊥ {⊤} = lift ∘ lower
x≮⊥ {[ _ ]} = lift ∘ lower

⊤⊥<-trans : ∀ {x y z} → x ⊤⊥< y → y ⊤⊥< z → x ⊤⊥< z
⊤⊥<-trans {⊥}     {y}     {⊥}     _        y<z = x≮⊥ {x = y} y<z
⊤⊥<-trans {⊥}     {_}     {⊤}     _        _   = _
⊤⊥<-trans {⊥}     {_}     {[ _ ]} _        _   = _
⊤⊥<-trans {⊤}     {_}     {_}     (lift ()) _
⊤⊥<-trans {[ _ ]} {y}     {⊥}     _        y<z = x≮⊥ {x = y} y<z
⊤⊥<-trans {[ _ ]} {_}     {⊤}     _        _   = _
⊤⊥<-trans {[ _ ]} {⊥}     {[ _ ]} (lift ()) _
⊤⊥<-trans {[ _ ]} {⊤}     {[ _ ]} _ (lift ())
⊤⊥<-trans {[ x ]} {[ y ]} {[ z ]} x<y y<z =
  IsStrictTotalOrder.trans isStrictTotalOrder x<y y<z

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)

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

pattern 0+ tr = false , tr
pattern 1+ tr = true , tr
```





```
rot^r : ∀ {lb ub rh v} {V : Key → Set v}
     → (k : Key)
     → V k
     → Tree V lb [ k ] (suc (suc rh))
     → Tree V [ k ] ub rh
     → Altered V lb ub (suc (suc rh))
rot^r x xv (node y yv ⟋ a b) c = 0+ (node y yv ⨪ a (node x xv ⨪ b c))
rot^r x xv (node y yv ⨪ a b) c = 1+ (node y yv ⟍ a (node x xv ⟋ b c))
rot^r x xv (node y yv ⟍ a (node z zv bl b c)) d =
  0+ (node z zv ⨪ (node y yv (⟍⇒⟋ bl) a b) (node x xv (⟋⇒⟍ bl) c d))


rot^l : ∀ {lb ub lh v} {V : Key → Set v}
     → (k : Key)
     → V k
     → Tree V lb [ k ] lh
     → Tree V [ k ] ub (suc (suc lh))
```





```
      → Altered V lb ub (suc (suc lh))
rotˡ x xv c (node y yv ⟍ b a) = 0+ (node y yv ⨪ (node x xv ⨪ c b) a)
rotˡ x xv c (node y yv ⨪ b a) = 1+ (node y yv ⟋ (node x xv ⟍ c b) a)
rotˡ x xv d (node y yv ⟋ (node z zv bl c b) a) =
  0+ (node z zv ⨪ (node x xv (⟍⇒⟋ bl) d c) (node y yv (⟋⇒⟍ bl) b a))


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
        → Altered V l u h
insert v vc f (leaf l<u) (l , u) = 1+ (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)
... | 0+ tl′ = 0+ (node k kc bl tl′ tr)
... | 1+ tl′ with bl
...     | ⟋ = rotʳ k kc tl′ tr
...     | ⨪ = 1+ (node k kc ⟋ tl′ tr)
...     | ⟍ = 0+ (node k kc ⨪ tl′ tr)
insert v vc f (node k kc bl tl tr) _
  | tri≈ _ refl _ = 0+ (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)
... | 0+ tr′ = 0+ (node k kc bl tl tr′)
... | 1+ tr′ with bl
...     | ⟋ = 0+ (node k kc ⨪ tl tr′)
...     | ⨪ = 1+ (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

  record Uncons {v} (V : Key → Set v) (lb : ⊤⊥) (ub : ⊤⊥ ) (h : ℕ) : Set (k ⊔ v ⊔ r) where
    constructor uncons
    field
      head : Key
      val : V head
      l<u : lb ⊤⊥< [ head ]
      tail : Altered V [ head ] ub h

  uncons′ : ∀ {lb ub h lh rh v} {V : Key → Set v}
         → (k : Key)
         → V k
         → ⟨ lh ⊔ rh ⟩≡ h
         → Tree V lb [ k ] lh
         → Tree V [ k ] ub rh
         → Uncons V lb ub h
  uncons′ k v bl tl tr = go k v bl tl tr id
    where
    go : ∀ {lb ub h lh rh v ub′ h′} {V : Key → Set v}
       → (k : Key)
       → V k
       → ⟨ lh ⊔ rh ⟩≡ h
       → Tree V lb [ k ] lh
       → Tree V [ k ] ub rh
       → (∀ {lb′} → Altered V [ lb′ ] ub h → Altered V [ lb′ ] ub′ h′)
       → Uncons V lb ub′ h′
    go k v ⨪ (leaf l<u) tr c = uncons k v l<u (c (0+ tr))
    go k v ⨪ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
      λ { (0+ tl′) → c (1+ (node k v ⋱ tl′ tr))
        ; (1+ tl′) → c (1+ (node k v ⨪ tl′ tr)) }
    go k v ⋱ (leaf l<u) tr c = uncons k v l<u (c (0+ tr))
    go k v ⋱ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
      λ { (0+ tl′) → c (rotˡ k v tl′ tr)
        ; (1+ tl′) → c (1+ (node k v ⋱ tl′ tr)) }
    go k v ⋰ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
```

```
      λ { (0+ tl′) → c (0+ (node k v ⨪ tl′ tr))
        ; (1+ tl′) → c (1+ (node k v ⟋ tl′ tr))}

  widen : ∀ {lb ub ub′ h v} {V : Key → Set v}
    → ub ⊤⊥< ub′
    → Tree V lb ub h
    → Tree V lb ub′ h
  widen {lb} ub<ub′ (leaf l<u) = leaf (⊤⊥<-trans {lb} l<u ub<ub′)
  widen ub<ub′ (node k v bl tl tr) = node k v bl tl (widen ub<ub′ tr)

  delete : ∀ {lb ub h v} {V : Key → Set v}
         → (k : Key)
         → Tree V lb ub (suc h)
         → Altered V lb ub h
  delete k (node k₁ v bl tl tr) with compare k k₁
  delete k (node {lh = zero}   k₁ v bl tl tr) | tri< _ _ _ = 1+ (node k₁ v bl tl tr)
  delete k (node {lh = suc lh} k₁ v bl tl tr) | tri< _ _ _ with delete k tl | bl
  ... | 0+ tl′ | ⟋ = 0+ (node k₁ v ⨪ tl′ tr)
  ... | 0+ tl′ | ⨪ = 1+ (node k₁ v ⟍ tl′ tr)
  ... | 0+ tl′ | ⟍ = rotˡ k₁ v tl′ tr
  ... | 1+ tl′ | _ = 1+ (node k₁ v bl tl′ tr)
  delete {lb} k (node {rh = zero} k v bl tl (leaf k<ub)) | tri≈ _ refl _ with bl | tl
  ... | ⟋ | _ = 0+ (widen k<ub tl)
  ... | ⨪ | leaf lb<k = 0+ (leaf (⊤⊥<-trans {lb} lb<k k<ub))
  delete k (node {rh = suc rh} k v bl tl (node kᵣ vᵣ blᵣ tlᵣ trᵣ)) | tri≈ _ refl _
    with bl | uncons′ kᵣ vᵣ blᵣ tlᵣ trᵣ
  ... | ⟋ | uncons k′ v′ l<u (0+ tr′) = rotʳ k′ v′ (widen l<u tl) tr′
  ... | ⟋ | uncons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ⟋ (widen l<u tl) tr′)
  ... | ⨪ | uncons k′ v′ l<u (0+ tr′) = 1+ (node k′ v′ ⟋ (widen l<u tl) tr′)
  ... | ⨪ | uncons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ⨪ (widen l<u tl) tr′)
  ... | ⟍ | uncons k′ v′ l<u (0+ tr′) = 0+ (node k′ v′ ⨪ (widen l<u tl) tr′)
  ... | ⟍ | uncons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ⟍ (widen l<u tl) tr′)
  delete k (node {rh = zero} k₁ v bl tl tr) | tri> _ ¬b c = 1+ (node k₁ v bl tl tr)
  delete k (node {rh = suc rh} k₁ v bl tl tr) | tri> _ ¬b c with delete k tr | bl
  ... | 0+ tr′ | ⟋ = rotʳ k₁ v tl tr′
  ... | 0+ tr′ | ⨪ = 1+ (node k₁ v ⟋ tl tr′)
  ... | 0+ tr′ | ⟍ = 0+ (node k₁ v ⨪ tl tr′)
  ... | 1+ tr′ | _ = 1+ (node k₁ v bl tl 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)
```