AVL Trees

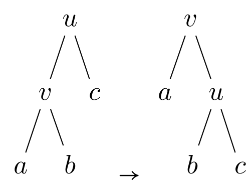
D Oisín Kidney

July 28, 2018

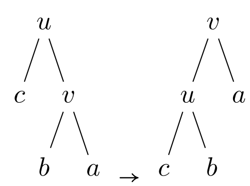
```agda
open import Relation.Binary
open import Relation.Binary.PropositionalEquality
open import Level using (Lift; lift; _⊔_; lower)
open import Data.Nat as ℕ using (ℕ; suc; zero; pred)
open import Data.Product
open import Data.Unit renaming (⊤ to 1)
open import Data.Maybe
open import Function
open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



```
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
     → Altered V lb ub (suc (suc rh))
rotʳ u uc (node v vc ⟋ ta tb) tc = 0+ (node v vc ⨪ ta (node u uc ⨪ tb tc))
rotʳ u uc (node v vc ⨪ ta tb) tc = 1+ (node v vc ⟍ ta (node u uc ⟋ tb tc))
rotʳ u uc (node v vc ⟍ ta (node w wc bw tb tc)) td =
  0+ (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))
     → Altered V lb ub (suc (suc lh))
rotˡ u uc tc (node v vc ⟍ tb ta) = 0+ (node v vc ⨪ (node u uc ⨪ tc tb) ta)
rotˡ u uc tc (node v vc ⨪ tb ta) = 1+ (node v vc ⟋ (node u uc ⟍ tc tb) ta)
rotˡ u uc td (node v vc ⟋ (node w wc bw tc tb) ta) =
  0+ (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
            → 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
          → ∃[ lb′ ] (V lb′ × lb ⊤⊥< [ lb′ ] × Altered V [ lb′ ] ub h)
uncons′ k v ∸ (leaf l<u) tr = k , v , l<u , 0+ tr
uncons′ k v ∸ (node k₁ v₁ bl tl₁ tr₁) tr with uncons′ k₁ v₁ bl tl₁ tr₁
... | k′ , v′ , l<u′ , 0+ tl′ = k′ , v′ , l<u′ , 1+ (node k v ⋱ tl′ tr)
... | k′ , v′ , l<u′ , 1+ tl′ = k′ , v′ , l<u′ , 1+ (node k v ∸ tl′ tr)
uncons′ k v ⋱ (leaf l<u) tr = k , v , l<u , 0+ tr
uncons′ k v ⋱ (node k₁ v₁ bl tl₁ tr₁) tr with uncons′ k₁ v₁ bl tl₁ tr₁
... | k′ , v′ , l<u′ , 0+ tl′ = k′ , v′ , l<u′ , rotˡ k v tl′ tr
... | k′ , v′ , l<u′ , 1+ tl′ = k′ , v′ , l<u′ , 1+ (node k v ⋱ tl′ tr)
uncons′ k v ⋰ (node k₁ v₁ bl tl₁ tr₁) tr with uncons′ k₁ v₁ bl tl₁ tr₁
... | k′ , v′ , l<u′ , 0+ tl′ = k′ , v′ , l<u′ , 0+ (node k v ∸ tl′ tr)
... | k′ , v′ , l<u′ , 1+ tl′ = k′ , v′ , l<u′ , 1+ (node k v ⋰ tl′ tr)

snoc : ∀ {lb ub ub′ h v} {V : Key → Set v}
        → ub ⊤⊥< ub′
        → Tree V lb ub h
        → Tree V lb ub′ h
snoc {lb} ub<ub′ (leaf l<u) = leaf (⊤⊥<-trans {lb} l<u ub<ub′)
snoc ub<ub′ (node k v bl tl tr) = node k v bl tl (snoc 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< a ¬b ¬c = 1+ (node k₁ v bl tl tr)
delete k (node {lh = suc lh} k₁ v bl tl tr) | tri< a ¬b ¬c with delete k tl
delete k (node {_} {suc lh} k₁ v ⋰ tl tr) | tri< a ¬b ¬c | 0+ tl′ = 0+ (node k₁ v ∸ tl′ tr)
delete k (node {_} {suc lh} k₁ v ∸ tl tr) | tri< a ¬b ¬c | 0+ tl′ = 1+ (node k₁ v ⋱ tl′ tr)
delete k (node {_} {suc lh} k₁ v ⋱ tl tr) | tri< a ¬b ¬c | 0+ tl′ = rotˡ k₁ v tl′ tr
delete k (node {_} {suc lh} k₁ v bl tl tr) | tri< a ¬b ¬c | 1+ tl′ = 1+ (node k₁ v bl tl′ tr)
delete k₁ (node {rh = zero} k₁ v ⋰ tl (leaf l<u)) | tri≈ ¬a refl ¬c = 0+ (snoc l<u tl)
delete {lb} k₁ (node {rh = zero} k₁ v ∸ (leaf l<u) (leaf l<u₁)) | tri≈ ¬a refl ¬c = 0+ (leaf (⊤⊥<-trans {lb}
delete k₁ (node {rh = suc rh} k₁ v ⋰ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c with uncons′ k v₁ bl tr tr₁
delete k₁ (node {_} {_} {suc rh} k₁ v ⋰ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 0+ tr′
delete k₁ (node {_} {_} {suc rh} k₁ v ⋰ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 1+ tr′
delete k₁ (node {rh = suc rh} k₁ v ∸ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c with uncons′ k v₁ bl tr tr₁
delete k₁ (node {rh = suc rh} k₁ v ∸ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 0+ tr′ = 1+
delete k₁ (node {rh = suc rh} k₁ v ∸ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 1+ tr′ = 1+
delete k₁ (node {rh = suc rh} k₁ v ⋱ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c with uncons′ k v₁ bl tr tr₁
delete k₁ (node {rh = suc rh} k₁ v ⋱ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 0+ tr′ = 0-
delete k₁ (node {rh = suc rh} k₁ v ⋱ tl (node k v₁ bl tr tr₁)) | tri≈ ¬a refl ¬c | k′ , v′ , l<u , 1+ tr′ = 1-
```

```
  delete k (node {rh = zero} k₁ v bl tl tr) | tri> ¬a ¬b c = 1+ (node k₁ v bl tl tr)
  delete k (node {rh = suc rh} k₁ v bl tl tr) | tri> ¬a ¬b c with delete k tr
  delete k (node {lh = _} {suc rh} k₁ v ⟋ tl tr) | tri> ¬a ¬b c | 0+ tr′ = rotʳ k₁ v tl tr′
  delete k (node {lh = _} {suc rh} k₁ v ∸ tl tr) | tri> ¬a ¬b c | 0+ tr′ = 1+ (node k₁ v ⟋ tl tr′)
  delete k (node {lh = _} {suc rh} k₁ v ⟍ tl tr) | tri> ¬a ¬b c | 0+ tr′ = 0+ (node k₁ v ∸ tl tr′)
  delete k (node {lh = _} {suc rh} k₁ v bl tl tr) | tri> ¬a ¬b c | 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)))
```

$\mathsf{member} : Key \to \langle\mathsf{Set}\rangle \to \mathsf{Bool}$
$\mathsf{member}\ k\ (\mathsf{tree}\ tr) = \mathsf{is\text{-}just}\ (\mathsf{Bounded.lookup}\ k\ tr)$