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)
```
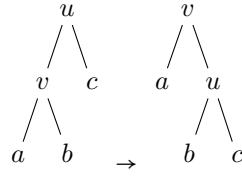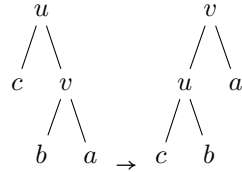
2

$(lk : \textsf{Tree } V\ l\ [\ k\ ]\ lh)$
$(ku : \textsf{Tree } V\ [\ k\ ]\ u\ rh) \to$
$\textsf{Tree } V\ l\ u\ (\textsf{suc } h)$

$\textsf{Altered} : \forall\ \{v\}\ (V : Key \to \textsf{Set } v)\ (l\ u : \lceil_\bot)\ (n : \mathbb{N}) \to \textsf{Set } (k \sqcup v \sqcup r)$
$\textsf{Altered } V\ l\ u\ n = \exists[\ inc\ ]\ (\textsf{Tree } V\ l\ u\ (\textsf{if } inc \textsf{ then suc } n \textsf{ else } n))$

$\textsf{pattern } 0+\ tr = \textsf{false}\ ,\ tr$
$\textsf{pattern } 1+\ tr = \textsf{true}\ ,\ tr$

$$
\begin{array}{ccc}
u & & v \\
/\ \backslash & & /\ \backslash \\
v\quad c & & a\quad u \\
/\ \backslash & & /\ \backslash \\
a\quad b & \to & b\quad c
\end{array}
$$

$\textsf{rot}^r : \forall\ \{lb\ ub\ rh\ v\}\ \{V : Key \to \textsf{Set } v\}$
$\quad \to (k : Key)$
$\quad \to V\ k$
$\quad \to \textsf{Tree } V\ lb\ [\ k\ ]\ (\textsf{suc } (\textsf{suc } rh))$
$\quad \to \textsf{Tree } V\ [\ k\ ]\ ub\ rh$
$\quad \to \textsf{Altered } V\ lb\ ub\ (\textsf{suc } (\textsf{suc } rh))$
$\textsf{rot}^r\ u\ uc\ (\textsf{node } v\ vc\ {\scriptstyle\nearrow}\ ta\ tb)\ tc = 0+\ (\textsf{node } v\ vc\ {\scriptstyle\div}\ ta\ (\textsf{node } u\ uc\ {\scriptstyle\div}\ tb\ tc))$
$\textsf{rot}^r\ u\ uc\ (\textsf{node } v\ vc\ {\scriptstyle\div}\ ta\ tb)\ tc = 1+\ (\textsf{node } v\ vc\ {\scriptstyle\searrow}\ ta\ (\textsf{node } u\ uc\ {\scriptstyle\nearrow}\ tb\ tc))$
$\textsf{rot}^r\ u\ uc\ (\textsf{node } v\ vc\ {\scriptstyle\searrow}\ ta\ (\textsf{node } w\ wc\ bw\ tb\ tc))\ td =$
$\quad 0+\ (\textsf{node } w\ wc\ {\scriptstyle\div}\ (\textsf{node } v\ vc\ ({\scriptstyle\searrow\Rightarrow\nearrow}\ bw)\ ta\ tb)\ (\textsf{node } u\ uc\ ({\scriptstyle\nearrow\Rightarrow\searrow}\ bw)\ tc\ td))$

$$
\begin{array}{ccc}
u & & v \\
/\ \backslash & & /\ \backslash \\
c\quad v & & u\quad a \\
/\ \backslash & & /\ \backslash \\
b\quad a & \to & c\quad b
\end{array}
$$

$\textsf{rot}^l : \forall\ \{lb\ ub\ lh\ v\}\ \{V : Key \to \textsf{Set } v\}$
$\quad \to (k : Key)$
$\quad \to V\ k$
$\quad \to \textsf{Tree } V\ lb\ [\ k\ ]\ lh$
$\quad \to \textsf{Tree } V\ [\ k\ ]\ ub\ (\textsf{suc } (\textsf{suc } lh))$
$\quad \to \textsf{Altered } V\ lb\ ub\ (\textsf{suc } (\textsf{suc } lh))$
$\textsf{rot}^l\ u\ uc\ tc\ (\textsf{node } v\ vc\ {\scriptstyle\searrow}\ tb\ ta) = 0+\ (\textsf{node } v\ vc\ {\scriptstyle\div}\ (\textsf{node } u\ uc\ {\scriptstyle\div}\ tc\ tb)\ ta)$
$\textsf{rot}^l\ u\ uc\ tc\ (\textsf{node } v\ vc\ {\scriptstyle\div}\ tb\ ta) = 1+\ (\textsf{node } v\ vc\ {\scriptstyle\nearrow}\ (\textsf{node } u\ uc\ {\scriptstyle\searrow}\ tc\ tb)\ ta)$
$\textsf{rot}^l\ u\ uc\ td\ (\textsf{node } v\ vc\ {\scriptstyle\nearrow}\ (\textsf{node } w\ wc\ bw\ tc\ tb)\ ta) =$
$\quad 0+\ (\textsf{node } w\ wc\ {\scriptstyle\div}\ (\textsf{node } u\ uc\ ({\scriptstyle\searrow\Rightarrow\nearrow}\ bw)\ td\ tc)\ (\textsf{node } v\ vc\ ({\scriptstyle\nearrow\Rightarrow\searrow}\ bw)\ tb\ ta))$

$\textsf{insert} : \forall\ \{l\ u\ h\ v\}\ \{V : Key \to \textsf{Set } v\}\ (k : Key)$

3

```
                  → 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
```

4

```
                    → ⟨ 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 k₁ v₁ bl tl₁ tr₁) tr c = go k₁ v₁ bl tl₁ tr₁
    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 k₁ v₁ bl tl₁ tr₁) tr c = go k₁ v₁ bl tl₁ tr₁
    λ { (0+ tl′) → c (rotˡ k v tl′ tr)
      ; (1+ tl′) → c (1+ (node k v ↘ tl′ tr)) }
  go k v ↗ (node k₁ v₁ bl tl₁ tr₁) tr c = go k₁ v₁ bl tl₁ tr₁
    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< 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 | 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 k₁ (node {rh = zero} k₁ v ↗ tl (leaf l<u)) | tri≈ ¬a refl ¬c = 0+ (widen 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}
```

5

```
delete k₁ (node {rh = suc rh} k₁ v bl tl (node k v₁ bl₁ tr tr₁)) | tri≈ ¬a refl ¬c with bl | uncons′ k v₁ bl₁ ⋮
... | ⋰ | 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> ¬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 | 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)
```