The free monoid on a type

Fosco Loregian

January 18, 2021

Recall that a *monoid* is a set endowed with an operation that is associative and has an identity element; we now want to define this (and others) property and prove that the `List` construction gives a monoid `List A` for every object/type `A`.

We start with a boilerplate declaration or Agda will complain:

```
module Monoidi where

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; sym)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _■)
```

Then we define the data type and the constructors for our wannabe monoid: lists of elements on A are either the empty tuple, or are construed inductively from an element `a : A` and a list `as : List A`, joined together with a `::` (pron. *cons*):

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

This definition matches the usual behaviour of lists that you might have known to love-hate from Haskell: the term `[1,2,3]` consists of `1 :: 2 :: 3 :: []`. One can define `head`, `tail`, etc. in the usual way; no heterogeneous lists, etc.

But this doesn't have to become a lesson on Functional Programming! There's another course for that. Instead, let's concentrate on the functions that will be useful for our proof: lists can be joined with the `++` operation (pron.: *concat*):

```
_++_ : {A : Set} → List A → List A → List A
[] ++ v = v
(u :: us) ++ v = u :: (us ++ v)
```

Easy peasy: concatenating an empty list with `v` yields just `v`, and the concat of an inhabited list with `v` is just the head of the first, cons the concat of its tail with `v`.

For the sake of completeness now let's define *properties* that operations might enjoy: the good ol' associativity and commutativity for an operation $A \times A \to A$.

What is beautiful about these declarations is that they define types that can be read as proofs that some propositions are true! The associative property says that

```
Associative :
  {A : Set} →
  -- for all given type A
  (f : A → A → A) → Set
  -- 'the function ∘ :  A × A → A
Associative {A} f =
  -- is associative' means that
  ∀ {x y z : A} → f (f x y) z ≡ f x (f y z)
  -- if (x y z :  A), then x ∘ (y ∘ z) = (x ∘ y) ∘ z
```

One interprets commutativity in a similar way.

```
Commutative : ∀ {A} → (A → A → A) → Set
Commutative {A} f = ∀ {x y : A} → f x y ≡ f y x
```

Now, a semigroup is just a set endowed with an associative binary operation:

```
record IsSemigroup {A} (_∘_ : A → A → A) : Set where
  field
    assoc : Associative {A} (_∘_)

record HasIdentity {A} (_∘_ : A → A → A) (z : A) : Set where
  field
    lId : {a : A} → z ∘ a ≡ a
    rId : {a : A} → a ∘ z ≡ a

record IsMonoid (A : Set) (_∘_ : A → A → A) (z : A) : Set where
  field
    isSemigroup : IsSemigroup {A} (_∘_)
    hasIdentity : HasIdentity {A} (_∘_) z
```

For the sake of completeness, let's state also the definition of a commutative ('abelian') monoid:

```
record IsAbelianMonoid (A : Set) (_∘_ : A → A → A) (z : A) : Set where
  field
    isMonoid : IsMonoid A (_∘_) z
    isAbelian : Commutative (_∘_)
```

1 Proof

Now for the proof that List A is a monoid for every A (in poorer typesetting, alas; also, note that the highlighted lines might need further refinement):

```
proof : ∀ {A : Set} → IsMonoid (List A) (_++_) ([])
proof {A} =
  record
    { isSemigroup =
        record { assoc = asso-proof }
    ; hasIdentity =
        record { lId = refl
               ; rId = rId-proof {A}
               }
    }
    where
      rId-proof : {A : Set} (a : List A) → (a ++ []) ≡ a
      rId-proof {A} [] = refl
      rId-proof {A} (a :: as) =
        begin
          (a :: as) ++ []
            ≡⟨ cong (\x → a :: x) (rId-proof {A} as) ⟩
          (a :: as)
        ■
      emma : {a b c : List A} → (x : A) →
        (x :: (a ++ b)) ++ c ≡ ((x :: a) ++ b) ++ c
      emma {[]} {b} {c} x = refl
      emma {(a :: as)} {b} {c} x =
        begin
          (x :: ((a :: as) ++ b)) ++ c
            ≡⟨ cong (\t → (x :: t) ++ c) refl ⟩
          (x :: (a :: (as ++ b))) ++ c
            ≡⟨ cong ((\t → (x :: t) ++ c)) refl ⟩
          (x :: ((a :: as) ++ b)) ++ c
        ■
      asso-proof : Associative {List A} (_++_)
      asso-proof {[]} {y} {z} = refl
      asso-proof {x :: xs} {[]} {z} =
        begin
          ((x :: xs) ++ []) ++ z
            ≡⟨ cong (\t → t ++ z) (rId-proof {A} (x :: xs)) ⟩
          (x :: xs) ++ ([] ++ z)
        ■
      asso-proof {x :: xs} {y :: ys} {[]} =
        begin
          ((x :: xs) ++ (y :: ys)) ++ []
            ≡⟨ rId-proof {A} ((x :: xs) ++ (y :: ys)) ⟩
          (x :: xs) ++ (y :: ys)
            ≡⟨ cong (\t → (x :: xs) ++ t) (sym (rId-proof {A} (y :: ys))) ⟩
          (x :: xs) ++ ((y :: ys) ++ [])
        ■
      asso-proof {x :: xs} {y} {z} =
        begin
          ((x :: xs) ++ y) ++ z
            ≡⟨ cong (\t → t ++ z) refl ⟩
```

```
    (x :: (xs ++ y)) ++ z
      ≡⟨ emma {xs} {y} {z} x ⟩
    x :: ((xs ++ y) ++ z)
      ≡⟨ cong (\t → x :: t) (asso-proof {xs} {y} {z}) ⟩
    x :: (xs ++ (y ++ z))
      ≡⟨ refl ⟩
    (x :: xs) ++ (y ++ z)
■
```