```
module GaleShapley where

open import Agda.Builtin.String
open import Agda.Builtin.Equality
open import Data.Empty
open import Data.Maybe
open import Data.Nat
open import Data.Nat.Properties
open import Data.Product
open import Data.Bool
open import Data.List
open import Data.Sum
open import Relation.Nullary
open import Induction.WellFounded
open import Relation.Binary.PropositionalEquality
import Data.Nat.Solver
open Data.Nat.Solver.+-*-Solver
  using (prove; solve; _:=_; con; var; _:+_; _:*_; :-_; _:-_)

-- lem2 : (4 + 6  10)
-- lem2 = solve 0 (con 4 :+ con 6 := con 10) refl

lem3 : (x : ) → (2 * (x + 4)  8 + 2 * x)
lem3 = solve 1 (λ x' → con 2 :* (x' :+ con 4) := con 8 :+ con 2 :* x') refl

infix 3 _>just_
infix 4 _from>_

lengthPrefs : List ( × List ) →
lengthPrefs [] = 0
lengthPrefs ((_ , l)  xs) = length l + lengthPrefs xs

compSumPrefLists : (freeMen engagedMen : List ( × List ) ) →
compSumPrefLists freeMen engagedMen = lengthPrefs freeMen + lengthPrefs engagedMen

record MatchingState : Set where
  constructor mkState
  field
    men : List ( × List )
    freeMen : List ( × List )
    engagedMen : List ( × List )
    women : List ( × List )
    couples : List ( × )
    sumPrefLists :
    sumEq : sumPrefLists  lengthPrefs freeMen + lengthPrefs engagedMen
```

```
-- Code by @yeputons on Stack Overflow :D
is- :  →  → Bool
is- a b with a ? b
... | Dec.yes _ = true
... | Dec.no _ = false

-- Helper function to search for an element in a list of natural numbers.
positionInListHelper :  → List  →  → Maybe
positionInListHelper x [] _ = nothing --searched everywhere but couldn't find it!
positionInListHelper x (xs  xss) zero with compare x xs
... | equal _ = just zero --found man at the tip of the list!
... | _       = positionInListHelper x xss (suc zero) --accumulate and keep searching...
positionInListHelper x (xs  xss) (suc n) with compare x xs
... | equal _ = just (suc n) --found man somewhere in the list
... | _       = positionInListHelper x xss (suc (suc n)) --accumulate and keep searching

positionInList :  → List  → Maybe
positionInList n ns = positionInListHelper n ns zero

-- m is the proposing man
-- h is the current husband of the woman
-- prefs is the preference list of the woman
-- returns true if the woman prefers m over h
propose : (m : )(h : )(prefs : List ) → Bool
-- Woman compares
propose man h preferenceList with positionInList man preferenceList | positionInList h preferenceList
... | just p | just q   = is- p q --does she prefer the new guy to the current one?
... | just p | nothing = false  --shouldn't happen in an ideal world : woman received a proposa
... | nothing | just q = false  --shouldn't happen in an ideal world : woman married an unknowr
... | nothing | nothing = false --shouldn't happen in an ideal world : who are these men??

-- We assume the "husbands" to be the proposing side,
-- therefore if women propose the pair looks like (wife, husband)
getHusband :  → List ( × ) → Maybe
getHusband woman [] = nothing --not married yet and this is the first proposal in the algo
getHusband woman ((m , w)  []) with compare woman w
... | equal _ = just m --found your husband!
... | _      = nothing --not married yet
getHusband woman ((m , w)  (c  cs)) with compare woman w
... | equal _ = just m --found your husband!
... | _      = getHusband woman (c  cs) --keep searching

getWife :  → List ( × ) → Maybe
getWife man [] = nothing
getWife man ((m , w)  []) with compare man m
... | equal _ = just w --found your wife!
... | _      = nothing --not married yet
```

```
getWife man ((m , w)  (c  cs)) with compare man m
... | equal _ = just w --found your wife!
... | _       = getHusband man (c  cs) --keep searching

-- Simply extract a preference list from the scheme of indexed lists.
getPreferenceList :  → List ( × List ) → List
getPreferenceList person [] = [] --dummy case
getPreferenceList person ((p , preferences)  ps) with compare person p
... | equal _ = preferences
... | _       = getPreferenceList person ps

-- Safely adding couples : previous marriages are unmade
safeAddNewCouple : (newCouple :  × )(previousCouples : List ( × )) → List ( × )
safeAddNewCouple (m , w) [] = (m , w)  []
safeAddNewCouple (m , w) ((a , b)  []) with compare w b
... | equal _ = (m , w)  []
... | _       = (m , w)  (a , b)  []
safeAddNewCouple (m , w) ((a , b)  (c  cs)) with compare w b
... | equal _ = (m , w)  c  cs
... | _       = (a , b)  safeAddNewCouple (m , w) (c  cs)

p : (l : List ( × ))(a : ) → a  3 → safeAddNewCouple (2 , 3) ((1 , a)  l)  ((2 , a)  l)
p [] a e with compare 3 a
p [] zero () | w
p [] (suc zero) () | w
p [] (suc (suc zero)) () | w
p [] (suc (suc (suc zero))) e | equal .3 = refl
p [] (suc (suc (suc (suc a)))) () | w
p (x  l) zero ()
p (x  l) (suc .2) refl = refl

-- Safely adding new engaged men to the list : dumped man is removed
safeAddNewEngagedMan : (newEngagedMan : ( × List ))(prevFiance : )(prevEngagedMen : List ( × List
-- Dummy case: this function is only invoked if a woman is already married, so the list
safeAddNewEngagedMan (newFiance , prefs) prevFiance [] = ((newFiance , prefs)  [] , (0 , []))

safeAddNewEngagedMan (newFiance , prefs) prevFiance ((m , prefsM)  []) with compare prevFiance m
... | equal _ = ((newFiance , prefs)  [] , (m , prefsM)) --kick him out!
... | _       = (((newFiance , prefs)  (m , prefsM)  []) , (0 , [])) --safe to keep after all
safeAddNewEngagedMan (newFiance , prefs) prevFiance ((m , prefsM)  ms  engagedMen) with compare p
... | equal _ = ((newFiance , prefs)  ms  engagedMen , (m , prefsM)) --kick him out!
... | _       = (m , prefsM)  (proj (safeAddNewEngagedMan (newFiance , prefs) prevFiance (ms  engagedM

step : MatchingState → MatchingState
-- When there are no more free men, the matching is stable and this is the last step.
step (mkState men [] engagedMen women couples k p) = mkState men [] engagedMen women couples k p
```

```
-- Dummy case : the function shouldn't really be invoked with a man with empty preferen
-- But otherwise Agda would question the completeness of our pattern matching.
step (mkState men ((n , [])  freeMen) engagedMen women couples k p) = mkState men ((n , [])  freeMen

-- Proposal step
step (mkState men ((n , w  prefs)  freeMen) engagedMen women couples k p) with getHusband w couples
... | just h with propose n h (getPreferenceList w women) --Woman has a husband, represented by hi
... | true = mkState men freeMenUpdated engagedMenUpdated women (safeAddNewCouple (n , w) couples
          where
            freeMenUpdated = proj_2 (safeAddNewEngagedMan (n , prefs) h engagedMen)  freeMen --(
            engagedMenUpdated = proj (safeAddNewEngagedMan (n , prefs) h engagedMen)
... | false = mkState men ((n , prefs)  freeMen) engagedMen women couples (compSumPrefLists ((n , pref
-- Woman didn't have a husband yet (represented by zero) : must accept proposal
step (mkState men ((n , w  prefs)  freeMen) engagedMen women couples k p) | nothing = mkState men fr
                                                                                                   (com

+-zero :   n k → k  n + 0 → k  n
+-zero zero zero p = p
+-zero zero (suc k) p = p
+-zero (suc n) zero ()
+-zero (suc n) (suc .(n + 0)) refl = cong suc (+-right-identity n)

+-move-zero :   n k → k  n + 0 → n  k + 0
+-move-zero zero zero refl = refl
+-move-zero zero (suc k) ()
+-move-zero (suc n) zero ()
+-move-zero (suc n) (suc .(n + 0)) refl = cong suc (+-move-zero n (n + 0) refl)

nn+m :   m n → n  n + m
nn+m zero zero = -refl
nn+m zero (suc m) = -reflexive (cong suc (sym (+-right-identity m)))
nn+m (suc n) zero = zn
nn+m (suc n) (suc m) = ss (nn+m (suc n) m)

lengthPrefsOneSide :   (freeMen engagedMen : List (  × List  ))(k :  ) → k  compSumPrefLists freeMen engag
lengthPrefsOneSide [] [] k p = zn , zn
lengthPrefsOneSide [] ((fst , [])  engagedMen) k p = zn , proj_2 (lengthPrefsOneSide [] engagedMen k p)
lengthPrefsOneSide [] ((fst , x  snd)  engagedMen) k p = zn , -reflexive (sym p)
lengthPrefsOneSide ((fst , [])  freeMen) [] k p = -reflexive (+-zero k (lengthPrefs freeMen) (+-move-zero (l
lengthPrefsOneSide ((fst , x  snd)  freeMen) [] k p = -reflexive (+-zero k (lengthPrefs ((fst , x  snd)  freeM
lengthPrefsOneSide ((fst , [])  freeMen) ((fst , [])  engagedMen) k p = lengthPrefsOneSide freeMen engage
lengthPrefsOneSide ((fst , [])  freeMen) ((fst , x  snd)  engagedMen) .(lengthPrefs freeMen + suc (lengthPr
lengthPrefsOneSide ((fst , x  snd)  freeMen) ((fst , [])  engagedMen) .(suc (lengthPrefs ((fst , snd)  freeMe
lengthPrefsOneSide ((fst , x  snd)  freeMen) ((fst , x  snd)  engagedMen) p refl = nn+m (suc (lengthPrefs

stepsWithPrefs :   m n → m  n → m  1 + n
stepsWithPrefs zero zero zn = zn
```

```
stepsWithPrefs zero (suc n) zn = zn
stepsWithPrefs (suc m) zero ()
stepsWithPrefs (suc m) (suc n) (ss p) = ss (stepsWithPrefs m n p)
```

lengthPrefsExtLemma : (x : × List)(xs : List (× List))(n :) → lengthPrefs xs n + lengthPrefs (x

```
-- Proofs! :D

-- The type of elements that belong to a list.
data ___ {A : Set}(a : A) : List A → Set where
  now : (as : List A) → a (a as)
  later : {a' : A}{as : List A} → a as → a (a' as)

-- Bigger than comparison for Maybe -typed elements.
data _>just_ : Maybe → Maybe → Set where
  _from>_ : {m n : } → m > n → just m >just just n
```

– Given a man and a woman and their preferences, the condition of stability is satisfied if – another m' and v

```
leftinv : (a : ) → zero + a a
leftinv a = refl

rightinv : (a : ) → a + zero a
rightinv zero = refl
rightinv (suc a) = cong suc (rightinv a)
```