```
{-
    Agda in a hurry
    ---------------

    Martín Hötzel Escardó
    5th October 2017, updated 10th November 2020 and 9th December 2021.

    An html rendering with syntax highlighting and internal links is available at
    https://www.cs.bham.ac.uk/~mhe/fp-learning-2017-2018/html/Agda-in-a-Hurry.html

    This is a brief introduction to Agda, originally written for
    (Haskell) functional programming students.

    Agda is a language similar to Haskell, but it includes dependent
    types, which have many uses, including writing specifications of
    functional programs.

    http://wiki.portal.chalmers.se/agda/pmwiki.php
    http://agda.readthedocs.io/en/latest/

    This tutorial doesn't cover the interactive construction of types
    and programs (and hence propositions and proofs).

    This file is also an introduction to dependent types.

    Organization:

      1. We first develop our own mini-library.

      2. We then discuss how to encode propositions as types in Agda.

      3. We then discuss list reversal, in particular the correctness
         of the "clever" algorithm that runs in linear, rather than
         quadratic, time.

      4. We conclude by giving a non-trivial example of the
         specification and proof of a non-trivial functional program
         with binary trees. This program is a direct translation of a
         Haskell program.

-}

{-

    What Agda calls sets is what we normally call types in programming:

-}

Type = Set

{-

    The type of booleans:

-}

data Bool : Type where
  False True : Bool

if_then_else_ : {A : Type} → Bool → A → A → A
if False then x else y = y
if True  then x else y = x

{-
    Curly braces "{" and "}" are for implicit arguments, that we don't
    mention explictly, provided Agda can infer them. If it cannot, we
    have to write them explicitly when we call the function, enclosed
    in curly braces.

    http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.ImplicitArguments
```

```
    There is a slightly more general version of if_then_else_, using
    dependent types, which says that if we have P False and P True,
    then we have P b for any given b : Bool:
-}

Bool-induction : {P : Bool → Type}

              → P False
              → P True
              --------------------
              → (b : Bool) → P b

Bool-induction x y False = x
Bool-induction x y True  = y


data Maybe (A : Type) : Type where
  Nothing : Maybe A
  Just    : A → Maybe A


Maybe-induction : {A : Type} {P : Maybe A → Type}

               → P Nothing
               → ((x : A) → P(Just x))
               ------------------------
               → (m : Maybe A) → P m

Maybe-induction p f Nothing  = p
Maybe-induction p f (Just x) = f x

{-
    This corresponds to Haskell's Either type constructor:
-}

data _+_ (A B : Type) : Type where
  inl : A → A + B
  inr : B → A + B


+-induction : {A B : Type} {P : A + B → Type}

            → ((x : A) → P(inl x))
            → ((y : B) → P(inr y))
            -----------------------
            → ((z : A + B) → P z)


+-induction f g (inl x) = f x
+-induction f g (inr y) = g y

{-
    Maybe A ≅ A + 𝟙, where 𝟙 is the unit type.
-}

data 𝟙 : Type where
  ₀ : 𝟙

{-
    The empty type has no constructors:
-}

data ∅ : Type where

{-
    The empty type has a function to any other type, defined by an empty
    set of equations. Its induction principle is a generalization of that.
-}

∅-induction : {A : ∅ → Type} → (e : ∅) → A e
```

```
∅-induction ()

{-
    A pattern () means that what is required is impossible, because
    the type under consideration is empty.
-}

∅-elim : {A : Type} → ∅ → A
∅-elim {A} = ∅-induction {λ _ → A}


data List (A : Type) : Type where
  []  : List A
  _∷_ : A → List A → List A


{-
    When we want to use an infix operator as a prefix function, we
    write underscores around it. So for example (_∷_ x) is equivalent
    to λ xs → x ∷ xs. However, we prefer to write (cons x) in this
    particular case, as (_∷_ x) occurs often in the following
    development, and we find (cons x) more readable:
-}

singleton : {A : Type} → A → List A
singleton x = x ∷ []


_++_ : {A : Type} → List A → List A → List A
[]       ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)


-- Induction over lists is a functional program too:

List-induction : {A : Type} {P : List A → Type}

              →  P []                                     -- base case
              → ((x : A) (xs : List A) → P xs → P(x ∷ xs))   -- induction step
              ----------------------------------------------
              → (xs : List A) → P xs

List-induction base step []       = base
List-induction base step (x ∷ xs) = step x xs (List-induction base step xs)

{-
    Just as if_then_else_ is a particular case of Bool-induction, the
    infamous function foldr is a particular case of List-induction,
    where the type family P is constant:
-}

foldr : {A B : Type} → (A → B → B) → B → List A → B
foldr {A} {B} f y = List-induction {A} {P} base step
  where
    P : List A → Type
    P _ = B

    base : B
    base = y

    step : (x : A) (xs : List A) → P xs → P(x ∷ xs)
    step x _ y = f x y

{-
    To convince ourselves that this really is the usual function
    foldr, we need to define the equality type (also known as the
    identity type), whose only constructor is refl (for reflexivity of
    equality):
-}

data _≡_ {A : Type} : A → A → Type where
```

```
  refl : (x : A) → x ≡ x


≡-induction : {A : Type} {P : {x y : A} → x ≡ y → Type}

            → ((x : A) → P(refl x))
            -------------------------------
            → (x y : A) (p : x ≡ y) → P p

≡-induction r x .x (refl .x) = r x


{-
    The following can be defined from ≡-induction, but pattern
    matching on refl probably gives clearer definitions.

    The dot in front of a variable in a pattern is to deal with
    non-linearity (multiple occurrences of the same variable) of the
    pattern. The first undotted variable is pattern matched, and the
    dotted one assumes the same value.
-}

trans : {A : Type} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans (refl x) (refl .x) = refl x

sym : {A : Type} {x y : A} → x ≡ y → y ≡ x
sym (refl x) = refl x

ap : {A B : Type} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
ap f (refl x) = refl (f x)

transport : {A : Type} (P : A → Type) {x y : A} → x ≡ y → P x → P y
transport P (refl x) p = p

{-
   We can now formulate and prove the claim that the above
   construction does give the usual function foldr on lists:
-}

foldr-base : {A B : Type} (f : A → B → B) (y : B)

          → foldr f y [] ≡ y

foldr-base f y = refl y


foldr-step : {A B : Type} (f : A → B → B) (y : B) (x : A) (xs : List A)

          → foldr f y (x ∷ xs) ≡ f x (foldr f y xs)

foldr-step f y x xs = refl _

{-
    In the above uses of refl in the right-hand side of an equation,
    Agda normalizes the two sides of the equation, and sees that they
    are the same, and accepts refl as a term with the required type.

    An underscore in the right-hand side of a definition represents a
    term that we don't bother to write because Agda can infer it. In
    the last example, the term can be either side of the equation we
    want to prove.

    But it has to be remarked that not all equations can be proved
    with refl. Here is an example, which needs to be done by induction
    on the list xs. But instead of using List-induction directly, we
    can have a proof by recursion on xs, like that of List-induction
    itself:
-}

++assoc : {A : Type} (xs ys zs : List A)
```

```
         → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

++assoc []       ys zs = refl (ys ++ zs)
++assoc (x ∷ xs) ys zs = conclusion
  where
    IH : xs ++ ys ++ zs ≡ xs ++ (ys ++ zs)
    IH = ++assoc xs ys zs

    conclusion' : x ∷ (xs ++ ys ++ zs) ≡ x ∷ (xs ++ (ys ++ zs))
    conclusion' = ap (x ∷_) IH

    conclusion : ((x ∷ xs) ++ ys) ++ zs ≡ (x ∷ xs) ++ (ys ++ zs)
    conclusion = conclusion'

{-
    Although the types of goal' and goal look different, they are the
    same in the sense that they simplify to the same type, applying
    the definition of _++_. In practice, we avoid conclusion' and
    define conclusion directly.
-}

{-
    In Haskell, it is possible to explicitly indicate the type of a
    subterm. In Agda we can achieve this with a user defined syntax
    extension. We use the unicode symbol "∶", which looks like the
    Agda reserved symbol ":".

    What the following says is that when we write "x ∶ A", what we
    actually mean is "id {A} x", where id is the identity function.
-}

id : {A : Type} → A → A
id {A} x = x

syntax id {A} x = x ∶ A

have : {A B : Type} → A → B → B
have _ y = y

{-
    Notice that we can also write x ∶ A ∶ B (which associates as
    (x ∶ A) ∶ B) when the types A and B are "definitionally equal",
    meaning that they are the same when we expand the definitions.

    We exploit this to shorten the above proof while adding more
    information that is not needed for the computer to the reader:
-}

++assoc' : {A : Type} (xs ys zs : List A)

         → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

++assoc' []       ys zs = refl (ys ++ zs)
                        ∶  ([] ++ ys) ++ zs ≡ [] ++ (ys ++ zs)

++assoc' (x ∷ xs) ys zs = (have(++assoc' xs ys zs ∶ (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)))
                          ap (x ∷_) (++assoc' xs ys zs)
                          ∶ x ∷ ((xs ++ ys) ++ zs) ≡ x ∷ (xs ++ (ys ++ zs))
                          ∶ ((x ∷ xs) ++ ys) ++ zs ≡ (x ∷ xs) ++ (ys ++ zs)

{-
    The computer can do away with this additional information via type
    inference, and so can we in principle, but not always in practice.
-}

++assoc'' : {A : Type} (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++assoc'' []       ys zs = refl _
++assoc'' (x ∷ xs) ys zs = ap (x ∷_) (++assoc'' xs ys zs)

{-
```

```
    We now show that xs ++ [] for any list xs, by induction on xs. We
    do this in a terse way, given the above explanations.

-}

[]-right-neutral : {X : Type} (xs : List X) → xs ++ [] ≡ xs
[]-right-neutral [] = refl []
[]-right-neutral (x ∷ xs) = ap (x ∷_) ([]-right-neutral xs)

{-

List reversal.

-}

rev : {A : Type} → List A → List A
rev []       = []
rev (x ∷ xs) = rev xs ++ (x ∷ [])

rev-++ : {A : Type} (xs ys : List A) → rev (xs ++ ys) ≡ rev ys ++ rev xs
rev-++ [] ys       = sym ([]-right-neutral (rev ys))
rev-++ (x ∷ xs) ys = conclusion
 where
  IH : rev (xs ++ ys) ≡ rev ys ++ rev xs
  IH = rev-++ xs ys

  a : rev (xs ++ ys) ++ (x ∷ []) ≡ (rev ys ++ rev xs) ++ (x ∷ [])
  a = ap (_++ (x ∷ [])) IH

  b : (rev ys ++ rev xs) ++ (x ∷ []) ≡ rev ys ++ (rev xs ++ (x ∷ []))
  b = ++assoc (rev ys) (rev xs) (x ∷ [])

  conclusion : rev (xs ++ ys) ++ (x ∷ []) ≡ rev ys ++ (rev xs ++ (x ∷ []))
  conclusion = trans a b

rev-involutive : {A : Type} (xs : List A) → rev (rev xs) ≡ xs
rev-involutive []       = refl (rev (rev []))
rev-involutive (x ∷ xs) = conclusion
 where
  IH : rev (rev xs) ≡ xs
  IH = rev-involutive xs

  a : rev (rev (x ∷ xs)) ≡ rev (rev xs ++ (x ∷ []))
  a = refl _

  b : rev (rev xs ++ (x ∷ [])) ≡ rev (x ∷ []) ++ rev (rev xs)
  b = rev-++ (rev xs) (x ∷ [])

  c : rev (x ∷ []) ++ rev (rev xs) ≡ rev (x ∷ []) ++ xs
  c = ap (rev (x ∷ []) ++_) IH

  conclusion : rev (rev (x ∷ xs)) ≡ (x ∷ xs)
  conclusion = trans a (trans b c)

{-

 The above reversal function is quadratic time. It is well known that
 it can be defined in linear time using rev-append. Let's prove this.

-}

rev-append : {A : Type} → List A → List A → List A
rev-append []       ys = ys
rev-append (x ∷ xs) ys = rev-append xs (x ∷ ys)

rev-linear : {A : Type} → List A → List A
rev-linear xs = rev-append xs []

rev-append-spec : {A : Type}
                  (xs ys : List A)
                → rev-append xs ys ≡ rev xs ++ ys
```

```
rev-append-spec []       ys = refl ys
rev-append-spec (x ∷ xs) ys = conclusion
 where
  IH : rev-append xs (x ∷ ys) ≡ rev xs ++ (x ∷ ys)
  IH = rev-append-spec xs (x ∷ ys)

  a : rev xs ++ ((x ∷ []) ++ ys) ≡ (rev xs ++ (x ∷ [])) ++ ys
  a = sym (++assoc (rev xs) (x ∷ []) ys)

  conclusion : rev-append (x ∷ xs) ys ≡ rev (x ∷ xs) ++ ys
  conclusion = trans IH a

rev-linear-correct : {A : Type}
                     (xs : List A)
                   → rev-linear xs ≡ rev xs
rev-linear-correct xs = trans (rev-append-spec xs []) ([]-right-neutral (rev xs))

{-
    We now define the usual map function on lists in the two usual ways.
-}

map' : {A B : Type} → (A → B) → List A → List B
map' f []       = []
map' f (x ∷ xs) = f x ∷ map' f xs

map'' : {A B : Type} → (A → B) → List A → List B
map'' f = foldr (λ x ys → f x ∷ ys) []

maps-agreement : {A B : Type}
                 (f : A → B)
                 (xs : List A)
               → map' f xs ≡ map'' f xs
maps-agreement f []       = refl []
maps-agreement f (x ∷ xs) = conclusion
 where
  IH : map' f xs ≡ map'' f xs
  IH = maps-agreement f xs

  conclusion : f x ∷ map' f xs ≡ f x ∷ map'' f xs
  conclusion = ap (f x ∷_) IH

{-
    Can choose below whether we want map = map' or map = map''.

    The proofs given below about them don't not need to be changed as
    the definition with foldr has the same definitional behaviour, as
    illustrated by the theorems foldr-base and foldr-step
    above.
-}

map : {A B : Type} → (A → B) → List A → List B
map = map'

{-

    Some properties of map

-}

map-id : {A : Type}
         (xs : List A)
       → map id xs ≡ xs
map-id []       = refl []
map-id (x ∷ xs) = ap (x ∷_) (map-id xs)

_∘_ : {A B C : Type} → (B → C) → (A → B) → (A → C)
g ∘ f = λ x → g (f x)


map-∘ : {A B C : Type}
        (g : B → C)
```

```
        (f : A → B)
        (xs : List A)
      → map (g ∘ f) xs ≡ map g (map f xs)
map-∘ g f []       = refl []
map-∘ g f (x ∷ xs) = conclusion
 where
  IH : map (g ∘ f) xs ≡ map g (map f xs)
  IH = map-∘ g f xs

  conclusion : g (f x) ∷ map (g ∘ f) xs ≡ g (f x) ∷ map g (map f xs)
  conclusion = ap (g (f x) ∷_) IH

{-
    We now define binary trees and a function to pick subtrees
    specified by a list of directions left and right.
-}


data BT (A : Type) : Type where
  Empty : BT A
  Fork  : A → BT A → BT A → BT A


BT-induction : {A : Type} {P : BT A → Type}
             →  P Empty                                       -- base
             → ((x : A) (l r : BT A) → P l → P r → P(Fork x l r))  -- step
             -----------------------------------------------------
             → (t : BT A) → P t

BT-induction {A} {P} base step Empty        = base : P Empty

BT-induction {A} {P} base step (Fork x l r) = step x l r (BT-induction base step l : P l)
                                                         (BT-induction base step r : P r)
                                             : P(Fork x l r)


data Direction : Type where
  L R : Direction


Address : Type
Address = List Direction


subtree : {A : Type} → Address → BT A → Maybe(BT A)
subtree []       t            = Just t
subtree (_ ∷ _)  Empty        = Nothing
subtree (L ∷ ds) (Fork _ l _) = subtree ds l
subtree (R ∷ ds) (Fork _ _ r) = subtree ds r


isValid : {A : Type} → Address → BT A → Bool
isValid []       _            = True
isValid (_ ∷ _)  Empty        = False
isValid (L ∷ ds) (Fork _ l _) = isValid ds l
isValid (R ∷ ds) (Fork _ _ r) = isValid ds r

{-
    If an addrees is invalid, then the function subtree gives Nothing:
-}

false-premise : {P : Type} → True ≡ False → P
false-premise ()

invalid-gives-Nothing : {A : Type} (ds : Address) (t : BT A)

                     → isValid ds t ≡ False → subtree ds t ≡ Nothing

invalid-gives-Nothing {A} [] Empty p
  = false-premise(p : isValid {A} [] Empty ≡ False)
```

```
invalid-gives-Nothing (d ∷ ds) Empty (refl False)
  = refl Nothing
  :  Nothing ≡ Nothing
  :  subtree (d ∷ ds) Empty ≡ Nothing

invalid-gives-Nothing [] (Fork x l r) p
  = false-premise(p : isValid [] (Fork x l r) ≡ False)

invalid-gives-Nothing (L ∷ ds) (Fork x l r) p
  = invalid-gives-Nothing ds l (p : isValid (L ∷ ds) (Fork x l r) ≡ False)
  : subtree ds l ≡ Nothing
  : subtree (L ∷ ds) (Fork x l r) ≡ Nothing

invalid-gives-Nothing (R ∷ ds) (Fork x l r) p
  = invalid-gives-Nothing ds r (p : isValid (R ∷ ds) (Fork x l r) ≡ False)
  : subtree ds r ≡ Nothing
  : subtree (R ∷ ds) (Fork x l r) ≡ Nothing

{-

    Or, in concise form:

-}

invalid-gives-Nothing' : {A : Type} (ds : Address) (t : BT A)

                     → isValid ds t ≡ False → subtree ds t ≡ Nothing

invalid-gives-Nothing' []       Empty        ()
invalid-gives-Nothing' (d ∷ ds) Empty        (refl False) = refl Nothing
invalid-gives-Nothing' []       (Fork x l r) ()
invalid-gives-Nothing' (L ∷ ds) (Fork x l r) p            = invalid-gives-Nothing' ds l p
invalid-gives-Nothing' (R ∷ ds) (Fork x l r) p            = invalid-gives-Nothing' ds r p

{-
    We now show that if an address ds is valid for a tree t, then
    there is a tree t' with subtree ds t = Just t'.

    "There is ... with ..." can be expressed with Σ types, which we
    now explain and define.

    Given a type A and a type family B : A → Type, the type Σ {A} B
    has as elements the pairs (x , y) with x : A and y : B x.

    The brackets in pairs are not necessary, so that we can write just
    "x , y". This is because we define comma to be a constructor,
    written as a binary operator in infix notation:
-}

record Σ (A : Type) (B : A → Type) : Type where
  constructor
    _,_
  field
    x : A
    y : B x


{-
   We define special syntax to be able to write expressions involving
   Σ in a more friendly way, so that, for example,

       Σ x : A , B x

   is the type of pairs (x,y) with x : A and y : B x.

   Notice that, for some reason, the syntax declaration is backwards
   (what is defined in on the right rather than the left:
-}

syntax Σ A (λ x → y) = Σ x : A , y
```

```
pr₁ : {A : Type} {B : A → Type} → (Σ x : A , B x) → A
pr₁ (x , y) = x

pr₂ : {A : Type} {B : A → Type} → ((x , y) : Σ x : A , B x) → B x
pr₂ (x , y) = y

{-
    Induction on Σ is uncurry:
-}

Σ-induction : {A : Type} {B : A → Type} {P : Σ A B → Type}

            → ((x : A) (y : B x) → P(x , y))
            ---------------------------------
            → (z : Σ A B) → P z

Σ-induction f (x , y) = f x y

{-
    We could have defined the projections by induction:
-}

pr₁' : {A : Type} {B : A → Type} → Σ A B → A
pr₁' {A} {B} = Σ-induction {A} {B} {λ _ → A} (λ x y → x)


pr₂' : {A : Type} {B : A → Type} → (z : Σ A B) → B(pr₁ z)
pr₂' {A} {B} = Σ-induction {A} {B} {λ z → B (pr₁ z)} (λ x y → y)

{-
    A particular case of Σ {A} C is when the family C : A → Type is constant,
    that is, we have C x = B for some type B, in which case we get the
    cartesian product A × B.
-}

_×_ : Type → Type → Type
A × B = Σ x : A , B

{-
    We can now formulate and prove that if an address is valid, then
    it gives some subtree.
-}

false-premise' : {P : Type} → False ≡ True → P
false-premise' ()


valid-gives-just : {A : Type} (ds : Address) (t : BT A)

                 → isValid ds t ≡ True
                 → Σ t' : BT A , (subtree ds t ≡ Just t')

valid-gives-just {A} [] Empty p
  = (have(p : isValid {A} [] Empty ≡ True))
    (Empty , (refl _ : subtree [] Empty ≡ Just Empty))

valid-gives-just {A} (d ∷ ds) Empty p
  = false-premise'(p : isValid {A} (d ∷ ds) Empty ≡ True)

valid-gives-just {A} [] (Fork x l r) p
  = Fork x l r , (refl _ : subtree [] (Fork x l r) ≡ Just (Fork x l r))
  : Σ t' : BT A , (subtree [] (Fork x l r) ≡ Just t')

valid-gives-just {A} (L ∷ ds) (Fork x l r) p
  = valid-gives-just ds l (p : isValid (L ∷ ds) (Fork x l r) ≡ True
                             : isValid ds l ≡ True)
  : (Σ t' : BT A , (subtree (L ∷ ds) (Fork x l r) ≡ Just t'))
  :  Σ t' : BT A , (subtree ds l ≡ Just t')

valid-gives-just {A} (R ∷ ds) (Fork x l r) p
  = valid-gives-just ds r (p : isValid (R ∷ ds) (Fork x l r) ≡ True
```

```
                                : isValid ds r ≡ True)
  : (Σ t' : BT A , (subtree (R ∷ ds) (Fork x l r) ≡ Just t'))
  :  Σ t' : BT A , (subtree ds r ≡ Just t')


{-
    Or, in concise form:
-}

valid-gives-Just' : {A : Type} (ds : Address) (t : BT A)

                 → isValid ds t ≡ True → Σ t' : BT A , (subtree ds t ≡ Just t')

valid-gives-Just' []       Empty        p = Empty , (refl _)
valid-gives-Just' (d ∷ ds) Empty       ()
valid-gives-Just' []       (Fork x l r) p = Fork x l r , (refl _)
valid-gives-Just' (L ∷ ds) (Fork x l r) p = valid-gives-Just' ds l p
valid-gives-Just' (R ∷ ds) (Fork x l r) p = valid-gives-Just' ds r p

{-
    We have been working with "propositions as types" and "proofs as
    (functional) programs", also known as the Curry--Howard
    interpretation of logic.

     A implies B                     A → B
     A and B                         A × B
     A or B                          A + B
     for all x : A, P(x)             (x : A) → P x
     there is x : A with P(x)        Σ x : A , P x
     false                           ∅
     not A                           A → ∅

    (https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence)

    We now construct the list of valid addresses for a given tree. The
    construction is short, but the proof that it does produce
    precisely the valid addresses is long and requires many lemmas,
    particularly in our self-imposed absence of a library.
-}


validAddresses : {A : Type} → BT A → List Address
validAddresses Empty        = singleton []
validAddresses (Fork _ l r) = (singleton [])
                           ++ (map (L ∷_) (validAddresses l))
                           ++ (map (R ∷_) (validAddresses r))

{-
    The remainder of this tutorial is devoted to showing the following:

        For any given address ds and tree t : BT A,

            isValid ds t ≡ True if and only if ds is in validAddresses t.

    This is formulated and proved in the function main-theorem below.


    We define when an element x : A is in a list xs : List A, written
    x ∈ xs, borrowing the membership symbol ∈ from set theory:
-}

data _∈_ {A : Type} : A → List A → Type where
  inHead : (x : A) (xs : List A) → x ∈ x ∷ xs
  inTail : (x y : A) (xs : List A) → y ∈ xs → y ∈ x ∷ xs

{-
    This is a so-called inductive definition of a predicate (the
    membership relation).

    (1) x ∈ x ∷ xs
```

```
    (2) y ∈ xs → y ∈ x ∷ xs

    We construct a proof of (1) with inHead, and a proof of (2) with
    inTail.

    The following proof is by induction on "x ∈ xs".
-}

mapIsIn : {A B : Type} (f : A → B) {x : A} {xs : List A}

        → x ∈ xs → f x ∈ map f xs

mapIsIn f (inHead x xs) = inHead (f x) (map f xs)
                        : f x ∈ f x ∷ map f xs
                        : f x ∈ map f (x ∷ xs)

mapIsIn f (inTail x y xs i) = (have(i : y ∈ xs))
                              (have(mapIsIn f i : f y ∈ map f xs))
                              inTail (f x) (f y) (map f xs) (mapIsIn f i)
                            : f y ∈ f x ∷ map f xs
                            : f y ∈ map f (x ∷ xs)

{-
    Or, in concise form:
-}

mapIsIn' : {A B : Type} (f : A → B) {x : A} {xs : List A}

        → x ∈ xs → f x ∈ map f xs

mapIsIn' f (inHead x xs) = inHead (f x) (map f xs)

mapIsIn' f (inTail x y xs i) = inTail (f x) (f y) (map f xs) (mapIsIn' f i)

{-
    Even more concise:
-}

mapIsIn'' : {A B : Type} (f : A → B) {x : A} {xs : List A}

        → x ∈ xs → f x ∈ map f xs

mapIsIn'' _ (inHead _ _) = inHead _ _

mapIsIn'' _ (inTail _ _ x_ i) = inTail _ _ _ (mapIsIn'' _ i)

{-
    Which means that we could have made all the "_" arguments into
    implicit arguments (greatly sacrificing clarity). Also, there is
    no guarantee that these implicit arguments will be inferrable in
    other contexts. In any case, it seems to be an art to decide which
    arguments should be left implicit with a good balance of
    conciseness and clarity.
-}


equal-heads : {A : Type} {x y : A} {xs ys : List A} → (x ∷ xs) ≡ (y ∷ ys) → x ≡ y
equal-heads {A} {x} {.x} {xs} {.xs} (refl .(x ∷ xs)) = refl x


equal-tails : {A : Type} {x y : A} {xs ys : List A} → (x ∷ xs) ≡ (y ∷ ys) → xs ≡ ys
equal-tails {A} {x} {.x} {xs} {.xs} (refl .(x ∷ xs)) = refl xs


isInjective : {A B : Type} → (A → B) → Type
isInjective f = {x y : _} → f x ≡ f y → x ≡ y


cons-injective : {A : Type} {x : A} → isInjective(x ∷_)
cons-injective {A} {x} = equal-tails {A} {x}
```

```
{-
    The following is by induction on xs. We introduce an auxiliary
    function g to do this induction, with the other parameters
    fixed. Because we need to pattern-match on the value of (map f xs)
    in the induction, we introduce an extra parameter for the value ys
    of (map f xs). Agda has a special keywork "with" for that purpose,
    but we don't discuss it in this brief introduction.
-}

mapIsIn-converse : {A B : Type} {f : A → B} {x : A} {xs : List A}

                  → isInjective f → f x ∈ map f xs → x ∈ xs

mapIsIn-converse {A} {B} {f} {x} {xs} inj = g xs (map f xs) (refl _)
  where
    g : (xs : List A) (ys : List B) → ys ≡ map f xs → f x ∈ ys → x ∈ xs
    g [] .(f x ∷ xs) () (inHead .(f x) xs)
    g [] .(x' ∷ xs) () (inTail x' .(f x) xs i)
    g (x' ∷ xs) .(f x ∷ ys) e (inHead .(f x) ys) = conclusion
      where
        a : f x ≡ f x'
        a = have(e : f x ∷ ys ≡ map f (x' ∷ xs)
                   : f x ∷ ys ≡ f x' ∷ map f xs)
            equal-heads e
        b : x ≡ x'
        b = inj a
        c : x ∈ x ∷ xs
        c = inHead x xs
        conclusion : x ∈ x' ∷ xs
        conclusion = transport (λ x' → x ∈ x' ∷ xs) b c
    g (x' ∷ xs) .(y ∷ ys) e (inTail y .(f x) ys i) = conclusion
      where
        et : ys ≡ map f xs
        et = have(e : y ∷ ys ≡ map f (x' ∷ xs)
                    : y ∷ ys ≡ f x' ∷ map f xs)
             equal-tails e
        IH : f x ∈ ys → x ∈ xs
        IH i = g xs ys et i
        conclusion : x ∈ x' ∷ xs
        conclusion = inTail x' x xs (IH i)

{-
    "Not A" holds, written, ¬A, if A is empty, or equivalently if there
    is a function A → ∅:
-}

¬ : Type → Type
¬ A = A → ∅


{-
    By induction on xs:
-}

not-in-map-if-not-in-image : {A B : Type} {f : A → B} {y : B}

                           → ((x : A) → ¬(f x ≡ y)) → (xs : List A) → ¬(y ∈ map f xs)

not-in-map-if-not-in-image {A} {B} {f} {y} ni = g
  where
    remark : (x : A) → f x ≡ y → ∅
    remark = ni

    g : (xs : List A) → y ∈ map f xs → ∅
    g []       ()
    g (x ∷ xs) (inHead .(f x) .(map f xs))      = ni x (refl (f x))
    g (x ∷ xs) (inTail .(f x) .y .(map f xs) i) = g xs i


{-
```

```
    By induction on zs:
-}

left-if-not-in-image : {A B : Type} {f : A → B} {y : B} (xs : List A) {zs : List B}

                     → ((x : A) → ¬(f x ≡ y)) → y ∈ zs ++ map f xs → y ∈ zs

left-if-not-in-image {A} {B} {f} {y} xs {zs} ni = g zs
  where
    g : (zs : List B) → y ∈ zs ++ map f xs → y ∈ zs
    g [] i = (have (i : y ∈ map f xs))
             ∅-elim (not-in-map-if-not-in-image ni xs i) : y ∈ []
    g (z ∷ zs) (inHead .z .(zs ++ map f xs)) = inHead z zs : z ∈ z ∷ zs
    g (z ∷ zs) (inTail .z y .(zs ++ map f xs) i) = inTail z y zs (g zs i : y ∈ zs)

{-
    By induction on xs:
-}

right-if-not-in-image : {A B : Type} {f : A → B} {y : B} (xs : List A) {zs : List B}

                      → ((x : A) → ¬(f x ≡ y)) → y ∈ map f xs ++ zs → y ∈ zs

right-if-not-in-image {A} {B} {f} {y} xs {zs} ni = g xs
  where
    g : (xs : List A) → y ∈ map f xs ++ zs → y ∈ zs
    g [] i = i
    g (x ∷ xs) (inHead .(f x) .(map f xs ++ zs)) = ∅-elim (ni x (refl (f x)))
    g (x ∷ xs) (inTail .(f x) y .(map f xs ++ zs) i) = g xs i


{-
    By induction on xs:
-}

inLHS : {A : Type} (x : A) (xs ys : List A) → x ∈ xs → x ∈ xs ++ ys
inLHS {A} x xs ys i = g xs i
  where
    g : (xs : List A) → x ∈ xs → x ∈ xs ++ ys
    g (x ∷ xs) (inHead .x .xs)     = inHead x (xs ++ ys)
    g (x ∷ xs) (inTail .x y .xs i) = inTail x y (xs ++ ys) (g xs i)


{-
    Agda checks that the patterns in any definition are exhaustive.
    Notice that the function g doesn't have a case for the empty list
    because this case is impossible and Agda can see that from the
    definition of ∈.

    By induction on xs:
-}

inRHS : {A : Type} (x : A) (xs ys : List A) → x ∈ ys → x ∈ xs ++ ys
inRHS {A} x xs ys i = g xs i
  where
    g : (xs : List A) → x ∈ ys → x ∈ xs ++ ys
    g [] i = i
    g (x' ∷ xs) i = inTail x' x (xs ++ ys) (g xs i)

{-
    With the above lemmas, we can finally prove our main theorem. We
    prove each direction separately.
-}


isValid-∈-validAddresses : {A : Type} (ds : Address) (t : BT A)

                         → isValid ds t ≡ True → ds ∈ validAddresses t

isValid-∈-validAddresses [] Empty e              = inHead [] _ : [] ∈ singleton []
isValid-∈-validAddresses [] (Fork x l r) e       = inHead [] _ : [] ∈ validAddresses (Fork x l r)
```

```
isValid-∈-validAddresses (L ∷ ds) Empty ()
isValid-∈-validAddresses (L ∷ ds) (Fork x l r) e = inTail [] _ _ lemma
 where
  IH : ds ∈ validAddresses l
  IH = isValid-∈-validAddresses ds l (e : isValid (L ∷ ds) (Fork x l r) ≡ True)

  a : L ∷ ds ∈ map (L ∷_) (validAddresses l)
  a = mapIsIn (L ∷_) IH

  lemma : L ∷ ds ∈ map (L ∷_) (validAddresses l) ++ map (R ∷_) (validAddresses r)
  lemma = inLHS (L ∷ ds) (map (L ∷_) (validAddresses l)) (map (R ∷_) (validAddresses r)) a

isValid-∈-validAddresses (R ∷ ds) Empty ()
isValid-∈-validAddresses (R ∷ ds) (Fork x l r) e = inTail [] _ _ lemma
 where
  IH : ds ∈ validAddresses r
  IH = isValid-∈-validAddresses ds r (e : isValid (R ∷ ds) (Fork x l r) ≡ True)

  a : R ∷ ds ∈ map (R ∷_) (validAddresses r)
  a = mapIsIn (R ∷_) IH

  lemma : R ∷ ds ∈ map (L ∷_) (validAddresses l) ++ map (R ∷_) (validAddresses r)
  lemma = inRHS (R ∷ ds) (map (L ∷_) (validAddresses l)) (map (R ∷_) (validAddresses r)) a


∈-validAddresses-implies-isValid : {A : Type} (ds : Address) (t : BT A)

                                 → ds ∈ validAddresses t → isValid ds t ≡ True

∈-validAddresses-implies-isValid {A} [] t i = refl (isValid [] t)
∈-validAddresses-implies-isValid {A} (L ∷ ds) Empty (inTail _ _ _ ())
∈-validAddresses-implies-isValid {A} (L ∷ ds) (Fork x l r) (inTail _ _ _ i) = conclusion
  where
    IH : ds ∈ validAddresses l → isValid ds l ≡ True
    IH = ∈-validAddresses-implies-isValid ds l

    remark : L ∷ ds ∈ map (L ∷_) (validAddresses l) ++ map (R ∷_) (validAddresses r)
    remark = i

    c : (ds : _) (vl : _) (vr : _)
      → L ∷ ds ∈ map (L ∷_) vl ++ map (R ∷_) vr → L ∷ ds ∈ map (L ∷_) vl
    c ds vl vr = left-if-not-in-image vr ni
      where
        ni : (es : _) → ¬((R ∷ es) ≡ (L ∷ ds))
        ni es ()

    b : (ds : _) (vl : _) (vr : _) → L ∷ ds ∈ map (L ∷_) vl ++ map (R ∷_) vr → ds ∈ vl
    b ds vl vr i = mapIsIn-converse cons-injective (c ds vl vr i)

    a : ds ∈ validAddresses l
    a = b ds (validAddresses l) (validAddresses r) i

    conclusion : isValid ds l ≡ True
    conclusion = IH a


∈-validAddresses-implies-isValid {A} (R ∷ ds) Empty (inTail _ _ _ ())
∈-validAddresses-implies-isValid {A} (R ∷ ds) (Fork x l r) (inTail _ _ _ i) = conclusion
  where
    IH : ds ∈ validAddresses r → isValid ds r ≡ True
    IH = ∈-validAddresses-implies-isValid ds r

    remark : R ∷ ds ∈ map (L ∷_) (validAddresses l) ++ map (R ∷_) (validAddresses r)
    remark = i

    c : (ds : _) (vl : _) (vr : _)
      → R ∷ ds ∈ map (L ∷_) vl ++ map (R ∷_) vr → R ∷ ds ∈ map (R ∷_) vr
    c ds vl vr = right-if-not-in-image vl ni
      where
        ni : (es : _) → ¬((L ∷ es) ≡ (R ∷ ds))
        ni es ()
```

```
    b : (ds : _) (vl : _) (vr : _) → R ∷ ds ∈ map (L ∷_) vl ++ map (R ∷_) vr → ds ∈ vr
    b ds vl vr i = mapIsIn-converse cons-injective (c ds vl vr i)

    a : ds ∈ validAddresses r
    a = b ds (validAddresses l) (validAddresses r) i

    conclusion : isValid ds r ≡ True
    conclusion = IH a

{-
    We now package the last two facts into a single one, to get our main theorem.
-}

_⇔_ : Type → Type → Type
A ⇔ B = (A → B) × (B → A)


main-theorem : {A : Type} (ds : Address) (t : BT A)

            → isValid ds t ≡ True ⇔ ds ∈ validAddresses t

main-theorem ds t = isValid-∈-validAddresses ds t , ∈-validAddresses-implies-isValid ds t

{-
    We conclude by declaring the associativity and precedence of the
    binary operations defined above, so that many round brackets can
    be avoided. Without this, we would get syntax errors above.
-}

infixl 3 _++_
infix  2 _≡_
infix  2 _∈_
infix  1 _⇔_
infixr 1 _,_
infixl 0 id
```