Chapter 20: Internal Verification

Zhenjiang Hu, Wei Zhang

School of Computer Science, PKU

November 16, 2022

External and Internal Proofs

- External verification: proofs are external to programs.
  - Algebraic properties are usually proved externally

- Internal verification: write functions with more semantically expressive types.
  - Can be applied for essential invariants of datatypes
  - Easier to apply for complex programs
  - Harder to read

The Vector Datatype

```
data 𝕍 {ℓ} (A : Set ℓ) : ℕ → Set ℓ where
    [] : 𝕍 A 0
    _::_ : {n : ℕ} → A → 𝕍 A n → 𝕍 A (suc n)
```

```
test-vector : 𝕍 𝔹 4
test-vector = ff :: tt :: ff :: ff :: []

test-vector2 : 𝕃 (𝕍 𝔹 2)
test-vector2 = (ff :: tt :: []) ::
               (tt :: ff :: []) ::
               (tt :: ff :: []) :: []

test-vector3 : 𝕍 (𝕍 𝔹 3) 2
test-vector3 = (tt :: tt :: tt :: []) ::
               (ff :: ff :: ff :: []) :: []
```

Functions over Vectors

```
_++𝕍_ : ∀ {ℓ} {A : Set ℓ}{n m : ℕ} →
        𝕍 A n → 𝕍 A m → 𝕍 A (n + m)
[] ++𝕍 ys = ys
(x :: xs) ++𝕍 ys = x :: xs ++𝕍 ys

head𝕍 : ∀ {ℓ} {A : Set ℓ}{n : ℕ} → 𝕍 A (suc n) → A
head𝕍 (x :: _) = x

tail𝕍 : ∀ {ℓ} {A : Set ℓ}{n : ℕ} → 𝕍 A n → 𝕍 A (pred n)
tail𝕍 [] = []
tail𝕍 (_ :: xs) = xs

map𝕍 : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'}{n : ℕ} →
       (A → B) → 𝕍 A n → 𝕍 B n
map𝕍 f [] = []
map𝕍 f (x :: xs) = f x :: map𝕍 f xs
```

Functions over Vectors

```
concatV : ∀{ℓ}{A : Set ℓ}{n m : ℕ} →
          V (V A n) m → V A (m * n)
concatV [] = []
concatV (x :: xs) = x ++V (concatV xs)

nthV : ∀ {ℓ} {A : Set ℓ}{m : ℕ} →
       (n : ℕ) → n < m ≡ tt → V A m → A
nthV 0 _ (x :: _) = x
nthV (suc n) p (_ :: xs) = nthV n p xs
nthV (suc n) () []
nthV 0 () []

repeatV : ∀ {ℓ} {A : Set ℓ} → (a : A)(n : ℕ) → V A n
repeatV a 0 = []
repeatV a (suc n) = a :: (repeatV a n)
```

Binary Search Trees

```
module bst (A : Set)
           (_≤A_ : A → A → 𝔹)
           (≤A-trans : transitive _≤A_)
           (≤A-total : total _≤A_) where

data bst : A → A → Set where
  bst-leaf : ∀ {l u : A} → l ≤A u ≡ tt → bst l u
  bst-node : ∀ {l l' u' u : A}(d : A) →
             bst l' d → bst d u' →
             l ≤A l' ≡ tt → u' ≤A u ≡ tt →
             bst l u
```

Searching for an Element in a Binary Search Tree

```
bst-search : ∀{l u : A}(d : A) →
               bst l u → maybe (Σ A (λ d' → d iso𝔹 d' ≡ tt))
bst-search d (bst-leaf _) = nothing
bst-search d (bst-node d' L R _ _) with keep (d ≤A d')
bst-search d (bst-node d' L R _ _) | tt , p1 with keep (d' ≤A d)
bst-search d (bst-node d' L R _ _)
    | tt , p1 | tt , p2 = just (d' , iso𝔹-intro p1 p2)
bst-search d (bst-node d' L R _ _)
    | tt , p1 | ff , p2 = bst-search d L
bst-search d (bst-node d' L R _ _)
    | ff , p1 = bst-search d R
```

Sigma Types

A Σ-type is a generalization of the usual Cartesian product type A × B, and is often referred to as a dependent sum type.

```
data Σ {ℓ ℓ'} (A : Set ℓ) (B : A → Set ℓ') : Set (ℓ ⊔ ℓ') where
    _,_ : (a : A) → (b : B a) → Σ A B


_×_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
A × B = Σ A (λ x → B)
```

Sigma Types: Nonzero Nat

```
ℕ⁺ : Set
ℕ⁺ = Σ ℕ (λ n → iszero n ≡ ff)

suc⁺ : ℕ⁺ → ℕ⁺
suc⁺ (x , p) = (suc x , refl)

_+⁺_ : ℕ⁺ → ℕ⁺ → ℕ⁺
(x , p) +⁺ (y , q) = x + y , iszerosum2 x y p

_*⁺_ : ℕ⁺ → ℕ⁺ → ℕ⁺
(x , p) *⁺ (y , q) = (x * y , iszeromult x y p q)
```

Why Sigma and Pi?

- Σ-types (dependent sum type) can be thought of as generalizing disjoint unions

  A ⊎ B:

$$(\{0\} \times A) \cup (\{1\} \times B)$$

- Dependent function type: (x:A) → B

  (or written mathematically as $\Pi x : A.\ B$) is another generalization of Cartesian products.

$$\sum_{x:A} B(x) = B(x_1) + B(x_2) + \cdots$$

$$\prod_{x:A} B(x) = B(x_1) \times B(x_2) \times \cdots$$

$$A = Bool$$

$$\sum A\ B = \sum_{x:A} B(x) = B(true) + B(false)$$

$$\prod A\ B = \prod_{x:A} B(x) = B(true) \times B(false)$$

Braun Tree (B-tree)

- **Braun trees** are either empty or else a node consisting of some data, and a left and a right subtree. There are two properties we want for Braun trees.
  - Ordering property. The data at each node is less than or equal to the data in the left and right subtrees of that node.
  - Structural property. For each node in the tree, either the size of the left and right subtrees of that node are the same, or else the size of the left is one bigger.

The braun-tree Datatype, and Sum Types

```
module braun-tree{ℓ} (A : Set ℓ) (_<A_ : A → A → 𝔹) where

data braun-tree : (n : ℕ) → Set ℓ where
   bt-empty : braun-tree 0
   bt-node : ∀ {n m : ℕ} →
             A → braun-tree n → braun-tree m →
             n ≡ m ∨ n ≡ suc m →
             braun-tree (suc (n + m))
```

```
data _⊎_ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') : Set (ℓ ⊔ ℓ') where
   inj₁ : (x : A) → A ⊎ B
   inj₂ : (y : B) → A ⊎ B

_∨_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
_∨_ = _⊎_
```

```
bt-insert : ∀ {n : ℕ} → A → braun-tree n → braun-tree (suc n)

bt-insert a bt-empty = bt-node a bt-empty bt-empty (inj₁ refl)

bt-insert a (bt-node{n}{m} a' l r p)
    rewrite +comm n m with p | if a <A a' then (a , a') else (a' , a)
bt-insert a (bt-node{n}{m} a' l r _) | inj₁ p | (a1 , a2)
    rewrite p = (bt-node a1 (bt-insert a2 r) l (inj₂ refl))
bt-insert a (bt-node{n}{m} a' l r _) | inj₂ p | (a1 , a2) =
    (bt-node a1 (bt-insert a2 r) l (inj₁ (sym p)))
```

Homework

20.1. Using the vector type V in a nested fashion, fill in the hole below to define a type for matrices of natural numbers, where the type lists the dimensions of the matrix:

_by_matrix : N → N → Set

n by m matrix = ?

20.2. Define the following basic operations on matrices, using the definition you propose in the previous problem. You should first figure out the types of the operations, of course, and then write code for them (possibly using helper functions).

(a) zero-matrix, which takes in the desired dimensions and produces a matrix of those dimensions, where every value in the matrix is zero.

(not finished yet, see next page)

(问题还没结束，下页继续)

Homework

(b) matrix-elt, which takes in an n by m matrix and a row and column index within those bounds, and returns the element stored at that position in the matrix.

(c) diagonal-matrix, which takes in an element d and a dimension n, and returns the n by n matrix which has zero everywhere except d down the diagonal of the matrix. Use this to define a function

identity-matrix

returning a diagonal matrix where the diagonal is 1.

(d) transpose, which turns an n by m matrix into a m by n matrix by switching the rows and columns.

(e) _._, the dot product of two vectors.