Chapter 20:
Internal Verification

Zhenjiang Hu, Wei Zhang

School of Computer Science, PKU

November 26, 2025

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
```

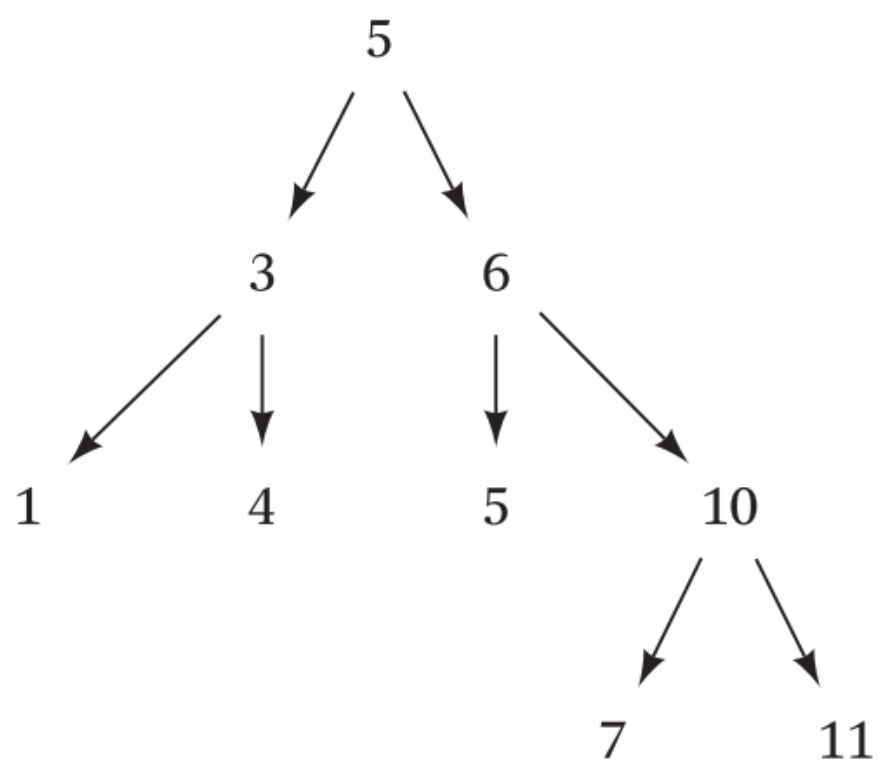
Functions over Vectors

```
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



Relations

```
module relations {ℓ ℓ' : level}{A : Set ℓ}
                 (_≥A_ : A → A → Set ℓ') where

reflexive : Set (ℓ ⊔ ℓ')
reflexive = ∀ {a : A} → a ≥A a

transitive : Set (ℓ ⊔ ℓ')
transitive = ∀ {a b c : A} → a ≥A b → b ≥A c → a ≥A c
```

Boolean Relations

```
module bool-relations {ℓ : level}{A : Set ℓ} (_≤A_ : A → A → 𝔹) where

open import relations (λ a a' → a' ≤A a ≡ tt) public using
    (reflexive ; transitive)

total : Set ℓ
total = ∀ {a b : A} → a ≤A b ≡ ff → b ≤A a ≡ tt

total-reflexive : total → reflexive
total-reflexive tot {a} with keep (a ≤A a)
total-reflexive tot {a} | tt , p = p
total-reflexive tot {a} | ff , p = tot p

_iso𝔹_ : A → A → 𝔹
d iso𝔹 d' = d ≤A d' && d' ≤A d

iso𝔹-intro : ∀{x y : A} → x ≤A y ≡ tt → y ≤A x ≡ tt → x iso𝔹 y ≡ tt
iso𝔹-intro p1 p2 rewrite p1 | p2 = refl
```

Binary Search Trees

```
open import bool-relations using (transitive ; total)

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)
```

Inserting an Element into a BST

Specification:

$$\text{bst-insert} : \forall \{l\,u : A\}(d : A) \longrightarrow \text{bst}\; l\; u \longrightarrow \text{bst}\;\; (\min d\; l)\;\; (\max d\; u)$$

```
min : A → A → A
min = λ x y → if x ≤A y then x else y

max : A → A → A
max = λ x y → if x ≤A y then y else x
```

Some Properties about min/max

```
min-≤1 : ∀{x y : A} → min x y ≤A x ≡ tt

min-≤2 : ∀{x y : A} → min x y ≤A y ≡ tt

max-≤1 : ∀{x y : A} → x ≤A max x y ≡ tt

max-≤2 : ∀{x y : A} → y ≤A max x y ≡ tt

min2-mono : ∀{x y y' : A} → y ≤A y' ≡ tt →
            min x y ≤A min x y' ≡ tt

max2-mono : ∀{x y y' : A} → y ≤A y' ≡ tt →
            max x y ≤A max x y' ≡ tt
```

Two Helper Functions for Changing Bounds

```
bst-dec-lb : ∀ {l l' u' : A} → bst l' u' →
             l ≤A l' ≡ tt → bst l u'
bst-dec-lb (bst-leaf p) q = bst-leaf (≤A-trans q p)
bst-dec-lb (bst-node d L R p1 p2) q =
  bst-node d L R (≤A-trans q p1) p2
```

```
bst-inc-ub : ∀ {l' u' u : A} → bst l' u' →
             u' ≤A u ≡ tt → bst l' u
bst-inc-ub (bst-leaf p) q = bst-leaf (≤A-trans p q)
bst-inc-ub (bst-node d L R p1 p2) q =
  bst-node d L R p1 (≤A-trans p2 q)
```

Final Implementation

```
bst-insert : ∀{l u : A}
               (d : A) → bst l u → bst (min d l) (max d u)
bst-insert d (bst-leaf p) =
    bst-node d (bst-leaf ≤A-refl) (bst-leaf ≤A-refl)
             min-≤1 max-≤1
bst-insert d (bst-node d' L R p1 p2) with keep (d ≤A d')
bst-insert d (bst-node d' L R p1 p2) | tt , p with bst-insert d L
bst-insert d (bst-node d' L R p1 p2) | tt , p | L' rewrite p =
    bst-node d' L' (bst-inc-ub R (≤A-trans max-≤2 p2))
             (min2-mono p1) ≤A-refl
bst-insert d (bst-node d' L R p1 p2) | ff , p with bst-insert d R
bst-insert d (bst-node d' L R p1 p2) | ff , p | R' rewrite p =
    bst-node d' (bst-dec-lb L p1) R'
              min-≤2 (max2-mono p2)
```

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.