# Introducción a la Verificación Formal

Clase 5 – 25/09/2025

# Clase pasada: BSTs

- Los definimos mediante un simple tipo inductivo
- Pudimos demostrar algunas propiedades como:
  - `forall x t. member x (insert x t)`
  - Otras sobre tamaño y altura
- La propiedad `member x (insert y (insert x t))` puede demostrarse… pero requiere razonar sobre la forma del árbol
- La propiedad `delete x (insert x t) == t` no vale, dado que la *forma* del árbol puede cambiar. Dos BST pueden ser equivalentes sin ser iguales.

```
type bst =
  | L
  | N of bst & int & bst
```

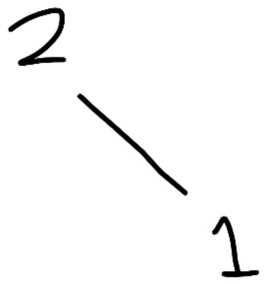
# ¿BSTs?

```
type bst =
  | L
  | N of bst & int & bst
```

- ¿Realmente estábamos verificando BSTs?
- ¿Se puede demostrar la propiedad siguiente?

```
let delete_not_mem (x y : int) (t : bst)
 : Lemma (requires not (member x t))
         (ensures (not (member x (delete y t)))) = …

let falso () : Lemma False =
  delete_not_mem 1 2 (N (L, 2, N (L, 1, L)))
```




- Todo lo que demostramos hasta ahora no hace uso del *invariante* de los BST (o no podríamos haberlo demostrado)

# Refinando la estructura BSTs

- Una forma usual de trabajar con estructuras con invariantes es definir primero la versión “base”, y luego refinarla.
- Las funciones relevantes suelen definirse sobre la versión base, y luego se demuestra que preservan los invariantes.
- Las versiones refinadas de las funciones son operacionalmente iguales a las versiones base
  - Las pruebas y los refinamientos se borran

```
type bst0 =
  | L
  | N of bst0 & int & bst0

let rec all_lt (x: int) (t: bst0) : bool =
  match t with
  | L -> true
  | N (l, y, r) -> all_lt x l && y < x && all_lt x r

let rec all_gt (x: int) (t: bst0) : bool =
  match t with
  | L -> true
  | N (l, y, r) -> all_gt x l && y > x && all_gt x r

let rec is_bst (t: bst0) : bool =
  match t with
  | L -> true
  | N (l, x, r) -> is_bst l && is_bst r && all_lt x l && all_gt x r
```

```
type bst = b:bst0{is_bst b}
```

```
let rec insert0 (x: int) (t: bst0) : bst0 =
  ...

let rec insert0_bst (x:int) (t:bst0)
  : Lemma (requires is_bst t)
          (ensures is_bst (insert0 x t)) =
  ...

let insert (x:int) (t:bst) : bst =
  insert0_bst x t;
  insert0 x t
```

# Un poco sobre la codificación a SMT

- SMT = *satisfacibilidad módulo teorías*
  - Esencialmente un SAT solver con anabólicos.
  - Un SAT solver resuelve el problema de satisfacibilidad booleana: dada una fórmula con variables, `&&`, `||`, `not` nos da:
    - Un modelo (asignación booleana) que satisface la formula
    - O, contesta que el problema es *insatisfacible* (**unsat**)
    - El problema es decidible, pero NP-completo. Aun así, hoy en día los SAT solver tienen buena performance en general.
  - SMT = SAT + *cuantificadores* (FOL) + otras teorías (enteros, reales, listas, tipos algebraicos, etc)
    - No es decidible (FOL ya no lo es).
    - Contesta **sat**, **unsat**, o **unknown**.
    - En el caso **sat**, nos puede devolver un **módelo**. Típico ejemplo: resolver sudokus.

```
(declare-datatypes () ((Val V1 V2 V3 V4 V5 V6 V7 V8 V9)))
(declare-fun board (Int Int) Val)

(define-fun valid_index ((i Int)) Bool
  (and (>= i 0) (< i 9)))

;; All values in a row are unique.
(assert
  (forall ((row Int) (i Int) (j Int))
    (=>
      (and
        (not (= i j))
        (valid_index row)
        (valid_index i)
        (valid_index j))
      (not (= (board row i)
              (board row j))))))

;; All values in a column are unique.
(assert
  (forall ((col Int) (i Int) (j Int))
    (=>
      (and
        (not (= i j))
        (valid_index col)
        (valid_index i)
        (valid_index j))
      (not (= (board i col)
              (board j col))))))

;; All values in each box are unique.
(assert
  (forall ((row1 Int) (col1 Int)
           (row2 Int) (col2 Int))
    (=>
      ;; If the below 3 are true...
      (and
        ;; 1. Row and column indices are in bounds.
        (valid_index row1) (valid_index col1)
        (valid_index row2) (valid_index col2)
        ;; 2. They are not the same elements.
        (or (not (= row1 row2))
            (not (= col1 col2)))
        ;; 3. They do exist in the same box.
        (= (div row1 3) (div row2 3))
        (= (div col1 3) (div col2 3)))
      ;; ... then the values must differ!
      (not (= (board row1 col1)
              (board row2 col2))))))
```

```
;; Use this board:
; -------------------------
; |       | 8   6 | 7     |
; |       | 2 9   |   4   |
; | 9     |     7 |     6 |
; |-----------------------|
; | 5     |     8 | 4     |
; | 6     |       | 2 5   |
; |     2 | 7 6   |       |
; |-----------------------|
; |       | 6     | 1 7   |
; |       |   7   | 5   4 |
; |   4   |   1   |   9   |
; -------------------------
;; Row 0
(assert (= (board 0 3) V8))
(assert (= (board 0 5) V6))
(assert (= (board 0 6) V7))
;; Row 1
(assert (= (board 1 3) V2))
(assert (= (board 1 4) V9))
(assert (= (board 1 7) V4))
;; Row 2
(assert (= (board 2 0) V9))
(assert (= (board 2 5) V7))
(assert (= (board 2 8) V6))
;; Row 3
(assert (= (board 3 0) V5))
(assert (= (board 3 5) V8))
(assert (= (board 3 6) V4))
;; Row 4
(assert (= (board 4 0) V6))
(assert (= (board 4 6) V2))
(assert (= (board 4 7) V5))
;; Row 5
(assert (= (board 5 2) V2))
(assert (= (board 5 3) V7))
(assert (= (board 5 4) V6))
;; Row 6
(assert (= (board 6 3) V6))
(assert (= (board 6 6) V1))
(assert (= (board 6 7) V7))
;; Row 7
(assert (= (board 7 4) V7))
(assert (= (board 7 6) V5))
(assert (= (board 7 8) V4))
;; Row 8
(assert (= (board 8 1) V4))
(assert (= (board 8 4) V1))
(assert (= (board 8 7) V9))
```

```
(check-sat)
;(get-model)

(echo "Row:0")
(get-value ((board 0 0) (board 0 1) (board 0 2) (board 0 3) (board 0 4) (board 0 5) (board 0 6) (board 0 7) (board 0 8)))
(echo "Row:1")
(get-value ((board 1 0) (board 1 1) (board 1 2) (board 1 3) (board 1 4) (board 1 5) (board 1 6) (board 1 7) (board 1 8)))
(echo "Row:2")
(get-value ((board 2 0) (board 2 1) (board 2 2) (board 2 3) (board 2 4) (board 2 5) (board 2 6) (board 2 7) (board 2 8)))
(echo "Row:3")
(get-value ((board 3 0) (board 3 1) (board 3 2) (board 3 3) (board 3 4) (board 3 5) (board 3 6) (board 3 7) (board 3 8)))
(echo "Row:4")
(get-value ((board 4 0) (board 4 1) (board 4 2) (board 4 3) (board 4 4) (board 4 5) (board 4 6) (board 4 7) (board 4 8)))
(echo "Row:5")
(get-value ((board 5 0) (board 5 1) (board 5 2) (board 5 3) (board 5 4) (board 5 5) (board 5 6) (board 5 7) (board 5 8)))
(echo "Row:6")
(get-value ((board 6 0) (board 6 1) (board 6 2) (board 6 3) (board 6 4) (board 6 5) (board 6 6) (board 6 7) (board 6 8)))
(echo "Row:7")
(get-value ((board 7 0) (board 7 1) (board 7 2) (board 7 3) (board 7 4) (board 7 5) (board 7 6) (board 7 7) (board 7 8)))
(echo "Row:8")
(get-value ((board 8 0) (board 8 1) (board 8 2) (board 8 3) (board 8 4) (board 8 5) (board 8 6) (board 8 7) (board 8 8)))
```

```
$ z3 sudoku.smt2
sat
Row:0
(((board 0 0) V1) ((board 0 1) V5) ((board 0 2) V3) ((board 0 3) V8) ((board 0 4) V4) ((board 0 5) V6) ((board 0 6) V7) ((board 0 7) V2) ((board 0 8) V9))
Row:1
(((board 1 0) V8) ((board 1 1) V7) ((board 1 2) V6) ((board 1 3) V2) ((board 1 4) V9) ((board 1 5) V1) ((board 1 6) V3) ((board 1 7) V4) ((board 1 8) V5))
Row:2
(((board 2 0) V9) ((board 2 1) V2) ((board 2 2) V4) ((board 2 3) V3) ((board 2 4) V5) ((board 2 5) V7) ((board 2 6) V8) ((board 2 7) V1) ((board 2 8) V6))
Row:3
(((board 3 0) V5) ((board 3 1) V3) ((board 3 2) V9) ((board 3 3) V1) ((board 3 4) V2) ((board 3 5) V8) ((board 3 6) V4) ((board 3 7) V6) ((board 3 8) V7))
Row:4
(((board 4 0) V6) ((board 4 1) V1) ((board 4 2) V7) ((board 4 3) V4) ((board 4 4) V3) ((board 4 5) V9) ((board 4 6) V2) ((board 4 7) V5) ((board 4 8) V8))
Row:5
(((board 5 0) V4) ((board 5 1) V8) ((board 5 2) V2) ((board 5 3) V7) ((board 5 4) V6) ((board 5 5) V5) ((board 5 6) V9) ((board 5 7) V3) ((board 5 8) V1))
Row:6
(((board 6 0) V2) ((board 6 1) V9) ((board 6 2) V5) ((board 6 3) V6) ((board 6 4) V8) ((board 6 5) V4) ((board 6 6) V1) ((board 6 7) V7) ((board 6 8) V3))
Row:7
(((board 7 0) V3) ((board 7 1) V6) ((board 7 2) V1) ((board 7 3) V9) ((board 7 4) V7) ((board 7 5) V2) ((board 7 6) V5) ((board 7 7) V8) ((board 7 8) V4))
Row:8
(((board 8 0) V7) ((board 8 1) V4) ((board 8 2) V8) ((board 8 3) V5) ((board 8 4) V1) ((board 8 5) V3) ((board 8 6) V6) ((board 8 7) V9) ((board 8 8) V2))
```

# Descargando VCs

- Una *condición de verificación (VC)* es una fórmula que debe valer para que un programa sea correcto.
  - F* computa VCs como parte del proceso de chequeo de tipos.
  - Normalmente, se acumulan durante todo el chequeo de una definición top-level.
  - Finalmente, se envía al SMT solver.
    - Concretamente, para una VC ϕ, se agrega `(assert (not φ))` al contexto del SMT. Si contesta **unsat**, la VC es verdadera (¡tercero excluído!).
  - Si el SMT contesta **unknown**, se rechaza la definición.
  - Nunca contesta **sat** para queries de F*, pero en este caso es cuando otras herramientas intentan construir un contraejemplo en términos fuente.

# Computando VCs

- En general, complejo. Estas son algunas reglas de a pie.
- Un argumento refinado `x:t{p x}` agrega una hipótesis `p x`
  - `ϕ ~> (p x ==> ϕ)`
- Una precondición (`requires`) es similar. Un assume es similar.
- Una postcondición p es una nueva obligación
  - `ϕ ~> ϕ` /\ p
- Un `assert p` agrega un *corte* en la prueba.
  - `ϕ ~> p` /\ (p ==> ϕ)
  - "Formalmente" no agrega poder. En la práctica es muy útil.
- El SMT, por su cuenta, analiza por casos, destruye constructores, razona sobre aritmética, etc.
  - **No** hace inducción, ni razona bien sobre funciones recursivas.
  - En general, tenemos que usar lemas auxiliares.

# Tareas

- Clase05.*.fst: completar removiendo los admits/assume.