Plan

- Why3
- demo with merge sort
- and dfs for graphs
- conclusions

Goal

Write elegant proofs for elegant programs

- training in program proofs checked by computers
- useful to teach algorithms

.. with Chen Ran (Iscas)

Why3

  LRI (orsay) + Inria + Cnrs [Filliâtre, Paskevich, Marché…]
- small Pascal-like imperative programming language
  [ with ML syntax ]
- invariants + assertions in Hoare logic
  [ + recursive functions, inductive datatypes, inductive predicates ]
- interfaces with modern automatic provers
  [ alt-ergo, cvc3, cvc4, eprover, gappa, simplify, spass, yices, z3, … ]
- interfaces with interactive proof assistants
  [ coq, pvs, isabelle ]
**MLW programming language**

```ocaml
let swap (a: array int) (i: int) (j: int) =  
  let v = a[i] in  
  a[i] <- a[j];  
  a[j] <- v;  

let selection_sort (a: array int) =  
  for i = 0 to length a - 1 do  
    let imin = ref i in  
    for j = i + 1 to length a - 1 do  
      if a[j] < a[imin] then imin := j  
    done;  
    swap a !imin i  
  done
```

**Why3 theories**

- theories about arrays

```ocaml
let swap (a: array int) (i: int) (j: int) =  
  requires { 0 <= i < length a /
              0 <= j < length a }  
  ensures { exchange (old a) a i j }  
  let v = a[i] in  
  a[i] <- a[j];  
  a[j] <- v;  
```

(see the why3 libraries)


**Hoare logic**

```ocaml
let swap (a: array int) (i: int) (j: int) =  
  let v = a[i] in  
  a[i] <- a[j];  
  a[j] <- v;  

let selection_sort (a: array int) =  
  for i = 0 to length a - 1 do  
    let imin = ref i in  
    for j = i + 1 to length a - 1 do  
      invariant { i <= imin < j }  
      invariant { forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }  
      invariant { forall k: int. i <= k < j -> a[kimin] <= a[k] }  
      if a[j] < a[kimin] then imin := j  
    done;  
    swap a !imin i  
  done
```

**Full program**

```ocaml
let selection_sort (a: array int) =  
  ensures { sorted a /
            permut (old a) a }  
  'l':  
  for i = 0 to length a - 1 do  
    invariant { sorted_sub a 0 i /
                permut (at a 'l) a }  
    invariant { forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }  
    let imin = ref i in  
    for j = i + 1 to length a - 1 do  
      invariant { i <= imin < j }  
      invariant { forall k: int. i <= k < j -> a[kimin] <= a[k] }  
      if a[j] < a[kimin] then imin := j  
    done;  
    swap a !imin i  
  done
```
An example

Mergesort (1/3)

Mergesort (2/3)

Mergesort (3/3)
Full program (1/2)

```plaintext
let rec mergeSort (a: array int) (lo hi: int) = 
    requires { Array.length a = Array.length b ∧ 
        0 <= lo <= (Array.length a) ∧ 0 <= hi <= (Array.length a) }
    ensures { sorted_sub a lo hi ⇐ modified_inside (old a) a lo hi }
if lo = 1 ∧ hi then
    let m = div (lo+hi) 2 in 
    assert { lo = m < hi );
    mergeSort a a m; 
    I2: mergeSort a b m hi;
    assert { array_eq_sub (at a 'I2) a lo m };
    for i = lo to m-1 do 
        invariant { array_eq_sub b a lo i } 
        b[i] <= a[i];
    done;
    assert { array_eq_sub b a lo m }; 
    assert { sorted_sub b lo m };
    for j = m to hi-1 do 
        invariant { array_eq_sub_rev_offset b a m j (hi-j) } 
        b[j] <= a[lo + hi - 1 - j ];
    done;
    assert { array_eq_sub b a m lo m };
    assert { sorted_sub b lo m };
    assert { array_eq_sub_rev_offset b a m hi 0 };
    assert { dsorted_sub b a m hi };
```

Full program (logic 1/2)

```plaintext
module MergeSort

use import int.Int
use import int.EuclideanDivision
use import int.Div2
use import ref.Ref
use import array.Array
use import array.ArraySorted
use import array.ArrayPerm
use import array.ArrayEq
use import map.Map

clone map.MapSort as N with type elt = int, predicate le = (<=)

predicate map_eq_sub_rev_offset (a1 a2: M.map int int) (l u: int) (offset: int) = 
    forall i: int. 1 <= i < u → M.get a1 i = N.get a2 (offset + i + u - 1 - i)

predicate array_eq_sub_rev_offset (a1 a2: array int) (l u: int) (offset: int) = 
    map_eq_sub_rev_offset a1.elts a2.elts l u offset

predicate dsorted_sub (a: M.map int int) (l u: int) = 
    forall i1 i2: int. 1 <= i1 <= i2 < u → M.get a i1 <= M.get a i2

predicate dsorted_sub (a: array int) (l u: int) = 
    map_dsorted_sub a.elts l u
```

Full program (2/2)

```plaintext
' I4: let i = ref lo in
    let j = ref hi in
    for k = lo to hi-1 do 
        invariant{ [lo < hi ∧ lo <= lj <= hi] }
        invariant{ sorted_sub a lo k } 
        invariant{ forall k1 k2: int. lo <= k1 < k ∧ k1 <= k2 < hi → a[k1] <= b[k2] } 
        invariant{ bitonic b [l [j] ] } 
        invariant{ modified_inside a (at a 'I4) lo hi } 
        assert { [i < j] };
        if b[i] < b[j] - 1 then
            begin
                a[k] <- b[i];
                i := i + 1 end
        else
            begin
                j := j - 1;
                a[k] <- b[j] end
    done

let mergeSort (a: array int) =
    assert { sorted a };
    let n = Array.length a in
    let b = Array.make n 0 in 
    mergeSort1 a b 0 n
```

Full program (logic 2/2)

```plaintext
predicate map_bitonic_sub (a: M.map int int) (l u: int) = 
    exists l: int. 1 <= l < u ∧ N.sorted_sub a l [l] ∧ map_dsorted_sub a l u

predicate bitonic (a: array int) (l u: int) = 
    map_bitonic_sub a.elts l u

lemma map_bitonic_incr : forall a: M.map int int, l u: int. 
    map_bitonic_sub a l u → map_bitonic_sub a (l+1) u

lemma map_bitonic_decr : forall a: M.map int int, l u: int. 
    map_bitonic_sub a l u → map_bitonic_sub a (l-1) u

predicate modified_inside (a1 a2: array int) (l u: int) = 
    array_eq_sub a1 a2 0 1 ∧ array_eq_sub a1 a2 u (Array.length a1)
```
Examples with Graphs

Depth-first search in graphs (1/4)

- reachability [the ‘white path theorem’]
- non white-to-black edges in undirected graphs
- acyclicity test
- articulation points
- strongly connected components

Kosaraju, Tarjan
Depth-first search in graphs (2/4)

- representation as array of lists of successors

Depth-first search in graphs (3/4)

- spanning trees = call graph of DFS

Undirected graph: no W2B arc (1/3)

function order (g: graph) : int = length g
predicate vertex (g: graph) (x: int) = 0 <= x < order g
predicate out (g: graph) (x: int) =
for all y in sons do
  if c[y] = WHITE then dfs y c;
done;
c[x] = BLACK

let dfs (g: graph) (x: int) (c: array color) =
c[x] = GRAY;
let sons = ref (g[x]) in
while !sons do
  match !sons with
  | Nil -> ()
  | Cons y sons' ->
    if c[y] = WHITE then dfs y c;
    sons := sons';
  end;
done;
c[x] = BLACK

let dfs_main (g: graph) =
let n = length (g) in
let c = Array.make n WHITE in
for x = 0 to n - 1 do
  if c[x] = WHITE then
dfs g x c
done

Depth-first search in graphs (4/4)

- spanning trees = call graph of DFS in mlw

let rec dfs (g: graph) (x: int) (c: array color) =
c[x] = GRAY;
let sons = ref (g[x]) in
while !sons do
  match !sons with
  | Nil -> ()
  | Cons y sons' ->
    if c[y] = WHITE then dfs g y c;
    sons := sons';
  end;
done;
c[x] = BLACK

let dfs_main (g: graph) =
let n = length (g) in
let c = Array.make n WHITE in
for x = 0 to n - 1 do
  if c[x] = WHITE then
dfs g x c
done

Undirected graph: no W2B arc (1/3)

function order (g: graph) : int = length g
predicate vertex (g: graph) (x: int) = 0 <= x < order g
predicate out (g: graph) (x: int) =
for all y in sons do
  if c[y] = WHITE then dfs y c;
done;
c[x] = BLACK

let rec dfs (g: graph) (x: int) (c: array color) =
c[x] = GRAY;
let sons = ref (g[x]) in
while !sons do
  match !sons with
  | Nil -> ()
  | Cons y sons' ->
    if c[y] = WHITE then dfs y c;
    sons := sons';
  end;
done;
c[x] = BLACK

let dfs_main (g: graph) =
let n = length (g) in
let c = Array.make n WHITE in
for x = 0 to n - 1 do
  if c[x] = WHITE then
dfs g x c
done

Undirected graph: no W2B arc (1/3)

function order (g: graph) : int = length g
predicate vertex (g: graph) (x: int) = 0 <= x < order g
predicate out (g: graph) (x: int) =
for all y in sons do
  if c[y] = WHITE then dfs y c;
done;
c[x] = BLACK

let rec dfs (g: graph) (x: int) (c: array color) =
c[x] = GRAY;
let sons = ref (g[x]) in
while !sons do
  match !sons with
  | Nil -> ()
  | Cons y sons' ->
    if c[y] = WHITE then dfs y c;
    sons := sons';
  end;
done;
c[x] = BLACK
Undirected graph: no W2B arc (2/3)

```ocaml
let rec dfs (g: graph) (x: int) (c: array color) =
  requires [wf g /\ vertex g x /\ length c = order g]
  requires [noW2BEdge g c]
  ensures [old c][x] = WHITE -> c[x] = WHITE]
  ensures [white_monotony g (old c) c]
  ensures [noW2BEdge g c]
  'L:
  c[x] <- GRAY;
  let sons = ref (g[x]) in
  while !sons <> Nil do
    invariant { white_monotony g (at c 'L) c}
    invariant {forall y: int. men y !sons -> edge x y}
    invariant {forall y: int. edge x y -> c[y] = WHITE -> mem y !sons}
    invariant {noW2BEdge g c}
    match !sons with
    | Nil -> ()
    | Cons y sons' ->
      if c[y] = WHITE then dfs g y c;
      sons := sons';
    end;
  done;
  c[x] <- BLACK
```

White paths (1/2)

- if white path between x and y, then dfs(x) flips y to black

```ocaml
let rec dfs (g: graph) (x: int) (c: array color) =
  requires [wf g /\ vertex g x /\ Array.length c = order g]
  requires [c[x] = WHITE]
  ensures [white_monotony g (old c) c]
  ensures [node_flip_whitepath x g (old c) c] (*new*)
  ensures [whitepath_node_flip x g (old c) c]
  'L0:
  c[x] <- GRAY;
  assert [forall y z: int, l: list int. mem y g[x] ->
      whitepath y l z g -> whitepath x (Cons x l) z g (at c 'L0)];
  'L:
```

Undirected graph: no W2B arc (3/3)

```ocaml
let dfs_main (g: graph) =
  requires [wf g]
  let n = length (g) in
  let c = make n WHITE in
  for x = 0 to n - 1 do
    invariant {noW2BEdge g c}
    if c[x] = WHITE then
      dfs g x c
    done
```

White paths (2/2)

- why these invariants?
- are they natural?
- can be found automatically?
Strongly connected components (1/2)

• in spanning trees, no left-to-right edge [Wengener, Pottier]
• SCC are prefixes of subtrees

Strongly connected components (2/2)

• if y connected to x by nodes less than x in post-order traversal of spanning tree, then x is connected to y [Kosaraju]
• if x cannot reach a node y less than x in pre-order traversal of spanning tree, then x is the root of its component [Tarjan]

Conclusions

Conclusion 1

• **Automatic** part of proof for **tedious** case analyzes
• **Interactive** proofs for the **conceptual** part of the algorithm
  
  - the ideal world
• From interactive part, one can call the automatic part
  
  - possible extensions of Why3 theories
  
  - but typing problems (inside Coq)
Conclusion 2

- Hoare logic prevents to write awkward denotational semantics
- Nobody cares about termination 😞
- Explore simple programs about algorithms before jumping to large programs.
- Why3 memory model is naive. It’s a «back-end for other systems».
- Also experimenting on graph algorithms and prove all algorithms in Sedgewick’s book.

Conclusion 3

- Why3 is excellent for mixing formal proofs and SMT’s calls
- Still rough for beginners
- Concurrency ?
- Functional programs ?
- Hoare logic vs Type refinements (F* [MSR])
- Frama-C project at french CEA extends Why3 to C programs.