Incorrectness & QuickCheck

Related Works

- Verifying Correct Usage of Context-Free API Protocols
  - Looks very similar with symbolic execution, but with abstract domain which over-approximate the condition of the branches
  - The benchmarks are kind of simple: lock&unlock, restore&save… I think the protocol is a simple version of specifications
- Experiences with QuickCheck: Testing the Hard Stu and Staying Sane
  - The queue example inspired me.
  - QuickCheck generate sequence of library calls, actually has no client(or the client are actually the “properties”)

Example: Library

When the idx is larger than the length of allocated array, will cause “index out of bounds” error.

```
module Stack = struct
  type 'a t = ('a array) * int

  let empty s default = Array.make s default, 0

  let is_empty (_, s) = s <= 0

  let mem (a, s) x =
    let rec aux idx =
      if idx == s then false else
      if Array.get a idx == x then true else
        aux (idx + 1)
    in
    aux 0
  let push (a, s) x =
    let _ = Array.set a s x in
    a, s + 1
  let pop (a, s) =
    let x = Array.get a (s - 1) in
    (a, s - 1), x
  let top (a, s) = Array.get a (s - 1)
end
```

Example Client

Find an element in graph which is less than “target” by dfs.

Input is a graph, start node, “target”

Output true such element exists

The stack has a fixed size(4).

The graph which makes stack over bound is not easy to find.

```
let dfs g start target =
  let checked = Set.empty (G.num_nodes g) in
  let st = Stack.empty 4 0 in
  if start <= target then true else
    let checked = Set.insert checked start in
    let rec loop (checked, st, pre, cur) =
      let rec next iter =
        if iter == (G.num_nodes g) then None else
        if G.if_edge g pre iter && (not (Set.mem checked
iter)) then
          Some iter
        else
          next (iter + 1)
      in
      match next cur with
      | None ->
        if Stack.is_empty st then false
        else
          let cur = pre in
          let st, pre = Stack.pop st in
          loop (checked, st, pre, cur)
      | Some iter' ->
        if iter' <= target then true else
          let checked = Set.insert checked iter' in
          let st = Stack.push st cur in
          loop (checked, st, iter', 0)
    in
    loop (checked, st, start, 0)
```

Random Generator

For convenience, the number of each node in graph is distinct, from 0 to n

Random generate adjacency matrix, start node and “target”.

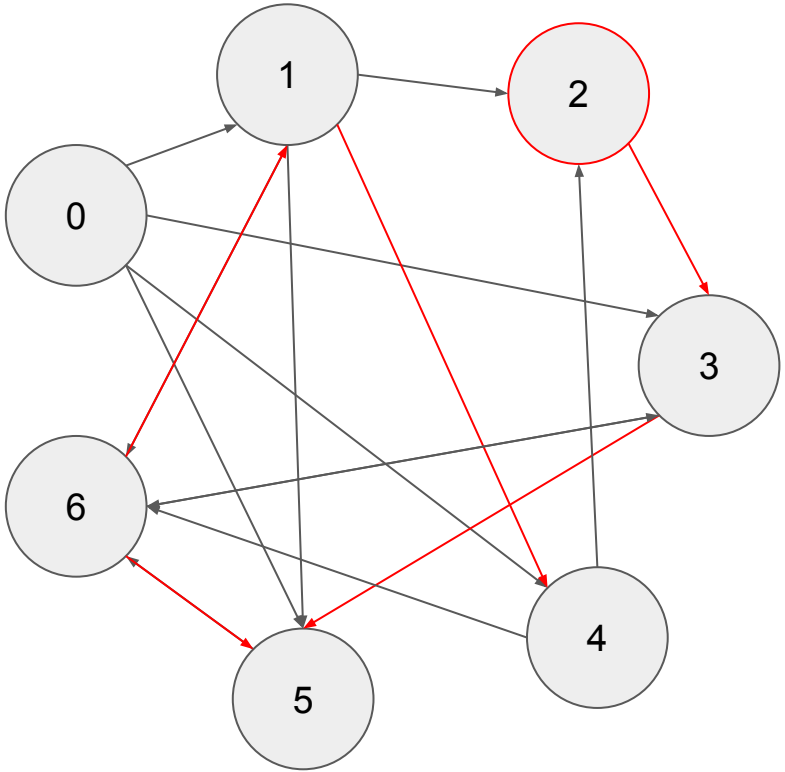
Uniform distribution.

Assume start node and “target” less than the number of the nodes.

Bad input

- The graph should be deep enough(depth is greater than 4)
- The path should not contains elements less than “target”
- Depends on the order of searching(dfs implementation), if there are 3 neighborhoods, which one search first.

Example
[false; true; false; true; true; true; false;
false; true; true; false; true; true; true;
false; false; true; true; false; false; false;
false; false; false; false; false; true; true;
false; false; true; false; true; false; true;
false; false; false; false; false; true; true;
false; true; false; true; false; true; true];
start = 2; x = 0



2 -> 3 -> 5 -> 6 -> 1 -> 4

Spec

The the dfs client, the error in pop will never happen, as we check is the stack is empty before pop. The error in push may happen.

One execution path can lead the error:

push st (x1); … push st (x2); … push st (x3); … push st (x4); … push st (x5); error

```
module Stack = struct
…
  {s == length a} push (a, s) x {error: index out of bounds}
  {s < length a} push (a, s) x {ok: s' = s + 1 /\ length a = length a'}
  let push (a, s) x =
    let _ = Array.set a s x in
    a, s + 1

  {s <= 0} push (a, s) x {error: index out of bounds}
  {s > 0} push (a, s) x {ok: s' = s - 1 /\ length a = length a'}
  let pop (a, s) =
    let x = Array.get a (s - 1) in
    (a, s - 1), x
  let top (a, s) = Array.get a (s - 1)
end
```

Method

Sample “push” and “pop”, try to infer specs:

1. Consistent with samples
2. There exists a path on client side, the specs on this path will cause an error.

Samples

Input:
  a: random allocated array,
  s: random integer
  x: random integer

```
let push (a, s) x =
    let _ = Array.set a s x in
    a, s + 1
```

a = [|1;2;3|], s = 3, x = 0 ⇒ error
a = [|1;2;3|], s = 2, x = 0 ⇒ a' = [|1;0;3|], s = 3
a = [|1;2;3;3|], s = 5, x = 3 ⇒ error
...

Feature:
  length: array -> int
  Equality: int -> int -> bool
  Less than: int -> int -> bool

Bad under-approximation:

{s >= length a} push (a, s) x {error: index out of bounds}

Over-approximate too much

Client

Path:
Stack.empty 4 0 … Stack.push st x1 …
end

Constraint of Path:
st = empty 4 0 /\ start > target /\ next cur
== Some iter' /\ push st x1 = st'

However, this path will not lead error. We can use this path and to generate corresponding inputs, if within a bound, we cannot find a bug, we believe the spec is wrong.

Add these safe inputs back to the sampling of "push" to refine under-approximation spec.(The grey sample)

```
let dfs g start target =
  let checked = Set.empty (G.num_nodes g) in
  let st = Stack.empty 4 0 in
  if start <= target then true else
    let checked = Set.insert checked start in
    let rec loop (checked, st, pre, cur) =
      let rec next iter =
        if iter == (G.num_nodes g) then None else
        if G.if_edge g pre iter && (not (Set.mem checked
iter)) then
          Some iter
        else
          next (iter + 1)
      in
      match next cur with
      | None ->
        if Stack.is_empty st then false
        else
          let cur = pre in
          let st, pre = Stack.pop st in
          loop (checked, st, pre, cur)
      | Some iter' ->
        if iter' <= target then true else
          let checked = Set.insert checked iter' in
          let st = Stack.push st cur in
          loop (checked, st, iter', 0)
    in
    loop (checked, st, start, 0)
```