Homework 1
AAA616, Fall 2022

Hakjoo Oh

Due: 10/30, 23:59

Problem 1 수업 시간에 디자인 한 인터벌 분석기(interval analyzer)를 구현해보자. While 언어의 문법 구조를 아래와 같이 정의하자.

$$\begin{array}{rcl} a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\ b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ c & \rightarrow & x := a \mid c_1; c_2; \ldots; c_k \mid \text{if } b\ c_1\ c_2 \mid \text{while } b\ c \end{array}$$

OCaml 데이터 타입으로 표현하면 아래와 같다.

```
type aexp =
  | Const of int
  | Var of string
  | Plus of aexp * aexp
  | Mult of aexp * aexp
  | Sub of aexp * aexp

type bexp =
  | True
  | False
  | Equal of aexp * aexp
  | Le of aexp * aexp
  | Not of bexp
  | And of bexp * bexp

type cmd =
  | Assign of string * aexp
  | Seq of cmd list
  | If of bexp * cmd * cmd
  | While of bexp * cmd
```

1. 첫 번째 스텝은 프로그램을 제어 흐름 그래프(control flow graph)로 변환하는 것이다. 아래 타입을 가지는 노드 및 그래프 구현은 제공된다.

   ```
   module type Node = sig
     type instr =
     | I_assign of string * aexp
     | I_assume of bexp
     | I_skip
     type t
     val create_assign : string -> aexp -> t
     val create_assume : bexp -> t
   ```

```
  val create_skip : unit -> t
  val get_nodeid : t -> int
  val get_instr : t -> instr
  val to_string : t -> string
  val compare : t -> t -> int
end

module type Cfg = sig
  type t
  val empty : t
  val nodesof : t -> Node.t list
  val succs : Node.t -> t -> NodeSet.t
  val preds : Node.t -> t -> NodeSet.t
  val add_node : Node.t -> t -> t
  val add_nodes : Node.t list -> t -> t
  val add_edge : Node.t -> Node.t -> t -> t
  val print : t -> unit
  val dot : t -> unit
end
```

프로그램을 제어 흐름 그래프로 변환하는 함수 `cmd2cfg`를 작성하시오:

```
cmd2cfg : cmd -> Cfg.t
```

2. 두 번째 스텝으로 요약 값(abstract value)들을 정의하자. 아래 타입을 가지는 두 모듈 `AbsBool` 및 `Interval`을 완성하시오.

```
module type AbsBool = sig
  type t = Top |  Bot | True | False
  val not : t -> t
  val band : t -> t -> t
end

module AbsBool : AbsBool = struct
  type t = Top | Bot | True | False
  let not b = b (* TODO *)
  let band b1 b2 = b1 (* TODO *)
end

module type Interval = sig
  type t
  val bottom : t
  val to_string : t -> string
  val alpha : int -> t
  val alpha_to : int -> t
  val alpha_from : int -> t
  val order : t -> t -> bool
  val join : t -> t -> t
  val meet : t -> t -> t
  val widen : t -> t -> t
  val narrow : t -> t -> t
  val add : t -> t -> t
  val mul : t -> t -> t
  val sub : t -> t -> t
```

```
  val equal : t -> t -> AbsBool.t
  val le : t -> t -> AbsBool.t
  val ge : t -> t -> AbsBool.t
end

module Interval : Interval = struct
  type t = Bot (* TODO *)
  let bottom = Bot
  let to_string i = "" (* TODO *)
  let alpha n = Bot (* TODO *)
  let alpha_to n = Bot (* TODO *)
  let alpha_from n = Bot (* TODO *)
  let order a b = true (* TODO *)
  let join a b = a (* TODO *)
  let meet a b = a (* TODO *)
  let widen a b = a (* TODO *)
  let narrow a b = a (* TODO *)
  let add a b = a (* TODO *)
  let mul a b = a (* TODO *)
  let sub a b = a (* TODO *)
  let equal a b = AbsBool.Top (* TODO *)
  let le a b = AbsBool.Top (* TODO *)
  let ge a b = AbsBool.Top (* TODO *)
end
```

3. 요약 메모리(abstract memory)를 정의하자. 아래와 같은 타입을 가지는 모듈로 정의한다.

```
module VarMap = Map.Make(String)
module type AbsMem = sig
  type t = Interval.t VarMap.t
  val empty : t
  val add : string -> Interval.t -> t -> t
  val find : string -> t -> Interval.t
  val join : t -> t -> t
  val widen : t -> t -> t
  val narrow : t -> t -> t
  val order : t -> t -> bool
  val print : t -> unit
end

module AbsMem : AbsMem = struct
  type t = Interval.t VarMap.t
  let empty = VarMap.empty
  let add x v m = m (* TODO *)
  let find x m = Interval.bottom (* TODO *)
  let join m1 m2 = m1 (* TODO *)
  let widen m1 m2 = m1 (* TODO *)
  let narrow m1 m2 = m1 (* TODO *)
  let order m1 m2 = true (* TODO *)
  let print m = VarMap.iter (fun x v ->
   prerr_endline (x ^ " |-> " ^ Interval.to_string v)) m
end
```

4. 마지막으로 워크리스트를 이용하는 고정점 계산 알고리즘을 구현해보자. 분석기는 노드에서 요약 메모리로 가는 테이블을 계산해야 한다. 테이블 구현은 아래와 같이 제공된다.

```
module type Table = sig
  type t = AbsMem.t NodeMap.t
  val empty : t
  val add : Node.t -> AbsMem.t -> t -> t
  val init : Node.t list -> t
  val find : Node.t -> t -> AbsMem.t
  val print : t -> unit
end

module Table : Table = struct
  type t = AbsMem.t NodeMap.t
  let empty = NodeMap.empty
  let add = NodeMap.add
  let init ns = List.fold_right (fun n -> add n AbsMem.empty) ns empty
  let find : Node.t -> t -> AbsMem.t
  =fun n t -> try NodeMap.find n t with _ -> AbsMem.empty
  let print t = NodeMap.iter (fun n m ->
    prerr_endline (string_of_int (Node.get_nodeid n));
    AbsMem.print m;
    prerr_endline "") t
end
```

분석 함수 `analyze`를 구현하시오.

```
analyze : Cfg.t -> Table.t
```

실행 예 아래 프로그램

```
x := 0;
y := 0;
while (x <= 9) {
    x := x + 1;
    y := y + 1;
}
```

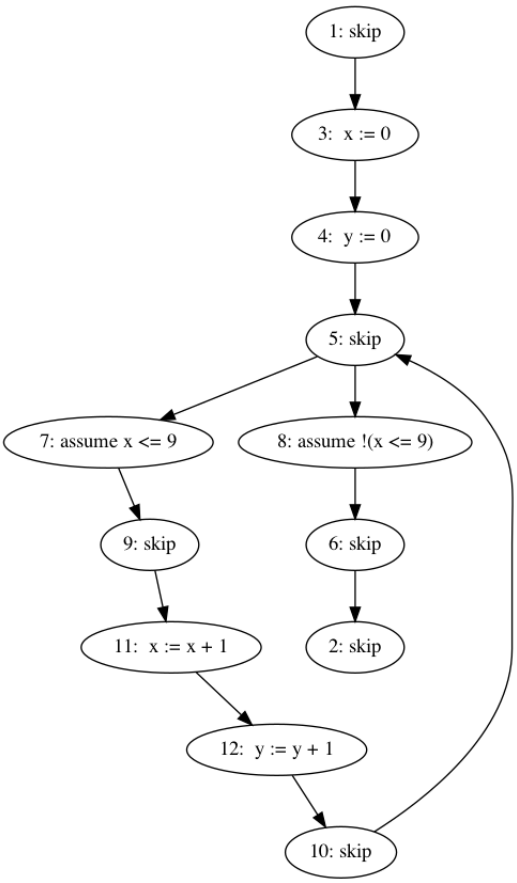
은 아래와 같이 OCaml 데이터 타입으로 표현할 수 있고

```
let pgm =
  Seq [
    Assign ("x", Const 0);
    Assign ("y", Const 0);
    While (Le (Var "x", Const 9),
      Seq [
        Assign ("x", Plus (Var "x", Const 1));
        Assign ("y", Plus (Var "y", Const 1));
      ]);
  ]
```

다음과 같이 분석하고 결과를 확인 할 수 있다.

```
let cfg = cmd2cfg pgm
let table = analyze cfg
let _ = Table.print table
```

생성된 그래프 `cfg`는 다음과 같을 것이고 (`cmd2cfg` 구현에 따라 생김새는 다를 수 있다),



분석 결과 `table`은 아래와 같이 출력되어야 한다.

```
1

2
x |-> [10, 10]
y |-> [0, +oo]

3
x |-> [0, 0]

4
x |-> [0, 0]
y |-> [0, 0]

5
x |-> [0, 10]
y |-> [0, +oo]

6
x |-> [10, 10]
y |-> [0, +oo]

7
x |-> [0, 9]
y |-> [0, +oo]
```

```
8
x |-> [10, 10]
y |-> [0, +oo]

9
x |-> [0, 9]
y |-> [0, +oo]

10
x |-> [1, 10]
y |-> [1, +oo]

11
x |-> [1, 10]
y |-> [0, +oo]

12
x |-> [1, 10]
y |-> [1, +oo]
```

제출 요령

- 아래 주소에 있는 템플릿 파일 `interval.ml`을 사용한다. (다음 페이지의 '부록'에 있는 코드와 동일하다)

  `https://github.com/hakjoooh/AAA616-2022Fall/blob/main/interval.ml`

- `interval.ml` 파일 하나에 모두 구현하도록 하고, 외부 라이브러리 의존성이 없어야 한다. 즉, 다음과 같은 커맨드로 문제 없이 실행되어야 한다.

  `ocaml interval.ml`

- 완성된 `interval.ml` 파일 하나만 제출한다.

1 부록

```
type aexp =
  | Const of int
  | Var of string
  | Plus of aexp * aexp
  | Mult of aexp * aexp
  | Sub of aexp * aexp

type bexp =
  | True
  | False
  | Equal of aexp * aexp
  | Le of aexp * aexp
  | Not of bexp
  | And of bexp * bexp

type cmd =
  | Assign of string * aexp
  | Seq of cmd list
  | If of bexp * cmd * cmd
  | While of bexp * cmd

let rec string_of_aexp a =
  match a with
  | Const n -> string_of_int n
  | Var x -> x
  | Plus (a1, a2) -> string_of_aexp a1 ^ " + " ^ string_of_aexp a2
  | Mult (a1, a2) -> string_of_aexp a1 ^ " * " ^ string_of_aexp a2
  | Sub (a1, a2) -> string_of_aexp a1 ^ " - " ^ string_of_aexp a2

and string_of_bexp b =
  match b with
  | True -> "true"
  | False -> "false"
  | Equal (a1, a2) -> string_of_aexp a1 ^ " == " ^ string_of_aexp a2
  | Le (a1, a2) -> string_of_aexp a1 ^ " <= " ^ string_of_aexp a2
  | Not b -> "!(" ^ string_of_bexp b ^ ")"
  | And (b1, b2) -> string_of_bexp b1 ^ " && " ^ string_of_bexp b2

module type Node = sig
  type instr =
  | I_assign of string * aexp
  | I_assume of bexp
  | I_skip
  type t
  val create_assign : string -> aexp -> t
  val create_assume : bexp -> t
  val create_skip : unit -> t
  val get_nodeid : t -> int
  val get_instr : t -> instr
  val to_string : t -> string
  val compare : t -> t -> int
end
```

```
module Node : Node = struct
  type instr =
  | I_assign of string * aexp
  | I_assume of bexp
  | I_skip
  type t = int * instr
  let new_id : unit -> int =
    let id =  ref 0 in
      fun _ -> (id := !id + 1; !id)
  let create_assign x a = (new_id(), I_assign (x, a))
  let create_assume b = (new_id(), I_assume b)
  let create_skip () = (new_id(), I_skip)
  let get_nodeid (id, _) = id
  let get_instr (_, instr) = instr
  let compare = Stdlib.compare
  let to_string n =
    match n with
    | (id, I_assign (x, a)) ->
      string_of_int id ^ ": " ^ " " ^ x ^ " := " ^ string_of_aexp a
    | (id, I_assume b) ->
      string_of_int id ^ ": " ^ "assume"  ^ " " ^ string_of_bexp b
    | (id, I_skip) ->
      string_of_int id ^ ": " ^ "skip"
end

module NodeSet = Set.Make(Node)
module NodeMap = Map.Make(Node)

module type Cfg = sig
  type t
  val empty : t
  val nodesof : t -> Node.t list
  val succs : Node.t -> t -> NodeSet.t
  val preds : Node.t -> t -> NodeSet.t
  val add_node : Node.t -> t -> t
  val add_nodes : Node.t list -> t -> t
  val add_edge : Node.t -> Node.t -> t -> t
  val print : t -> unit
  val dot : t -> unit
end

module Cfg : Cfg = struct
  type t = {
    nodes : NodeSet.t;
    succs : NodeSet.t NodeMap.t;
    preds : NodeSet.t NodeMap.t }
  let empty = {
    nodes = NodeSet.empty;
    succs = NodeMap.empty;
    preds = NodeMap.empty }

  let nodesof : t -> Node.t list
  =fun t -> NodeSet.elements t.nodes
```

```
  let succs : Node.t -> t -> NodeSet.t
  =fun n g -> try NodeMap.find n g.succs with _ -> NodeSet.empty

  let preds : Node.t -> t -> NodeSet.t
  =fun n g -> try NodeMap.find n g.preds with _ -> NodeSet.empty

  let add_node : Node.t -> t -> t
  =fun n g -> { g with nodes = NodeSet.add n g.nodes }

  let add_nodes : Node.t list -> t -> t
  =fun ns g -> g |> (List.fold_right add_node ns)

  let (|>) x f = f x
  let add_edge : Node.t -> Node.t -> t -> t
  =fun n1 n2 g ->
    g
    |> add_nodes [n1;n2]
    |> (fun g -> { g with
          succs = NodeMap.add n1 (NodeSet.add n2 (succs n1 g)) g.succs })
    |> (fun g -> { g with
          preds = NodeMap.add n2 (NodeSet.add n1 (preds n2 g)) g.preds })

  let print g =
    print_endline "** Nodes **";
    NodeSet.iter (fun n ->
      print_endline (Node.to_string n)
    ) g.nodes;
    print_endline "";
    print_endline "** Edges **";
    NodeMap.iter (fun n succs ->
      NodeSet.iter (fun s ->
        print_endline (string_of_int (Node.get_nodeid n) ^ " -> " ^
          string_of_int (Node.get_nodeid s))
      ) succs
    ) g.succs

  let dot g =
    print_endline "digraph G {";
    NodeSet.iter (fun n ->
      print_string (string_of_int (Node.get_nodeid n) ^ " ");
      print_string ("[label=\"" ^ Node.to_string n ^ "\"]");
      print_endline ""
    ) g.nodes;
    NodeMap.iter (fun n succs ->
      NodeSet.iter (fun s ->
        print_endline (string_of_int (Node.get_nodeid n) ^ " -> " ^
          string_of_int (Node.get_nodeid s))
      ) succs
    ) g.succs;
    print_endline "}"
end
```

```
let cmd2cfg : cmd -> Cfg.t
=fun cmd -> Cfg.empty (* TODO *)

module type AbsBool = sig
  type t = Top |  Bot | True | False
  val not : t -> t
  val band : t -> t -> t
end

module AbsBool : AbsBool = struct
  type t = Top | Bot | True | False
  let not b = b (* TODO *)
  let band b1 b2 = b1 (* TODO *)
end

module type Interval = sig
  type t
  val bottom : t
  val to_string : t -> string
  val alpha : int -> t
  val alpha_to : int -> t
  val alpha_from : int -> t
  val order : t -> t -> bool
  val join : t -> t -> t
  val meet : t -> t -> t
  val widen : t -> t -> t
  val narrow : t -> t -> t
  val add : t -> t -> t
  val mul : t -> t -> t
  val sub : t -> t -> t
  val equal : t -> t -> AbsBool.t
  val le : t -> t -> AbsBool.t
  val ge : t -> t -> AbsBool.t
end

module Interval : Interval = struct
  type t = Bot (* TODO *)
  let bottom = Bot
  let to_string i = "" (* TODO *)
  let alpha n = Bot (* TODO *)
  let alpha_to n = Bot (* TODO *)
  let alpha_from n = Bot (* TODO *)
  let order a b = true (* TODO *)
  let join a b = a (* TODO *)
  let meet a b = a (* TODO *)
  let widen a b = a (* TODO *)
  let narrow a b = a (* TODO *)
  let add a b = a (* TODO *)
  let mul a b = a (* TODO *)
  let sub a b = a (* TODO *)
  let equal a b = AbsBool.Top (* TODO *)
  let le a b = AbsBool.Top (* TODO *)
  let ge a b = AbsBool.Top (* TODO *)
end
```

```
module VarMap = Map.Make(String)
module type AbsMem = sig
  type t = Interval.t VarMap.t
  val empty : t
  val add : string -> Interval.t -> t -> t
  val find : string -> t -> Interval.t
  val join : t -> t -> t
  val widen : t -> t -> t
  val narrow : t -> t -> t
  val order : t -> t -> bool
  val print : t -> unit
end

module AbsMem : AbsMem = struct
  type t = Interval.t VarMap.t
  let empty = VarMap.empty
  let add x v m = m (* TODO *)
  let find x m = Interval.bottom (* TODO *)
  let join m1 m2 = m1 (* TODO *)
  let widen m1 m2 = m1 (* TODO *)
  let narrow m1 m2 = m1 (* TODO *)
  let order m1 m2 = true (* TODO *)
  let print m = VarMap.iter (fun x v -> prerr_endline
    (x ^ " |-> " ^ Interval.to_string v)) m
end

module type Table = sig
  type t = AbsMem.t NodeMap.t
  val empty : t
  val add : Node.t -> AbsMem.t -> t -> t
  val init : Node.t list -> t
  val find : Node.t -> t -> AbsMem.t
  val print : t -> unit
end

module Table : Table = struct
  type t = AbsMem.t NodeMap.t
  let empty = NodeMap.empty
  let add = NodeMap.add
  let init ns = List.fold_right (fun n -> add n AbsMem.empty) ns empty
  let find : Node.t -> t -> AbsMem.t
  =fun n t -> try NodeMap.find n t with _ -> AbsMem.empty
  let print t = NodeMap.iter (fun n m ->
    prerr_endline (string_of_int (Node.get_nodeid n));
    AbsMem.print m;
    prerr_endline "") t
end

let analyze : Cfg.t -> Table.t
=fun g -> Table.empty (* TODO *)

let pgm =
  Seq [
```

```
    Assign ("x", Const 0);
    Assign ("y", Const 0);
    While (Le (Var "x", Const 9),
      Seq [
        Assign ("x", Plus (Var "x", Const 1));
        Assign ("y", Plus (Var "y", Const 1));
      ]);
  ]

let cfg = cmd2cfg pgm
let table = analyze cfg
let _ = Table.print table
```