Homework 3
CSE6049 Program Analysis, Spring 2021
Woosuk Lee

due: 5/10(Mon), email-to-TA (bbumbuul@yahoo.com)

- The goal of this assignment is to design and implement a language interpreter and a sound static analyzer for the simple imperative language covered in the lecture.
- Skeleton code is provided and accessible through the course website. Before you start, see `README.md` to understand how to proceed.
- You have to heavily use OCaml modules. You may refer to an introduction to the concept from slides `http://psl.hanyang.ac.kr/~wslee/courses/cse6049/ocaml_module.pdf`.
- Please send a ZIP file titled "HW3_[Your Student ID].zip" to TA via email, and the zipped file should contain
  - OCaml source files `interpreter.ml` and `domains.ml` for Exercises 1,2,3, and 5
  - A PDF document file for Exercises 4

Background. Consider the `miniC` language used in the lectures. The language features arithmetic operations, loops, and conditionals.

The syntax is depicted in Figure 1. The `input` command reads an integer from external input. You may assume the value from external input ranges between -5 and 5.

The language is represented as the following data type in OCaml.

```
type var = string
type program = cmd
```

n	$\in$	$\mathbb{V}$	scalar values
$\mathtt{x}$	$\in$	$\mathbb{X}$	program variables
$\odot$	::=	$+ \mid - \mid *$	binary operators
$\olessthan$	::=	$< \mid \leq \mid == \mid > \mid \neq$	comparison operators
E	::=		scalar expressions
	$\mid$	n	scalar constant
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$E \odot E$	binary operation
B	::=		boolean expressions
	$\mid$	$\mathtt{x} \olessthan n$	comparison of a variable with a constant
	$\mid$	$\neg B$	negated condition
C	::=		commands
	$\mid$	**`skip`**	command that ”does nothing”
	$\mid$	C; C	sequence of commands
	$\mid$	$\mathtt{x} := E$	assignment command
	$\mid$	**`input`**(x)	command reading of a value
	$\mid$	**`if`**(B){C}**`else`**{C}	conditional command
	$\mid$	**`while`**(B){C}	loop command
	$\mid$	**`print`**(E)	print command

Figure 1: Grammar of the `miniC` language

```
and cmd =
  | SKIP
  | IF of cond * cmd * cmd
  | WHILE of cond * cmd
  | ASSIGN of var * exp
  | READ of var (* input(x) *)
  | SEQ of cmd * cmd
  | PRINT of exp

and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp

and cond =
  | TRUE
  | FALSE
  | LE of var * int
  | EQ of var * int
  | GT of var * int
```

```
  | NEQ of var * int
  | NOT of cond
```

A program generates an output memory state for a given input memory state. The set of memory states $\mathbb{M}$ is defined by:

$$\mathbb{M} = \mathbb{X} \to \mathbb{V}_\perp$$

where $\mathbb{V}_\perp$ is the "lifted" (also called flat) integer domain (i.e., $\mathbb{V}_\perp = \mathbb{V} \cup \{\perp\}$) which is a CPO.

Throughout this assignment, the following module signature will be used to define the interface for CPOs.

```
module type DOMAIN =
  sig
    type elt (* the type of abstract domain elements *)
    val bot: elt
    val join: elt -> elt -> elt (* least upper bound *)
    val leq: elt -> elt -> bool (* less than or equal to *)
    val string_of_elt: elt -> string
    val add: elt -> elt -> elt (* addition between elements *)
    val sub: elt -> elt -> elt (* subtraction between elements *)
    val mul: elt -> elt -> elt (* multiplication between elements *)
  end
```

The lifted integer domain that follows the `DOMAIN` interface is already defined in the skeleton code by the following OCaml module.

```
module IntCPO : DOMAIN =
struct
  type elt = Bot | Int of int
  ...
end
```

The following *functor* MakeMemCPO returns a module that can be used for function domain $\mathbb{X} \to D$ for a given CPO D.

```
module VarMap = Map.Make (struct
    type t = var
    let compare = String.compare
  end)

module MakeMemCPO (D : DOMAIN) =
struct
  (* see https://ocaml.org/api/Map.Make.html *)
  include VarMap

  type t = D.elt VarMap.t (* type : string -> D.elt *)
  ...
```

The data structure for memories is also defined in the skeleton code by

```
module Mem = MakeMemCPO(IntCPO).
```

Exercise 1 Implement a language interpreter by writing a function

```
interpret :  program -> Mem.t -> Mem.t
```

that takes a program and an input memory state (initially empty memory) and returns an output memory state. The function should be defined in file `interpreter.ml`.

Exercise 2 Implement a collecting semantics-based interpreter that collects *all possible values* that may be computed during program execution *for each variable*. In other words, it should compute a collecting state in $\mathbb{X} \to 2^{\mathbb{V}}$.

The power set of values (i.e., integers) can be defined by any module that follows the following interface.

```
module type INTSET_DOMAIN =
  sig
    include DOMAIN
    val filter: (int -> bool) -> elt -> elt
    val make: int list -> elt
  end
```

where `filter f s` returns the set of all elements in `s` that satisfy predicate `f` and `make` generates a set of integers from a list of integers. The `filter` function will be useful for handling conditional commands.

Define a module

```
module IntsetCPO : INTSET_DOMAIN
```

in file `domains.ml`. Then, the function domain for states in $\mathbb{X} \to 2^{\mathbb{V}}$ can be defined by

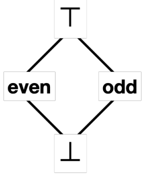
```
module IntsetMem = MakeMemCPO(IntsetCPO)
```

Then, write a function

```
interpret_collect :  program -> IntsetMem.t -> IntsetMem.t.
```

The function should be defined in file `interpreter.ml`.

Exercise 3 Design an abstract interpreter (i.e., static analyzer) that determines the parity (i.e., evenness or oddness) of a value of each program variable after the program execution. The parity abstract domain $D_P = \{\bot, \texttt{even}, \texttt{odd}, \top\}$ is characterized by the following hasse diagram and galois connection.



$$2^{\mathbb{Z}} \underset{\alpha_P}{\overset{\gamma_P}{\leftrightarrows}} D_P$$

where

$$\alpha_P(Z) = \begin{cases} \bot & (Z = \emptyset) \\ \texttt{even} & (Z \subseteq \mathbb{Z}_{even}) \\ \texttt{odd} & (Z \subseteq \mathbb{Z}_{odd}) \\ \top & (\text{otherwise}) \end{cases}$$

$$\gamma_P(P) = \begin{cases} \emptyset & (P = \bot) \\ \mathbb{Z}_{even} & (P = \texttt{even}) \\ \mathbb{Z}_{odd} & (P = \texttt{odd}) \\ \mathbb{Z} & (\text{otherwise}) \end{cases}$$

and $\mathbb{Z}_{even}$ (resp. $\mathbb{Z}_{odd}$) is the set of even (resp. odd) integers.

Define the abstract semantics for the parity analysis and prove the soundness of the static analysis.

Exercise 4 Implement your own abstract interpreter for the parity analysis. The parity domain can be defined by any module that follows the following interface.

```
module type PARITY_DOMAIN =
  sig
    include DOMAIN
    val top: elt
    val meet: elt -> elt -> elt (* greatest lower bound *)
    val make: int -> elt
  end
```

where the `meet` function is for computing the greatest lower bound ($\sqcap$) of two elements. For example, $\top \sqcap \texttt{even} = \texttt{even}$ and $\texttt{even} \sqcap \texttt{odd} = \bot$. The `meet` function will be useful for handling conditionals.

Define a module

```
module ParityCPO : PARITY_DOMAIN
```

in file `domains.ml`. Then, the function domain for states in $\mathbb{X} \to D_P$ can be defined by

```
module ParityMem = MakeMemCPO(ParityCPO)
```

Then, write a function

```
interpret_parity :  program -> ParityMem.t -> ParityMem.t
```

The function should be defined in file `interpreter.ml`.

Exercise 5 Implement an abstract interpreter based on the intervals abstraction.

The interval abstract domain D_I can be defined by any module that follows the following interface.

```
module type INTERVAL_DOMAIN =
  sig
    include DOMAIN
    type bound = Z of int | Pinfty | Ninfty
    val top: elt
    val meet: elt -> elt -> elt (* greatest lower bound *)
    val make: bound -> bound -> elt
    val widen: elt -> elt -> elt
    val narrow: elt -> elt -> elt
  end
```

where the `make` function is for constructing an abstract element (i.e., interval). For example, one can construct an interval $[1, +\infty]$ by `make (Z 1) Pinfty` and $[-\infty, 0]$ by `make Ninfty (Z 0)`. The `widen` and `narrow` functions are for widening and narrowing respectively.

Define a module

```
module IntervalCPO : INTERVAL_DOMAIN
```

in file `domains.ml`. Then, the function domain for states in $\mathbb{X} \to D_I$ can be defined by

```
module IntervalMem = MakeMemCPO(IntervalCPO)
```

Then, write a function

```
interpret_interval :  program -> IntervalMem.t -> IntervalMem.t
```

The function should be defined in file `interpreter.ml`.