EC3204: Programming Languages and Compilers (Fall 2024)
Homework 2: Interval Analyzer

Due: 12/11, 23:59 (submit on GIST LMS)

Instructor: Sunbeom So

Important Notes

- **Evaluation criteria**
  The correctness of your implementation will be evaluated using testcases:

$$\frac{\#\text{Passed}}{\#\text{Total}} \times 100$$

  - "Total": testcases prepared by the instructor (not disclosed before the evaluation).
  - "Passed": testcases where expected outputs match your outputs.

- **Compilable**
  Make sure that your submission is successfully compiled. If your code cannot be compiled, you will get 0 points for the corresponding HW.

- **Academic Integrity**
  Violating academic integrity will result in an F, even after the end of the semester.

  Click here to check the rules related to academic integrity.

  Be aware that proving your academic integrity is entirely your responsibility.

- **No Changes on Templates, File Names, and File Extensions**
  Your job is to complete `(* TODO *)` parts in provided code templates. You should not modify the other templates. Do not change the file names. The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`, `.zip`, `.tar`).

- **Regarding Using LLM-based Tools**
  You can use LLM-based tools to complete your HW, as long as (1) you use LLMs by yourself, (2) you can reproduce all the steps assisted by LLMs, and (3) you can clearly explain all the details of your implementation (including LLM-generated parts).

  Even though you claim that you have completed HW with the help of LLMs, if your submission is highly similar to other students' code and you cannot reproduce certain steps by any reason (including the nondeterminism of LLMs), I will consider that you cheated on your assignments.

1 Goal

Your goal is to implement an interval domain-based semantic analyzer to prove assertion safety.

2 Structure of the Project

You can find the following files in the directory `hw2/code`.

- `main.ml`: contains the driver code.
- `lang.ml`: contains the definition of our target language.

```
1 type pgm = fid * param list * cmd
2 and fid = string
3 and param = typ * var
4 and var = string
5
6 and typ = ...
```

As implemented in the code, in our language, a program is defined by the 3-tuple:

$$(\mathit{fid}, (\mathit{typ}\ x)^*, c)$$

where *fid* is a function identifier (name), $(\mathit{typ}\ x)^*$ is a sequence of input parameters (variables annotated with their types), and c is a command. In our language, a variable can be either integer-typed or integer array-typed, i.e., $\mathit{typ} \rightarrow \texttt{int} \mid \texttt{int}[n]$ where n denotes a positive integer constant. Note that our language supports statically-sized arrays only. The command c is defined below, together with l-values (lv), arithmetic expressions (a), and boolean expressions (b).

$$\begin{array}{rcl} lv & \rightarrow & x \mid x[a] \\ a & \rightarrow & n \mid \texttt{len}(x) \mid lv \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\ b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 \prec a_2 \mid \neg b \mid b_1\ \&\&\ b_2 \mid b_1 \mid\mid b_2 \\ c & \rightarrow & \mathit{typ}\ x \mid lv = a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ e\ c_1\ c_2 \mid \texttt{while}\ b\ c \mid \texttt{assert}(b) \end{array}$$

In the language syntax, $\prec$ denotes the standard binary comparison operators ($==, !=, <, \leq, >, \geq$). The intended semantics must be clear through the following explanations.

- $\texttt{len}(x)$: returns the size of an array-typed variable x. We assume ill-typed expressions like $\texttt{len}(y)$ where y is an integer-typed will not appear.
- $\mathit{typ}\ x$ (in c): represents a variable declaration, where a variable gets assigned a default value. The default values for `int` and $\texttt{int}[n]$ are 0 and an array containing n zeros, respectively. For example, given a declaration command $\mathit{int}\ x$, 0 should be assigned to x. As another example, given a declaration command $\mathit{int}[10]\ x$, 0 should be assigned to $x[0], \cdots, x[9]$.
- $lv = a$: is an assignment. For simplicity, we assume an assignment of the form $x = y$, where both x and y are array-typed variables, will not appear. That is, we do not consider aliases or (shallow/deep) copies between array-typed variables. The other cases are standard.

  - `assert`(b): is an assertion that contains properties to prove and does not affect the program semantics. That is, `assert` just expresses properties expected to hold at specific locations, and nothing happens at runtime even if the conditions in `assert` do not hold.

- `lexer.mll`: contains the lexical specification in `ocamllllex` for automatically generating a lexer. The provided implementation is complete, so you do not need to modify this file.

- `parser.mly`: contains the syntactic specification in `ocamlyacc` for automatically generating a parser. **Your job is to complete the translation rules (concrete syntax to abstract syntax) for `aexp` (a) and `bexp` (b)**:

```
...
aexp: (* TODO *)
  | MINUS NUM {L.Int ($2* (-1))}
  | NUM {L.Int $1}

bexp: (* TODO *)
  | TRUE {L.True}
...
```

- `absDom.ml`: implements the abstract domains and abstract operators.

  Your job is to replace "raise NotImplemented" with your implementations. You do not need to introduce new helper functions.

  - `module Itv`: implements an interval domain.
  - `module AbsBool`: implements an abstract domain for boolean values.
  - `module AbsMem`: aims to implement an abstract memory state $\widehat{m} \in \widehat{\mathbb{M}}$, a mapping from each variable to an interval.

- `itvAnalysis.ml`: should contain the abstract transformers (abstract semantic functions).

  Your job is to replace "raise NotImplemented" with your implementation. You may need to introduce several new functions.

  - *Hint:* One simple approach to abstract an integer array as an interval is to represent all elements as a single interval value. For example, given an array a containing three elements $\{1, 5, 10\}$, the interval values of $a[0]$, $a[1]$, and $a[2]$ are all $[1, 10]$.
  - To implement the abstract semantics for while-loops (`while` b c), you should consider widening and narrowing to ensure termination. Moreover, to achieve high precision as possible, you may consider applying widening operators only when you do not reach a fixed point after x iterations, where x is a predetermined threshold (e.g., $x = 1000$).

3 How to Build

(1) Install the dependencies using the following commands.

```
$ sudo apt-get update
$ sudo apt-get install -y opam ocamlbuild ocaml-findlib
$ opam init
$ eval $(opam env)
$ opam update
$ opam install -y batteries
```

(2) Activate the build script by running the below command in the directory `hw2/code`.

```
$ chmod +x build
```

(3) After completing the three files (`parser.mly`, `absDom.ml`, `itvAnalysis.ml`), run the following command to compile the project.

```
$ ./build
```

Then, the executable `main.native` will be generated. You can run it as follows.

```
$ ./main.native -input TESTFILE
```

where `TESTFILE` is a program file written in our target language. Example program files can be found in the directory `hw2/code/test`.

4 Running Example

Consider the test program in `hw2/code/test/loop1`:

```
1  loop1 (int n) {
2    int i;
3    int j;
4
5    i = 0;
6    j = 0;
7
8    while (i!=n) {
9      i = i+1;
10     j = j+1;
11     assert(1<=i); (* safe, provable *)
12     assert(1<=j); (* safe, provable *)
13   }
14
15   assert (i==n); (* safe, but cannot be proven using our interval analysis *)
16   assert (i==j); (* safe, but cannot be proven using our interval analysis *)
17 }
```

If you run the command

```
$ ./main.native -input test/loop1
```

you should obtain the following result:

```
...
========== Result ==========
- # Proven : 2
- Time : ...
```

where 2 indicates the number of proven assertions (lines 11, 12).