EC4219: Software Engineering (Spring 2024)
Homework 2: Houdini Algorithm for Invariant Inference

100 points
Due: 6/12, 23:59 (submit via GIST LMS)

Instructor: Sunbeom So

Important Notes

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

  $$\frac{\#\text{Passed}}{\#\text{Total}} \times \text{point per problem}$$

  - "Total" indicates a set of testcases prepared by the instructor (undisclosed before the evaluation).
  - "Passed" indicates testcases whose expected outputs match with the outputs produced by your implementations.

- **Executable**
  Before you submit your code, please make sure that your code can be successfully compiled. That is, the command `./build` should not report any errors. Otherwise, you will get 0 points for that HW.

- **No Plagiarism and No Discussion**
  Cheating (i.e., copying assignments by any means) will get you an F. See the slides for Lecture 0. Code-clone checking will be conducted irregularly. Furthermore, discussions at all levels are strictly disallowed.

- **No Changes on Template/File Name/File Extension Changes**
  Your job is to complete `(* TODO *)` parts in provided templates; you should not modify the other existing code templates. Do not change the file names. The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`, `.zip`, `.tar`).

- **No Posting on the Web**
  You should not post your implementations on public websites (e.g., public GitHub repositories). Violating this rule gets you an F, even after the end of the semester.

1 Goal

In Hw1, we assumed loop invariants are properly annotated in our programs to verify. Your job in Hw2 is to implement the Houdini-style algorithm for automatically generating such invariants.

2 Structure of the Project

You can find the following files in the `hw2` directory. **Your job is to complete and submit the three files**:

`verify2.ml`, `guess.ml`, `verify.ml`

- `verify2.ml`: This module aims to implement the verification procedure with Houdini-style invariant inference below:

Algorithm 1 Verification with Automatic Invariant Inference

```
Input: A program $P$ containing holes to verify
Output: An annotated program $P'$ and its verification result
$\mu \leftarrow$ enumerate speculated invariants                ▷ guess.ml, $\mu : hid \rightarrow \mathcal{P}(\mathsf{FOL})$
while true do                                                  ▷ the function synthesize in verify2_template.ml
    refuted $\leftarrow \mathsf{Verify}(P, A)$                  ▷ the function verify_candidates in verify2_template.ml
    if refuted $= \emptyset$ then
        break
    for each $(hid, \phi) \in$ refuted do                      ▷ the function refine in verify2_template.ml
        $\mu[hid] \leftarrow \mu[hid] \setminus \{\phi\}$
$P' \leftarrow$ fill in the holes of $P$ using $\mu$            ▷ the function fill in verify2_template.ml
return ($P'$, verification result of $P'$)
```

  - Tip 1: In `verify2_template.ml`, `pregen`$'$ is a function that generates verification conditions by treating each atomic candidate invariant as an independent assertion. For example, for each $i \in \mu(hid)$, you may generate following verification conditions:

$$\{\bigwedge_{i' \in \mu(hid)} i' \land e\}\ c\ \{i\}$$

  - Tip 2: You may partly use `pregen` from `verify.ml` when implementing `pregen`$'$.

- `guess.ml`: This module aims to generate candidate invariants mined from code text. More specifically, its goal is to construct a mapping from hole identifiers (`hid` in the file) to a set of atomic formulas. Recall that Houdini algorithm aims to find conjunctive invariants. Thus, a candidate invariant can be represented as a set of atomic formulas, where all elements are connected with $\land$.

  In Hw2, for each hole identifier (*hid*), the search space for candidate invariants is defined as follows:

$$\begin{array}{rl}
& \{\forall j. x \leq j \land j < y \rightarrow a[j] \neq z \mid x, y, z \in \mathbb{X}_i, a \in \mathbb{X}_a\} \\
\cup & \{x \prec y \mid x, y \in \mathbb{X}_i, \prec \in \{\leq, =\}\} \\
\cup & \{c \leq x \mid x \in \mathbb{X}_i, c \in \{0, 1\}\} \\
\cup & \{x < \mathsf{len}(a) \mid x \in \mathbb{X}_i, a \in \mathbb{X}_a\} \\
\cup & \{\mathsf{sorted}(a, x, \mathsf{len}(a) - 1) \mid x \in \mathbb{X}_i, a \in \mathbb{X}_a\} \\
\cup & \{\mathsf{partitioned}(a, 0, x, x + 1, \mathsf{len}(a) - 1) \mid x \in \mathbb{X}_i, a \in \mathbb{X}_a\} \\
\cup & \{\mathsf{partitioned}(a, 0, x - 1, x, x) \mid x \in \mathbb{X}_i, a \in \mathbb{X}_a\}
\end{array}$$

where $\mathbb{X}_i$ and $\mathbb{X}_a$ are integer-typed and integer-array-typed variables collected from a program text.

- `verify.ml`: This is a file that you implemented for Hw1. **You can submit a revised version that is different from the one you submitted for Hw1.**
- `formula.ml`: contains the definition of FOL formulas. The difference compared to Hw1 is that, a "hole" is added into the definition of the FOL formula, in order to allow loop invariants to be unspecified.
- The rest files are the same as Hw1 (except for subsequent changes due to the introduction of the hole).

3 How to Build

Once you complete the three files (`verify2.ml`, `guess.ml`, `verify.ml`), you can build the project as follows.

```
$ ./build
```

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

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

4 Running Example

If you run the command

```
$ ./main.native -input test/loop2
```

you should obtain the following result:

```
...
[INFO] iter: ...
[INFO] inductive invariants found!
...
=== Verificaiton Result ===
true, 2
...
```

where `true` indicates that the function is partially correct, and `2` is the number of proven assertions.