Invariant Generation for Complexity Analysis of Python Programs

Adam Blank (adamblan@cs) and Michael Arntzenius (daekharel@gmail.com)

Project Description

The high level goal of this project is to implement several functions. Given a Python program P,

- `ub_time_complexity(P)` returns a symbolic upper bound on the *run time* of P.
- `ub_output_complexity(P)` returns a symbolic upper bound on the *size* of the output of P.
- `lb_time_complexity(P)` returns a symbolic lower bound on the *run time* of P.
- `lb_output_complexity(P)` returns a symbolic lower bound on the *size* of the output of P.

For example, consider the following code:

```
def f(n):
   out = 1
   for i in range(n):
      out = out * n
   return out
```

We can see that `ub_time_complexity(f)` should return $n \in \mathcal{O}(n)$, because the loop runs n iterations. However, `f` *returns* `out` which is 1 initially and is multiplied by n, n times. So, `ub_output_complexity(f)` $= n^n \in \mathcal{O}(n^n)$.

0.1 Restrictions

In order to keep the scope of this assignment reasonable, we will restrict input programs in the following ways:

- All functions will be of type `int`, `int`, ..., `int` $\to$ `int`.
- We will only simplify recursive programs of several particular forms (discussed below).
- We will ignore certain dynamic features of Python.
- If termination/invariant detection is too difficult to infer, we will output `?`, meaning "we do not know".

The difficulty arises in handling loops and recursion.

0.2 Loops

To effectively bound the runtime of a loop, we must bound the number of iterations. We propose doing this by using a variety of AI techniques and ideas from CITE PAPER HERE to generate invariants on the values of variables after multiple iterations. In particular, we aim to be able to handle *binary search*-like functions, and *early terminations* of loops. We intend to use search techniques on the AST of the loop, planning to prove the invariants we find correct, and possibly machine learning to classify programs as terminating or not.

0.3 Recursion

To effectively bound the runtime of a recursive function, we need to be able to bound the number of recursive calls. We only consider the class of programs where one of the following is true, for $f(\mathbf{x})$. We say $f(a_1, a_2, \ldots, a_n) \rightsquigarrow f(b_1, b_2, \ldots, b_n)$, when $f(\mathbf{b})$ is called as part of executing $f(\mathbf{a})$:

- $\exists i, m.\ \forall \mathbf{y}.\ f(\mathbf{x}) \rightsquigarrow f(\mathbf{y}) \implies m \leq y_i < x_i$
- $\exists i, M.\ \forall \mathbf{y}.\ f(\mathbf{x}) \rightsquigarrow f(\mathbf{y}) \implies x_i < y_i \leq M$

As with loops, we will use planning and search to determine and prove whether programs are of this form or not.

References