Verification and Abstract Interpretation Project
Typestate Analysis

Haggai Eran 043425321 Shaked Flur 036333094

June 29, 2010

Short Description

We implmented an intraprocedural typestate analysis, with support for different state transitions in branches. We use the Spark points-to analysis available in Soot.

Lattice

For each allocation site we have a lattice of the form $ASInfo : \langle unique, states \rangle$.

- The *unique* bit determines if there is only one object of this site, as seen in class.
- The *states* set contains the possible states the objects of this site may be in.

The analysis' lattice contains an $ASInfo$ tuple for each allocation site.

Branches

We wanted to support different state transisions in branches, such as the case below:

```
Stack s = new Stack();
boolean b = s.empty();
if (!b)
    s.pop();
```

Basic typestate analysis will find an error on the `s.pop()` call, since `s` is in state MAYBE_EMPTY. We overcome this problem by introducing branch dependant transisions: when the '`if`' body is executed, it is always after the `empty()` method returned false, so the state of `s` inside the '`if`' is changed to NOT_EMPTY.

In order to add such support, we want to relate between the condition `!b` in the `if` statement, and `s` and the method call `empty()`. To do that we added to the lattice a variation of reaching definition analysis. This part of the lattice maps for each boolean local variable, its defining statement.

Automata

Here is an (incomplete) example of our automata syntax. The first two lines define the special states initial and error. Each state is defined in a `state` block, containing a list of transitions stated as *method signature* `->` *next state*.

```
automaton Stack {
    initial = MAYBE_EMPTY;
    error = ERROR;
    state MAYBE_EMPTY {
        Object push(Object) -> NON_EMPTY;
        Object pop() -> ERROR;
        boolean empty() returns false -> NON_EMPTY;
```

```
        }
        state NON_EMPTY {
            void clear() -> MAYBE_EMPTY;
            Object pop() -> MAYBE_EMPTY;
        }
        state ERROR {}
    }
```

In addition to the ordinary method transitions, the definition contains a transition depending on a branch, for the `empty()` method. If the current state is MAYBE_EMPTY, and a branch is taken indicating the return value of an `empty()` call was false, the next state will be NON_EMPTY.

This technique is useful when there are states such that one is semantically contained in another, as is the case here where MAYBE_EMPTY contains the NON_EMPTY state.

Methods that do not appear in a state transitions list are assumed to keep the current state.

Lattice Transformers

For method invocation, if there is only a single object of the given allocation site (determined by the *unique* bit), and only one allocation site, we perform a strong update, and change the *states* part to next states. If *unique* bit is false, or there are more than one allocation site, a weak update is performed, and the resulting *states* contains both the previous states and the next ones.

In allocation sites, the site's ASInfo is set to the automaton's initial state. The unique bit is set or unset as needed.

For assignment statements where a method result is assigned to a local boolean, we set the statement as the local's defining statement. If later a method is invoked on the base object's allocation site then the defining statement is set to $\top$.

If statements are handled by checking if their condition is a simple boolean check on a local. If so, we find the defining statement of that local, and see if it is a method invocation. If it is, we set the output *states* set differently in the branch case and the fall case, according to the automaton. In other cases the if is treated as a nop.

In cases where a data member of our class is read or written to, we cannot guarentee that the state will not change, so to preserve soundness we set the ASInfo value of the allocation site associated with the object to $\top$. This is done also when our class is passed as an argument to a method call, and when it is stored in another object's field.

Lattice Join

For each allocation site, the join is done separately:

The *states* sets' union is taken, and the result of the join is *unique* if both are set.

For boolean locals, if a single boolean local has two different defining statements, one from each side of the join, the result is $\top$.