Statement Transformations

Alon Mishne

November 11, 2010

1 Definitions

State (S) A *History Collection*.

HistoryCollection (HC) A set of *Abstract Object-History* pairs.

Abstract Object (AO) Represents a single runtime object. Composed of a set of *Access Path*s the object is currently known to dwell at.

Access Path (AP) An $\langle anchor, field\,list \rangle$ tuple where anchor is local / static variable and the list of composed of fields.

- We will represent an access path by concatenating the anchor and fields together with dots, e.g. `x.y.z`.
- The form `x.y.*` is used to represent the group of all possible access paths with prefix `x.y`.
- We will allow concatenating two access paths by using dot as well.

History (H) A weighted automaton where each node represents a type state and each edge contains a method signature. We will display a history in the form of a set of (Node, Edge, Node) tuples.

- The $EmptyHistory$ symbol is used to describe an empty history set.

History Node (N) Represents a specific type state. Nodes are identified by an "H" followed by a serial number.

History Edge (E) A method signature and an associated weight (concatenated together via an asterisk in this document), or a question mark. A question mark represents one or more unknown methods that were invoked on the object.

thus, a state can be represented in the form:

$$S = \{(AO_1, H_1), (AO_2, H_2), ...\}$$

or

$$S = \{(\{x, x.y\}, \{(H0, < init > *3, H1), (H1, flush()*1, H2)\}), (\{z, w\}, \{(H0, f(int, String)*1, H1)\})\}$$

2 Function definitions

We shall define a few function for representing the transformations.

$kill(AO, AP)$ Returns a new abstract object in which each access path which is prefixed by AP is removed.

$$kill(AO, AP) = \{ap \in AO | ap \notin AP.*\}$$

$extend(H, < method\ signature >)$ TBD

$type(args)$ TBD

3 Transformations

History collection before the statement is always marked HC. `args` specifies the group of argument access paths sent to a method during invocation.

As our initial input as actually bytecode, we ignore any statement which is more complex than those appearing below, assuming at has already been decomposed - specifically, we assume no statement contains more than one assignment and more than one method invocation.

3.1 Assignment

All intermediate-language assignments will be of the form AP_1 `=` AP_2. In any case which isn't self assignment ($AP_1 = AP_2$), we need to remove all access paths starting with the left-hand side, AP_1, from all abstract objects.

3.1.1 Assigning null: AP `= null` (e.g. `x.y = null`)

$$\{(AO', H) | (AO, H) \in HC, AO' = AO \setminus AP.*\}$$

Explanation: this is the equation for simply removing any access path starting with `x.y` from every abstract object containing it. This will also appear in any other transformation for assignment.

3.1.2 Assigning another access path: AP_1 `=` AP_2 (e.g. `x.y = a.b`)

$$\{(AO', H) | (AO, H) \in HC, AO' = (AO \setminus AP_1.*) \cup \{AP_1.AP' | AP_2.AP' \in AO\}\}$$

Explanation: we have actually created an alias for every object containing an access path starting with, the union here adds these aliases to all affected object. We must preserve the suffixes of these access paths, however.

3.1.3 Assigning a new object: AP `= new T(args)` (e.g. `x.y = new Date(5,3,2010)`)

$$\{(AO', H) | (AO, H) \in HC, AO' = AO \setminus AP.*\} \cup \{(\{AP\}, extend(EmptyHistory, \texttt{<init>}(type(args)))\}$$

Explanation: we add a new abstract object with the access path assigned into. The new object's history starts with the constructor signature.

3.2 Method invocation

Method invocation actually concerns too parts we care about:

1. Since a method have been invoked on a receiver we need to extend its history by that new method. Also, we must be aware the receiver is not always contained in our state.
2. If we know which implementation of that method will be chosen and we have access to it, we should also extend our state with anything that has occurred in that method (intra-procedural analysis).

In any case, we treat the analysis as if all types are tracked for method invocations and all methods are "walked into". In practice, not only is it impossible to "walk into" many methods (because of polymorphism, missing code, etc.) but our implementation also does not record method invocations on untracked types and can filter out methods we are not interested in digging into — e.g. methods from the standard Java library.

In addition, our intra-procedural analysis implementation cannot handle recursive circles. When we reach an invocation of a method that we are currently in the process of analyzing, we filter that invocation out and do not walk into it.

When encountering a method call AP`.f(args)`, the following steps are performed in order:

1. If no abstract object contains AP, add $(AP, EmptyHistory)$ to the state.
2. For each abstract object containing AP, extend it with the signature `f`$(type(args))$.
3. Analyze the invoked method, but start the analysis not with an empty history collection, but with a history collection generated from the current history collection. After the analysis completes, the next state will be generated by converting back the final state reached when analyzing the invoked method. The conversions are details in the next sections.

3.2.1 Converting a state *to* a called method

The challenge is that while the actual histories of the abstract objects do not change, the access paths do. Any local access path will be removed, access paths that are available to method the caller and the called methods (e.g. globals) need to be preserved, and access paths of parameters in the called methods need to be added to the appropriate abstract objects.

We rely on the fact that every access path actually also contains its scope - i.e. the class (and method, for locals) containing it. That means that the first two problems - hiding some existing access and preserving others depending on the called methods - are actually trivially solved by doing nothing.

What remains is dealing with parameter, and that is actually identical to the behavior for general assignment, when the parameter access path is the left-hand side and the argument access path the right-hand side.

3.2.2 Converting a state *from* a called method

Unlike when converting the state *to* called methods, returning from a method does require we remove all access paths which are local to that method, because of cases such as:

```
f(); // analyze f()
f(); // analyze f() again
```

Since in that case we need to analyze f() twice, but the second call should not preserve histories to local f variables from the first call.

3.2.3 Dealing with return values

TBD

3.2.4 Static methods

Static methods are not recorded in the history, but are walked into.

4 Implementation notes

4.1 Histories

4.1.1 Memory layout

Histories are represented in memory as a graph with every node having a map of outgoing edges and a multimap of ingoing edges - that's because a given object state cannot be changed to two different states by calling the same method, while identical method calls can *lead* to the same state.

4.1.2 History operations

TBD

Most history operations such as merging two histories together, printing out histories, copying histories etc. are implemented by the use of the visitor pattern.