McSaF quick start

Prepared by Ismail Badawi – ismail@badawi.io

Introduction

This handout is a quick, largely standalone, overview of McSAF, the McLab toolkit's static analysis framework. It covers

- how to convert an analysis from Laurie's Six Step Method to Java code using McSAF (with examples)
- important changes that have been made to the framework since Jesse Doherty's Master's thesis

Implementing analyses

In class, you'll have seen how to precisely specify a dataflow analysis via Laurie's six steps:

1. State the problem precisely
2. Define the domain (e.g. for liveness analysis, sets of strings representing variable names)
3. Is this a forward or backward analysis?
4. Write down flow equations for different language constructs
5. What's the merge operation?
6. What's the initial dataflow fact?

We first briefly go over how a flow analysis is represented in McSAF, then spell out the mapping between the six steps and code.

In McSAF, a flow analysis is represented by a Java class which extends either `ForwardAnalysis` or `BackwardAnalysis` (two classes provided by the framework). The analysis class is generic; the generic type parameter represents the flow data domain. `ForwardAnalysis` and `BackwardAnalysis` declare three abstract methods that analyses must implement; `A merge(A in1, A in2)`, `A copy(A src)`, and `A newInitialFlow()`.

We're going to implement (a simplistic version of) reaching defs. Here, the six items are

1. A definition (for our purposes) is an assignment of a value to a variable. A definition d : `x = E` *reaches* a program point P if there is at least one control flow path from d to P along which `x` is not redefined.

2. The domain is sets of definitions (i.e. sets of assignment statements)

3. This is a forward analysis

4. The rule for an assignment d : `x = E` is:

   ```
   out(d) = (in(d) - kill(d)) + gen(d)
   ```

   where `kill(d)` is the set of previous definitions of `x`, and `gen(d)` is just `{d}`.

5. The merge operation is set union.

6. The initial dataflow fact is `in(entry) = {}`.

Here it is:

```
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

import analysis.ForwardAnalysis;
import ast.AssignStmt;
import ast.ASTNode;
import ast.Stmt;

public class ReachingDefinitions extends ForwardAnalysis<Set<AssignStmt>> {
    public ReachingDefinitions(ASTNode tree) {
        super(tree);
    }

    // The following three methods must be implemented to correctly extend ForwardAnalysis.
    @Override public Set<AssignStmt> newInitialFlow() { return new HashSet<>(); }

    @Override public Set<AssignStmt> copy(Set<AssignStmt> src) { return new HashSet<>(src); }

    @Override public Set<AssignStmt> merge(Set<AssignStmt> in1, Set<AssignStmt> in2) {
        Set<AssignStmt> out = new HashSet<>(in1);
        out.addAll(in2);
        return out;
    }

    @Override public void caseStmt(Stmt node) {
        inFlowSets.put(node, copy(currentInSet));
        currentInSet = copy(currentOutSet);
        outFlowSets.put(node, copy(currentOutSet));
    }

    @Override public void caseAssignStmt(AssignStmt node) {
        inFlowSets.put(node, copy(currentInSet));

        // Compute (in - kill) + gen
        // out = in
        currentOutSet = copy(currentInSet);
        // out = out - kill
        Set<String> namesToKill = node.getLValues();
        Set<AssignStmt> kill = new HashSet<>();
        for (AssignStmt def : currentOutSet) {
            Set<String> names = def.getLValues();
            names.retainAll(namesToKill);
            if (!names.isEmpty()) {
                kill.add(def);
            }
        }
        currentOutSet.removeAll(kill);
        // out = out + gen
        currentOutSet.add(node);

        outFlowSets.put(node, copy(currentOutSet));
    }
}
```

This is a simplistic implementation because (among other things):

- it only considers assignments; there are other constructs that introduces names, like global declarations, or function input parameters.
- it treats arrays, structs, and cell arrays naively. Traditionally, assigning to an element of an array (as in `a(1) = 2`) is considered a use of that array, not a definition. In MATLAB however, assigning to arrays makes them spring into existence if they didn't exist, so that assignment could be considered an implicit definition. A more sophisticated approach might try to determine whether the array definitely exists before the assignment, and in that case not consider it a definition.
- it doesn't correctly handle assignments with multiple names on the left hand side. A definition like `x = 4` will kill previous definitions like `[x, y] = f()`, which can lead to wrong results for `y`. One way to deal with this is to have multiple copies of a definition for each of the names it defines – we can do this by using `Map<String, Set<AssignStmt>>` as the flow data instead of a `Set<AssignStmt>`.

Some remarks code-wise:

- You would run this analysis with `ReachingDefinitions rd = new ReachingDefinitions(ast); rd.analyze();`, and access results with `rd.getOutFlowSets().get(Stmt)`.
- We use the `AssignStmt#getLValues()` method to get at the names of variables being assigned to. There could be multiple (e.g. `[x, y] = somefunction()`), and there could also be an array, cell array, or struct on the left hand side (e.g. `a(i) = 5`, `a.i = 5`, or `a{i} = 5`), and of course you could combine those (`[a(i), b{j}, c.k] = f()`). For that last example `getLValues()` will return the set `{'a', 'b', 'c'}`.
- The `new HashSet<>()` syntax is only valid in Java 7 and later – you can omit the type parameters and they'll be inferred. At this point the McLab framework requires Java 7, so you may as well use this syntax.
- Some parts of this are clumsy because I wanted to use only standard Java (+ McLab), without assuming knowledge of Guava. It would be possible to write the `caseAssignStmt` method much more succinctly.

The mapping between the six steps and the code is as follows:

1. State the problem precisely. This isn't expressed in code, but it's something you should do for yourself to nail down what code you want to write.
2. Define the domain. This is the generic type parameter – `Set<AssignStmt>`.
3. Is this a forward or backward analysis? This is the choice to extend `ForwardAnalysis`.
4. Write down flow equations for different language constructs. These are the `case` methods, e.g. `caseAssignStmt` here.
5. What's the merge operation? This is the the `merge` method, which implements set union by calling `Set#addAll()`.
6. What's the initial dataflow fact? This is the `newInitialFlow()` method, which return an empty hash set.

Changes from Jesse Doherty's thesis

A good resource on McSAF is Jesse Doherty's Master's thesis. However, since its publication, the framework has undergone a few important API changes. This means that most of the code samples and sample analyses won't run against the current version of McSAF.

Flow data representation

Chapter 5.2.1 of the McSAF thesis covers special collection types introduced by McSAF – `HashSetFlowSet`, `HashMapFlowMap`, and so on. These are all gone; you should use plain old `java.util` collections (`HashSet`, `HashMap`, etc.) instead.

`java.util.Set` implementations support union, intersection and difference via the `addAll`, `retainAll` and `removeAll` methods. Alternatively, since the Natlab.jar embeds Google's Guava libraries, you can use methods like `Sets.union()`, `Sets.intersection()`, and so on. You'll usually want to copy the result of these though, since they return "views" on the results of the operation.

For maps, the McSAF collections had this notion of mergers, and the union and intersection operations were capable of merging common values for you. The replacement for this is the `natlab.toolkits.analysis.MergeUtil` class and its `mergePut` and `unionMerge` methods.

API changes

McSAF flow analyses are expected to implement `merge` and `copy` methods that tell the framework how to merge the flow data at control flow merge points, and how to copy values, respectively. In the thesis, these had the following signatures:

```
void merge(A in1, A in2, A out);
void copy(A src, A dest);
```

These have been changed to:

```
A merge(A in1, A in2);
A copy(A src);
```

In other words, you now create the return values yourself instead of filling in a supplied output parameter, which turned out to be error-prone for various reasons.

Naming changes

The cumbersome `AbstractSimpleStructuralForwardAnalysis` and `AbstractSimpleStructuralBackwardAnalysis` were renamed to `ForwardAnalysis` and `BackwardAnalysis`, respectively.