The Static Analysis Framework OPAL

Intraprocedural Data Flow Analysis

The Monotone Framework

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

> If you have questions don't hesitate to join our public chat: Gitter
> If you find any issues, please directly report them: GitHub

Forwards Data Flow Analysis

Given the CFG of a method (be it in three-address code or in bytecode) it then is possible to perform a forward data flow analysis using:

```
<CFG>.performForwardDataFlowAnalysis[Facts >: Null <: AnyRef](
      seed: Facts,
      t:    (Facts, I, PC, CFG.SuccessorId) ⇒ Facts,
      join: (Facts, Facts) ⇒ Facts
): (Array[Facts], /*normal return*/ Facts, /*abnormal return*/ Facts)
```

- `Facts` is the type of the facts collected for each instruction; typically a set of facts (`Set[Fact]`).
- `seed` is the initial fact that holds on entry.
- `t` is the transfer function.
- `join` is the function to join the data flow facts if multiple control flow paths converge. Typically, a set union or intersection.

The transfer function is given the current facts, the current instruction, the program counter (pc) of the current instruction and the id of the successor instruction. The id is either:

- `CFG.NormalReturnId` (`== Int.MaxValue`) if the successor is the unique exit node representing normal returns.
- `CFG.AbnormalReturnId` (`== Int.MinValue`) if the successor is the unique exit node representing abnormal returns (i.e., an uncaught exception will be thrown by the instruction).
- [-65535,-1] to identify the catch handler that handles the exception thrown by the current instruction. The pc of the first instruction of the catch handler is (`- successorId`).
- the (valid) pc of the next instruction (the normal case.)

Based on the successor id it is then possible to decide which facts should be propagated. For example, if the successor id identifies an exception handler (i.e., we have an exceptional control flow) then we now that the evaluation of the instruction has failed. In case of the constructor call of an object we then now that the object is not properly initialized.

The transfer function is typically just a *big* pattern match over the relevant instructions that affect the set of facts.

Backwards Data Flow Analysis

Given the CFG of a method (be it in three-address code or in bytecode) is is possible to perform a backward data flow analysis using:

```
<CFG>.performBackwardDataFlowAnalysis[Facts >: Null <: AnyRef](
      seed: Facts,
      t:    (Facts, I, PC, CFG.PredecessorId) ⇒ Facts,
      join: (Facts, Facts) ⇒ Facts
): (Array[Facts], /*init*/ Facts)
```

- `Facts` is the type of the facts collected for each instruction; typically, a set of facts (`Set[Fact]`).
- `seed` is the initial fact (given that we have a backward analysis, it is the fact that holds at the exit nodes of the method).
- `t` is the transfer function.
- `join` is the function to join the data flow facts if multiple control flow paths converge. Typically, a set union or intersection.

The transfer function is given the current facts, the current instruction, the program counter (pc) of the current instruction and the id (either a pc or -1 if the current instruction is the first instruction) of the predecessor instruction. Please, recall that the first instruction can have multiple predecessors.

CFG nodes that are not reached from an exit node (e.g., if you have an infinite loop) will not be analyzed!

Examples

To get a deeper understanding how to instantiate the framework consider studying concrete implementations. In OPAL, we have implemented some analyses using the framework:

- Project Demos: org.opalj.tac.VeryBusyExpressions
- Project Demos: org.opalj.tac.AvailableExpressions