Static Analysis of Embedded DSL-s

Aivar Annamaa
University of Tartu

aivar.annamaa@gmail.com

February 6th, 2010

Problem

- DSL-s are often embedded as string literals in a GPL
  - SQL, RegEx, HTML
- Mistakes pop up at runtime
- Especially error prone together with conditional concatenation

Example: SQL in Java

```
...

String sql = "select id, name from persons";
if (dept != null) {
    sql += "where dept = ?";
}

// following may give runtime error
PreparedStatement stmt = conn.prepareStatement(sql);
...
```

Static analyzer for SQL embedded into Java

Should detect SQL errors at compile time

- Locate hotspots ie. method calls that cause runtime errors when given bad SQL as argument (eg. `Connection.prepareStatement`)
- Construct abstract value of argument expression
- Check abstract value for errors:
  - perform exhaustive testing on possible concrete values against real DB
  - (or try to parse the abstract value directly)
- (Analyze correct usage of ResultSet)
- (Keep track of different DB schemas used in the program)

Aims

- Be sound: no errors from analyzer $\Rightarrow$ no SQL prepare errors at runtime
- Be fast enough for on-line usage (while typing), even in case of big projects
- Be precise for common idioms of SQL construction
  - single literals and unconditional *intra*procedural concatenation (90%)
  - concatenations with few conditions or simple *inter*procedural constructions (9%)
- Be tolerable in rare complex cases (loops, many conditions, deep chains of method calls, etc.)

Conceptual framework for constructing abstract string

- Extract program slice for string expression at hotspot
- Perform constant propagation analysis (on that slice)
  - for each CFG node compute abstract environment – a mapping from string variables to abstract strings

```
Env: Var -> AbsStr
```

```
AbsStr ::= ConstStr String
         | Seq AbsStr AbsStr
         | Choice AbsStr AbsStr
         | IntStr
         | AnyStr
```

Expression evaluator

Computes abstract value of given expression in given environment

```
eval (StringLiteral s)  env = ConstStr s
eval (Var n)            env = env n
eval (Concat exp1 exp2) env =
          Seq (eval exp1 env) (eval exp2 env)

eval (IntExp e)    _ = IntStr
eval _             _ = AnyStr
```

Environment transformer for statements

Start at entry node with empty environment and work towards hotspot using environment transformer (`tr`) at each statment

```
tr (Assign var expr) oldEnv = update in var (eval expr)

tr (Block []) oldEnv = oldEnv
tr (Block s:ss) oldEnv = tr (Block ss) (tr s oldEnv)

tr (IfElse ifBlock elseBlock) oldEnv =
    merge (tr ifBlock oldEnv) (tr elseBlock oldEnv)
```

`merge` unions two environments pointwise using `Choice`

Handling loops using cheating approach

- For efficiency (and termination), pretend that loop bodies execute always once or twice
  - no need for fixpoint computation
- For soundness add `AnyStr` as choice to all variables assigned in the loop-body

```
tr (Loop header body) oldEnv =
                merge (merge onceEnv twiceEnv) anyEnv
                where
                    onceEnv = tr body oldEnv
                    twiceEnv = tr body onceEnv
                    anyEnv = anyStrForAllAss body
```

Going interprocedural

- Expression may use current method parameters
  - actual arguments at all possible callsites are analyzed
- Expression may include method calls
  - All possible target methods get evaluated context-sensitively
- In both cases, same evaluation procedure is used recursively
- Depth of such recursion is limited:
  - when limit is reached, then `AnyStr` is returned
  - gains efficiency in deep chains of method calls and avoids problems with recursive methods
- Needs class hierarchy analysis for better precision in case of polymorphic methods

Interpretation of the result

- Constructing abstract string always terminates, because of special treatments of loops and limited depth in interprocedural analysis
- If resulting abstract string contains `AnyStr`, then corresponding hotspot is reported as possible source of errors
- Otherwise:
  - all possible concrete strings are generated from abstract string (`IntStr` gets translated to '1')
  - each string is sent to DB for parsing and validating
  - if any of them raises an error, then hotspot is reported as possible source of errors

Opportunity for modularity

```
...
String getQuery(String grouper) {
    String sql = "select " + grouper + " as gr,"
      + "sum(income) as total_income "
      + "from results ";

     if (!grouper.lowercase().equals("dept")) {
         sql += " where period_year > 1970";
     }

     sql += "group by " + grouper;
}
...
stmt = conn.parseStatement(getQuery("dept"));
...
stmt = conn.parseStatement(getQuery("year"));
...
```

Modular dataflow analysis

- Continuous analysis (while typing) would be really nice
- Doing full-program analysis after each code edit may not be feasible
- General idea of modular interprocedural dataflow analysis:
  - each relevant method is analyzed independently and abstract summary of it's effect is cached (eg. in form of a table or graph)
  - later, if analysis of this method is needed in some context then it's cached summary is interpreted (instead of analyzing it again)
- Opportunity for metaprogramming:
  - compiling method summaries to real Java methods might give better performance than interpreting summary data each time

Current implementation

- Implemented in Java as an Eclipse JDT plugin
- Works in "bacth-mode", no modular on-line analysis yet
- Program slicing not explicitly present in the algorithm
- Working directly on AST, without separate CFG
- Abstract string construction works from hotspot backwards
- Can analyse business module of Compiere ERP system (200K LOC, 250 hotspots) in less than a minute
  - for 20 hotspots, result included `AnyStr` ie. at some point analyzer had said "not sure"
  - remaining 230 results expanded to 260 different concrete strings
  - 8 concrete strings didn't pass validation by DB
  - 4 of them real bugs

A screenshot