Automatic Repair of Static Analysis Violations Using Datalog

Anton Lyxell <alyxell@kth.se>

Supervisor Fernanda Madeiral / Examiner Martin Monperrus

2020-04-15

Motivation - Why Consider Datalog?

The Project

Datalog Evaluation

Program Repair in Datalog

Motivation - Why Consider Datalog?

Motivation - Why Consider Datalog?

> Working at the Datalog level eliminated much of the artificial complexity of a points-to analysis implementation, allowing us to concentrate on indexing optimizations and on the algorithmic essence of each analysis.
>
> – Smaragdakis and Bravenboer on DOOP (2010)

We argue that Datalog is so well suited to the implementation of a disassembler that it represents a qualitative change in what is possible in terms of accuracy and efficiency.
– Flores-Montoya and Schulte (2020)

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.
— Moseley and Marks in Out of the Tar Pit (2006)

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the ~~handling of state~~ and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are ~~code volume~~, and ~~explicit concern with the flow of control through the system~~.
— Moseley and Marks in Out of the Tar Pit (2006)

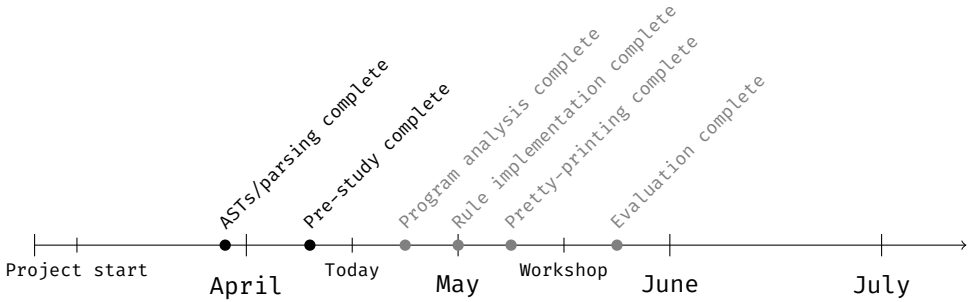
The Project

Automatic Repair of Static Analysis Violations Using

- Pre-study **(DONE)**
- Implementation **(IN PROGRESS)**
  - ASTs in Datalog **(DONE)**
  - Program analysis (type checking, dataflow) **(IN PROGRESS)**
  - Transformation rule implementation **(TODO)**
  - Pretty-printing **(TODO)**
- Evaluation **(TODO)**

Timeline

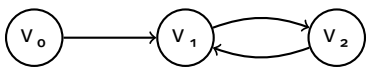
“The June schedule”



Datalog Evaluation

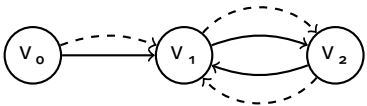
Datalog Evaluation

```
reachable(X, Z)  :-  reachable(X, Y), connection(Y, Z).
reachable(X, Y)  :-  connection(X, Y).
```



connection	
v_0	v_1
v_1	v_2
v_2	v_1

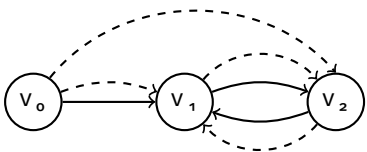
```
reachable(X, Z)  :-  reachable(X, Y), connection(Y, Z).
reachable(X, Y)  :-  connection(X, Y).
```



connection	
v_0	v_1
v_1	v_2
v_2	v_1

reachable	
v_0	v_1
v_1	v_2
v_2	v_1

```
reachable(X, Z)  :-  reachable(X, Y), connection(Y, Z).
reachable(X, Y)  :-  connection(X, Y).
```



connection	
v_0	v_1
v_1	v_2
v_2	v_1

reachable	
v_0	v_1
v_1	v_2
v_2	v_1
v_0	v_2

Program Repair in Datalog

Example, rewrite:

before: `if (list.size() == 0) {…}`

after: `if (list.isEmpty()) {…}`

before: `if (list.size() == 0) …`

```
... :-
    equals_expression(id, left, right),
    ...
```

after: `if (list.isEmpty()) …`

before: `if (list.size() == 0) …`

```
... :-
    equals_expression(id, left, right),
    method_invocation(left, object, "size", []),
    ...
```

after: `if (list.isEmpty()) …`

before: `if (list.size() == 0) …`

```
... :-
    equals_expression(id, left, right),
    method_invocation(left, object, "size", []),
    literal_integer(right, 0),
    ...
```

after: `if (list.isEmpty()) …`

before: `if (list.size() == 0) …`

```
... :-
    equals_expression(id, left, right),
    method_invocation(left, object, "size", nil),
    literal_integer(right, 0),
    has_java_collection_type(left).
```

after: `if (list.isEmpty()) …`

before: `if (list.size() == 0) …`

```
method_invocation(id, object, "isEmpty", nil) :-
    equals_expression(id, left, right),
    method_invocation(left, object, "size", nil),
    literal_integer(right, 0),
    has_java_collection_type(left).
```

after: `if (list.isEmpty()) ...`

Is this fast?

- Yes, the Datalog engine Soufflé compiles the Datalog code to C++
- If done right performs on par or better than hand-written C++ code
- Profiling and tuning is important

Tree flattening

Subtract(Add(Int(2), Int(3)), Int(3))

becomes

A. Subtract(B, C)
B. Add(D, E)
C. Int(3)
D. Int(2)
E. Int(3)

i.e. one linear preprocessing pass lets us do AST matching in constant time

Questions?

Automatic Repair of Static Analysis Violations Using Datalog

- Anton Lyxell <alyxell@kth.se> (email me!)