Combined Static and Dynamic Mutability Analysis

SHAY ARTZI, ADAM KIEZUN, DAVID GLASSER, MICHAEL D. ERNST

ASE 2007

PRESENTED BY: MENG WU

About Author

Program Analysis Group,

Computer Science and

Artificial Intelligence Laboratory

Students of Prof. Michael Ernst

Shay Artzi

VP Development, CTO at Zappix

Postdoc at IBM research

Adam Kiezun

Computational biologist in the Cancer Program at Broad Institute of Harvard and MIT

Postdoc at Division of Genetics of Brigham and Women's Hospital and Harvard Medical School

Outline

- Parameter Mutability Definition and Applications
- Classifying Parameters:
  - Staged analysis
  - Dynamic analyses
  - Static analyses
- Evaluation
- Conclusions

Parameter Mutability

Parameter P of method M is:

- Mutable if some execution of m can change the **state of an object** that at run-time corresponds to p
- Immutable if no such execution exists
- All parameters are considered to be fully un-aliased on top-level method entry

A method is pure (side-effect free) if:

- All its parameters are immutable (Including receiver and global state)

Parameter Mutability

The state of an object *o* is the part of the heap that is reachable from *o* by following references, and includes the values of reachable primitive fields.

Thus, reference immutability is **deep**—it covers the entire abstract state of an object, which includes the fields of objects reachable from the object.

```
void f(C c) {
D d = c.d;
E e = d.e;
e.f = null;
}
```

C is mutable.
It is mutated by c.d.e.f = null

Example and Classification

```
1   class C {
2      public C next;
3   }
4
5   class Main {
6      void modifyParam1(C p1, boolean doIt) {
7         if (doIt) {
8            p1.next = null;
9         }
10     }
11
12     void modifyParam1Indirectly(C p2, boolean doIt) {
13        modifyParam1(p2, doIt);
14     }
15
16     void modifyAll(C p3, C p4, C p5, boolean doIt) {
17        C c = p3.next;
18        p4.next = p5;
19        c.next = null;
20        modifyParam1Indirectly(p3, doIt);
21     }
22
23     void doNotModifyAnyParam(C p6) {
24        if (p6.next == null)
25           System.out.println("p6.next is null");
26     }
27     void doNotModifyAnyParam2(C p7) {
28        doNotModifyAnyParam(p7);
29     }
30  }
```

- p1 is directly mutable, in line 8
- p2 is indirectly mutable
- p3 is mutable because line 19 modifies p3.next.next.
- p4is directly modified in modifyAll (line 18), because the mutation occurs via reference p4
- p5 is mutable, cause p3 and p4 may be aliased
- p6, p7 is immutable, no execution of either method could modify an object passed to them

Using Mutability Information in Other Applications

- Program comprehension (Dulado 03)
- Modeling (Burdy 05)
- Verification (Tkachuk 03)
- Compiler optimization (Clausen 97)
- Program transformation (Fowler 00)
- Regression oracle creation (Marini 05, Xie 06)
- Invariant detection (Ernst 01)
- Specification mining (Dallmeier 06)
- Model-based Test Input Generation(Palulu, Artzi 06)

Staged Mutability Analysis

Idea:

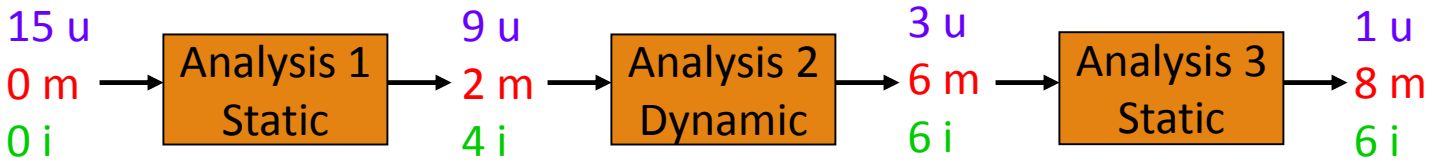
- Combine simple analyses to outperform a complicated one
- Harvest the power of both static and dynamic analyses
- Optionally use unsound analyses. Unsoundness is mitigated by other analysis in the combination and is acceptable for some

Advantages:

- Improve overall accuracy
- Scaleable

Pipeline approach:

- Connect a series of scalable analyses in a pipeline.
- The i/o of each analysis is a classification of all parameters.
- Analyses represent imprecision using the *unknown* classification



Staged Mutability Analysis

The problem of mutability inference is undecidable, so no analysis can be both sound and complete.

- An analysis is i-sound if it never classifies a mutable parameter as immutable .
- An analysis is m-sound if it never classifies an immutable parameter as mutable .
- An analysis is complete if it classifies every parameter as either mutable or immutable .

Staged Mutability Analysis

Dynamic Analysis

- Observe execution to classify as mutable
- Performance optimizations
- Optional accuracy heuristics
- Random generation of inputs

Static Analysis

- Intra-procedural points-to analysis
- Inter-procedural propagation phase

Dynamic Mutability Analysis

Parameter p of method m is classified as mutable if:

I. the transitive state of the object that p points to changes during the execution of m , and

II. p is not aliased to any other parameter of m

Implementation:
When object o is modified in method m during the execution of program, calculate reach(m,p) for each p, which is the set of objects that are transitively reachable from each parameter p. if o ∈ reach(m,p), and p is not aliased to other parameters, classify p as mutable.

Dynamic Mutability Analysis

How to make it fast:

- the analysis determines object reachability by maintaining and traversing its own data structure that mirrors the heap, which is

  faster than using reflection

- the analysis computes the set of reachable objects

  lazily, when a modification occurs

- the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set is modified

Dynamic Analysis Heuristic:

1. Classify a parameter as immutable at the end of/during the execution
   - All unknown parameters in methods that were executed more than N times and block coverage at least t%

Advantages:

- Algorithm classifies parameters as immutable in addition to mutable (adds 6% correctly classified immutable parameters to the best pipeline)

Disadvantages:

- May classify mutable parameters as immutable (0.2% misclassification in our experiments)

Dynamic Analysis Heuristic:

2. Using Known Mutable Parameters

- Treat object passed to a mutable parameter as if it is immediately mutated

Advantages:

- Can discover possible mutation that do not necessarily happen in the execution (In practice it was not highly effective in a pipeline due to the static analysis).
- Minor performance improvement by not waiting for the field write and storing less information (1% -5% in our experiments)

Disadvantages:

- Can propagate misclassification if the input classification contains misclassification (did not happen in our experiments)

Dynamic Analysis Heuristic:

3. Classifying aliased mutated parameters

- classifies a parameter p as mutable if the object that p points to is modified, regardless of whether the modification happened through an alias to p or through the reference p itself

Advantages:

- identify mutability in advance of further static analysis

Disadvantages:

- The heuristic is i-sound but m-unsound.

Randomly Generated Inputs to the Dynamic Analysis

Provided by user

- Exercises complex behavior
- May have limited coverage

Generated Randomly (Pacheco 06)

- Fully automatic process
- Focus on unclassified parameters
- Dynamic analysis can be iterated

Dynamic analysis in the pipeline	Correct %	Misclassified %
Using user input	86.9	3.8
Using focused Iterative random generated	90.2	2.8
Using both user & random generated input	91.1	3.4

Static Analysis

Analyses:

- Intra-procedural points-to analysis
- Inter-procedural propagation analysis

Very simple

- Scales well
- Very coarse pointer analysis
- No complicated escape analysis
- No use of method summaries

Designed to be used in conjunction with other analyses

- Other analyses make up for its weak points, and vice versa

Intraprocedural Static Analysis

Pointer Analysis

- Calculate which parameters each Java local may point to
- Assume that method calls alias all parameters

Intraprocedural Static Analysis

- Mark direct field and array mutations as mutable
- Mark parameters that aren't directly mutated and don't escape through method calls as immutable
- Leave the rest as unknown

Interprocedural Propagation

Uses the same pointer analysis

Constructs a Parameter Dependency Graph

- Shows how values get passed as parameters

Propagates mutability through the graph

- Unknown parameters that are only passed to immutable positions should be immutable
- Unknown parameters that are passed to mutable positions should be mutable

Static Analysis Example

```
class List {
    int size(List this){ return n;}
    void add(List this, Object o) {…
        this.array[index] = o; …
    }
    void addAll(List this, List l){…
        this.add(x); …
    }
}
```

Static Analysis Example

```
class List {
    int size(List this){ return n;}
    void add(List this, Object o) {…
        this.array[index] = o; …
    }
    void addAll(List this, List l){…
        this.add(x); …
    }
}
```

Initial classification – all unknown

Static Analysis Example

```
class List {
    int size(List this){ return n;}
    void add(List this, Object o) {…
        this.array[index] = o; …
    }
    void addAll(List this, List l){…
        this.add(x); …
    }
}
```

After Intra-Procedural analysis

Static Analysis Example

```
class List {
    int size(List this){ return n;}
    void add(List this, Object o) {…
        this.array[index] = o; …
    }
    void addAll(List this, List l){…
        this.add(x); …
    }
}
```

After Inter-Procedural propagation

Evaluation

- Pipeline construction
- Accuracy
- Scalability
- Applicability

6 subject programs; largest 185KLOC

Pipeline Construction

Always start with the intra-procedural analysis:

- Sound and very simple.
- Discover the classification of many trivial or easily detectable parameters

Always run propagation after each analysis:

- Only the first time is expensive (parameter graph is only calculated once)
- Adds between 0.3%-40% to the correct classification results of a pipeline ending with a non propagation, classification changing analysis

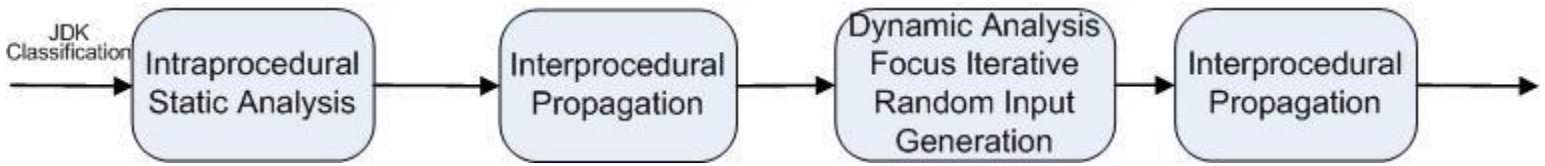
Static analyses should be the first part of the pipeline for the best combination of dynamic and static analysis

- The static analyses sets the ground for the following dynamic analysis reducing its imprecision by 75%

Pipeline Construction

- Random input generation is more effective then user input
- Use all the dynamic analysis heuristics

Best Combination (out of 168):



Accuracy Evaluation

Results for eclipse compiler (107 KLOC):

Analysis	Correct %	Imprecise %	Misclassified %	
			m/i	i/m
JPPA (Salcianu)	27.5	72.5	0.0	0.0
JPPA + Heuristic (mutable if possible write)	88.1	8.0	3.9	0.0
Staged Analysis	90.2	7.0	2.7	0.1

Scalability Evaluation

Execution times on Daikon (185KLOC):

Analysis	Total (s)
JPPA	5586
Intra-Procedural	167
… + Inter-Procedural	564 (mostly call graph construction)
… + Dynamic (Random focused iterative)	1484
… + Inter-Procedural	1493

Applicability Evaluation

Client application:
Palulu, Artzi 06: Model-based test input generation

Smaller Model ！ Less Generation Time !

Analysis	Nodes	Edges	Time (s)
No mutability information	444729	624767	6703
JPPA	131425	210354	4626
Staged Analysis	124601	201327	4271

Conclusions

Framework for a staged mutability analysis

Novel dynamic analysis
- Iterative random input generation is competitive with user input

Combination of lightweight static and dynamic analysis
- Scalable
- Accurate

Evaluation
- Sheds light into the complexity of the problem and the effectiveness of the analyses applied to it
- Investigate tradeoffs between analysis complexity and precision

Improve client applications
- Reduce model size for test generation

Thanks & Questions?

Reference:

“A formal definition and evaluation of parameter immutability” by Shay Artzi, Jaime Quinonez, Adam Kieżun, and Michael D. Ernst. Automated Software Engineering, vol. 16, no. 1, 2009, pp. 145-192.

http://publications.csail.mit.edu