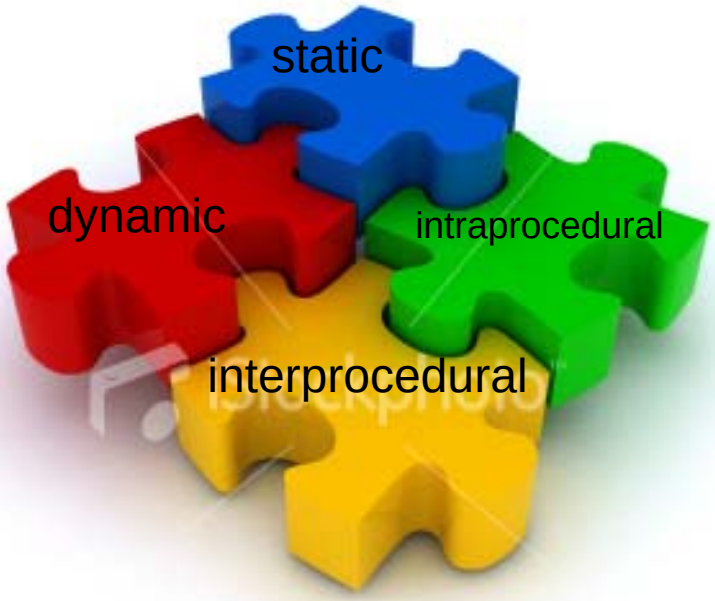


Combined Static and Dynamic Mutability Analysis

Shay Artzi, Adam Kiezun,
David Glasser Michael D. Ernst
CSAIL, MIT

Mutability/Immutability Definition

- Parameter P of method M is:
  - Mutable if some execution of M can change the state of P's referent object using P
  - Immutable if no such execution exists
- A method is pure (side-effect free) if:
  - All its parameters are immutable (including receiver and global state)

Mutability Example

```
class List {
    …
    int size(List this){ return n;}
    void add(List this, Object o) {…}
    List addAll(List this, List l){…}
    List clone(List this){
      return new List().addAll(this); }
    void remove(List this, Object O) {
      if(!this.contains(o)) return;
      …
   }
}
```

Mutability Example

```
class List {
    …
    int size(List this){ return n;}
    void add(List this, Object o) {…}
    List addAll(List this, List l){…}
    List clone(List this){
      return new List().addAll(this); }
    void remove(List this, Object O) {
      if(!this.contains(o)) return;
      …
    }
}
```

* Immutable

Mutability Example

```
class List {
    …
    int size(List this){ return n;}
    void add(List this, Object o) {…}
    List addAll(List this, List l){…}
    List clone(List this){
      return new List().addAll(this); }
    void remove(List this, Object O) {
      if(!this.contains(o)) return;
      …
    }
}
```

* Immutable
* Mutable

Mutability Example

```
class List {
    …
    int size(List this){ return n;}
    void add(List this, Object o) {…}
    List addAll(List this, List l){…}
    List clone(List this){
      return new List().addAll(this); }
    void remove(List this, Object O) {
      if(!this.contains(o)) return;
      …
   }
}
```

* Immutable
* Mutable
* Pure Method

Uses of Mutability Information

- Requires sound mutability information:
  - Modeling (Burdy 05)
  - Compiler optimizations (Clausen 97)
  - Verification (Tkachuk 03)
  - Typestate checker (Deline 04)
- Can use unsound mutability information:
  - Regression oracle creation (Marini 05, Xie 06)
  - Test input generation (Artzi 06)
  - Invariant detection (Ernst 01)
  - Specification mining (Dallmeier 06)
  - Program comprehension (Dolado 03)
  - Refactoring tools

Application: Test Input Generation

- Palulu tool for test generation (Artzi et al.,06)
  - Generate tests based on a model
    - Models describe legal sequences of calls
  - Smaller models
    - Faster systematic exploration
    - Greater search space exploration by random generator
    - Easier to compare models
  - Prune models
    - Removing calls that do not mutate objects
    - Preserving the state
    - Can reduce model by 90%

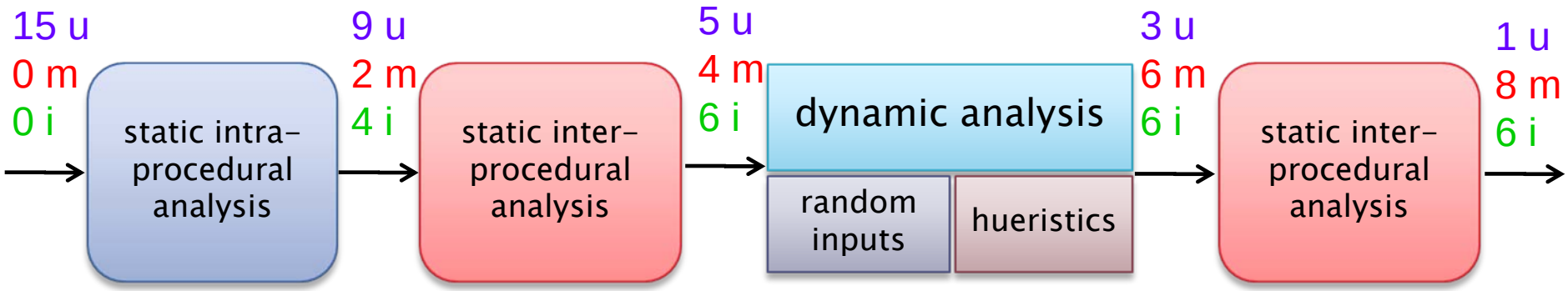
Outline

- Mutability Definition and Applications
- Technique:
  - Staged analysis
  - Static analyses
  - Dynamic analyses
- Evaluation
- Conclusions

Staged Analysis

- Connect a series of scalable analyses in a pipeline
- Has advantages of both static and dynamic analyses :
  - Accurate
  - Scalable analysis
- Sound analysis
  - Combining sound components
  - Optionally use unsound heuristics
    - Improve recall
    - Precision loss is mitigated by other analyses
    - Precision loss is acceptable for some uses

Pipeline Approach- Best Pipeline



- The i/o of each analysis is a classification of all parameters
- Analyses represent imprecision using the *unknown* classification

Static Analysis

- Analyses:
  - Intra-procedural analysis
  - Inter-procedural propagation analysis
- Properties
  - Simple points-to analysis
  - No complicated escape analysis
  - Scales well
- Designed to be used with other analyses
  - Other analyses make up for its weak points, and vice versa

Points-to Analysis

- Calculate which parameters each Java local may point to
- Properties
  - Context-insensitive
  - Flow-insensitive
    - Flow-sensitive until the first backward jump target
  - 1-level field-sensitive
- Computes two results
  - Over-estimate: Method calls alias all parameters
  - Under-estimate: Method calls don't alias parameters

Intra-procedural Analysis

- Algorithm
  - Use over-estimate points to sets
  - Mark direct field and array mutations as mutable
  - Mark parameters that aren't directly mutated and don't escape as immutable
  - Leave the rest as unknown
- Soundness
  - i-sound, m-unsound

Interprocedural Propagation

- Constructs a Parameter Dependency Graph (PDG)
  - Shows how values get passed as parameters
- Propagates mutability through the graph
  - unknown → all immutable becomes immutable
    - In an over-approximated PDG (over-estimate points-to sets)
  - unknown → any mutable becomes mutable
    - In an under-approximated PDG (under-estimate points-to sets)
- Soundness
  - i-sound (given i-sound input classification)
  - m-unsound

Dynamic Mutability Analysis

- Observe program execution
- On field/array write:
  - Mark as mutable all *non-aliased* formal parameters that transitively point to the mutated object

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```

Line 7 modifies the receiver of `add`.
Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```

Line 7 modifies the receiver of `add`.
Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);                    ←
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```

Line 7 modifies the receiver of `add`.
Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```


Line 7 modifies the receiver of `add`.
Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```


Line 7 modifies the receiver of `add`. Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Example

```
1. void addAll(List this, List other) {
 2.     for(Object o:other) {
3.         add(this, o);
 4. }
5. void add(List this, Object o) {
 6.     ...
7.     this.array[index] = o;
 8. }
```


Line 7 modifies the receiver of `add`.
Thus, the receivers of `add` and `addAll` are classified as mutable.

Dynamic Analysis Optimizations

- Maintain a model of the heap
  - Allows searching backwards in the object graph
  - Avoids using reflection for searching
- Caching
  - For each object, cache the set of corresponding formal parameters, and reachable objects
- Lazy computation
  - Compute reachable set only when a write occurs

Heuristic: Classifying Parameters as Immutable

- Classify a parameter of method M as immutable if
  1. M executed > *N* times
  2. M coverage > T%
  - At the end of the pipeline/component analysis
  - During analysis
- Advantages:
  - Algorithm classifies parameters as immutable in addition to mutable
    - adds 6% correctly classified immutable parameters to the best pipeline
  - Improve performance
- Disadvantages:
  - May classify mutable parameters as immutable
    - 0.1% misclassification in our experiments)

Immutable Misclassification Example

```
void remove(List this, Object O) {
  if (!contains(o)) return;
  ...
}
void main() {
   List lst = new List();
   lst.remove(5);
}
```

- The receiver of **remove** may be classified as immutable
- Unlikely in practice: only if every call to **remove** is a no-op

Heuristic: Using Known Mutable Parameters

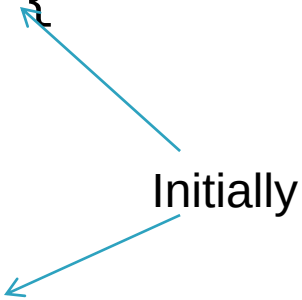
- Treat object passed to a mutable parameter as if it is immediately mutated
- Advantages:
  - Can discover potential mutation that does not happen in the execution
  - Minor performance improvement
- Disadvantages:
  - Can propagate misclassifications
    - Did not happen in our experiments

Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```

Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```



Using Known Mutable Parameters Improves Accuracy

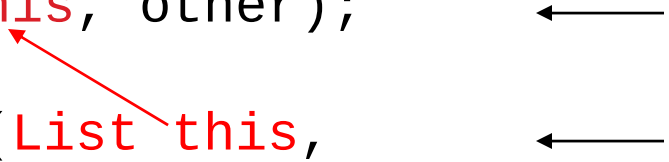
```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```

Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);        <----
}
void addAll(List this,         <----
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```

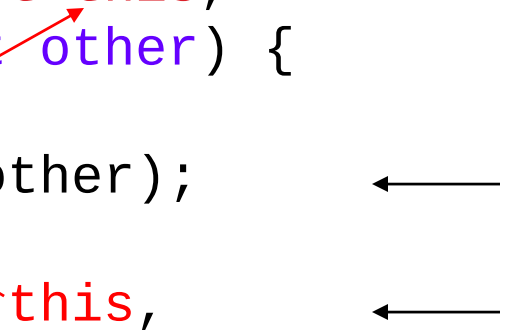
Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```



Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```



Using Known Mutable Parameters Improves Accuracy

```
void main() {
   List lst = new List();
   List lst2 = new List();
   lst2.copyInto(lst);
}
void copyInto(List this,
              List other) {
   ...
   addAll(this, other);
}
void addAll(List this,
            List other) {
  for(Object o:other) {
    add(this, o);
  }
}
```

Dynamic Analysis Input (Execution)

- Provided by user
  - May exercise complex behavior
  - May have limited coverage
- Generated randomly (Pacheco et al. 06)
  - Fully automatic
  - Focus on unclassified parameters
  - Dynamic analysis can be iterated
- Focused iterative random input outperformed user input

Evaluation

- Pipeline construction
  - Finding the best pipeline
- Accuracy
- Scalability
- Applicability

Subject Programs

program	kLOC	# non–trivial params
jolden*	6	470
Sat4j	15	1136
tinysql	32	1708
htmlparser	64	1738
eclipse compiler*	107	7936
daikon	185	13319

* Determined correct classification for the all parameters

Pipeline Construction

- Start with the intra-procedural static analysis:
  - Accurate and very simple.
  - Classify many trivial or easily detectable parameters
- Run inter-procedural static analysis after each analysis:
  - Only the first time is expensive (PDG creation)
  - Improves the classification by 0.3%-40%
- Static analyses should precede dynamic analysis
  - Reduces imprecision by 75%

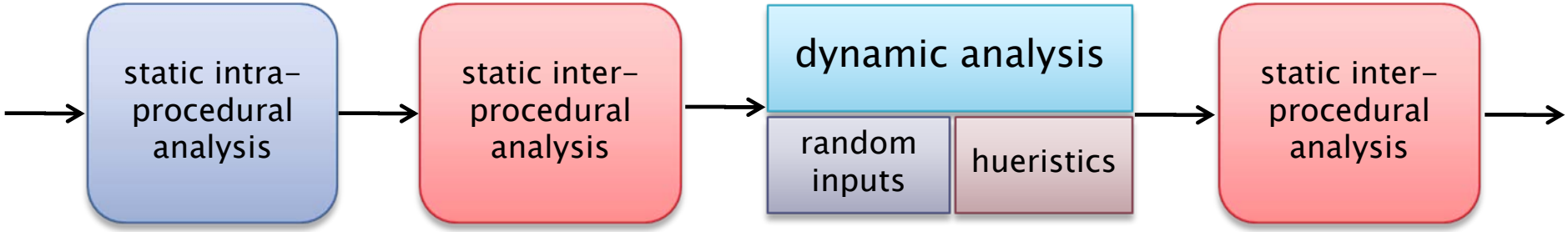
Pipeline Construction (Dynamic)

- Random input generation is more effective than user input

input to the dynamic analysis	Correct %	Misclassified %
user input	86.9	3.8
focused iterative random generated	**90.2**	**2.8**
both user & random generated input	91.1	3.4

- dynamic analysis heuristics are useful
  - Recall : +0.151%
  - Precision: -0.004%

Best Pipeline



- Out of 192 different pipelines

Accuracy Results

- Results for the eclipse compiler
  - 107 KLOC
  - Correct classification of~8000 parameters

Analysis	immutable		mutable	
	Recall	Precision	Recall	Precision
Best recall staged	0.928	0.996	0.907	0.971
Sound staged	0.781	1.00	0.915	0.956
JPPA	0.734	0.998	n/a	n/a

Scalability Evaluation

- Execution times on Daikon
  - 185KLOC
  - Largest subject program

Analysis	Total (s)
JPPA	5586
Staged Analysis	1493

Applicability Evaluation

- Client application:
  - Palulu (Artzi et al.,06): Model-based test input generation

Subject program	Analysis	Nodes	Edges
eclipse compiler + daikon + jolden	No mutability	445k	625k
	Staged Analysis	125k	201k
	JPPA	131k	210k
tinysql + htmlparser + sat4j	No mutability	49k	68k
	Staged Analysis	8k	13k
	JPPA	N/A	N/A

Previous Work

Static inference tools

- Properties
  - Requires precise pointer analysis
  - Conservative
  - Scalability issues
- Systems
  - Rountev 04
  - Salcianu and Rinard 05
  - Quinonez et al 07 (javarifier)
  - Greenfieldboyce and Foster 07 (jqual)

Dynamic inference tools

- Properties
  - Classify method only
- Systems
  - Xu et al 07
  - Dallamier and Zeller 07

Type/annotation systems

- Properties
  - Approximation of the immutable definition
  - Readable
  - Checkable
- Systems
  - Javari – Tschantz et al 05
  - IGJ – Zibin et al 07
  - JML – Burdy et al 03

Conclusions

- Framework for staged mutability analysis
- Novel dynamic analysis
- Combination of lightweight static and dynamic analysis
  - Compute both mutable and immutable
  - Scalable
  - Accurate
- Evaluation
  - Evaluated many sound and unsound instantiations
  - Investigate complexity vs. precision tradeoffs
  - Aids client applications