Collections Frameworks for Points-to Analysis

Tobias Gutzmann, Jonas Lundberg, and Welf Löwe

(tobias.gutzmann|jonas.lundberg|welf.lowe)@lnu.se

September 23, 2012

Introduction & Motivation

- Points-to analysis (P2A): Static program analysis computing reference information
  - What objects are possibly referenced by a field?
  - Used, e.g., for call graph construction, statically resolving polymorphic calls, down-cast safety
- Collections frameworks: Part of almost each standard library of programming languages
- Points-to analysis has a hard time analyzing collections frameworks
  - Lots of features for programmers, but P2A only needs to know: What goes into a collection object, also can get out of it
- → special handling of collections classes for improving both performance and precision

Special handling of collection classes in Points-to analysis

- Points-to information of collection classes often not of interest
- Basic idea (Liang et al., PASTE'01): replace calls to methods of collection classes with field accesses:
  - $c.add(o) \rightarrow c.elem = o$
  - $o = c.get() \rightarrow o = c.elem$

Special handling of collection classes in Points-to analysis

- Points-to information of collection classes often not of interest
- Basic idea (Liang et al., PASTE'01): replace calls to methods of collection classes with field accesses:
  - $c.add(o) \rightarrow c.elem = o$
  - $o = c.get() \rightarrow o = c.elem$
- Drawbacks:
  - Not sound: callbacks not taken into consideration
  - Must be implemented for each P2A implementation separately

Our approach

- Basic observation: No strong updates in P2A, so use base type fields instead of arrays
    - Possible as backing data-structures in collection classes are well encapsulated
- Iterators etc. just expose the elements of the collection objects, just through a different API
- Note: Some preconditions must be fulfilled, please cf. paper

Replacement classes by example

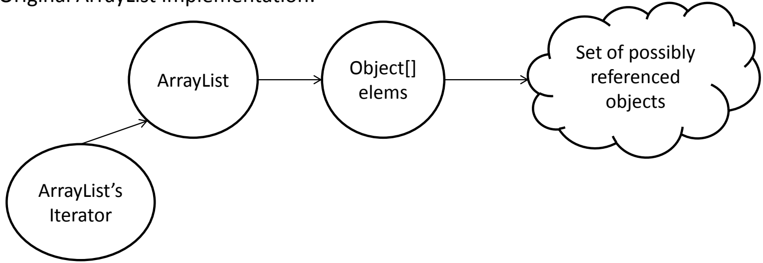
```
class ArrayList extends AbstractList {

    private Object elems; // non-array field
    Object get(int i) { return elems; }
    void add(Object o) { elems = o; }
    // ...




}
```

Replacement classes by example

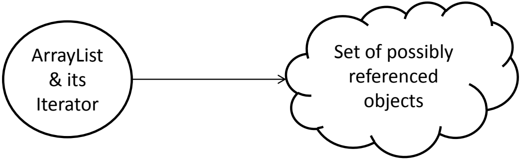
```
class ArrayList extends AbstractList
              implements Iterator {
    private Object elems; // non-array field
    Object get(int i) { return elems; }
    void add(Object o) { elems = o; }
    // ...
     Iterator iterator() { return this; }

    boolean hasNext() { return true; }
    Object next() { return elems; }
}
```

Reference modeling in P2A

Original ArrayList implementation:



Replacement ArrayList implementation:



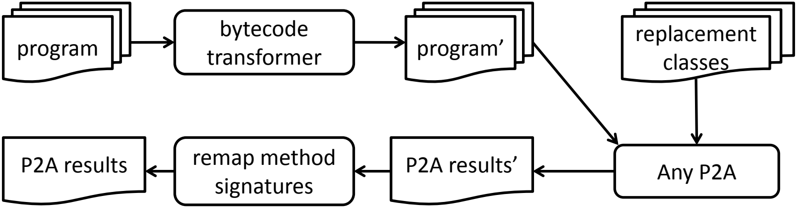
Changed method signatures

- Some classes now implement conflicting interfaces. Needed to change return types of some methods.

class and method	return type change	reason
Collection.remove(Object)	boolean → Object	Map.remove(Object)
Iterator.remove()	void → Object	Queue.remove()
ListIterator.add(Object)	void → boolean	Collection.add(Object)

→ programs must be transformed prior to being analyzed

Engineering Process



- P2A implementation never knows, no adaptation required

Evaluation

Setup[1]

- Experiments with P2SSA (our own P2A) with two different settings, as well as Spark and Paddle (Soot framework).
- 9 benchmark programs. Note: Two of them make (almost) no use of collection classes in application code.
- Metrics:
  - Call graph
  - Object call graph: a more fine-grained version of call graph
  - Heap: Size of abstract heap
- Validated by comparing with results from dynamic analysis
- Spark: no improvements, not further discussed.

[1] All experiments performed on a Standard Desktop PC, Intel Core 2 Quad Q9550, 2.83Ghz, 4GB RAM, 32-bit Windows XP, JDK 1.6.0 22, with JVM arguments -Xmx1200M -Xss30M. All results are average of three runs.

Evaluation II

Performance

- Transformation of classes took 1.1 seconds on average
- Paddle $\sim$24%, P2SSA$_1$ $\sim$9%, P2SSA$_2$ $\sim$17% faster on average

Precision

- P2SSA$_1$ hardly any improvements, not reflected below
- Call graph: on average improved by $\sim$1% (nodes) resp. $\sim$2% (edges) (Paddle, P2SSA$_2$)
- Object call graph: on average improved by $\sim$1.5% (nodes) resp. $\sim$4% (edges) (P2SSA$_2$)
- Heap: on average improved by $\sim$7% (P2SSA$_2$)

Conclusion

- Improved precision while at the same time reduced costs

Other aspects

- Even better results with inlining of collection classes methods (but that's specific to each P2A implementation); cf paper
- Works with application-specific collection classes, as they are not replaced
- Preliminary home: `http://homepage.lnu.se/staff/tgumsi/collections/`
- *Applicable to other static analyses !?*

The End

Thank you very much for your attention!