**Objectives**

Spark is a modular toolkit for flow-insensitive may points-to analyses for Java, which enables experimentation with:

- various parameters of pointer analyses
- various implementations of pointer analyses
- various client analyses

Which affect the accuracy, efficiency, and size of the result of a pointer analysis.

Spark allows experimentation with the following parameters that affect the accuracy, efficiency, and size of the result of a pointer analysis:

- Is the initial call graph built by CHA, RTA, VTA, ...?
- Is an initial call graph required, or is it constructed during the analysis?
- Are declared types and casts respected?
- Are variables in SSA form, UD-DU webs, or as in original source?
- Are object instances distinguished in field/array references?
- Appropriate level of context sensitivity?
- Subset (Andersen) or unification (Steensgaard)?

**Pointer Analysis Parameters**

- Subset (Andersen) or unification (Steensgaard)?
- Appropriate level of context sensitivity?
- Are object instances distinguished in field/array references?
- Are variables in SSA form, UD-DU webs, or as in original source?
- Are declared types and casts respected?
- Is an initial call graph required, or is it constructed during the analysis?
- Is the initial call graph built by CHA, RTA, VTA, ...?
Spark is a component of Soot, a framework for analyzing, optimizing, annotating, and decompiling Java class files. It supports a number of intermediate representations, ranging from stack-based, bytecode-like Baf to structured, Java-source-like Dava. Most analyses and transformations in Spark, in particular, operate on Jimple, a stackless, typed, three-address intermediate representation of the program. Spark reads the Jimple representation produced by Soot, and computes may points-to information for all pointer variables in the program. This information allows client analyses to determine whether two pointer variables may be aliased, as well as the types that may reach each variable. The results of these client analyses can be used further by Soot, or encoded in class file attributes for use by a JIT compiler or other optimizer.
The pointer assignment graph is a flow-insensitive representation of the program source. Simple (p) or field reference (p.f) nodes (depending on pointer analysis parameters) represent all locations storing pointers. For a context sensitive analysis, multiple nodes may represent a single variable in different contexts. Edges represent not just explicit assignments, but also flow through method parameters, return values, and exceptions. Some or all edges can be made bi-directional for a unification-based analysis. Every node has a declared type which the solver may use.
Java class files may contain optional named attributes with arbitrary data. Attributes can be used to communicate results of analyses to a virtual machine, JIT compiler, or other optimizer. Soot provides an annotation framework which allows annotations to be associated with classes, fields, methods, or individual statements, and propagated cleanly between its intermediate representations. We are experimenting with encoding side-effect information in attributes for use by JIT compilers. Such side-effect information can also be encoded in the class file as annotations for use by a virtual machine or JIT compiler. If we can determine that neither the write to q.f nor the call to getX() access p.f, then we can move the redundant load and store of p.f outside the loop. Such side-effect information is one example client of points-to analysis. Side-effect analysis computes a dependence graph between statements possibly reading or writing fields. This information is used by other analyses within Soot, such as common subexpression elimination. It can also be encoded in the class file as annotations.

```java
int getX() {
    return this.x;
}

O foo(O p, O q) {
    int ret = 1000000;
    while (ret > 0) {
        p.f = ret;
        q.f = getX();
        ret = p.f - 1;
    }
    return ret;
}
```

1. For each statement, Spark encodes numbered nodes representing locations read and written. Repeated uses of the same reference use the same node. This information is used by other analyses. The dependence graph between statements possibly reading or writing fields is used by other analyses. Such side-effect analysis is one example client of points-to analysis.
In the solver, we experiment with implementation details which affect efficiency.

The solver can collapse strongly connected components and rooted DAGs of simple nodes in the pointer assignment graph. It can then propagate points-to sets for simple nodes in a single pass. It must iteratively propagate sets for field reference nodes to their aliases. Respecting declared types increases precision, and prevents blowup in the number of aliased field references, but reduces set propagation opportunities. The solver can collapse strongly connected components and rooted DAGs of simple nodes in the pointer assignment graph. It may be compiled to execute efficiently on a single pass. It can then propagate points-to sets for simple nodes in a single pass. It must iteratively propagate sets for field reference nodes to their aliases. Some analyses result in more complex graph representations, which may reduce graph simplification opportunities. The solver is available for other researchers to implement their own points-to analyzers.

We have developed a framework which allows disparate flow-insensitive analyses that must consider the effects of native methods to share a single library of simulated native methods. Each native method is represented by a concrete subclass of the Naive Method Simulation abstract class, in which the method's effects are described in terms of abstract operations such as object allocation, assignments, and field accesses. The Naive Method Simulation abstract class is defined in terms of abstract operations such as object allocation, assignments, and field accesses. Each native method is represented by a concrete subclass of the Naive Method Simulation abstract class, in which the method's effects are described in terms of abstract operations such as object allocation, assignments, and field accesses. The Naive Method Simulation abstract class is defined in terms of abstract operations such as object allocation, assignments, and field accesses. We will use Spark to answer these questions:

- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyzers?
- How can analysis results be communicated securely to JIT compilers?
- How can analysis results be used effectively by JIT compilers?
- Which pointer analysis implementations fit well with which client analyzers?
- Which pointer analysis parameters are applicable for different analyzers?
- Which pointer analysis parameters are applicable for Java?
- Which pointer analysis parameters are applicable for different analyizers?
Credits

Ondřej Lhoták
Feng Qian
John Jorgensen
Laurie Hendren

Sable Research Group, School of Computer Science
McGill University, Montreal, CANADA

www.sable.mcgill.ca

This work was funded in part by NSERC, a Richard H. Thomison Fellowship, and an IBM Faculty Development Grant.