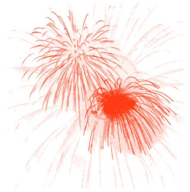


SPARK: Soot Pointer Analysis Research Kit

Ondřej Lhoták

Objectives

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

- various parameters of pointer analyses which affect accuracy and efficiency
- various implementations of pointer analyses
- various client analyses

Pointer Analysis Parameters

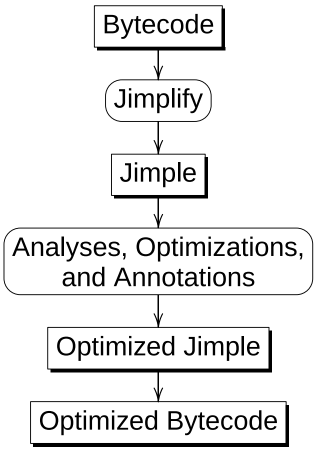
Spark allows experimentation with the following parameters that affect the accuracy, efficiency, and size of the result of a pointer analysis.

- Subset (Andersen) or unification (Steensgard)?
- Appropriate level of context sensitivity?[a]
- 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 pointer analysis?
- Is the initial call graph built by CHA, RTA, VTA, ... ?

---

[a] Not yet implemented in current version

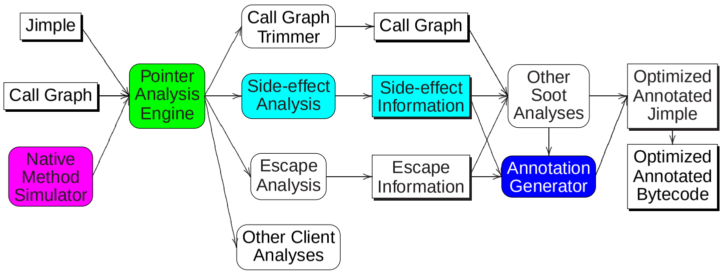
Soot Overview



Spark is a component of Soot, a framework for analyzing, optimizing, annotating, and decompiling Java class files. It supports a number of intermediate representations of varying levels, ranging from stack-based, bytecode-like Baf to structured, Java-source-like Dava.

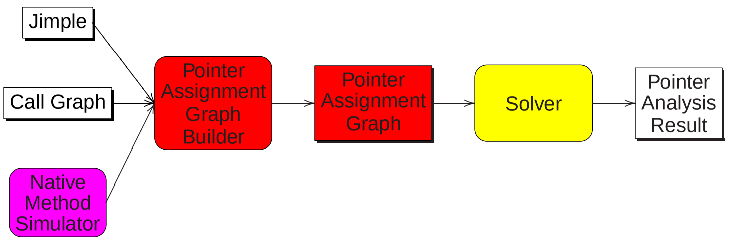
Most analyses and transformations in Soot, and the Spark framework in particular, operate on Jimple, a stackless, typed, three-address intermediate representation.

Spark in Soot



Spark reads the Jimple intermediate 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.

Pointer Analysis Engine



The pointer analysis engine consists of two main components:

- The builder builds a simple intermediate representation of the flow of pointers in the program. Pointer analysis parameters determine how the features of the program are represented.
- A solver produces the required may points-to information. The solver is mostly independent of the pointer analysis parameters. We can experiment with various solver implementations.

Pointer Assignment Graph

p ← new C	p ← q
p ← q.f	p.f ← q

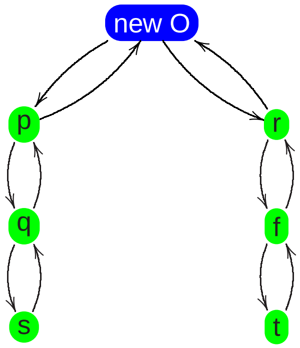
The pointer assignment graph is a flow-insensitive representation of the program source.

- 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.

Example 1

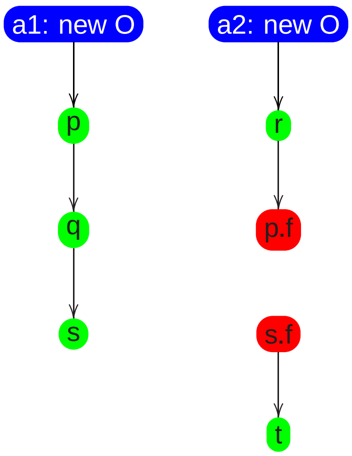
```
static void foo() {
a1: p = new O();
    q = p;
a2: r = new O();
    p.f = r;
    t = bar( q );
}

static O bar( O s ) {
    return s.f;
}
```



This graph would be made from the code fragment by a unification-based analysis, represented by bi-directional edges. Object instances are not distinguished, so a single simple node represents all instances of field `f`. All allocation sites of each type are grouped together in a common node.

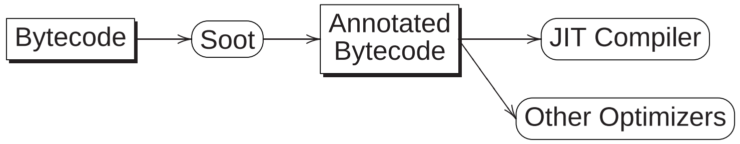
Example 2



This is the pointer assignment graph that would be produced from the example code using different settings of pointer analysis parameters. A subset-based analysis is being done in this case, so edges are directed. Object instances are distinguished, so separate nodes represent `s.f` and `p.f`. Each allocation site is represented using its own node.

Annotations

- 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.



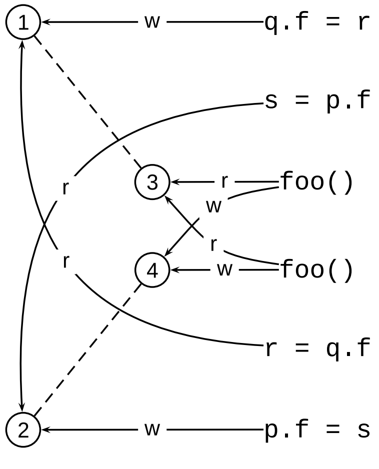
Side-effect Analysis

```
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;
}
```

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 analysis 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 for use by a virtual machine, JIT compiler, or optimizer.

Side-effect Information in Attributes



1. For each statement, Spark encodes numbered nodes representing locations read and written. Repeated uses of the same reference use the same node.
2. Nodes representing overlapping sets of locations are connected in a graph (dashed lines).

Here, `foo()` reads locations pointed to by `q.f`, and writes those pointed to by `p.f`.

Solver

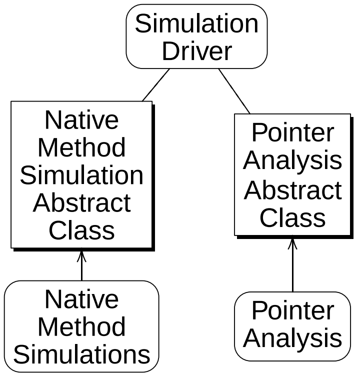
In the solver, we experiment with implementation details which affect efficiency.

- 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 graph simplification opportunities.

⋆ Implementation of points-to sets has a huge effect on analysis times. We are investigating alternative implementations such as OBDDs.

Native Method Simulator



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 Native Method Simulation abstract class, in which the method's effects are described in terms of abstract operations such as object allocations, assignments, and field reads and writes. Each analysis provides an implementation of the Pointer Analysis abstract class, where it defines its representation of the abstract operations.

Experiments and Future Work

We will use Spark to answer these questions:

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

Spark is available for other researchers to implement their points-to analyses within it, so that they may be compared in a common context. It is included in version 1.2.4 of Soot, available under the LGPL at `www.sable.mcgill.ca/soot`.

Credits

Ondřej Lhoták
Feng Qian
John Jorgensen
Laurie Hendren

Sable
McGill

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. Tomlinson Fellowship, and an IBM Faculty Development grant.