Using Soot as a Program Optimizer

Patrick Lam (plam@sable.mcgill.ca)

March 23, 2000

1 Goals

This tutorial describes the use of Soot as an optimization tool. After completing this tutorial, the user will be able to use Soot to optimize classfiles and whole applications.

Prerequisites The user should have a working installation of Soot; successful completion of the introduction is one way to exercise one's installation of Soot.

2 Classfile Optimization

Soot is able to optimize individual classfiles. Some of the transformations which can be carried out on individual classfiles include: copy propagation, constant propagation and folding, conditional branch folding, dead assignment elimination, unreachable code elimination, unconditional branch folding, unreachable code elimination and unused local elimination.

Common subexpression elimination has also been developed, but the current implementation is slow; it must be explicitly enabled. (This is described in the phase-options document.) Partial-redundancy elimination is being developed.

In order to optimize the `Hello` example from the previous tutorial, we issue the command:

```
> java soot.Main -O Hello
Transforming Hello...
```

Soot will then leave a new, improved `Hello.class` file in the current directory. For this class, the improvement after Sootification is not so obvious. Soot does, however, eliminate unused locals. Try adding an unused local to `Hello` and giving this command:

```
> java soot.Main -O --jimple Hello
Transforming Hello...
```

You should see that the unused local is no longer present.

Any number of classfiles can be specified to Soot in this mode, as long as they are in the `CLASSPATH`.

Hidden Trap Note that your classfile may belong to some package; it may be called, for instance, `soot.Scene`. This indicates that the `Scene` class belongs to the `soot` package. It will be in a `soot/` subdirectory. In order to Sootify this file, you must be in the parent directory (not `soot/`), and you must specify `java soot.Main -O soot.Scene`.

Unfortunately, our current optimizations with `-O` tend to have little effect on the program execution time.

3 Program Optimization

Soot has another mode of operation; it can run in *application* mode. In that case, the user specifies the main classfile, and Soot will load all needed classes.

This allows Soot to carry out whole-program transformations; for instance, method inlining requires the whole program to correctly resolve virtual method calls.

To specify that Soot should do whole-program optimizations (`-W`), as well as single-class optimizations, use the command:

```
> java soot.Main --app -W Hello
Transforming Hello...
```

Soot will write out all classes except those in the `java.*`, `javax.*` and `sun.*` packages.

The default behaviour of `-W` is to statically inline methods. Soot is also capable of static method binding; use

```
> java soot.Main --app -p wjop.smb disabled:false -p wjop.si disabled -W -O
       Hello
```

This type of optimization has produced significant speedups on some benchmarks.

4 Summary

This lesson has described how Soot can be used to optimize classfiles and whole applications.

5 History

- March 14, 2000: Initial version.
- March 23, 2000: Changed documentation to reflect fact that -W includes -O.