

Building Static Analysis With ExtendJ

Jesper Öqvist, PhD Student, LU

About Me

- PhD student on Compiler Construction
- Intern at Google in Silicon Valley this summer, worked on static analysis
- Been working on ExtendJ and JastAdd
- I am the current maintainer of ExtendJ

Static Analysis at Google

30K commits each day, analyzed by Tricorder.

A single large codebase, [~2 billion lines of code](#).

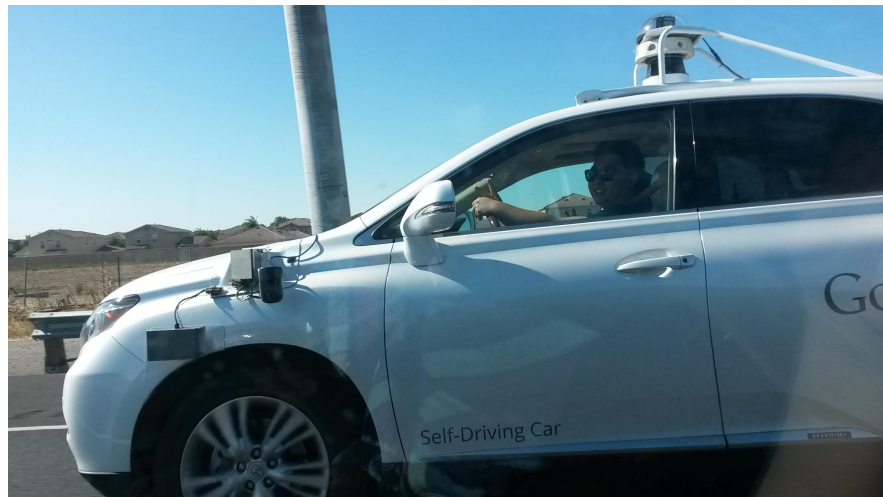
What I did at Google:

- Tricorder - added new analyzers
- Shipshape - integrated analyzers, improved Jenkins plugin
- ExtendJ for writing analyzers



Work at Google!

<https://www.google.se/about/careers/students/>



What is ExtendJ?

An extensible Java compiler using JastAdd attribute grammars.

- Declarative - side-effect free evaluation removes lots of headache
- Modular - build with only the parts you need
- Extensible - add your own module without touching old code

Previously known as “JastAddJ”, developed at Lund University by Torbjörn Ekman, et al.

Why do we want another Java compiler?

ExtendJ has many uses that are difficult or time consuming to implement in Javac or JDT:

- Experimenting with new language features
- Building source transformation tools
- Building static analysis

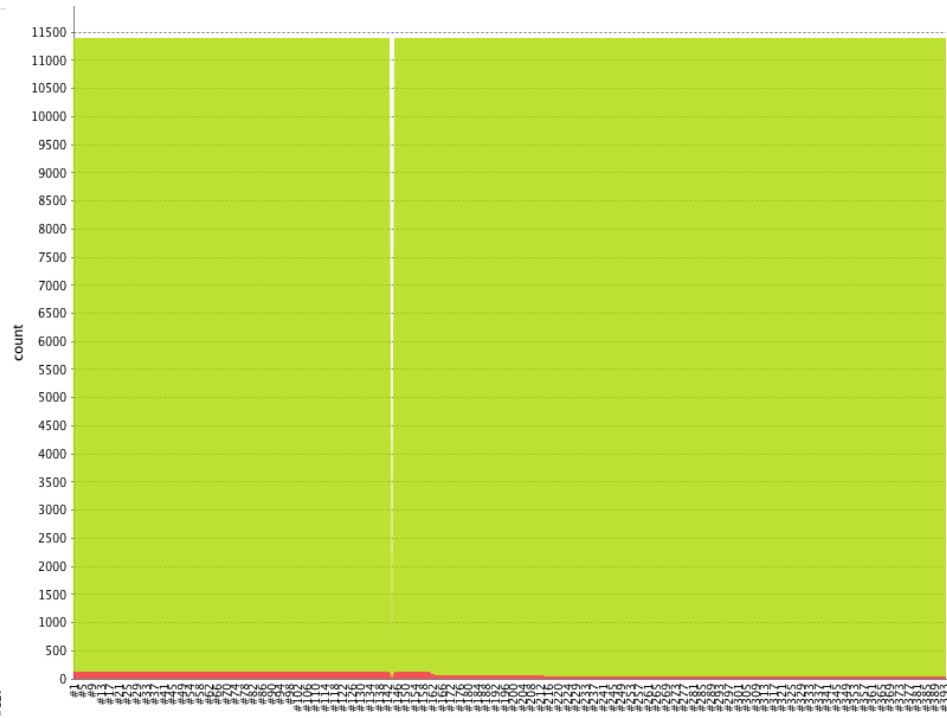
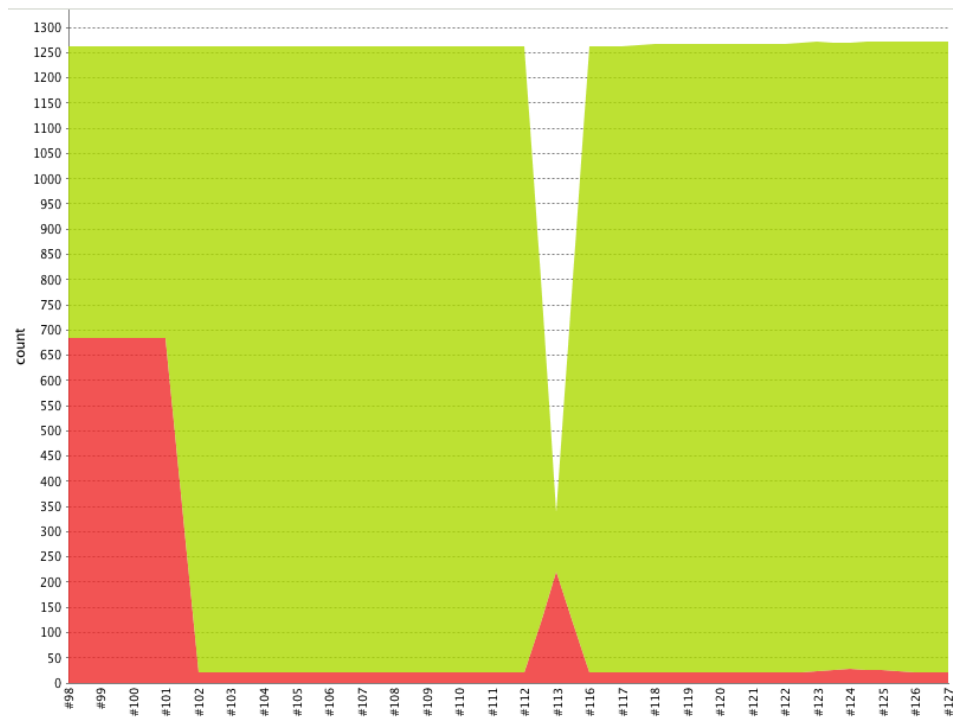
Other extensible compilers like Polyglot:

- Not as high Java conformance



Regression Testing

Testing is essential for compiler development!



API Overview

`Program ::= CompilationUnit*; // Program is a list of Java source files.`

`CompilationUnit ::= TypeDecl*; // ClassDecl, EnumDecl, InterfaceDecl.`

`TypeDecl ::= BodyDecl*; // MethodDecl, ConstructorDecl, FieldDecl.`

`abstract Stmt;`

`IfStmt : Stmt ::= Condition:Expr Then:Block [Else:Block];`

<http://jastadd.org/releases/jastaddj/7.1/doc/index.html>

API Overview

Attributes are used to look up semantic information. Some examples:

```
decl().type(); // Get the type of a declaration.
```

```
lookupType("java.util", "Collection"); // Lookup specific type.
```

```
lookupMethod(access); // Find method for access.
```

```
decl().hostType();
```

Final Checker Example <https://bitbucket.org/extendj/final-checker>

A simple extension to check for variables that could safely be declared 'final'.

Final means that a variable can only be assigned once.

```
public class Test {  
  
    // 'args' is never assigned, so it is effectively final.  
    public static void main(String[] args) {  
        int a = args.length;  
        for (int i = 0; i < a; ++i) {  
            a = m(i, a);  
        }  
    }  
}  
...
```

Final Checker Example

The final checker extension is a good starting point to make your own extension.

```
static int m(final int i, int a) {  
    a = i;  
    int c;  
    c = a; // only assigned once  
    int d = a + i;  
    return d;  
}
```

```
11:21:05 jesper@yolo ~/git/final-checker $ java -jar finalchecker.jar Test.java  
checking Test.java  
Test.java:4: parameter args is effectively final  
Test.java:15: variable c is effectively final  
Test.java:17: variable d is effectively final  
11:21:25 jesper@yolo ~/git/final-checker $
```

Final Checker Implementation

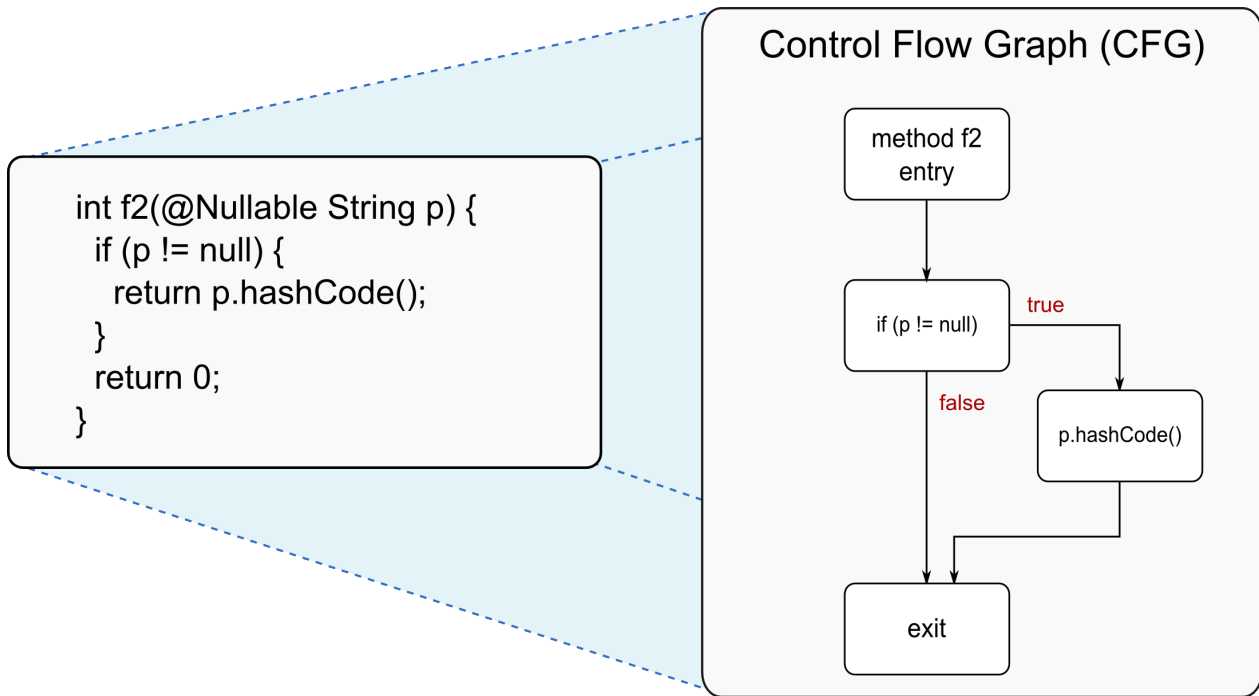
```
coll java.util.List<Variable> CompilationUnit.effectivelyFinalVariables()  
  [new LinkedList<Variable>()]  
  with add  
  root CompilationUnit;
```

```
VariableDecl contributes declaration()  
  when !declaration().isFinal() && declaration().isEffectivelyFinal()  
  to CompilationUnit.effectivelyFinalVariables()  
  for compilationUnit();
```

```
ParameterDeclaration contributes this  
  when !isFinal() && isEffectivelyFinal()  
  to CompilationUnit.effectivelyFinalVariables()  
  for compilationUnit();
```

Control Flow Graphs (CFGs)

A CFG, captures the possible control flow in a program:



SimpleCFG

Control flow graphs are useful for static analysis.

We have an intraflow module already for ExtendJ, but it generated dense graphs - collected control flow between every expression.

SimpleCFG is a new CFG module where you instead select which statements or expressions should appear in the graph.


The idea is to insert nonterminal attributes where needed to represent CFG nodes.

Nullable Dereference

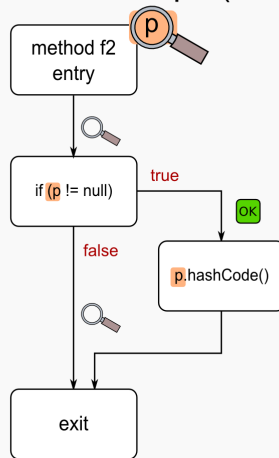
An analyzer testing for dereferences of parameters declared as @Nullable

Uses CFG module to do control flow sensitive analysis.

```
int f2(@Nullable String p) {  
    if (p != null) {  
        return p.hashCode();  
    }  
    return 0;  
}
```



Control Flow Graph (CFG)



Control flow dependent analysis:

- search in the CFG after dereferences of **p**
- skip branches guarded by null tests

Benefits of implementing analysis in ExtendJ

- Pick only the parts of the compiler that are needed
- I stripped out most type lookup for the Nullable Dereference analyzer
- Code generation not necessary
- Analysis is very fast!

Extension Architecture

