Compiler projects 2024

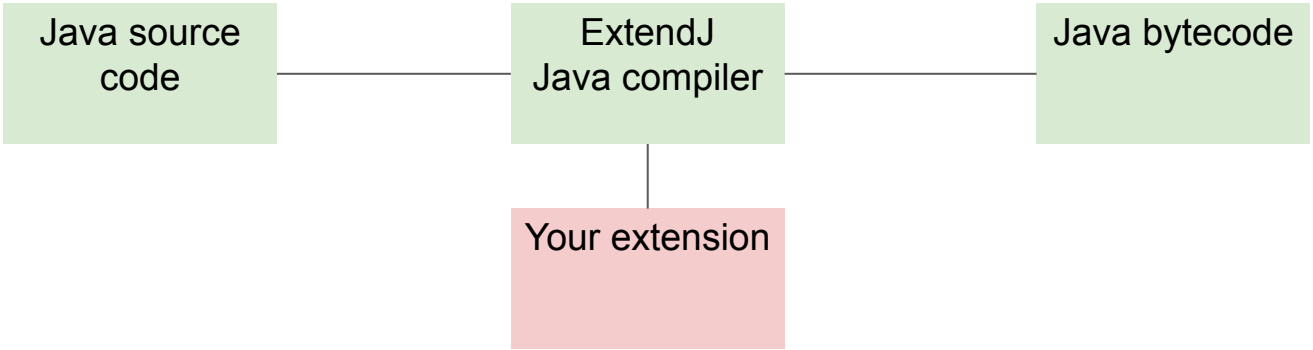
EDAN70, Lund University

Extensions to ExtendJ

Some of the projects will be extensions to the extensible Java compiler called ExtendJ

See:

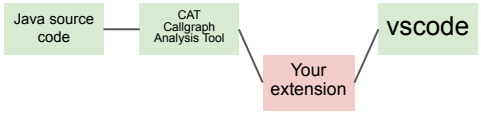
- Website: extendj.org. Look at Getting Started for advice on how to make an experimental small extended compiler.
- Articles:
  - Ekman, Hedin: The JastAdd Extensible Java Compiler, OOPSLA 2007 (doi)
  - Öqvist, Hedin: Extending the JastAdd Extensible Java Compiler to Java 7, PPPJ 2013 (doi)



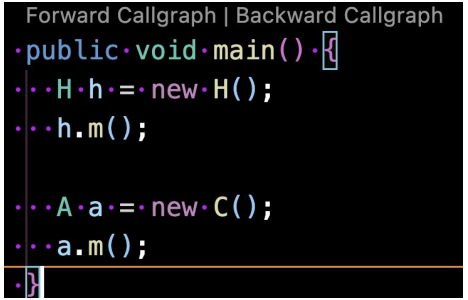
Projects

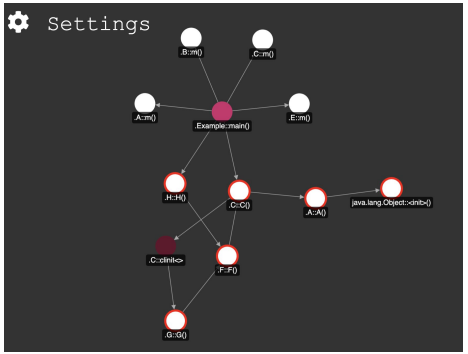
1. Call Graph Analysis Results in LSP
2. Projection Boxes in CodeProber
3. JastAddBridge 0.2 (JastAdd ↔ LSP extension)
4. Points-To Analysis in ExtendJ
5. Declarative Bug Checkers for C in Clog

Call Graph Analysis Results in VS Code



Goal: Build support for displaying call graph analysis results in VS Code. The idea is to build this by reusing CAT, a call graph analysis tool for Java. CAT is built using JastAdd, as an extension to the ExtendJ Java compiler. The connection to VS Code will be done using LSP (Language Server Protocol). As an example usage, the developer should be able to click on a method in VS Code, and get the call graph in rendered in another VS Code tab.





Concrete steps/goals:

- Get hover to work in the editor, displaying the list of directly called methods as hover text.
- Rendering a very simple graph in HTML and display in a separate tab in VS Code.
- Support the graph of directly called methods.
- Extract sourcepath/classpath from Java Language Server extension.

Depending on time, one or more of the following goals:

- Support virtual calls, showing all possible method implementations that may be called.
- Support the full transitive graph, adding support for condensing it in various ways
- Support for the backward graph (display callers of the method instead of callees)
- Explore further VSCode features (e.g., Explorer panel)
- Extract more information from CAT if needed.

Evaluation:

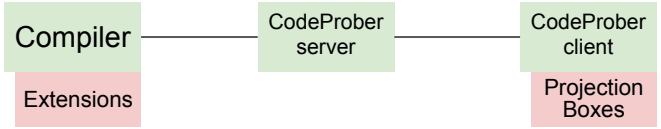
- ***Proof of concept***: Show that the tool works for interesting examples.

Articles and other resources:

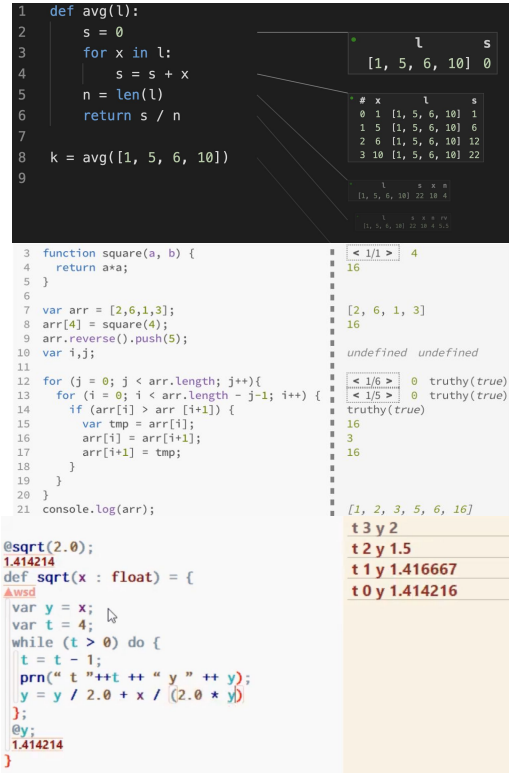
- CAT: Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis
- Call graphs for java

Supervisor: ***Idriss Riouak, idriss.riouak@cs.lth.se***

Projection boxes in CodeProber



Goal: Implement support for alternate presentation styles of diagnostic information in CodeProber. The goal is to produce something similar to **projection boxes**. CodeProber currently supports displaying information as squiggly lines and arrows. Projection boxes enable data such as execution traces to be displayed in a nice way.



Concrete steps/goals

- Extend the Monaco editor used in CodeProber with a custom view that displays values alongside the source code.
- Add attributes that collect relevant information in at least **two** compilers/analyzers. For example:
  - Collect variable values from the interpreter in a SimpliC implementation (lab 5 in EDAN65).
  - Collect all compile-time constant values from a method in ExtendJ
    - (The *constant* attribute is already implemented, you just need to collect the values)
- Connect the collected data and the Monaco extension together inside CodeProber.
- *Stretch goal*: turn this into a more generic extension to the **tracing** system in JastAdd. Imagine inspecting “IdUse.lookup”, and seeing the traced intermediate steps displayed right next to the corresponding source code. That could be incredibly helpful!

Evaluation:

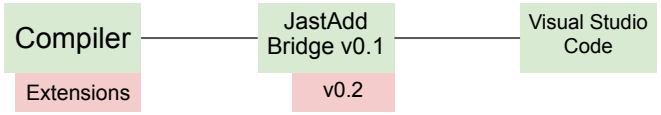
- User study, to determine usefulness (target audience is e.g students that took the compiler course)
- Performance measurements (does the trace collection/visualisation work well for larger programs?)

Articles and other resources:

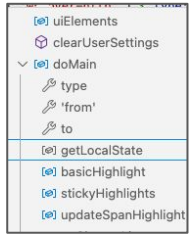
- Projection boxes demo: https://proj-boxes.goto.ucsd.edu/
- Publication that introduced projection boxes: https://doi.org/10.1145/3313831.3376494
- CodeProber: https://codeprober.org

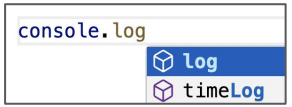
Supervisor: ***Anton Risberg Alaküla***

JastAddBridge 0.2



Goal: Extend JastAdd Bridge, an IDE extension developed by students from last years project course. The extension connects to a jar file (often “compiler.jar”) and adapts certain attributes into LSP interactions. This enables JastAdd tools to provide hover information, go to definition, etc in Visual Studio Code, without having to write an extension of their own. The current implementation only supports a small subset of LSP, and does not handle multiple files. This project revolves around extending and improving the current implementation.





Concrete steps/goals

- Add **at least** three new LSP interactions to JastAdd Bridge. For example:
  - Semantic highlighting, Document Symbol provider (“outline”) and Code completion (“autocomplete”)
- Implement the added interactions in at least two compilers to prove that it is reusable. Suggested compilers:
  - SimpliC (from EDAN65)
  - ExtendJ (https://extendj.org/)
- Implement a way for handling multiple files. Implement an interaction in ExtendJ that uses multiple files, for example “go to definition” for a variable/method that is defined in a separate file.

Evaluation:

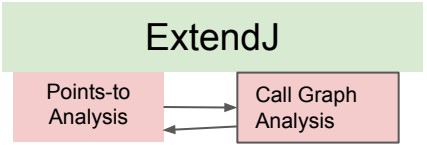
- Performance (does the interaction scale well to larger source files/projects?)
- Functional (does it correctly handle multiple files? Does it handle updating the underlying tool file on recompiles, like CodeProber does?)

Articles and other resources:

- JastAdd Bridge report from last year: https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2023/Reports/hardt-hemberg.pdf
- Semantic highlighting project from 2021: https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2021/Reports/ringstrom-frisk.pdf
- CodeProber: https://codeprober.org

Supervisor: ***Anton Risberg Alaküla***

Points-To Analyses in ExtendJ



Goal: A *points-to analysis* is a program analysis that approximates what possible “memory locations” (objects, in Java) a reference variable may point to. This information can greatly improve the precision of bug detectors and security analyses. In an object-oriented language like Java, such an analysis additionally requires a *call graph analysis*, to resolve dynamic dispatch, which in turn depends on the points-to analysis.

Concrete steps/goals

- Implement Steensgaard’s points-to analysis in ExtendJ, using circular reference attributes that represent the abstract memory locations as AST nodes, ignoring method calls
  - Alternatively implement Andersen’s analysis
- Implement Rapid Type Analysis (the “optimistic” variant used by Bacon and Sweeney) to resolve dynamic dispatch, using your points-to information to determine the possible dynamic type of receiver objects like `a` in `a.foo()`
- Fully handle method calls in your points-to analysis

Evaluation:

- Execution time performance, compared to existing tools (SootUp)
- Precision and recall, compared to existing tools

Articles and other resources:

- Anders Møller and Michael I. Schwartzbach: Static Program Analysis
- Rapid Type Analysis on the EDAP15/2022 slides
- The SootUp library

Supervisor: ***Christoph Reichenbach***

```
class A {
    String foo() {
        return null;
}   }

class B extends A {
    String foo() {
        return "foo"; // [S0]
}   }

int f() {
    A a = new B(); // [B0]

    // points-to:  a ↦ [B0]
    // call graph: B::foo()
    // points-to:  s ↦ [S0]
    String s = a.foo();

    // Not a null pointer
    return a.length();
}
```

Declarative Bug Checkers for C in Clog

Goal: Clog is a declarative logic programming language with pattern matching support for C code that utilises the Clang C/C++ compiler frontend, intended to make it easy to build custom static program analyses. We want to understand how easy it is to write custom checkers in Clog, compared to the established “Coccinelle” tool, used in the Linux kernel.

Concrete steps/goals

- Select a subset of the Coccinelle bug checkers and a set of C benchmark programs on which those checkers are able to detect bugs
- Re-implement the Coccinelle checkers in Clog
- Optionally develop your own checkers in both languages
- Compare the two sets of bug checkers and the two specification languages

Evaluation:

- Treating your implementations as case studies, reflect on challenges observed while implementing the bug checkers
- Examine precision and recall of checkers built in Clog vs existing Coccinelle checkers

Articles and other resources:

- Clog: A Declarative Language for C Static Code Checkers [project on github]
- Coccinelle home page and article on bug finding
- Juliet test suite of C/C++ programs with security vulnerabilities

Supervisor: ***Christoph Reichenbach***

```
FunctionWithGoto($func, g, l) :-
                 <: $type $func(..) { .. } :>,
                 @$func g <: goto $label; :>,
                 @$func l <: $label : $substmt :>.

BackwardGoto(g, l) :-
                 g <: goto $label; :>,
                 l <: $label : $substmt :>,
                 c_src_line_start(g) > c_src_line_start(l).

LabeledReturn(l) :-
                 l <: $label : return $v;  :>.

LabeledReturn(l) :-
                 l <: $label : return ; :>.

WarnBackwardGoto(file, line) :-
                BackwardGoto(g, _),
                file = io_relative_path(c_src_file(g)),
                line = c_src_line_start(g).
```

Project name

TODO:
Architecture here

Goal: …

Concrete steps/goals

- …
-

Evaluation:

- …
- …

Articles and other resources:

- ...

Supervisor: ***…***