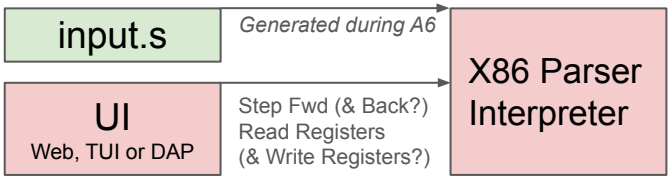
Compiler projects 2025

EDAN70, Lund University

Projects

1. x86 Visual Debugger and Emulator
2. MLIR generator for SimpliC
3. Declarative Bug Checkers for C in Clog
4. Emacs Lisp Reference Attribute Grammars
5. Data flow Analysis on the GPU
6. Add constructs to ExtendJ
7. Java Bug Finding
8. Extend SimpliC with more constructs

x86 Visual Debugger and Emulator



Goal: Build a visual emulator and debugger for x86 assembly code, supporting the instructions used in the EDAN65 Compilers course. The tool should support step-wise execution and viewing the contents of memory and registers. Potentially also breakpoints and/or more advanced concepts like reverse execution. Ideally this would be made as a 100% web-based tool to simplify distribution (this tool may be deployed in next years EDAN65 instance). Depending on experience with web development, other architectures are possible, for example using JastAdd and a text-based UI ("TUI").

Concrete steps/goals

- Implement a parser for a few x86 instructions, address modes, and registers.
- Implement an interpreter for the instructions.
- Implement a UI. Ideally web-based, but can also build a TUI, or use the Debug Adapter Protocol (DAP).

Evaluation:

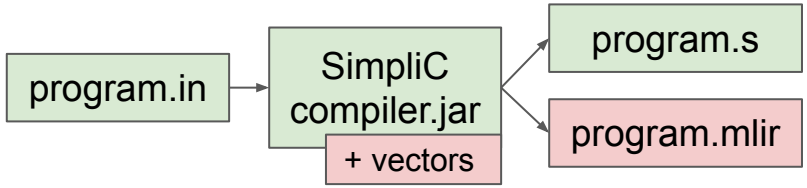
- Functional completeness for a given subset of instructions. Systematic test suite.
- Comparison to other assembly debuggers
- Small scale user study with students that recently took EDAN65.

Articles and other resources:

- Find articles on assembly debugging tools for students on Google Scholar
- A study on the use of CodeProber in EDAP15 (program analysis): https://programming-journal.org/2025/10/10/
- Lezer (a parser generator for JavaScript): https://lezer.codemirror.net/
- Debug Adapter Protocol: https://microsoft.github.io/debug-adapter-protocol/

Supervisor: ***Anton Risberg Alaküla***

MLIR generation for SimpliC



Goal: Generate MLIR intermediate code instead of x86 for SimpliC. Also extend extend SimpliC with vectors and perform some interesting optimization on the MLIR code.

Concrete steps/goals

- Extend your SimpliC implementation with vectors. Make them stack allocated to keep things simple.
- Get a (C++|Python) -> MLIR -> LLVM -> SomeArchitecture pipeline running, so that some tiny program can be run.
- Generate MLIR code from SimpliC to hook into this pipeline.
- Add a custom optimization pass to MLIR.
- If time permits:
  - Add more features to SimpliC. For example structs, heap allocation (+ pointers), 2d vectors, ….
  - Try to get your code running on non-CPU hardware. For example, use MLIR's gpu dialect to run your code on the GPU. It will probably be inefficient, but interesting.

Evaluation:

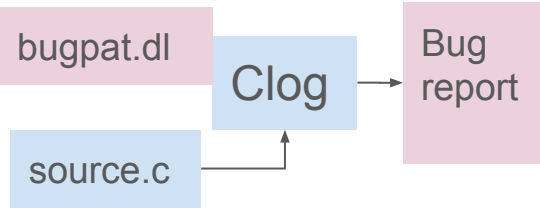
- Measure improvement your optimization pass brings. See if you can beat "normal" `gcc -O2 program.c` for some specific input file.
- Compare performance of existing SimpliC X86 pipeline, with new MLIR pipeline
- Compare performance of running on different hardware types (CPU/GPU/TPU/…).

Articles and other resources:

- MLIR: https://mlir.llvm.org/
- SSA form, articles on MLIR

Supervisor: ***Anton Risberg Alaküla***

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).
```

Emacs Lisp Reference Attribute Grammars

Goal: Implement Reference Attribute Grammars on top of Emacs Lisp with demand-driven evaluation and caching.

Prerequisite:
- **Prior Emacs Lisp programming experience**

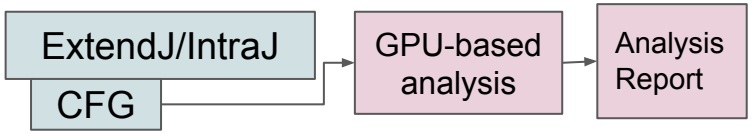
Concrete steps/goals
- Select some tree-structured target demonstrator, such as org-mode elements or tree-sitter AST nodes
- Introduce a macro to enable (cached) synthesised attribute construction
- Introduce a macro to define sufficient structure to allow tree traversal for inherited attributes
- Add inherited attributes
- Add circular attributes

Evaluation:
- Demonstrate with a suitable application (e.g., name analysis on tree-sitter AST nodes)
- Evaluate execution time against existing mechanisms (e.g., LSP)

Articles and other resources:
- `(info "(elisp)Parsing Program Source")`
- `(info "(org)")`
- `(describe-function 'defmacro)`
- `(describe-function 'cl-defmethod)`

Supervisor: tbd or ***Christoph Reichenbach***

Data flow Analysis on the GPU



Goal: Implement and benchmark data flow analyses on the GPU and compare against execution times of Java-based analyses.

Prerequisite:

- Familiarity with GPU kernel programming or other data-parallel programming

Concrete steps/goals

- Familiarise yourself with `taichi` or a similar DSL for building and orchestrating GPU kernel computation
- Manually encode a points-to analysis in a GPU kernel to validate
- Extract control-flow graph information and transfer functions from existing IntraJ/ExtendJ-based points-to analysis
- Integrate extracted information into your algorithm and validate
- Iterate your implementation to optimise: can you increase data parallelism?

Evaluation:

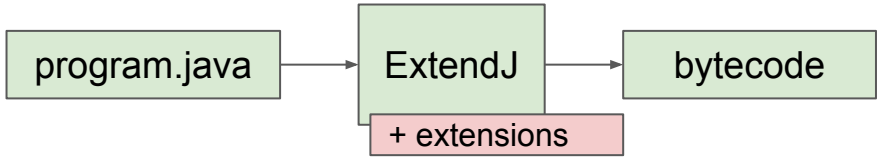
- Measure execution time performance and scalability vs. existing Java implementation

Articles and other resources:

- Efficient Demand Evaluation of Fixed Point Attributes using Static Analysis *(IntraJ/ExtendJ)*
- Taichi DSL (Python-hosted); home page

Supervisor: ***Christoph Reichenbach***

Add constructs to ExtendJ



Goal: The ExtendJ Java compiler currently supports most of Java 11. Newer Java versions have introduced a number of interesting constructs, including switch expressions, records, pattern matching, simple main methods, and compact source files. The goal is to extend ExtendJ with a simple version of one or more of these new constructs.

Concrete steps/goals

- Get ExtendJ running with a minimal example extension.
- Construct example programs and gradually implement support for them.

Evaluation:

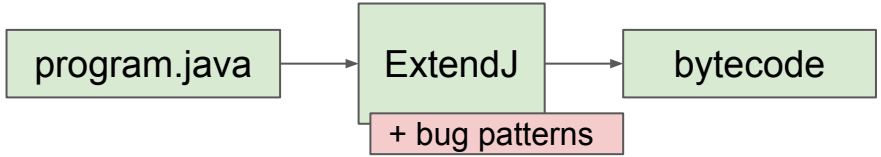
- Demonstrate on example programs.
- Add tests for the new features. Run existing regression test.
- Evaluate SLOC for extension. Could the extension be done modularly?

Articles and other resources:

- ExtendJ articles
- Relevant JEPs (see https://en.wikipedia.org/wiki/Java_version_history for an overview)

Supervisor: ***TBA***

Java Bug Finding



Goal: Add bug pattern detection to Java by extending the ExtendJ compiler

Concrete steps/goals

- Get ExtendJ running with a minimal example extension.
- Investigate other bug finders, e.g., ErrorProne, SpotBugs, and SonarQube to find examples of bugs to check for.
- Add a simple check to ExtendJ, e.g., flag calling toString() on an array as a warning.
- Select more bug patterns and implement them

Evaluation:

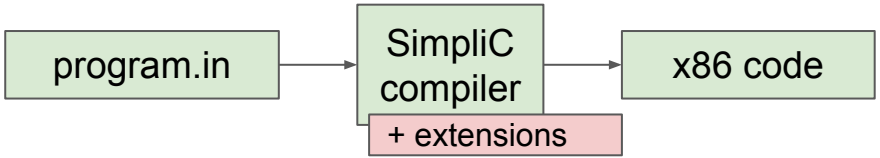
- Run on larger code samples and compare output to e.g, Error Prone, SpotBugs, and/or SonarQube
- Compare pattern detection implementation size/quality.

Articles and other resources:

- E. Aftandilian, et. al. Building Useful Program Analysis Tools Using an Extensible Java Compiler
- https://errorprone.info/bugpatterns
- Papers on ExtendJ

Supervisor: ***TBA***

Extend SimpliC with more constructs



Goal: Modular extension of your SimpliC with more language constructs, e.g., some of: floats, booleans, nested functions, arrays, structs, for loop, etc.

Concrete steps/goals

- Identify a simple construct to extend with, e.g., a for loop.
- Add more constructs gradually.

Evaluation:

- Demonstrate on example programs.
- Add tests for the new features. Run existing regression test.
- Evaluate SLOC for extension. Could the extension be done modularly?

Articles and other resources:

- JastAdd and ExtendJ articles.
- Other modular AG systems, e.g., Silver.

Supervisor: ***TBA***

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)