Project descriptions
compiler projects

This document describes available compiler projects for the course Project in Computer Science (EDAN70). All projects include constructing some artifact, evaluating it, and relating to some literature. All projects are related to ongoing teaching and research at the department. All artifacts will be open-sourced under the modified BSD license.

Projects

1. Support for CUP parser generator for JastAdd projects
2. Layout-sensitive parsing
3. Type inference for Dart
4. Type inference for Go
5. Static analysis: Bug detection
6. Static analysis: Dead method detection
7. Lab language compiler in Scala and Kiama
8. Xtext language-based editor
9. Spoofax language-based editor
10. Extending SimpliC with an LLVM backend
11. Extending SimpliC with a Java bytecode backend
12. JastAdd frontend to Java bytecode
1 Support for CUP parser generator for JastAdd projects

JastAddParser is a preprocessor for the Beaver parser generator, providing a slightly higher-level parser specification, suited for JastAdd. In particular, JastAddParser supports that a parser specification can be split into several modules, whereas Beaver supports only a single specification file. JastAddParser is used in many JastAdd projects, including the ExtendJ Java compiler and the JModelica.org compiler for Modelica and Optimica. JastAddParser is itself implemented using JustAdd (and JastAddParser).

CUP is an LALR(1) parser generator that is currently better maintained than Beaver. It might be beneficial to use CUP instead of Beaver as the backend for JastAddParser, in particular if the parsing speed of CUP turns out to be higher than that of Beaver.

The task is to implement a new backend to JastAddParser, generating specification files for CUP instead of for Beaver. The resulting parser should be evaluated on the test suite for JastAddParser, and obstacles to implementing a full CUP backend should be identified. If relevant, the test suite should be extended. A comparison should be done on the parsing speed of the generated CUP and Beaver parsers.

Preferably, the implementation should be done in such a way that both the CUP and Beaver backends can be easily maintained, and allowing new backends for other parser generators to be added.

If the resulting backend is sufficiently powerful, and if time permits, it would be very interesting to do an evaluation on the JastAddJ and JModelica parsers.

Resources:

- The JastAddParser project: [https://bitbucket.org/jastadd/jastaddparser](https://bitbucket.org/jastadd/jastaddparser)
- The Beaver project: [http://beaver.sourceforge.net](http://beaver.sourceforge.net)
- The CUP project: [http://www2.cs.tum.edu/projects/cup/](http://www2.cs.tum.edu/projects/cup/)
2 Layout-sensitive parsing

Many languages are layout-sensitive, i.e., they include whitespace and indentation in the syntax. Examples include Python and Markdown. There are several approaches to layout-sensitive parsing reported in the literature. The goal is to find out which techniques and tools that exist, and to try out the tools on example grammars. Subsets of Python and Markdown are suitable test languages.

Compare performance and specification size. Describe the techniques used and how they can be combined with normal parsing techniques based on context-free grammars.

Resources:

- S. Erdweg et al. Layout-Sensitive Generalized Parsing. SLE 2013
3 Type inference for Dart

Dart is a new language designed for programming the web. It has an optional type system: the programmer can leave out types, in which case the type checking is done dynamically. The programmer can also add types which are then checked statically.

It would be useful to infer types in Dart, so that some types can be computed automatically by the compiler.

The task is to implement a subset of Dart, using JastAdd, and to infer types.

The evaluation is primarily qualitative, showing examples of useful inference, and showing how inference can be implemented using reference attribute grammars.

Resources:

- Dart Language Specification [https://www.dartlang.org/docs/spec/]
- Wikipedia on type inference

4 Type inference for Go

Go is a new systems programming language. It has a type system where variables do not need to explicitly be declared with a type. The compiler instead infers the type, i.e., computes the type based on how the type is used.

The task is to implement a subset of Go, using JastAdd, including type inference.

The evaluation is primarily qualitative, showing examples of useful inference, and showing how inference can be implemented using reference attribute grammars.

Resources:

- The Go Programming Language [https://golang.org]
- Wikipedia on type inference
5 Static analysis: Bug detection

Bug detection tools analyze the source code to detect common bugs. Examples include FindBugs and ErrorProne. FindBugs analyzes Java bytecode. ErrorProne uses the javac compiler API to implement bug detectors.

The goal of the project is to reimplement a number of the bug detectors from ErrorProne by instead using JastAdd attribute grammars, on top of the ExtendJ Java compiler. It is interesting to find out how easy it is to implement these bug detectors, and how fast the analysis is. On the one hand, ExtendJ is slower than javac if you do a complete compilation of a program. On the other hand, ExtendJ might be faster than javac due to the demand analysis used in ExtendJ.

To evaluate the approach, the performance and specification size should be compared with that in ErrorProne.

If time permits, the new bug detector can be integrated into Google's new code review integration tool ShipShape (an open-source version of their in-house Tricorder tool).

Resources:

6 Static analysis: Dead method detection

Often, a Java program is packaged as a jar file, including all libraries used by the program. However, this will include also unused methods, resulting in a larger jar file than necessary. The goal of this project is to identify unused methods (dead methods), as seen from the main method. If the unused methods are known, a smaller "shrink-wrapped" jar-file could be produced. Such a computation can be done provided that Java reflection is not used, and the whole set of classes is known.

The goal of this project is to implement a detector of dead Java methods, by building a static analyzer on top of ExtendJ (the JastAdd Extensible Java Compiler). The static analyzer should use a call graph component, that takes virtual methods into account, i.e., the fact that a method call can go to a method implementation in a subclass. Only classes reachable from the main method should be included in the call graph. Other methods are dead.

This call graph computation corresponds to an algorithm called Rapid Type Analysis, originally formulated for C++, using an iterative algorithm with a worklist. In a JastAdd implementation, the algorithm will instead be implemented using circular attributes.

The project can be evaluated in several ways:

- compare the JastAdd formulation with the original Rapid Type Analysis algorithm, comparing performance and specification size.
- qualitative evaluation, showing how the algorithm can be formulated using attributes rather than imperatively, and discussing the differences
- use the algorithm to run on Java applications to see how much dead code they contain, i.e., how much would be gained by shrinkwrapping them.

Resources:

7 Lab language compiler in Scala and Kiama

Scala is an advanced programming language combining support for object-oriented and functional programming. Kiama is a Scala library for attribute grammars. It implements many of the constructs in JastAdd, including reference attributes, parameterized attributes, and circular attributes. However, collection attributes are not supported. For parsing, Kiama normally uses a Scala parser combinator library. Kiama also has support for AST transformations, and for abstract state machines.

The task is to implement the lab language in Kiama, making it pass the same test cases as your own lab implementation in JastAdd. To evaluate the approach, the implementations should be compared with respect to specification size and performance. If time permits, both implementations can be extended with more language constructs.

Resources:

- The Kiama library: https://bitbucket.org/inkytonik/kiama
8 Xtext language-based editor

Xtext is a framework for developing interactive language-based Eclipse editors, i.e., editors with syntax coloring, content assist, etc. Xtext uses the Eclipse Modeling Framework (EMF) for internal representation of ASTs. Xtext also has an API for solving common compilation problems, but which is not as general as attribute grammars.

The goal is to implement an Xtext editor for a small language. The editor should provide syntax coloring, and semantic support like content assist and compile-time error checking.

The small language can be either SimpliC, or another small language. A nice thing would be to create an editor for the .ast language (JastAdd’s abstract grammars), since this could become a useful tool for JastAdd programmers. The tool should preferably handle several .ast modules, since many JastAdd projects use multiple .ast files.

The evaluation is primarily qualitative, providing a proof-of-concept solution that shows that it is possible to construct an editor in this way, and discussing pros and cons of the solution. It is possible to do a small think-aloud user study for the editor.

Resources:

- The Xtext project: [http://www.eclipse.org/Xtext](http://www.eclipse.org/Xtext)

If there is time, it would be very interesting to experiment with doing some analysis in the editor by using JavaRAG. JavaRAG is a Java library that supports reference attribute grammars on any Java-based AST. (It would be difficult to use JastAdd here, since JastAdd and Xtext use different AST representations.) See

- The JavaRAG library: [https://bitbucket.org/javarag/javarag](https://bitbucket.org/javarag/javarag)
9 Spoofax language-based editor

Spoofax is a platform for developing interactive language editors as Eclipse plugins, including syntax highlighting and code completion. Spoofax supports parsing using GLR parsing (generalized LR parsing). Spoofax does not support attribute grammars, but instead uses a special name binding language (NaBL) for doing name analysis, and rewrite rules using the Stratego language, for other semantic analyses and code generation.

The task is to implement a Spoofax editor for a small language. The editor should provide syntax coloring, and semantic support like content assist and compile-time error checking.

The small language can be either SimpliC, or another small language. A nice thing would be to create an editor for the .ast language (JastAdd’s abstract grammars), since this could become a useful tool for JastAdd programmers. The tool should preferably handle several .ast modules, since many JastAdd projects use multiple .ast files.

The evaluation is primarily qualitative, providing a proof-of-concept solution that shows that it is possible to construct an editor in this way, and discussing pros and cons of the solution. It is possible to do a small think-aloud user study for the editor.

Resources:

- The spoofax platform: [http://strategoxt.org/Spoofax](http://strategoxt.org/Spoofax)

If there is time, it would be very interesting to experiment with doing some analysis in the editor by using JavaRAG. JavaRAG is a Java library that supports reference attribute grammars on any Java-based AST. (It would be difficult to use JastAdd here, since JastAdd and Xtext use different AST representations.) See

- The JavaRAG library: [https://bitbucket.org/javarag/javarag](https://bitbucket.org/javarag/javarag)
10 Extending SimpliC with an LLVM backend

LLVM is a compiler infrastructure project for programs written in arbitrary languages. Despite its acronym, it is not a virtual machine, but rather an optimization framework. It defines a low-level assembly-like language, but that is platform independent.

The task is to implement a new backend to the lab language (SimpliC), that generates code for LLVM instead of for x86, and extending the language to cover more constructs, for example arrays or structs. Performance measurements should be made, comparing the x86 implementation with the LLVM implementation, using LLVM tools to compile the LLVM IR code to optimized x86.

LLVM resources:

- LLVM site: [http://llvm.org](http://llvm.org)

11 Extending SimpliC with a Java bytecode backend

The task is to implement a new backend to the lab language (SimpliC), that generates Java bytecode instead of x86, and extending the language to cover more constructs, for example arrays or structs. Performance measurements should be made, comparing the x86 implementation with the JVM implementation.

Java Bytecode resources:

12 JastAdd frontend to Java bytecode

Many analyses are simpler to do on bytecode than on source code, and by doing the analysis on bytecode, the same analyzer can be applied to all languages that compile to the bytecode. JastAdd has powerful support for static analysis, so for this reason, it would be nice to have a JastAdd frontend to Java bytecode, and which can then be modularly extended with various analyses.

The goal of the project is to implement such a frontend, and to implement some very simple example bytecode analyses. Inspiration for what analyses to implement can be taken from FindBugs, which supports bytecode analysis.

The project can be evaluated by comparing it to other Java bytecode tools and libraries such as BCEL, ASM, and Javassist. Performance and ease of use for analysis are interesting to evaluate.

Resources:

- BCEL
- ASM
- Javassist
- Java Byte Code