Contributions to Declarative Implementation of Static Program Analysis

Öqvist, Jesper

2018

Document Version:
Publisher's PDF, also known as Version of record

Contributions to Declarative Implementation of Static Program Analysis

Jesper Öqvist

Doctoral Dissertation, 2019

Department of Computer Science
Lund University

ISBN 978-91-7753-944-5 (printed)
ISBN 978-91-7753-945-2 (electronic)
ISSN 1404-1219
Dissertation 61, 2019
LU-CS-DISS: 2019-01

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: jesper.oqvist@cs.lth.se

Typeset using LaTeX.
Printed in Sweden by Tryckeriet i E-huset, Lund, 2019.

Abstract

Programming languages are ever evolving, with new languages being invented to solve new problems, and old languages being extended to solve old problems in new ways. With the continued evolution of programming languages, and with new and improved static program analyses, we need flexible systems for building our static analyses and compilers. Declarative programming with *Reference Attribute Grammars* (RAGs) is a fruitful approach for building extensible and maintainable static program analyses and compilers. With declarative programming, code is easier to write, easier to understand, and easier to extend.

This thesis presents contributions to declarative static program analysis implementation with RAGs. In particular, I have developed new language extensions for the Java programming language, as well as new static analyses and tools for Java, based on the extensible Java compiler ExtendJ. Language extensions include an implementation of Java 7, and a new programming mechanism, *multiplicities*. A new static analysis-based tool presented in this thesis is a regression test selection tool, which reduces testing time for Java software development.

My contributions also include new design patterns for declaratively specifying static analyses in RAGs, and significant improvements to the ExtendJ compiler. My work in ExtendJ includes developing the first version of the compiler with fully declarative static analysis.

This dissertation also includes contributions to RAGs, including new algorithms for concurrent attribute evaluation with implementation in the JastAdd metacompiler. With these new algorithms, it is possible to parallelize any RAG-implemented analysis. In particular, I parallelized static analysis in the ExtendJ compiler to achieve a twofold speedup and orders of magnitude reduction of latency.

Acknowledgements

This work was in part funded by the Swedish Research Council under grant 621-2012-4727 and by a 2015 Google Faculty Research Award for supporting concurrent analyses in interactive programming tools.

This dissertation would not exist if not for Prof. Görel Hedin. Görel convinced me to start my PhD studies and supervised me along the way. She told me what to do, corrected my mistakes, and listened to my weird ideas. I owe Görel my greatest thanks in helping me complete all of this work.

I owe much gratitude to my beloved wife, Elise. She is the chillest wife, in the parlance of our times. Elise kept me sane and healthy while writing this thesis. I am also very grateful for our many fun travels and outings during my time as Ph.D. student.

Thank you Prof. Boris Magnusson for co-supervising me and for the test selection collaboration.

Thank you Emma Söderberg for co-supervising me, for getting me into running (`RuntimeException`), for hosting me at Google for my first internship, and for many fun collaborations related to JastAdd and other projects during my PhD studies.

Thank you Prof. Friedrich Steimann for the multiplicities collaboration. It was fun to visit your department in Germany and to work intensely to complete most of the implementation in a week.

Thank you Niklas Fors, Alfred Åkesson, and Christoff Bürger for collaborations related to JastAdd development and compiler construction.

For entertaining and enlightening discussions during the time-honored tradition of fika, I would like to thank Christoph Reichenbach, Flavius Gruian, Linus Åkesson, Maj Stenmark, Pierre Moreau, Jonas Skeppstedt, Gustav Cedersjö, Noric Couderc, and all of my other colleagues at the computer science department in Lund, past and present.

Thank you Erik Hogeman for the Java 8 project.

Thank you to all the excellent people whom I had the pleasure to work with at Modelon (Jesper, Jon, Jonathan, Axel) and Google Mountain View (Ciera, Karl, Valentyn, Hans, Eddie). Thank you Arun Chauhan for hosting me during my second internship.

The diagrams in this thesis would not look as nice without a few essential LaTeXpackages. Thus, I owe my gratitude to the authors of the PGF/Ti*k*Z packages, and especially Sašo Živanović for developing the excellent Forest package which I used for drawing all trees in the thesis.

I was considering making a list of enemies and placing Måns Magnusson on it for distracting me with programming problems. However, I finished the thesis on time so I think we can continue to be friends.

Contribution Statement

The following papers are included in this dissertation:

Paper I Jesper Öqvist and Görel Hedin. "Extending the JastAdd Extensible Java Compiler to Java 7". In *Proceedings of the 10th International Conference on Principles and Practicies of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13), ACM, pp. 147–152*. Stuttgart, Germany, 2013.

Paper II Friedrich Steimann, Jesper Öqvist, and Görel Hedin. "Multitudes of Objects: First Implementation and Case Study for Java". In *Journal of Object Technology, Vol. 13, no. 5 (November 2014), pp. 1:1–33*.

Paper III Jesper Öqvist, Görel Hedin, and Boris Magnusson. "Extraction-Based Regression Test Selection" In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'16), ACM, pp. 5:1–5:10*. Lugano, Switzerland, 2016.

Paper IV Jesper Öqvist and Görel Hedin. "Concurrent Circular Reference Attribute Grammars". In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017), pp. 151–162*. Vancouver, BC, Canada, 2017.

This thesis includes an extended version of this paper, with correctness proofs for the algorithms. This extended version of the paper was published as a technical report (report number 103, Lund University, 2017).

The table below indicates the responsibilities Jesper Öqvist had in writing each paper:

Paper	*Writing*	*Concepts*	*Implementation*	*Evaluation*	*Algorithm*
I	YES	YES	YES	YES	N/A
II	partial	no	YES	YES	N/A
III	yes	partial	YES	YES	yes
IV	YES	YES	YES	YES	YES

Capital letters indicate roles where Jesper Öqvist took primary responsibility for the given role. For Paper III, Jesper Öqvist wrote the first drafts of the paper, and then took main responsibility to finish the implementation and evaluation parts. For Paper II, Friedrich Steimann wrote the majority of the paper, with Jesper Öqvist taking responsibility for writing about implementation and evaluation as well as designing some of the figures/diagrams. For all papers, Jesper Öqvist developed the implementation and empirical evaluation. For Paper IV, Jesper Öqvist wrote all proofs, with feedback from Görel Hedin.

Contents

Introduction

INTRODUCTION

1 Introduction

Static Program Analysis is the automated analysis of a computer program before it is executed [MS18; Bin07]. Static program analysis is essential for software development in several ways: it is used for translating programs to executable machine code, uncovering bugs, optimizing program performance, and assisting software developers in constructing and testing programs. Programmers use static analysis via smart code editing tools in modern integrated development environments for efficient software editing and for improving code organization. Tools like declaration lookup and code completion suggestions are used for navigating and writing code, while code refactoring tools are used to improve code structure. In addition, static analysis is an important part of automated software quality assurance and testing at many software companies, like Google [Sad+18] and Facebook [Cal+15].

When developing static program analyses we face similar problems as in other types of software development, including the perennial challenge of building a flexible software architecture that enables future additions after initial development. Static analyses should have a flexible architecture to ease the process of expanding the analysis to support new language features as programming languages evolve and as new analyses are invented. The challenge of developing a flexible architecture can be addressed by *Declarative Programming*.

Declarative programming is a programming paradigm in which the programmer specifies the desired result of a computation rather than exactly enumerating the necessary steps to compute the result. A common view of declarative programming is that the programmer writes *what* should be computed, rather than *how* it is computed. Side effect-free code is an important aspect of declarative programming. Observable side effects are any effect a function call has on the state of the program which can change the result of other function calls [Nau05]. When side effects are present, the ordering of function calls is significant for the result of the program. A key advantage of declarative programming is that the programmer need not carefully order function calls to avoid unwanted side effects, instead they are free to focus on what should be computed.

Declarative programming benefits flexible software architectures by making it easy to break the code up into smaller parts which can be cleanly combined. This makes it possible to have clearly separated computations for separate tasks in the software, making it easier to extend the software with new features, since fewer existing parts of the software will need to be taken into account when making changes or additions. Furthermore, side effect freedom makes it easier to combine different computations, aiding analysis composition and extensibility.

One way of developing static analyses and compilers declaratively is by using *Attribute Grammars* (AGs), a declarative formalism proposed by Knuth [Knu68b] for describing the semantics of programming languages. Although plain AGs *can* be used to declaratively implement static analyses, the task of implementing a full programming language was onerous due to drawbacks like "repetition, overwhelming detail, and the interleaving of many activities" [DC90]. Furthermore, static analyses often rely on computations with information flow along a graph structure, and these are not well suited for classical AGs which work with information flow along a tree structure. *Reference Attribute Grammars* (RAGs) [Hed00] is an extension to plain AGs with attributes that can have reference values. These reference attributes enable us to compute graphs on top of a tree, and allow computations to flow along those graphs, substantially simplifying the task of implementing static analyses. Reference attributes proved very useful in practice for implementing real programming languages (in contrast to most previous attempts with classical AGs which were mostly minimal toy languages). Reference attributes are often very elegant, compactly defining the relations that are being computed. Compilation, and other static analyses, can be implemented with RAGs, leading to elegant declarative implementations which are easy to compose and extend.

Past development efforts have proven the feasibility of building large-scale compilers using RAGs. For example, the *JastAdd* [HM03a] metacompiler supports compiler development with RAGs and has been used to develop large-scale compilers for the Java and Modelica languages. In particular, ExtendJ [EH07b] is a full Java compiler, and JModelica.org [Åke+10] is a compiler for the Modelica modeling language. Both ExtendJ and JModelica.org are developed using JastAdd and using reference attributes for declarative static analysis. *Silver* [Wyk+10a] is another example of a metacompiler for RAGs, which has been used for implementing compilers for Java [Wyk+07], C [Kam+17], and PROMELA [MW11]. RAGs have also been implemented as embedded libraries, including Kiama (a Scala library) [Slo09], RACR (for Racket) [Bür15], and JavaRAG (a Java embedding) [FCH15]. With a few supporting features, like static Aspect-Oriented Programming, and an attribute replacement mechanism, it is possible to extend a compiler implemented with RAGs by adding new attributes [AET08a]. This can be used to implement new static analyses, tools, and language extensions upon an existing compiler, for example adding non-null type checking to a Java compiler [EH07a].

This thesis presents new applications and algorithms for RAGs. Central to the contributions are practical applications and tools implemented in the JastAdd metacompiler and the ExtendJ Java compiler. Specifically, my main contributions are the following:

- An extension of the ExtendJ compiler to support Java 7 (Paper I).
- An extension for ExtendJ with new static analyses and code generation for a new programming mechanism, *Multiplicities*, implemented as a Java language extension (Paper II).
- A new algorithm for regression test selection with implementation based on ExtendJ (Paper III).
- A new concurrent evaluation algorithm for RAGs with support for circular (fixpoint) attributes, implemented in JastAdd and evaluated by parallelizing ExtendJ (Paper IV).

My research shows that RAGs are very effective for building static analyses, programming language extensions, and static-analysis based tools. By using RAGs to develop declarative compiler extensions, I have demonstrated that the declarative nature of RAGs benefits composability and extensibility in static analysis. As part of my thesis work I have made substantial improvements to both ExtendJ and JastAdd. My improvements to ExtendJ have enabled it to support nearly all of Java 8, as well as fixing numerous bugs which prevented many real-world programs from compiling correctly. To verify the improvements to ExtendJ, I developed a large regression test suite for the compiler. The impact of my work is demonstrated by multiple research groups publishing results based on extensions for ExtendJ to support new language features and analyses.

An important improvement to ExtendJ was also to weed out some side effects that had been introduced in the specification in order to speed up compilation. I have managed to replace that code with completely declarative code with only negligible effects on performance. While those side effects were carefully crafted to not have any effect on sequential evaluation, they did not work correctly for parallel evaluation. Complete lack of observable side effects is crucial in order for the parallel evaluation algorithms to work correctly.

1.1 Methodology

The work in this dissertation was carried out using a problem-oriented methodology. I looked at different interesting problems, designed potential solutions, implemented promising solutions to explore what worked, and finally evaluated the effectiveness of the solution. Zobel describes a similar methodology [Zob14, p. 54].

Solutions were implemented using aspects of agile software development [Mar02], like iterative development and continuous testing. For example, I developed an extensive regression test suite for the ExtendJ compiler with about a thousand tests for bugs and features. One bug fix or feature was implemented at a time and the tests were run after each change to the compiler, to ensure there were no regressions.

I evaluated the correctness and performance of each compiler extension, static analysis, and algorithm presented in this thesis by compiling real-world Java programs. The subject programs I used were sourced from the Java open source community, and from curated corpora of Java programs for static analysis and runtime benchmarking, in particular:

DaCapo 2009 A corpus of Java programs for Java runtime benchmarking [Bla+06].

Qualitas Corpus 20130901 A collection of Java programs for static analysis and software studies [Tem+10].

I evaluated the correctness of compiler extensions in Paper I to III by comparing the compilation result to the ground-truth result of the OpenJDK compiler, the reference compiler for Java. For Paper III, I measured the test results of the subject programs to check that all tests that failed were identified by the test selection tool I developed for that paper (in this case, the subject programs were compiled with OpenJDK for test running). For performance evaluation, I measured overall compilation time (in Paper IV, I additionally measured individual attribute evaluation times). For performance evaluations, I used elements of the steady-state performance method described by Georges et al. [GBE07a]. The exact method varied between evaluations, but typically the solution was run on a subject program multiple times while measuring only the running time without Java start-up time. The steady-state performance method aims to eliminate variance between runs caused by the Java runtime system, including garbage collection and runtime code optimizations.

All implementations I developed for the included papers in this dissertation have been published as open source software, freely available online for other researchers to use and extend. Additionally, the implementation and empirical evaluation for Paper IV was accepted by a peer review process known as *artifact evaluation* [KV15], and the artifact itself was published together with the paper.

1.2 Thesis Outline

The rest of this thesis is organized as follows. Section 2 gives an introduction to static program analysis, in particular: applications and implementation concerns. Section 2.3 presents a basic theoretical background to reference attribute grammars, including most of the kinds of attributes supported by the JastAdd metacompiler. An introduction to the JastAdd metacompiler itself is given in Section 2.5.

Section 3 shows how declarative and extensible static analyses can be developed with RAGs using the JastAdd metacompiler. In this section, I also show a new pattern for generic tree traversal with RAGs.

Section 4 describes the ExtendJ compiler: a full declarative and extensible Java compiler which was built using JastAdd.

Section 5 details my contributions in this thesis, including, but not limited to, the contributions in the included papers.

Concluding remarks are in Section 6, and then the included papers follow.

2 Background

This section gives an introduction to static program analysis, reference attribute grammars, and the JastAdd metacompiler. These topics are needed for the presentation of the technical contributions in this dissertation. We will start with a high-level overview of static program analysis, then follows a theoretical background for static

analysis with reference attribute grammars. Finally, we introduce the JastAdd meta-compiler.

2.1 Static Program Analysis

Programs in general-purpose programming languages, like Java or C, are executed to perform some computation based on varying inputs. The computation is *dynamic*: its output may change for each set of inputs. The goal of static program analysis, or just *static analysis* for short, is to answer questions about *static* properties of programs, that is, properties that hold for all possible inputs.

Let us consider a question we can ask about a program: Can the program get stuck forever, or will it always halt? This question is about a static property of programs, so can we answer it with static analysis? Clearly, the question is easy to answer for *some* programs. For instance, this program is obviously non-halting:

```
while (true) { }
```

Unfortunately, it turns out to be a computationally intractable problem to answer this question for all possible programs. The question is a variant of the famous halting problem which was proven undecidable in the general case by Turing [Tur37].

Although many interesting static analysis questions are undecidable in the general case, we can often develop useful static analyses by limiting the scope of the question or making the analysis conservative such that it gives a definite answer only for a subset of all possible programs (some programs halt, some are definitely non-halting for certain inputs, others we don't know).

Static analysis originates from compiler construction. A compiler is a program that translates a computer program in source form (usually text) to an executable form (like machine code). We can view compilers as a static analysis that answers the question: What is the machine code translation of this program? Although this definition means that a compiler is just a type of static analysis, we normally reason about compilers not as a single analysis, but as several distinct and interdependent analyses.

With the advent of optimizing compilers, several new types of static analyses were developed for optimizing program performance. Performance optimizing analyses are often focused around removing redundant computations from a program. In more recent years, static analysis has also become the collective term for tools that analyze program correctness, often in integrated development tools and continuous integration systems. This dissertation focuses more on the latter form of analyses, as well as the original compiler-oriented static analyses (that is, the non-optimizing kinds). The next section gives an overview of these kinds of analyses.

Methods of Static Analysis

There are many different kinds of static analysis, with different applications. Compilers use static analysis to check the source program for errors and for generating executable code. Integrated development environments provide tools to the programmer based on static analysis, like refactoring tools, code navigation tools, and code

completion tools. After the program code has been edited by a programmer, it is often checked for problems like code duplication [RCK09; NR15] on continuous integration servers [DMG07; Sad+18]. Analyses that produce metrics about class encapsulation and complexity can help guide refactoring efforts to improve existing code in object-oriented languages [CK94; BBM96].

The field of static program analysis is diverse, with many different kinds of static analysis. The kinds that are discussed in this dissertation can be grouped into the following categories:

- Name Analysis,
- Type Analysis,
- Control Flow Analysis,
- Dataflow Analysis.

These are broad categories encompassing multiple different static analyses. In the following sections we will look at examples of typical analyses from each category.

Name Analysis

The goal of name analysis is to determine the meaning of variable names, function names, and type names in a program.[1] Name analysis is required for all other forms of useful program analysis: if we can not figure out the meaning of variable (or function) names in a program we cannot figure out anything else useful about it.

Consider the code fragment:

```
int a = -1;
int fn() {
  int a = 2;
  return a+1;
}
```

The expression `a+1` refers to some variable `a`. However, we can not be certain which one the programmer meant since there are two declarations for a variable named `a`. The expression value is either 3 or 0, depending on which declaration is used. A reasonable programming language specification disambiguates situations like these, and the compiler uses name analysis to implement the right behaviour. Name analysis is also used by the compiler to assign memory locations to variables: memory locations are assigned per declaration and each variable use refers to the memory location its declaration was assigned.

Java uses another type of name analysis, called *syntactic classification* [JLS7, §6.5.1], to determine what name scope the names in a program belong to. For example, in a qualified expressions like `a.b`, the `b` refers to a field in some class, but

[1] Interpreted languages can use environments at runtime to avoid performing full name analysis before running a program. Thus, interpreted languages can rely on dynamic name analysis rather than static name analysis.

it could be a static field or an instance field, depending on whether or not a is a class name in the current context. Syntactic classification disambiguates the expression by marking each name with the correct name kind.

Type Analysis

Types are used in programming for categorizing the objects of the program and controlling which operations are permitted on them. In Java, for example, the following statement is disallowed because it is ill-typed:

```
int a = "hi";
```

Type analysis is a broad category of analyses concerned with enforcing the typing rules of programming languages, or for providing additional type-based analysis on top of the rules of the language. Type analysis normally consists of the following steps:

Type Lookup Determine the type of all typenames in the program. Typenames are used, for example, in variable declarations and class instance expressions.

Type Analysis Assign result types for all expressions in the program. For some expressions the result type may not be specified by the language: they are marked as ill-typed.

Type Checking Check that the operands of each expression and statement are compatible according to the typing rules of the language. Report type errors for all ill-typed expressions.

Type Inference Reconstruct types for partially typed expressions, saving work for the programmer.

The first step, type lookup, often piggybacks on the name analysis in a compiler. The second step, type analysis, may produce some type errors. Type errors found during type analysis can either be reported directly or handled later in type checking. In the latter case the type analysis will assign some kind of wrong-type to an ill-typed expression, that is, a type that is only used for marking incorrectly typed expressions.

In languages using parametric polymorphism, many aspects of type analysis become much more involved. Type lookup for polymorphic typenames requires more work since these typenames can include other typenames as arguments. Another implementation issue for nominally typed[2] languages, like Java, with polymorphic types is to assign identities to different parameterizations of a single type.

A common feature in languages with parametric polymorphism is *Type Inference*. Type inference is intended to save the programmer the tedious work of typing out the full type for all variables and functions when the compiler could instead figure out the right type. For example, in Java 6 programs, statements similar to the following one are not uncommon:

[2]Nominally typed means that types are compared by name, rather than by structure. For an introduction to nominal and structural typing, see Pierce [Pie02, p. 251-254].

```
List<Map<Integer,String>> maps
    = new ArrayList<new HashMap<Integer,String>>();
```

There is some redundancy in this statement: a human familiar with Java 6 could easily deduce the meaning without the type on the first line or, alternatively, the type parameter to `ArrayList`. In Java 7, the programmer may instead utilize type inference and write just

```
List<Map<Integer,String>> maps = new ArrayList<>();
```

Similarly, the C++ language adopted type inference in C++11. The `auto` keyword from C++11 can be used to simplify a variable declaration like this

```
std::list<int>::iterator it = list.begin();
```

into this

```
auto it = list.begin();
```

As demonstrated above, type inference clearly saves some typing for the programmer. This in itself is often a motivation for the use of type inference. It could also be argued that reduced code size makes the code easier to read, with secondary effects of making the code easier to understand and modify for third parties. In the first case, the inferred type is obvious, since the type is repeated on the same line. In the second case, however, the inferred type depends on whatever the type of `list` is, possibly making the meaning less clear at a glance.

Control Flow Analysis

Control flow analysis is used for analyzing the possible orderings of expressions and statements in a program during execution. Each possible ordering is called a control-flow path, and these paths are represented implicitly by control flow graphs.

Control flow analysis can be used, for example, to determine if a piece of code is unreachable, or if all paths through a function have a return statement.

An example of control flow analysis in Java compilers is the exception handling analysis. This analysis ensures that all control flow paths where an exception can be raised, for certain types of exceptions, encounter a corresponding exception handler. Thus the exception handling analysis ensures that a program never gets interrupted in an unexpected way by certain exceptions.[3]

Control flow analysis normally refers to the analysis of control flow within a single procedure. Inter-procedural control flow analysis is a related, but quite different, topic. Rapid Type Analysis is an example of an inter-procedural analysis for object-oriented languages [BS96].

Dataflow Analysis

Dataflow analysis builds upon control flow analysis to find the possible ways data can flow through a program. Dataflow analysis gathers all possible control flow paths

[3]The exception (excuse the pun) to this rule is that non-exception throwable objects, and exceptions the programmer explicitly allowed to be thrown, may still interrupt the program.

and conservatively approximates the state of variables at all points in the program. For example, dataflow analysis can determine if a variable can ever contain a null reference at a certain point in the program.

Definite assignment analysis is a kind of dataflow analysis, in which we analyze variable uses to determine if the variable has definitely been assigned before that use. Definite assignment analysis can catch uninitialized variable errors, a well-known weakness of the C programming language.

Consider the following fragment of code:

```
void f() {
  int a;
  int b = a+1;
}
```

Here, variable a is used without being initialized. In the C programming language, this results in an undefined value assigned to variable b. Most C compilers allow this to compile, often without warning, which has led to a number of bugs in real-world programs.[4] In the worst case these bugs might not be caught in code review or testing and make it into a software release. As of May 2018 there were 27 documented vulnerabilities with descriptions explicitly mentioning "uninitialized variable" or "uninitialized stack variable" in the Common Vulnerabilities and Exposures database [CVE].

Program Models

As static analyses vary in domain and application, so do their implementations. However, all static analyses have in common a need for some kind of program model. Program models represent, in a structured form, the features of the program which are of interest for the analysis at hand. The program model used for control flow and dataflow analysis, for example, is the control flow graph.

In this dissertation we will mainly consider static analyses that are based on an *Abstract Syntax Tree* (AST) as program model. For control flow and dataflow analysis, the control flow graph can be extracted from the AST. Most compiler frontends[5] are primarily based on analyzing and transforming ASTs.

The state of the art in static program analysis based on ASTs is to use depth-first tree traversal to compute the necessary information at each node. In this thesis, we focus instead on computing properties of the AST with reference attribute grammars. The next section describes the AST in more detail, and the section after that introduces reference attribute grammars.

[4] GCC version 7.2.0, a popular C compiler, compiles the example code without warning. An optional warning can be enabled by the user, giving a warning for the example code.

[5] Frontend refers to the parts of a compiler that are not directly concerned with generating or optimizing the executable program output.

2.2 Abstract Syntax Trees

A programming language is a formal language[6] which follows a (usually unambiguous) context-free grammar that can be described in a syntactic metalanguage like, e.g., Extended Backus-Naur Form (EBNF).[7] This section presents an informal view of the formal notions of languages and grammars. See Hopcroft and Ullman [HU69] for a more formal presentation.

Formal notations like EBNF describe the syntax of a programming language in a way that can be mechanically processed and reasoned about (using a metacompiler). Here is a concrete example of part of an EBNF grammar for a simple procedural C-like programming language:

$$
\begin{array}{lll}
\textit{Program} & ::= & \textit{Function}* \\
\textit{Function} & ::= & \textit{Type}\ \langle \textit{ID} \rangle\ \texttt{`('}\ \textit{Parameter}*\ \texttt{`)'}\ \textit{Block} \\
\textit{Parameter} & ::= & \textit{Type}\ \langle \textit{ID} \rangle \\
\textit{Block} & ::= & \texttt{`\{'}\ \textit{Statement}*\ \texttt{`\}'} \\
\textit{Statement} & ::= & \textit{IfStmt} \mid \textit{Call} \\
\textit{IfStmt} & ::= & \texttt{`if'}\ \texttt{`('}\ \textit{Expr}\ \texttt{`)'}\ \textit{Block}\ [\ \texttt{`else'}\ \textit{Block}\ ] \\
\textit{Call} & ::= & \langle \textit{ID} \rangle\ \texttt{`('}\ \textit{Expr}*\ \texttt{`)'}\ \texttt{`;'} \\
\ldots & &
\end{array}
$$

Each line gives a *production rule* composed of symbols, specifying how to write a declaration or statement in the language. Each rule defines a so-called *nonterminal* symbol of the language, which is composed of other symbols (terminal and nonterminal). Terminal symbols are keywords (`'if'`), punctuation symbols (`'{'`), and identifiers ($\langle ID \rangle$). The left-most part of a rule gives the name of the nonterminal, then the component symbols follow on the right (after ::=). Programs in the language should exactly match a rule of the grammar; in the present example, a valid program should match the *Program* rule.

Any valid program can be proven to be part of a formally defined programming language by building a derivation tree of grammar rules matching the program. Parsers are programs that match a valid program to the nonterminals of the parsed programming language. A successful parse produces a rooted tree of the (non)terminals of the program. This tree is called an *Abstract Syntax Tree* (AST), where internal nodes are nonterminals and leaf nodes are terminals. An AST is abstract in the sense that it does not contain the punctuation symbols and keywords from the concrete grammar of the language and instead follows the *abstract grammar* for the language, from which all the punctuation/keyword symbols can be reconstructed (called unparsing).[8]

[6] As opposed to natural languages which are informal and highly ambiguous.

[7] Backus-Naur Form was first used in the Algol 60 report by Backus et. al. [Bac+60]. Extended Backus-Naur Form adds repetition and optional symbols.

[8] An alternative form of parsing output is the parse tree, which contains the literals and tokens of the concrete syntax. Either form constitutes a proof that the program belongs to the parsed programming language. The AST matches a derivation tree for the proof.

```
int greet(bool mom) {
  if (mom) {
    print("Hi Mom!");
  } else {
    print("Hi!");
  }
}
```

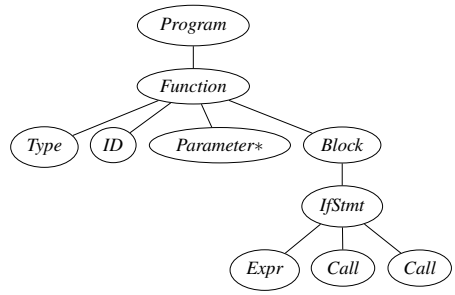


Figure 1: Simple imperative program. On the left: the source code of the program. On the right: a simplified AST matching the program.

An AST is a tree representation of the statements and expressions of a source program. The root of the AST represents the whole program. The next level typically contains global declarations like classes, constants, and functions. Figure 1 illustrates a typical AST for a small program in the language described by the above EBNF grammar.

A context-free grammar described in EBNF is a declarative description of a language. This in itself has all the advantages of declarative programming and, additionally, a formal language like EBNF can be mechanically processed by a type of metacompiler known as a *parser generator* to obtain an efficient parser for the language.[9] By using a parser generator, the language designer need only design the syntactic rules of their language and write them down in a suitable syntactic metalanguage. Certain parser generators can even guarantee, for certain classes of languages, that the language is unambiguous (if it is LR(1), for example). It is entirely feasible to hand-construct a parser, but this requires a larger amount of code to be written, and the resulting code is neither easy to manually prove correct nor readily machine-checkable for correctness.

2.3 Reference Attribute Grammars

Attribute Grammars (AGs) were introduced by Knuth [Knu68b] as a formalism for defining the semantics of a programming language with attributes on the nonterminals of an AST and equations on production rules. An attribute value can represent static properties like the type of an expression, or whether or not a variable use has a previous assignment. We can view attributes as derived properties of the AST: the attribute values are computed as functions of the AST and their values are used in computing other attribute values.

Reference Attribute Grammars (RAGs) are an extension of attribute grammars to object oriented programming proposed by Hedin [Hed00]. In RAGs, attributes can be references pointing to nonterminals. Reference attributes work well, for

[9] Some parser generators can even build parsers for ambiguous or non-deterministic grammars. For example, the Spoofax language workbench implements SGLR parser generation [KV10], based on generalized LR parsing which was invented by Lang [Lan74].

example, for computing graphs derived from the AST, like control flow graphs and class dependence graphs, among others.

RAGs are useful for building extensible and modular language implementations and static analyses, a task that was by some considered impractical in classical AGs.[10] Modularity has long been an active research topic in attribute grammars. Previous proposals for improving modularity in AGs include modular AGs by Dueck and Cormack [DC90], composable AGs by Farrow et al. [FMY92], and generic AGs by Saraiva and Swierstra [SS99], among others.[11] RAGs have been particularly successful compared to the previous approaches to modularity in AGs. The main difference is that RAGs use reference attributes which point to nodes in the AST. RAGs also incorporate higher-order attributes [VSK89a], a previous extension to AGs which was also useful for modularity.

In an object-oriented setting, nonterminals correspond to abstract classes, and production rules correspond to concrete subclasses. However, it is often convenient to create subclasses of concrete classes, and this does not directly correspond to nonterminals or productions. Thus, in the following, we will only view attributes and equations as properties of AST classes, or AST nodes when speaking of a particular instantiation.

An important property of attributes is that they are declarative: attribute equations specify what is computed, not the exact order of attribute evaluations needed to compute the result. An *attribute evaluator* is used to dynamically schedule attribute evaluation in order to compute the required attribute values.

Attribute evaluators are free to *memoize* attributes. Memoization, also referred to as *caching*, means that the attribute value is stored after it has been computed so that it can be reused the next time the attribute value is needed. Memoization can be used to minimize attribute computation time [Jou84].[12] In addition to memoization, attribute evaluators may even parallelize attribute computation, or incrementally update attributes when the underlying AST changes. My most important contribution in this thesis is concurrent evaluation algorithms for RAGs (Paper IV). Concurrent evaluation can be used for parallelizing an existing attribute-based compiler without editing attribute equations, provided that the attributes are well-defined. Parallelization is possible due to the fact that well-defined attributes are *observationally pure*, meaning that they always compute the same value given identical inputs (just like pure functions) [Nau05].

It is important to note that the presence of reference attributes makes it impossible to compute a static attribute evaluation schedule, as is possible with Knuth style AGs. Reference attributes can, in general, point to any nonterminal in the AST and it is not possible to know their dependencies a priori. Reference attributes are thus dynamically scheduled and evaluated on demand.

[10] Dueck and Cormack [DC90] mention some of the problems in using plain AGs for language implementation in practice. Additionally, Farrow and Stanculescu [FS89] reports advantages and disadvantages of using AGs for a large compiler implementation (40+ thousand lines).

[11] For an overview of modular and composable AGs, see Kastens and Waite [KW94].

[12] Söderberg developed a method to automatically select which attributes to memoize to improve performance in practice [SH10].

The following sub-sections introduce several different kinds of attributes which are part of RAGs and common extensions to AGs available in the JastAdd metacompiler. The JastAdd metacompiler itself is introduced in Section 2.5.

Synthesized Attributes

The simplest kind of attribute is the synthesized attribute. A synthesized attribute can be seen as propagating information upwards through the AST (towards the root). Synthesized attributes are defined by equations on AST nodes. An attribute equation can use other attributes or children of the node that the equation belongs to.

A simple synthesized attribute is declared using the following notation:

$$\begin{aligned} &\textbf{\textit{syn int}}\ A.x \\ &\textbf{\textit{eq}} \qquad A.x = 3 \end{aligned}$$

This declares a ***syn****thesized* attribute x on the AST class A. Any AST node of type A will have an instance of the attribute. The equation for the attribute is given on the second line. An equation must exist for each concrete subclass of A. If an equation is given on A itself, then classes lacking an equation will just use the equation from A.

For a more practical example of synthesized attributes, we will consider an attribute that determines if an expression has a static constant value. Such expressions are useful to find because they can be simplified by replacing them with the corresponding constant value, known as *constant value folding*. The following abstract grammar declares a small expression language for this example:

$$\begin{aligned} &\texttt{abstract}\ \textit{Expr} \\ &\textit{Add} : \textit{Expr} ::= \textit{Left:Expr Right:Expr} \\ &\textit{IntLiteral} : \textit{Expr} ::= \langle \textit{VALUE} \rangle \\ &\textit{VarUse} : \textit{Expr} ::= \langle \textit{NAME} \rangle \end{aligned}$$

In this grammar, *IntLiteral* represents constant numbers like 1024 or -37, *VarUse* represents variable uses like a, and *Add* means an addition expression like $a + 5$. Note that *Expr* is declared `abstract`: this just means that *Expr* itself can never occur in any valid expression, and thus we need not write equations for attributes on *Expr*.[13]

[13] In some cases it is convenient to have equations on abstract classes like *Expr*. This is discussed in more detail in Section 3.2.

For identifying constant expressions, observe that an addition expression has a constant value if both terms are constant. Second, a literal value is always constant. For example, the expression $2 + 3$ has a constant value of 5, as opposed to the expression $a + 5$ which can vary depending on the value of variable a. A synthesized attribute based on these ideas could look like this:

$$
\begin{aligned}
&\textbf{\textit{syn boolean}} && \mathit{Expr.isConstant}\\
&\textbf{\textit{eq}} && \mathit{Add.isConstant} = \mathit{Left.isConstant} \wedge \mathit{Right.isConstant}\\
&\textbf{\textit{eq}} && \mathit{IntLiteral.isConstant} = \textbf{\textit{true}}
\end{aligned}
$$

This declares the synthesized attribute *isConstant*. On the first line, the attribute is declared on *Expr* and its subtypes with type ***boolean***, after which equations are given for *Add* and *IntLiteral*. The equations are specified with the ***eq*** keyword and without repeating the attribute type. Note that the equations directly encode the facts about constant expressions which we observed previously.

From the above equations it appears as if there only exists constant expressions, but we have not given an equation for *isConstant* on variable uses yet. Our goal is to handle expressions with variables, like $5 + x$, as illustrated in Figure 2. To achieve this, we add a new attribute equation so that variables are not considered constant:

$$\textbf{\textit{eq}}\ \mathit{VarUse.isConstant} = \textbf{\textit{false}}$$

With just the three attribute equations above we have made a very simple but functional constant value analysis. Better yet, we could combine the above equations with a complete language specification to enable simple constant value folding in a real compiler. Constant value folding also requires that we compute the constant value, which can be done with these attributes:

$$
\begin{aligned}
&\textbf{\textit{syn int}} && \mathit{Expr.constValue}\\
&\textbf{\textit{eq}} && \mathit{Add.constValue} = \mathit{Left.constValue} + \mathit{Right.constValue}\\
&\textbf{\textit{eq}} && \mathit{IntLiteral.constValue} = \mathit{parseInt}(\mathit{VALUE})\\
&\textbf{\textit{eq}} && \mathit{VarUse.constValue} = 0
\end{aligned}
$$

The last equation here is meaningless and will never be used. However, we must include it or else the *constValue* attribute is incomplete as it is conceivable that it could be used in some computation.[14]

With the *isConstant* and *constValue* attributes, a compiler can perform constant folding of subexpressions by replacing each subtree which has $\mathit{isConstant} = \textbf{\textit{true}}$ with an *IntLiteral* containing the value of *constValue*. For instance, in our example language, the expression $a + 32 + 64$ can be replaced by $a + 96$ without affecting the meaning of the expression. Figure 3 shows the corresponding AST before and after constant folding.

[14]The metacompiler that compiles this attribute code does not use precise enough static analysis to allow omitting the equation.

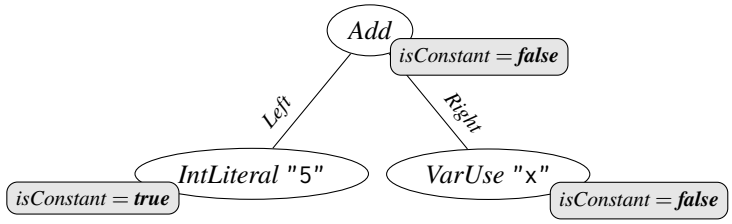


Figure 2: Attributed AST for the expression $5+x$. The *isConstant* attribute instances are displayed as gray boxes attached to nonterminals (AST nodes).

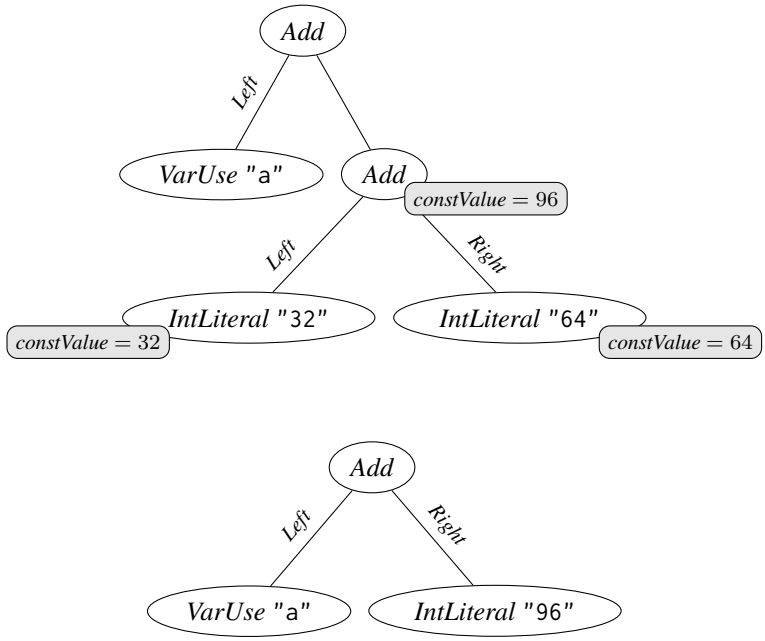


Figure 3: AST for the expression $a + 32 + 64$, before (above) and after (below) constant value folding.

Inherited Attributes

Inherited attributes are used to access information from a parent of an AST node. This is in contrast to synthesized attributes which only directly use values and attributes of the AST node they belong to.

The name for inherited attributes comes from the fact that they inherit their value through the AST structure. Note that this is different from object-oriented inheritance. In an object-oriented setting, it is also possible for a subclass to inherit an attribute equation from a superclass, through the class hierarchy. To avoid confusion I explicitly refer to the latter case as object-oriented inheritance.

Equations for an inherited attribute are specified on edges of the AST. For example, the following notation declares an inherited attribute x on class B, with a matching equation on the edge $A \rightarrow B$:

$$\begin{aligned} &\textbf{\textit{inh int}} \quad B.x \\ &\textbf{\textit{eq}} \qquad A.B.x = 3 \end{aligned}$$

This attribute is illustrated in Figure 4. In this case, A is a parent of B, in some AST following the abstract grammar. If B could also be a child of some other class P, say, then another equation must be added on edge $P \rightarrow B$.

For the simplest form of inherited attributes, an equation must exist for all possible parent edges. However, this may result in many so-called copy attributes which just pass an attribute value along down the AST. This can be neatly solved by a generalization of inherited attributes known as *broadcasting* [Hed11]. With broadcasting, an equation is needed only for at least one ancestor edge for each possible AST instance. Broadcasting is illustrated in Figure 5: node D inherits the value for attribute y through the equation on the edge $A \rightarrow C$. Without broadcasting, an equation would have been needed on edge $C \rightarrow D$. The closest ancestor edge with a matching equation is the one used to evaluate the attribute value. This means that a closer equation can shadow one that is further up in the tree, as in Figure 6.

For an example of a practical inherited attribute, consider the problem of finding the receiver expression of a method call in a language where methods are invoked on objects via dot expressions:

```
a.m().n()
```

The receiver expression is just the left-hand side of the dot. In the above case, `a` is the receiver expression in the call to `m()`, and `a.m()` is the receiver of `n()`.

Here is the relevant part of the abstract grammar for the small method call language:

```
abstract Expr
    Dot : Expr ::= Left:Expr Right:Expr
     Id : Expr
   Call : Expr
```

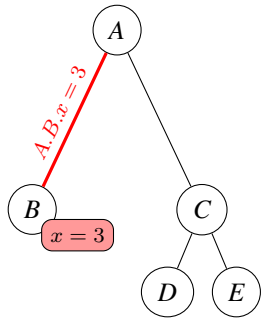


Figure 4: The value of $B.x$ is given by an equation on the edge $A \to B$.

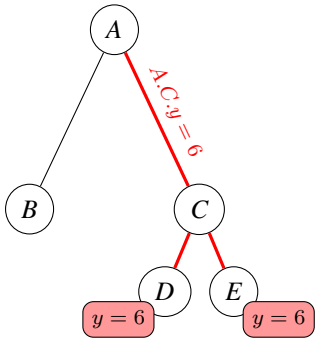


Figure 5: The values of $D.y$ and $E.y$ are given through broadcasting from the equation on edge $A \to C$.

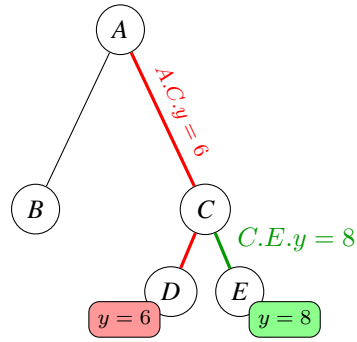


Figure 6: For node E, the equation on edge $C \to E$ shadows the one on edge $A \to C$.

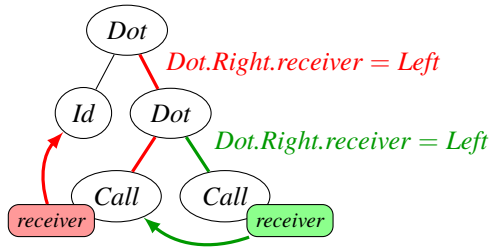


Figure 7: AST for the expression `a.m().n()`. Note that the receiver of the left-hand call is defined by the upper equation, whereas the receiver of the right-hand call is defined by the lower equation.

Assuming that AST nodes contain a parent reference, we could access the receiver without using attributes by following the parent link of a *Call* and then inspecting the left-hand child of the parent. However, we then have to consider the case when calls are chained, leading to calls occurring also as the left-hand side of a *Dot*. In the case with a call in the left-hand side we would have to go up one additional level to find the matching receiver expression. The problem of finding the receiver can more simply be expressed as an inherited attribute:

$$\begin{aligned} &\textbf{\textit{inh}}\ \mathit{Expr} && \mathit{Call.receiver} \\ &\textbf{\textit{eq}} && \mathit{Dot.Right.receiver} = \mathit{Left} \end{aligned}$$

This attribute specification works directly with chained calls. There will be one equation for each *Dot* but it applies only to the *receiver* attributes of the *Right* child. For the *Left* child, an equation of an enclosing *Dot* applies.[15] Figure 7 illustrates this situation.

Broadcasting is used in the *receiver* attribute to get the right reference for left-hand children of a *Dot* without directly giving an equation for that child. The equation given for the right child of a *Dot* does not distinguish between which part of the right child the attribute is in. Note that with broadcasting, an inherited attribute equation defines an attribute value for a subtree at a particular AST edge and broadcasts the value to matching attributes below that edge.

Parameterized Attributes

A natural extension to attributes is to allow parameterization of the attribute equation. This is useful when an attribute answers a question with one or more unknowns. A common example is an attribute that checks if a variable declarator declares a variable with a given name:

$$\begin{gathered} \mathit{Declarator} \ ::= \ \langle \mathit{NAME} \rangle \\ \textbf{\textit{syn boolean}}\ \mathit{Declarator.declares}(\mathit{String}\ n) = (n = \mathit{NAME}) \end{gathered}$$

Another application for parameterized attributes are attributes that select one of several possible values, like the individual declarations in a multiple variable declaration statement:

$$\begin{gathered} \mathit{VarDecl} \ ::= \ \mathit{Declarator}* \\ \textbf{\textit{syn}}\ \mathit{Declarator}\ \mathit{VarDecl.declAt}(i) = \mathit{Declarator}[i] \end{gathered}$$

Both synthesized and inherited attributes can be parameterized, as well as a few other types of attributes discussed in the following sections (circular and higher-order).

[15] This example assumes that a *Call* in this language is never allowed to be used outside a *Dot* expression. Otherwise, we would need additional equations for the *receiver* attribute to be complete.

Parameterized attributes have many practical uses in compilers and static analysis construction. A typical use case is for name lookup, where the name lookup attributes take as parameter the name being looked up. Other interesting applications for parameterized attributes occur when combined with higher-order attributes, for instance the parameterized type lookup attribute in ExtendJ (see Section 4.7).

In the formalization of RAGs by Buckley and Sloane [BS17], inherited attributes are implemented with parameterized synthesized attributes and by having a parent reference as an intrinsic attribute of each node in the AST.

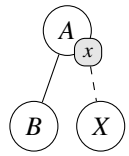
Higher-Order Attributes

Higher-Order Attributes (HOAs) [VSK89a] are attributes which compute a derived subtree of the AST. The derived subtree is considered as part of the AST it belongs to and can itself have attributes. Lazy evaluation is necessary for HOAs, because they can represent infinite trees.

HOAs, also known as *Non-Terminal Attributes* (they are both nonterminals and attributes), are declared by adding the ***nta*** keyword to an ordinary attribute declaration:

$$\textbf{\textit{syn nta}}\ X\ A.x = \textbf{\textit{new}}X$$

To illustrate HOAs, they are shown as part of the AST but connected to the attribute with a dashed edge, like this:



A HOA must compute a fresh object. That is, a new part of the AST must be built which is not allowed to link to an existing subtree of the AST – otherwise it would destroy the tree structure with a subtree that has multiple parents. Existing parts of the AST can be safely reused in a HOA by copying the relevant parts instead of linking them directly. This is a simple rule to follow, but easy to forget or accidentally break.

Higher-order attributes are useful for computing AST structures that were not built directly by the parser. This has many applications, for example: normalizing syntactic sugar and reifying implicit program elements. These applications are explained in more detail below.

Normalizing Syntactic Sugar Syntactic sugar is a term used to describe syntax elements in a programming language which map directly to other, more elementary, language constructs.[16] Examples include the += operator in the C language, which is equivalent to addition and assignment.

For program analysis it is useful to *desugar* syntactic sugar into a corresponding elementary form, so that fewer special cases must be handled by static analysis. This normalization consists of transforming the AST where the specialized syntax occurs.[17]

With HOAs, we can compute the normalized AST as an attribute. The normalized AST must be explicitly accessed to use the transformed version. This is a small overhead compared to using imperative tree transformations. *Forwarding* is an extension of HOAs in which the attribute implicitly replaces the sugared AST [Wyk+02].

Reifying Implicit Constructs Java is an example of an object-oriented language that has a default supertype of all classes, named `Object`. The `Object` class is never declared but behaves as any other declared class (except that it does not have a superclass). The `Object` class must be somehow represented internally in a Java compiler, and since it behaves just like any other Java class it should be placed in the class table among the other library classes (which are parsed from library class files).

A more complex application of reifying constructs is for polymorphic type instantiation. This use case is described in more detail in Section 4.11.

Turing Machines HOAs can be used to evaluate any Turing machine. Although this is quite an esoteric application, it illustrates how HOAs can evaluate infinite ASTs.

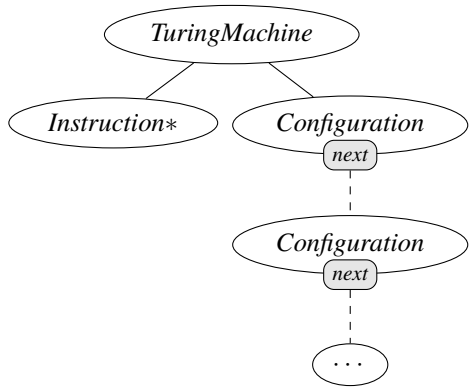
For the Turing machine encoding we need an attribute that computes the next configuration of a Turing machine:

***syn nta** Configuration Configuration.next*

Here, *Configuration* is a nonterminal containing the configuration of a Turing machine (tape content, head position, and current state). A Turing machine starts out in an initial configuration, and the successor configuration is computed by the higher-order *next* attribute, which itself has a successor configuration. To run a Turing machine until it halts, the successor attribute is evaluated until we reach a halting state. The *next* attribute is expanded indefinitely if the machine setup corresponds to a non-halting Turing machine. The following diagram illustrates the AST for a Turing machine:

[16]The term syntactic sugar was coined by Peter J. Landin in 1964.

[17]Normalization is additionally important during code generation, where the program is transformed into a very simple form to enable general optimizations that work on many different surface syntaxes.



The instruction list stored at the root of the AST, in *TuringMachine*, represents the action table of the Turing machine. When evaluating the next configuration, the matching instruction for the current state and symbol under the head is found in the instruction list. An inherited attribute is used to access the instruction list from any configuration.

An implementation of this encoding of Turing machines in RAGs is available at the following public repository: `https://bitbucket.org/joqvist/turing`.

Circular Attributes

Circular attributes are attributes which depend (indirectly) on themselves. The semantics of circularly attributes were first proposed for AGs by Farrow [Far86] and improved by Jones [Jon90]. Circular attributes were later adapted for RAGs by Magnusson and Hedin [MH07]. A circular attribute can be used to express fixpoint functions, which occur in control flow analysis, dataflow analysis, and type inference. An example of such a fixpoint function is the reachable procedures from a given procedure, which is conveniently defined as follows:

$$\mathit{calls}(p) = \{ \text{ procedures called by } p \ \}$$

$$\mathit{reachable}(p) = \{p\} \cup \left(\bigcup_{c \in \mathit{calls}(p)} \mathit{reachable}(c) \right)$$

Evaluation of circular attributes is done by fixpoint iteration: the equation is computed until its value reaches a fixed point. For circular attributes we require that the attribute is well-defined according to the following criteria: the attribute equation is monotone with values arrangeable in a lattice of finite height. This ensures that there is a single least fixed point.

To declare a circular attribute, the attribute is provided with an initial value for the fixpoint iteration. Given a well-defined circular attribute, the fixpoint iteration will terminate and give a single well-defined value.

```
// External declarations:
bool isleaf(Tree t);
int min(int a, int b);
void print(int a);

// Finding minimum leaf value:
int tmin(Tree t) {
  if (isleaf(t)) {
    return t.value;
  } else {
    return min(tmin(t.left), tmin(t.right));
  }
}

void findmin(Tree t) {
  print(tmin(t));
}
```

Procedure	*Calls*	*Reachable*
`findmin`	{`print`, `tmin`}	{`print`, `findmin`, `tmin`, `min`, `isleaf`}
`tmin`	{`tmin`, `min`, `isleaf`}	{`tmin`, `min`, `isleaf`}
`min`	{}	{`min`}
`isleaf`	{}	{`isleaf`}
`print`	{}	{`print`}

Figure 8: Reachable procedures for a small program for finding the minimum leaf value in a tree. Above: the source code of the program. Below: reachable procedure sets.

The reachable procedures function above can be implemented as a circular attribute in the following way:

$$\textbf{\textit{syn}}\ \mathit{Set}\langle \mathit{Procedure}\rangle\ \mathit{Procedure.reachable}\ \textbf{\textit{circular}}(\emptyset) =$$
$$\textbf{\textit{this}} \cup \left(\bigcup_{c \in \textbf{\textit{this}}.\mathit{calls}} c.\mathit{reachable} \right)$$

The ***circular***$(\emptyset)$ part gives the initial value for the fixpoint iteration. Figure 8 shows the reachable procedures in a small imperative program.

Collection Attributes

In static analysis it is often necessary to collect a multitude of values of some kind from different nodes in an AST. Examples include error messages, local procedure calls, and static type dependencies. *Collection attributes* are an extension for RAGs

by Magnusson et al. [MEH07] which supports these kinds of value collections. The attribute evaluator traverses the AST to collect contributions for the attribute value from disparate nodes in the tree.

To declare a collection attribute, we must declare what is to be collected and on which class:

coll *Collection⟨String⟩ Program.errors*

This attribute collects error messages on the *Program* class (the root of the AST). The meaning of the attribute is only implicit from the attribute name (*errors*); the actual meaning of the attribute is defined by *contribution statements* declared for all classes which may provide a value for this collection attribute. In this case, we need a contribution statement for each class that may report an error message:

VarUse ***contributes*** `"variable not initialized before use"`
  when $\neg$ *definitelyAssigned*
  to *Program.errors*

This contribution is *conditional*, where the ***when*** clause contains an expression that controls if the error message will be reported for the current node or not. It is also possible to declare unconditional contributions, for example when collecting static type dependencies.

2.4 Attribute-Controlled Rewrites

In *Rewritable Reference Attribute Grammars* [EH04], parts of the AST can be automatically transformed by using attribute-controlled rewrite rules. While higher-order attributes can also be used to transform part of the AST, they need to be explicitly referenced in order to access the transformed version of the AST. In contrast, rewrite rules are invisible to other attributes: they are automatically activated whenever the rewritable part of the AST is first accessed.

Rewrite rules can have an optional condition, which decides if the rewrite will be applied to the target node.

The following rewrite rule replaces any multiplication by zero with a constant zero literal:

rewrite *MulExpr*
  when *Left.isZero* $\vee$ *Right.isZero*
  to new *IntegerLiteral*(`"0"`)

Rewrite rules are a simple way of transforming parts of the AST. Unlike higher-order attributes, a rewrite rule replaces the original part of the AST that is rewritten. With a higher-order attribute, on the other hand, it is always possible to access the original AST, making it easier to pretty-print the original source form of the program.

Söderberg and Hedin [SH15] showed that rewrites are equivalent to circular higher-order attributes. This mapping makes it simple to implement rewrites in a RAG system which already has circular and higher-order attributes.

2.5 The JastAdd Metacompiler

JastAdd is a metacompiler[18] for *Reference Attribute Grammars* (RAGs) which has found success in modular programming language composition and for real-world programming language implementation [HM03a; Hed11; EH07c].

JastAdd generates Java AST classes to represent the nonterminals of an abstract grammar. Attributes are generated as Java methods in these AST classes, based on a set of attribute specifications. JastAdd attributes are specified in a domain-specific language with embedded Java code for attribute equations. This language uses *Inter-Type Declarations* (ITDs): a concept from *Aspect-Oriented Programming* (AOP) in which the attribute is declared separately from the class it belongs to. It is also possible to declare new utility methods in AST classes as ITDs.[19]

In previous attribute grammar examples, we have already used a simplified version of the JastAdd notation for attributes. Here is the literal JastAdd attribute syntax for a synthesized attribute:

```
syn int A.x() = 3;
```

This declares a synthesized attribute (hence, `syn`) belonging to AST class `A`. We will continue using a simplified version of the JastAdd syntax that removes semicolons and the parentheses for non-parameterized attributes.

Thanks to the use of inter-type declarations, attribute declarations can be organized into aspect files by whatever categorization is most appropriate. An aspect file has the following layout:

```
import java.util.*;
aspect MyAnalysis {
  <attributes and other ITDs>
}
```

The JastAdd abstract grammar syntax is similar to the previous grammar examples. However, there may only be one production rule per nonterminal, since each production rule corresponds to an AST class declaration. For alternatives, JastAdd grammars have object-oriented inheritance between grammar productions. For each alternative, we use separate productions with a common superclass. Binary expressions are a common example where alternatives are useful in the abstract grammar. In JastAdd, we can represent all binary expressions with a common supertype *Binary*, say, and we then have subtypes for each concrete kind of binary expression, like addition, subtraction, multiplication, etc.

$$\texttt{abstract}\ \mathit{Binary} : \mathit{Expr} \ ::= \ \mathit{Left{:}Expr}\ \mathit{Right{:}Expr}$$

The common supertype *Binary* does not represent any specific kind of binary expression, in fact it will never exist in any concrete AST so we declare it `abstract`. Concrete subclasses of *Binary* are the actual binary expressions of the language. Following are some common examples of binary expressions:

[18] A compiler that compiles compilers; a compiler for a metalanguage.

[19] The flavor of AOP used in JastAdd is similar to subject-oriented programming [HO93].

$$Add : Binary$$
$$Sub : Binary$$
$$Mul : Binary$$
$$\ldots$$

By inheriting from *Binary*, these classes automatically receive the child components of *Binary*. It is possible to add additional child components in a subtype. In the above example, *Binary* could alternatively have been declared as empty, and each subtype could have specified the *Left* and *Right* children. This would, however, remove the possibility for attributes common to all binary expressions to directly access the children, which would result in less reuse.

JastAdd abstract grammars may use optional, list, and token components:

$$A ::= B \quad [C] \quad D* \quad \langle E \rangle$$

Lists and optional children are wrapped by implicitly generated nonterminals *List* and *Opt*.

JastAdd supports all previously discussed attribute kinds: synthesized, inherited, parameterized, higher-order, circular, and collection attributes. JastAdd also allows automatic attribute-controlled rewriting of AST nodes, and JastAdd has an aspect-oriented mechanism for replacing existing attribute equations.

JastAdd generates AST classes and weaves attributes into them. Attributes are generated as methods of the AST classes. This forms a foundation for building compilers and static analyses: only a parser needs to be added in order to build a complete static analysis framework with JastAdd. In fact, JastAdd has been used to build compilers for several languages like Java, Modelica, and Bloqqi [EH07b; Åke+10; FH16]. In this dissertation we will look mainly at the ExtendJ Java compiler, but the techniques covered can be applied to any other JastAdd-based compiler project, or indeed, to other RAG-based compiler specifications.

JastAdd makes it eminently easy to build extensible compilers thanks to RAGs and aspect-oriented programming. The following section describes how JastAdd features are used for building extensible static analyses.

3 Using RAGs for Extensible Analyses

In this section, I show how some features of RAGs can be used to build extensible static analyses. This is based on my experiences from working with the JastAdd metacompiler. Although the following discussion focuses on JastAdd RAGs, many of the results apply to other RAGs as well.

The main features of JastAdd that are of interest from the perspective of extensibility are:

- inter-type declarations,
- object-oriented attribute inheritance,

- attribute replacement,
- structure-shy programming with inherited attribute broadcasting,
- desugaring for reusable code generation,
- collection attributes.

The following sub-sections examine each of these features in detail.

3.1 Inter-Type Declarations

As mentioned in Section 2.5, JastAdd attributes are specified with inter-type declarations (ITDs). By using ITDs, JastAdd solves the *expression problem*, a common yardstick when comparing extensibility of programming language implementations. The expression problem was defined by Wadler [Wad98] as follows:

> The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

With ITDs, new attributes (functions) can be added to pre-existing AST classes. New AST classes (datatype cases) are also easily added by new abstract grammar rules. There is one caveat: JastAdd RAGs require recompilation when the attribute grammar is changed or extended. Recompilation is needed because JastAdd generates the analysis or compiler by weaving the introduced attributes into AST class declarations during code generation.

For an example of the expression problem, I will demonstrate how to extend a small expression language with a new operator and new functionality by using JastAdd. We start with the expression language defined by the following abstract grammar:

$$
\begin{aligned}
\texttt{abstract}\ \mathit{Expr} & \\
\mathit{ParExpr} : \mathit{Expr} &::= \mathit{Expr} \\
\mathit{IntLiteral} : \mathit{Expr} &::= \langle \mathit{VALUE} \rangle \\
\texttt{abstract}\ \mathit{Binary} : \mathit{Expr} &::= \mathit{Left{:}Expr}\ \mathit{Right{:}Expr} \\
\mathit{Add} : \mathit{Binary} & \\
\mathit{Sub} : \mathit{Binary} &
\end{aligned}
$$

This language can easily be extended with a new factorial operator by appending the following abstract grammar rule:

$$\mathit{Factorial} : \mathit{Expr} ::= \mathit{Expr}$$

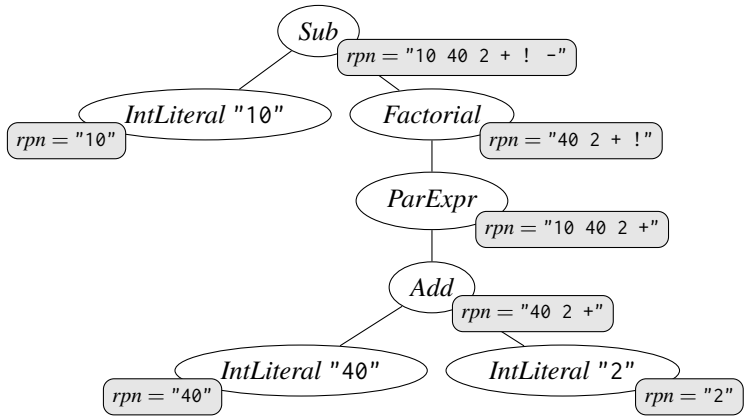


Figure 9: Attributed AST for the algebraic expression $10 - (40 + 2)!$. The *rpn* attribute shows the reverse polish notation at each subexpression. The RPN for the whole expression is `10 40 2 + ! -`.

We will now extend the behaviour of the language by adding attributes for printing an expression in *Reverse Polish Notation* (RPN), in which operands are written before operators. To this end, we declare the *rpn* attribute on *Expr*:

$$\textbf{\textit{syn}}\ \mathit{String}\ \mathit{Expr.rpn}$$

Equations for the new attribute are needed on each concrete subclass of *Expr*:

$$\begin{aligned}
\textbf{\textit{eq}} \quad & \mathit{Add.rpn} = \mathit{Left.rpn} + \texttt{" "} + \mathit{Right.rpn} + \texttt{" +"} \\
\textbf{\textit{eq}} \quad & \mathit{Sub.rpn} = \mathit{Left.rpn} + \texttt{" "} + \mathit{Right.rpn} + \texttt{" -"} \\
\textbf{\textit{eq}} \quad & \mathit{ParExpr.rpn} = \mathit{Expr.rpn} \\
\textbf{\textit{eq}} \quad & \mathit{IntLiteral.rpn} = \mathit{VALUE} \\
\textbf{\textit{eq}} \quad & \mathit{Factorial.rpn} = \mathit{Expr.rpn} + \texttt{" !"}
\end{aligned}$$

For the factorial operator, RPN coincides with the standard algebraic notation. For the binary expressions *Add* and *Sub*, the operands are printed first, then the operator. Parenthesis expressions (*ParExpr*), which are significant in algebraic notation, are not needed in RPN. Integer literals (*IntLiteral*) need no special handling for RPN output. Figure 9 shows the AST of an expression in the extended language and the corresponding RPN attribute values.

3.2 Object-Oriented Attribute Inheritance

When RAGs are combined with object-oriented programming, attributes can be inherited through the class hierarchy. This can be useful for factoring out common attributes to an AST superclass. For instance, consider the previous RPN example: the *Binary* class is a common superclass of both *Add* and *Sub* expressions. We can

factor out the separate equations of the *rpn* attribute from *Add* and *Sub* with a new equation on *Binary*, like this:

$$\textbf{\textit{eq}}\ \mathit{Binary.rpn} = \mathit{Left.rpn} + \texttt{" "} + \mathit{Right.rpn} + \texttt{" "} + \mathit{opSuffix}$$

Now we just need *opSuffix* on *Add* and *Sub*:

$$\begin{aligned}
&\textbf{\textit{syn}}\ \mathit{String}\ \mathit{Binary.opSuffix}\\
&\textbf{\textit{eq}} \qquad\qquad \mathit{Add.opSuffix} = \texttt{"+"}\\
&\textbf{\textit{eq}} \qquad\qquad \mathit{Sub.opSuffix} = \texttt{"-"}
\end{aligned}$$

Although we got rid of some duplicated code by moving the *rpn* equation to *Binary*, it is of little benefit in the present example, especially since we had to add a new *opSuffix* attribute to fill in the blank at *Binary*. This type of attribute factoring pays off to a much greater extent in more realistic programming languages which typically have many more binary operators.

3.3 Attribute Replacement

In addition to inter-type declarations, JastAdd has another AOP-inspired mechanism: it is possible to replace an existing attribute by using a ***refine*** declaration, like this:

$$\textbf{\textit{refine eq}}\ \mathit{Sub.opSuffix} = \texttt{"?"}$$

This changes the equation for an existing attribute *Sub.opSuffix*.

The ***refine*** mechanism is similar to point-cuts in AOP, but instead of inserting a new computation it replaces the whole computation.[20]

Replacing attributes like this may at first seem to provide little benefit for building static analyses with RAGs. However, it is very useful when extending an existing analysis. When extending an analysis, there is often a need to change the meaning of an existing attribute. This can either be accomplished by using a ***refine*** rule, or by creating a subclass of the corresponding AST class and overriding the attribute. The ***refine*** solution is often more lightweight: it requires less boilerplate code and specialization to handle the new AST class.

When extending a programming language, it is often necessary to change the behaviour of an existing attribute to accommodate some new language feature. This can either be done by refactoring the pre-existing attribute into smaller parts that can be overridden separately in an extension. However, this can reduce the readability of the original code which in may outweigh the benefit of reduced code duplication in the extension. Furthermore, if the base system can not be changed, use of ***refine*** can allow some changes which would otherwise not be possible.

[20] It is possible to reuse the old equation inside the new one by using the ***refined*** keyword.

3.4 Structure-Shy Programming with Inherited Attributes and Broadcasting

Inherited attributes with broadcasting embody a style of programming which was named *structure-shy* programming by Lieberherr [Lie96]. A structure-shy program specifies certain behaviour for some substructures and leaves handling of the rest to a generic solution, according to Cunha and Visser [CV11]. A typical example of a structure-shy program is an XPath query which matches only certain parts of an XML document tree and ignores the rest [XPath]. An XPath query is insensitive to irrelevant changes in the tree which the query does not directly match.

An inherited attribute equation (with broadcasting) is propagated to all matching attributes in the subtree below the edge where the equation is attached. If we add a new AST node inside some subtree which is already covered by an inherited equation, there is often no need to add a new equation for that attribute. In several cases, it is possible to add new language features that reuse existing inherited attributes with little effort.

The structure-shyness of inherited attributes is particularly apparent in lookup attributes. Name lookup is a typical example of a lookup attribute: an inherited attribute is used for finding a reference to the declaration of a variable use. Suppose we have the following abstract grammar:

$$
\begin{aligned}
&\texttt{abstract}\ \mathit{Expr}\\
&\quad \mathit{Let} : \mathit{Expr} \ ::= \ \langle \mathit{NAME} \rangle\ \langle \mathit{VALUE} \rangle\ \mathit{Expr}\\
&\quad \mathit{VarUse} : \mathit{Expr} \ ::= \ \langle \mathit{NAME} \rangle\\
&\quad \mathit{Add} : \mathit{Expr} \ ::= \ \mathit{Left{:}Expr}\ \mathit{Right{:}Expr}
\end{aligned}
$$

The *Let* expressions of this language are only allowed to define variables with constant values, and the expression inside the *Let* may only use variable names defined by some enclosing *Let*. An example of a valid expression in the language looks like this:

```
let x = 40 in
  let y = 2 in
    x + y
```

This expression computes the value 42.

The above language is almost entirely pointless. Nevertheless, name analysis for this language works much like name analysis in any other programming language. Here is a name lookup attribute for the `let`-expression language:

$$
\begin{aligned}
&\textbf{\textit{inh}}\ \mathit{Let} \quad \mathit{Expr.lookup}(n)\\
&\textbf{\textit{eq}} \quad \mathit{Let.Expr.lookup}(n) = \textbf{\textit{this}}\ \textbf{\textit{if}}\ n = \mathit{NAME};\ \textbf{\textit{else}}\ \mathit{lookup}(n)
\end{aligned}
$$

The lookup attribute at a declaration node (*Let* expression) will give a reference to the declaration if its name matches the sought-after variable. If the name n does not match the declaration, the attribute equation uses $\mathit{lookup}(n)$ of the *Let* expression itself to delegate name lookup to the enclosing scope.

The lookup attribute above is structure-shy in the sense that we could introduce new name declarations outside the expression, or irrelevant name declarations inside it, without affecting the lookup attribute value for previous nodes. For example, the following expression has the same name bindings as in the previous one:

```
let unused = 1 in
  let x = 40 in
    let abc = 0 in
      let y = 2 in
        x + y
```

We can add many language extensions to this `let`-language without affecting name lookup. For instance, if we wish to add a division operator to the language, we could add the following abstract grammar rule:

$$\mathit{Div} : \mathit{Expr} ::= \mathit{Left{:}Expr}\ \mathit{Right{:}Expr}$$

Now, we can write expressions like

```
let x = 355 in
  let y = 113 in
    x / y
```

Importantly, this extended language works as intended without having to add new attribute equations for the lookup attribute. This works despite the fact that the new *Div* node occurs between the pre-existing *Let* and *VarUse* constructs in the AST. Because we added a kind of expression which does not declare new names or alter name scoping rules, the old equations just work.

3.5 Desugaring with Higher-Order Attributes

Higher-order attributes are useful for developing code generation for programming language extensions, among other things. Code generation for a new language construct can often be conveniently implemented by mapping the new language mechanism to an equivalent form using pre-existing language features.

For an example of desugaring, we will look at a small extension to the ExtendJ compiler which overloads the multiplication operator for string repetition. Multiplying a string is not allowed in plain Java (as of the current latest version, Java 11). Our goal is to allow a string to be multiplied with an integer, resulting in the string repeated a number of times equal to the integer operand. For instance, the following expression should store the string `"gogogo"` in variable `msg`:

```
String msg = "go" * 3;
```

The above is equivalent to the following `for`-loop:

```
StringBuilder buf = new StringBuilder();
for (int i = 0; i < 3; ++i) {
  buf.append("go");
}
String msg = buf.toString();
```

If we wish to implement the code generation for this in ExtendJ, we could create a higher-order attribute to compute the desugared version:

$$\textbf{\textit{syn nta}}\ \textit{Stmt Mul.desugared} = \textbf{\textit{new}}\ \textit{ForStmt}(\ldots)$$

Code generation can be accomplished by reusing existing code generation on the desugared form, like this:

$$\textbf{\textit{eq}}\ \textit{Mul.code} = \textit{desugared.code}$$

The details of a complete implementation are just a little bit more involved. A functional implementation of this example, as a small extension to ExtendJ, is available as open source from the following public repository:

`https://bitbucket.org/extendj/string-repeat`

The ExtendJ compiler uses desugaring for a few important features. For example, lambda expressions in the Java 8 module are implemented by desugaring to anonymous classes. Another example is the try-with-resources statement, introduced in Java 7 [JLS7, §14.20.3], for which partial desugaring is used to simplify code generation.

3.6 Collection Attributes

Collection attributes receive their values from contribution statements, which are declared separately from the target collection attribute, using ITDs. This makes it easy to introduce new contributions to an existing collection attribute in an extension. In language extensions we often need to add new error messages to the compiler, which can be easily done by adding new contributions if the error messages are collected with a collection attribute.

For an example of adding a new error message to a compiler, suppose we want to report when strings are compared using the equals operator in Java:

```
if (name == "Not Sure") {
  ...
}
```

Normally, Java allows this code without warning, though it usually does not work the way the programmer intended. Strings should instead, in most cases, be compared with the `equals` method:

```
if (name.equals("Not Sure")) {
  ...
}
```

An error message can be added to the ExtendJ compiler by just adding a new contribution for a collection attribute, like this:

$$\begin{aligned}&\textit{EQExpr}\ \textbf{\textit{contributes}}\ \textit{error}(\texttt{"Incorrect string equality test!"})\\ &\quad\textbf{\textit{when}}\ \textit{Left.type.isString} \wedge \textit{Right.type.isString}\\ &\quad\textbf{\textit{to}}\ \textit{CompilationUnit.problems}\end{aligned}$$

Collections Over Higher-Order Attributes

Collection attributes do not normally search for contributions in higher-order attributes. This is due to the fact that higher-order attributes can expand indefinitely.

In some cases, however, it is useful to have collections which range over certain finite higher-order attributes. To this end, we can control the search during collection attribute evaluation by using a special JastAdd mechanism that I developed for JastAdd version 2.2.1. This new mechanism is a variation of the contributes statement that allows specifying which nodes to search for contributions. Here is an example from the ExtendJ compiler:

```
LambdaExpr contributes {
  toClass().collectContributions();
} to TypeDecl.nestedTypes();
```

I added this particular code fragment to the compiler to solve a problem in finding all anonymous classes induced by lambda expressions. In the ExtendJ Java compiler, extensions may need to generate new anonymous classes. In particular, the Java 8 module in ExtendJ uses anonymous classes to implement lambda expressions. Since these anonymous classes are built with higher-order attributes at various places in the AST (inside expressions), they need to be located during code generation in order for the anonymous class to be written to a class file. There was already such a collection attribute to gather ordinary anonymous class declarations. However, it did not locate the new lambda classes because they were nested inside higher-order attributes. This could be solved with a new contribution statement, except for lambda expressions nested in other lambda expressions. For nested lambdas, we have the problem of needing to look inside the anonymous class, which is a higher-order attribute, to find all anonymous classes. The above code fragment solved this issue by adding the anonymous class for a lambda expression, *LambdaExpr.toClass*, to be searched for contributions to the *nestedTypes* collection attribute (the attribute for finding anonymous classes).

3.7 Generic AST Traversal

In imperative programming, traversing a tree of nodes in a particular order is straightforward, for example using standard preorder traversal (as in depth-first search). However, this task is trickier with declarative attributes. Attributes typically hide their evaluation order, and have no direct mechanism for ordering attribute evaluations. So the question is: How can we use attributes to implement ordered traversal?

The solution is to use a reference attribute to point out the predecessor in the required traversal order. We can then implement our traversal using this predecessor attribute. Suppose that we need to number the nodes of an AST by their preorder number. Let *pred* be a reference attribute pointing to the predecessor in the traversal order. We can achieve the required preorder numbering with the following equations:

$$\textbf{\textit{syn int}}\ \mathit{ASTNode.dfnum} = \mathit{pred.dfnum} + 1$$
$$\textbf{\textit{eq}}\ \mathit{Root.dfnum} = 0$$

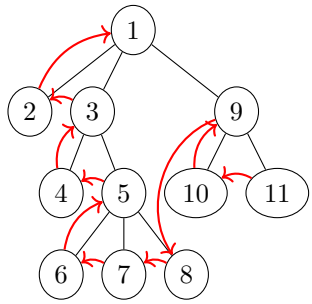


Figure 10: An AST with nodes labeled by their preorder number. The red arrows show the *pred* attribute references for all nodes except the root.

Here, *ASTNode* is the generic superclass of all AST classes, and *Root* is the root class of the AST (from the abstract grammar). The equation on *Root* is needed to give the base case for *dfnum*: a *Root* node is the first node in the traversal. Figure 10 shows the preorder numbering of a generic AST.

The implementation of the *pred* attribute is a little bit more involved. The *pred* attribute is an inherited attribute which uses *prevNode* to find the predecessor of the current node in its parent. We also need an attribute, *last*, to point to the last node inside a subtree. Here is the full implementation of *pred* for a preorder traversal:

$$
\begin{aligned}
&\textbf{\textit{inh}}\ \mathit{ASTNode}\ \mathit{ASTNode.pred} \\
&\textbf{\textit{eq}}\qquad\quad \mathit{ASTNode.child}[i].\mathit{pred} &&= \mathit{prevNode}(i) \\
&\textbf{\textit{syn}}\ \mathit{ASTNode}\ \mathit{ASTNode.prevNode}(i) &&= \mathit{child}[i-1].\mathit{last}\ \ \textbf{\textit{if}}\ \ i > 0;\ \textbf{\textit{else}}\ \ \textbf{\textit{this}} \\
&\textbf{\textit{syn}}\ \mathit{ASTNode}\ \mathit{ASTNode.last} &&= \mathit{prevNode}(|\mathit{child}|)
\end{aligned}
$$

Here, *child* is an array of the children in an *ASTNode*, and $|child|$ means the length of the child vector. This code uses a JastAdd mechanism which we have not previously discussed: the *pred* equation uses the index i of the child edge which the equation is evaluated on (this works similarly for list children).

We can apply this preorder traversal pattern to a number of useful tasks, like numbering local variable declarations (needed for code generation). Here is an example of a local variable numbering attribute, *varNum*, for a fictional procedural language:

$$
\begin{aligned}
\textbf{\textit{syn int}}\ \mathit{ASTNode}\ \mathit{ASTNode.varNum} &= \mathit{pred.varNum} \\
\textbf{\textit{eq}}\ \mathit{VarDecl.varNum} &= \mathit{pred.varNum} + 1 \\
\textbf{\textit{eq}}\ \mathit{Function.varNum} &= 0
\end{aligned}
$$

The equation on *Function* resets the numbering for each function, making variable numbering local to each function. The *varNum* attribute is structure-shy, as it is only affected by the presence of *VarDecl* and *Function* nodes in the AST, and insensitive to the structure of the rest of the tree. New language constructs can easily be added to the language without affecting the *varNum* attribute.

3.8 Discussion

While ITDs are useful for extensibility, it can been argued that AOP counters modularity. Parnas famously promotes information hiding as the main criteria for decomposing a system into modules [Par72]. However, few things can be effectively hidden in the presence of AOP [Ste06].

With ITDs we gain the ability to easily decompose a compiler into separate modules according to any cross-cutting concern. Additionally, AOP in JastAdd provides high composability: we can build components of attributes which are combined without having to pre-design extension points or callback mechanisms. The benefits of the limited form of AOP combined with attributes in JastAdd were investigated in a paper by Avgustinov et al. [AET08a].

In practice, extensibility need not be deliberately designed into a JastAdd-based compiler. Instead, extensibility occurs as a happy coincidence of using RAGs and AOP.

With attributes, there is no need for data structures like symbol tables, which are external to the AST. Instead, all information that is needed for compilation can be computed directly by attributes. Symbol tables can be replaced with reference attributes. Lookup tables indexed by AST nodes correspond directly to attributes where the attribute equation computes the value in the lookup table. This lack of specialized data structures is also a benefit from the perspective of extensibility as there are no existing data structures that need to be expanded to store more information than they were originally designed for.

4 The ExtendJ Java Compiler

ExtendJ (formerly, JastAddJ[21] [EH07b]) is an extensible Java compiler, implemented using the JastAdd metacompiler, supporting full Java source-to-bytecode compilation [ExJorg].

ExtendJ is free and open source, provided under The Modified BSD License [BSD]. The source code is available from the following repository:

```
https://bitbucket.org/extendj/extendj
```

With ExtendJ, it is possible to develop new language extensions, static analyses, source transformation tools, and other tools based on the Java language.

The rest of this section is organized as follows. The next sub-section describes the development history of ExtendJ, then interesting examples of extensions are presented, then continuing with a high-level overview of the design of the compiler, and some of the most important attributes.

[21] Originally known as the JastAdd Extensible Java Compiler, later shortened to JastAddJ. The compiler has occasionally been referred to as the JastAdd frontend for Java. The compiler was finally renamed to ExtendJ to avoid confusion with the JastAdd metacompiler.

4.1 Development History

ExtendJ was originally designed as a case study in using JastAdd to construct a practical compiler with RAGs. The result was a highly extensible compiler that proved suitable for building static analyses and language extensions for Java.

Torbjörn Ekman developed the first versions of ExtendJ, including the Java 1.4 and Java 5 versions [EH04; EH07b]. I later took over the project, implementing new versions for Java 6 (a minimal change to Java 5), and Java 7. Ekman had left the project before I started working on ExtendJ. I started by learning about RAGs and how the compiler worked. I received help from Görel Hedin and Anders Nilsson who had some knowledge of ExtendJ. The next major Java version, Java 8, was implemented by Erik Hogeman for his Masters Thesis project under my supervision [Hog14]. Hogeman did an excellent job, but there were some final enhancements which I completed for the Java 8 extension (improved type inference and code generation issues).

The following table summarizes the authorship of ExtendJ, as of version 8.1.2:

Author	*Commits*	*Inserted*	*Removed*
Jesper Öqvist	890	136 043	116 391
Torbjörn Ekman	401	69 606	38 231
Erik Hogeman	17	12 273	2 477
Max Schäfer	14	8 695	172
Pavel Avgustinov	9	1 730	865
Emma Söderberg	20	563	107

Inserted/Removed: number of lines inserted/removed across all commits.[22]

Sorted by decreasing number of inserted plus removed lines. Authors with fewer than 300 lines inserted plus removed are not listed (10 in total).

During my time working on ExtendJ, I have fixed many bugs in the compiler. This includes code generation errors for which the compiler created faulty bytecode, compile failures where the compiler failed to compile well-formed Java code, or incorrectly accepting code which should have caused a compile error. About 250 of the bugs I fixed are tracked on the current issue tracker for ExtendJ,[23] many bugs that I fixed were not tracked. I also developed the test framework and test suite for ExtendJ, which currently contains about 1700 tests. Erik Hogeman wrote 888 tests for the Java 8 implementation [Hog14].

In order to enable parallel compilation in ExtendJ, side effects had to be removed. Thus, an important part of my contributions in this dissertation is a fully declarative, side effect free, version of ExtendJ. This makes it possible to parallelize compilation, and to run concurrent analyses in interactive tools based on ExtendJ. Previously, ExtendJ had some known uses of side effects and imperative tree transformations which prevented concurrent evaluation.

[22]The number of lines were counted by using the `show` command for `git` with the `--numstat` option. Line counts include changes to non-source files like README and ChangeLog.

[23]The issue tracker is at the main code repository: `https://bitbucket.org/extendj/extendj`.

I have also made several other refactorings to the compiler in order to improve correctness or simplify compilation. Some of this work is presented in Section 5.

4.2 Extensions Overview

ExtendJ has been useful to researchers over several years, thanks to continued development of the compiler and support for newer Java versions. The ability to implement new Java versions efficiently, by only a few developers, was to a large extent afforded by the inherent extensibility in using JastAdd to build the compiler.

While other extensible Java compilers do exist, language extensions and tools based on ExtendJ are often smaller in terms of code size, and more maintainable [AET08a]. The smaller implementation size and declarative coding paradigm in ExtendJ seems to outweigh some of the drawback of using a more unconventional RAG-based compiler architecture. Researchers who are unfamiliar with RAGs will find learning the JastAdd code in ExtendJ to be an obstacle to overcome before they can start implementing a project in the compiler. Still, ExtendJ has been used by several researchers to implement their language extensions, static analyses, and other tools.

To illustrate the variety of research that has been done with ExtendJ, here are some of the interesting results that have been published, including work from our own research group as well as from others who worked independently:

Language extensions:

- Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble, "Modularity first: a case for mixing AOP and attribute grammars". In *AOSD*, 2008. [AET08a].
- (Paper II) Friedrich Steimann, Jesper Öqvist, and Görel Hedin. "Multitudes of Objects: First Implementation and Case Study for Java". In *Journal of Object Technology, Vol. 13, no. 5 (November 2014), pp. 1:1–33*.
- Sukyoung Ryu, "ThisType for Object-Oriented Languages: From Theory to Practice". In *TOPLAS*, 2016 [Ryu16].
- YungYu Zhuang and Shigeru Chiba, "Expanding Event Systems to Support Signals by Enabling the Automation of Handler Bindings". In *Journal of Information Processing*, 2016 [ZC16].
- Tetsuo Kamina and Tomoyuki Aotani, "Harmonizing Signals and Events with a Lightweight Extension to Java". In *Programming Journal, Vol. 2, no. 3*, 2018 [KA18].
- Jan C. Dageförde and Herbert Kuchen, "A constraint-logic object-oriented language". In *SAC*, 2018 [DK18].

Tools and analyses:

- Torbjörn Ekman and Görel Hedin, "Pluggable checking and inferencing of nonnull types for Java". In *Journal of Object Technology, Vol. 6, no. 9, 2007* [EH07a].
- Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson, "Extensible intraprocedural flow analysis at the abstract syntax tree level". In *Sci. Comput. Program.*, 2013 [Söd+13].
- (Paper III) Jesper Öqvist, Görel Hedin, and Boris Magnusson. "Extraction-Based Regression Test Selection" In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'16), ACM, pp. 5:1–5:10.* Lugano, Switzerland, 2016.
- Friedrich Steimann, Jörg Hagemann, and Bastian Ulke, "Computing repair alternatives for malformed programs using constraint attribute grammars". In *OOPSLA*, 2016 [SHU16].
- Mohammad R. Azadmanesh and Matthias Hauswirth, "Concept-Driven Generation of Intuitive Explanations of Program Execution for a Visual Tutor". In *VISSOFT*, 2017 [AH17].

At the computer science department at Lund University, we have for the past four years offered a course where students work in groups of two to implement small compiler-related projects. Some of the compiler projects are implemented as extensions to ExtendJ. Here are a few of the interesting projects that have been done by students so far in the course:

- Olle Tervalampi-Olsson and Marcus Lacerda, "Object-oriented metrics for Java programs", 2014.
- Joel Lindholm and Johan Thorsberg, "Package metrics on Java projects", 2014.
- Ella Eriksson and Zimon Kuhs, "Bug detection through static analysis", 2015.
- Hans Bjerndell and Linus Lexfors, "Extending Java with new operators", 2016.
- Sebastian Hjelm and Markus Olsson, "Extending the ExtendJ Java compiler to Java 9 support", 2017.
- Wawrzyn Chonewicz and Filip Stenström, "Extending Java with new operators using ExtendJ", 2017.

Although most of the student projects listed above were either very small extensions or partially implemented, they still demonstrate that it is reasonably easy for developers who are previously unfamiliar with ExtendJ to start implementing useful

extensions within a couple of weeks of work. The students spend about six to eight man-weeks of implementation work on the project (the rest of the time is spent on reading related work and writing a report). At the beginning of the project, they have a very basic understanding of JastAdd and RAGs from the prerequisite compilers course.

Most of the contributions in this dissertation were implemented and/or evaluated in ExtendJ. The Java 7 extension (Paper I) is a core extension for ExtendJ, in which I developed design patterns which were later used in the Java 8 extension. The Multiplicities case study (Paper II) was developed as an ExtendJ extension. The test selection project (Paper III) was developed as an ExtendJ extension. Finally, the concurrent evaluation algorithms for RAGs presented in Paper IV were evaluated in ExtendJ.

4.3 Compilation Passes

Compilers are typically organized into multiple passes consisting of various static analyses and AST transformations. Passes normally need to run in a specific order because some passes depend on transformations or analyses done by a previous pass. For example, name analysis is usually one of the first passes because most other analyses require name analysis information. Passes communicate information both through the AST and through other shared data structures like symbol tables and control flow graphs.

ExtendJ differs from conventional compilers by having relatively few passes, and by using practically no data structures apart from the AST. In fact, ExtendJ has only three passes: *parsing*, *error checking*, and *code generation*, as illustrated in Figure 11. Error checking is done by evaluating the collection attribute *problems* on each source file in the program. The *problems* attribute contains error messages and warnings for the current program. If there were error messages found, then ExtendJ proceeds by generating the necessary Java bytecode in the code generation pass.

The reason there are so few passes in ExtendJ is that attributes do most of the work: when the *problems* attribute is evaluated, it automatically causes the evaluation of all attributes it depends on. Code generation similarly relies on additional attributes, which are evaluated as needed. There are cyclic dependencies between a small number of attributes, which are solved by using fixpoint iteration (with circular attributes). Because attributes are declarative, we do not have to consider the ordering of attributes during evaluation. Any valid attribute evaluation order[24] always gives the same result.

An alternative viewpoint is to regard each attribute as a kind of mini-pass. In this sense, ExtendJ contains very many interleaved passes.

[24]Meaning a reverse dependency order, of which there are many.

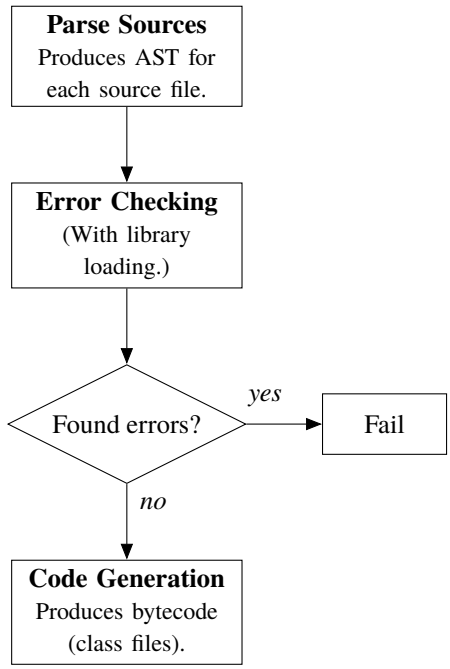


Figure 11: Flowchart for compilation in ExtendJ. Compilation is split into three explicit passes: *parsing*, *error checking*, and *code generation*. ExtendJ uses dynamic loading of library classes, interleaving some parsing (of bytecode and Java source code) with error checking.

4.4 Modular Architecture

ExtendJ is composed of a set of modules supporting different versions of Java. At the base is a Java 1.4 module, upon which Java 5 through 8 modules are added as extensions (each new extension version depending on the previous). The modules consist of JastAdd aspect files, abstract grammar, and separate scanning specification and parsing grammar. For each supported Java version, there are two modules: a frontend module for parsing and semantic analysis, and a backend module for generating Java bytecode.

The following table summarizes the contents of the current modules in ExtendJ, version 8.1.2:

Module	*LOC*	*LOC%*	*AST*	*Classes*	*Attrs*	*Refines*
java4 f	9 129	51%	169	49	797	0
java4 b	4 146		0	27	150	0
java5 f	5 829	27%	64	8	350	53
java5 b	1 248		0	5	46	47
java6 f	21	0%	0	0	0	1
java6 b	17		0	1	0	2
java7 f	1 324	6%	8	3	101	13
java7 b	249		1	2	12	3
java8 f	3 934	16%	22	2	207	38
java8 b	309		0	0	11	5
Σ	26 206		264	97	1 674	162

The *f*/*b* suffix indicates frontend/backend module. *LOC* is the number of lines of aspect code (excluding lines with only comments, spaces, and braces). *AST* is the number of grammar classes added in each module. *Classes* are non-grammar classes and interfaces. *Attrs* is the number of attribute declarations, and *Refines* is the number of replaced attribute equations and methods.[25]

4.5 Abstract Grammar

This section gives a high-level overview of the abstract grammar of ExtendJ, omitting many details which do not impact the rest of the discussion. This section can be skipped if you are not interested in the technical details of the ExtendJ implementation.

The abstract grammar in ExtendJ is not too different from any other Java compiler. Most of the AST class names are derived from the Java specification [JLS7]. A somewhat typical ExtendJ AST is shown in Figure 12.

At the top level, we have the main AST root node, *Program*, which contains multiple compilation units:

$$Program \ ::= \ CompilationUnit*$$

[25]The number of lines of code were counted with a tool based on the lexer from ExtendJ.

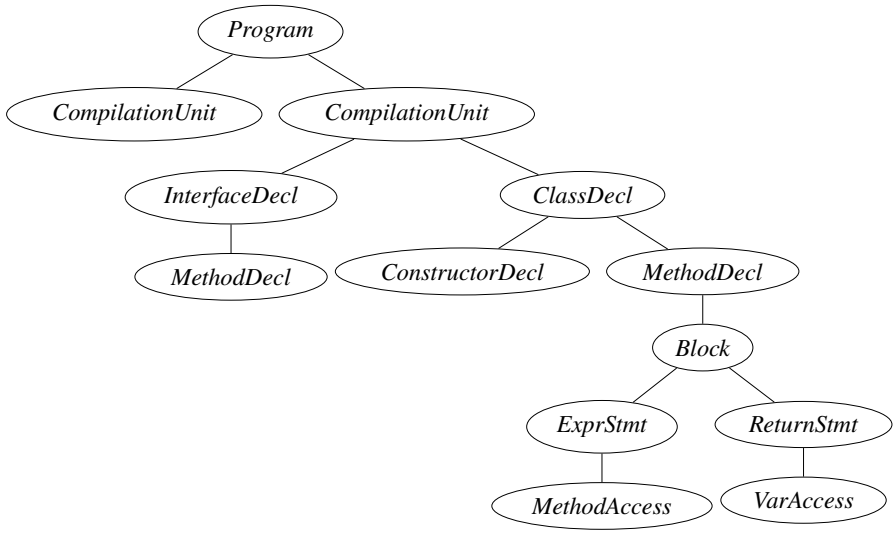


Figure 12: A minimal ExtendJ AST, exhibiting most of the typical high-level structure.

Each compilation unit represents one Java source file, containing a list of type declarations:

$$\mathit{CompilationUnit} \ ::= \ [\,\mathit{PackageDecl}\,]\ \mathit{ImportDecl}*\ \ \mathit{TypeDecl}*$$

Type declarations include, among others, class declarations and interface declarations:

$$\begin{aligned}
\texttt{abstract}\ \mathit{TypeDecl} \ &::= \ \mathit{Modifiers}\ \langle \mathit{ID} \rangle\ \mathit{BodyDecl}* \\
\texttt{abstract}\ \mathit{ReferenceType} &: \mathit{TypeDecl} \\
\mathit{ClassDecl} : \mathit{ReferenceType} \ &::= \ \ldots \\
\mathit{InterfaceDecl} : \mathit{ReferenceType} \ &::= \ \ldots \\
\mathit{EnumDecl} &: \mathit{ClassDecl} \\
\mathit{PrimitiveType} &: \mathit{TypeDecl}
\end{aligned}$$

Each type declaration contains a list of member declarations like methods, fields, constructors, and so on:

$$\begin{aligned}
\texttt{abstract}\ \mathit{BodyDecl} & \\
\texttt{abstract}\ \mathit{MemberDecl} &: \mathit{BodyDecl} \\
\mathit{ConstructorDecl} : \mathit{BodyDecl} \ &::= \ \ldots \\
\mathit{MethodDecl} : \mathit{MemberDecl} \ &::= \ \ldots \\
\mathit{FieldDecl} : \mathit{MemberDecl} \ &::= \ \ldots
\end{aligned}$$

The difference between *BodyDecl* and *MemberDecl* is minor. There are many similar cases where we use multiple levels of abstract superclasses in the full ExtendJ grammar. These are used both for sharing attribute code between similar statements, and for preventing some kinds of constructs occurring in certain places of the tree.

Inside body declarations are statements and expressions. As in most programming languages, methods and constructors have a list of statements, and statements contain expressions. Statements include typical imperative programming language statements such as for-loops, if-statements, switch statements, etc.

```
abstract Stmt
IfStmt : Stmt                ::= Condition:Expr Then:Stmt [Else:Stmt]
WhileStmt : BranchTargetStmt ::= Condition:Expr Stmt
ForStmt : BranchTargetStmt   ::= InitStmt:Stmt* [Condition:Expr]
                                 UpdateStmt:Stmt* Stmt
BreakStmt : Stmt             ::= ⟨Label⟩
ThrowStmt : Stmt             ::= Expr
TryStmt : Stmt               ::= Block CatchClause*
                                 ExceptionHandler:Block
```

The expression grammar is much larger than the statement grammar. Seen from a high level, there are two kinds of expressions:

Access A reference to some entity: a type, variable (or parameter, field), method (a call), `this` pointer, qualified expression (dot).

Non-Access A non-reference expression, e.g., literal values, arithmetic expressions, type casts, etc.

Both access and non-access expressions are subclasses of the *Expr* class, and all access expressions inherit from *Access*.

```
abstract Expr
abstract Access : Expr
```

There are many subtypes of *Access* for the different kinds of access expressions. The following list includes the most important kinds of access expressions:

```
Dot : Access ::= Left:Expr Right:Access
VarAccess : Access ::= ⟨ID⟩
MethodAccess : Access ::= ⟨ID⟩ Arg:Expr*
ConstructorAccess : Access ::= ⟨ID⟩ Arg:Expr*
TypeAccess : Access ::= ⟨Package⟩ ⟨ID⟩
ThisAccess : Access
```

SuperAccess : *Access*
PackageAccess : *Access* ::= ⟨*Package*⟩
ArrayAccess : *Access* ::= *Expr*
ParseName : *Access*

There are many different subtypes of *TypeAccess*, for different kinds of references to types including polymorphic type accesses. The *ParseName* access is a special kind of ambiguous name that is reclassified automatically by *syntactic classification*, as described in Section 4.7.

Continuing with the expression grammar, we have non-access expressions. These types of expressions follow a typical imperative programming language expression grammar:

`abstract` *AssignExpr* : *Expr* ::= *Dest:Expr Source:Expr*
`abstract` *PrimaryExpr* : *Expr*
ParExpr : *PrimaryExpr* ::= *Expr*
`abstract` *Binary* : *Expr* ::= *LeftOperand:Expr RightOperand:Expr*
`abstract` *ArithmeticExpr* : *Binary*
`abstract` *AdditiveExpr* : *ArithmeticExpr*
AddExpr : *AdditiveExpr*
SubExpr : *AdditiveExpr*

The *AdditiveExpr* class is an abstract class used solely for the convenience of being able to declare a shared attribute on a single superclass: there are several attributes that are common between additive expressions and thus the *AdditiveExpr* class lets us specify only one attribute equation for all of them.

A special kind of statement, named *ExprStmt*, links expressions and statements. Its purpose is to allow single expressions to be treated as statements. This is necessary, e.g., for method calls:

ExprStmt : *Stmt* ::= *Expr*

4.6 Attributes

The attributes in ExtendJ can roughly be grouped into these categories:

Semantic Attributes Attributes that implement a distinct part of the Java specification [JLS7]. Some of these attributes follow the specification closely, and can almost be read out loud as if part of the specification. Others diverge a bit more from the specification or are simply less readable than the corresponding natural language specification. An example of semantic attributes are the attributes that implement definite assignment analysis [JLS7, §16]: they closely follow the specification and the equations are often very readable.

Utility Attributes Attributes that are mainly concerned with collecting or organizing auxiliary information for the semantic attributes. A typical example is the *TypeDecl.supertypes* attribute which collects all supertypes of the receiver type. Utility attributes are often useful in extensions.

Transformation Attributes Higher-order attributes that transform part of the AST, either to implement a transformation required by the Java specification, or to simplify the work for other attributes. Examples include enum constructor transformation, implicit diamond access methods, etc.

The attributes in ExtendJ are divided into aspect files based on what Java version and which kind of analysis they implement. The next sections present some of the most important attributes in ExtendJ, and give an overview of the major static analyses in ExtendJ.

4.7 Name Analysis

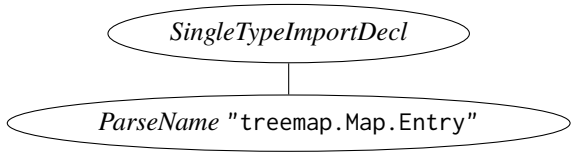
The primary name analysis tasks in Java compilers are binding uses of named entities to corresponding declarations [JLS7, §6], and syntactic classification [JLS7, §6.5.1].

The purpose of syntactic classification is to determine the meaning of all names in the program, classifying them as variable names, type names, method names, or package names. Consider the following import declaration:

```
import treemap.Map.Entry;
```

The meaning of `Map` is ambiguous here: it could refer to a package `treemap.Map`, or a class inside the `treemap` package.

The parser produces a *ParseName* node when a name is used in a context where the parser is not able to unambiguously decide which kind of name it is. For example, the ExtendJ parser builds the following AST subtree from the above import statement:



Syntactic classification is performed in ExtendJ by using the JastAdd rewrite mechanism: a rewrite rule for the *ParseName* class automatically transforms each *ParseName* node into an appropriate *Access* when it is first referenced from an attribute [EH04]. The *ParseName* node is present in the parsed AST but all attributes are oblivious to it because they can only observe the syntactically classified result.[26]

Name analysis is needed to find matching declarations for variable and type names. Name analysis is done with the idiomatic lookup pattern for JastAdd, in which inherited attributes are used for looking up names from enclosing scopes [Hed11]. The main attributes used for finding declarations via name analysis are:

syn *Variable*	*VarAccess.decl*
syn *MethodDecl*	*MethodAccess.decl*
syn *ConstructorDecl*	*ConstructorAccess.decl*
syn *ConstructorDecl*	*ClassInstanceExpr.decl*
syn *TypeDecl*	*TypeAccess.decl*

These attributes find matching declarations for different kinds of named entities. The *Variable* type is an interface used for local variables, fields, and parameter declarations. Method and constructor lookup can involve overload resolution, shadowing, and type inference. The *decl* attributes point to a single declaration node, but if the declaration is undefined or ambiguous it will point to a singleton representing an unknown declaration, using the null object pattern.

The *decl* attributes are implemented by lower-level attributes which are parameterized by the name being looked up (except for constructor lookup where the name is not needed). These lower-level lookup attributes are:

inh *Set⟨Variable⟩ Expr.lookupVariable*(*String name*)

inh *Set⟨MethodDecl⟩ Expr.lookupMethod*(*String name*)

inh *Set⟨ConstructorDecl⟩ ConstructorAccess.lookupConstructor*

inh *TypeDecl Expr.lookupType*(*String packageName, String typeName*)

As a part of type analysis, typenames are resolved by name lookup via the *lookupType* attribute. The type name may match an existing type declaration somewhere in the program AST, or it can match a library class (either in a user library or system library included with the Java runtime), or else it is an unknown type. For unknown types, the *UnknownType* singleton is returned.

It would be wasteful to load all available libraries before starting compilation, so instead ExtendJ uses demand-loading of libraries. Only the library types needed for the current compilation task will actually be loaded. This dynamic library loading is implemented by parameterized higher-order attributes: the attribute

syn *CompilationUnit Program.getLibCompilationUnit*(*String name*)

is responsible for loading a library type by its fully qualified type name. This higher-order attribute builds an implicit part of the AST, so that the loaded type becomes part of the full program AST once loaded and subsequent accesses to the same library type reuse the already-loaded type.

[26] If syntactic classification fails, the *ParseName* is replaced by an *AmbiguousAccess*.

4.8 Type Analysis

ExtendJ represents each Java type by a subclass of the AST class *TypeDecl*. For example, Java classes are represented by the AST class *ClassDecl*, interfaces by *InterfaceDecl*, and enum types by *EnumDecl*. Each type in a Java program has a corresponding *TypeDecl* node somewhere in the AST. Even primitive types like `int` and `boolean` have a corresponding type declaration node (reified as higher-order attributes of type *PrimitiveType*) and behave for the most part like any other kind of type declaration. In this system, user types are first-class citizens and, for the most part, indistinguishable from library classes.

In most places where a type is used in a Java program, it is referred to by name, as a *TypeAccess*. When analyzing Java code, we often need to look up the type declaration for a given name in order to compare types or query any property of some named type. To this end, name lookups are used to find a matching type declaration for typenames. In particular, the attribute *TypeAccess.decl* and *Expr.lookupType* are used for finding the type declaration matching a typename. For parameterized types, like `List<Integer>`, the particular parameterization used must be instantiated. This process is described in more detail in Section 4.11.

The attribute for computing the type of a general expression is

syn *TypeDecl Expr.type*

For variables (and similarly for fields and parameter uses), the type is computed by looking up the variable declaration. The declared type in the variable declaration is then used to compute a reference to the relevant *TypeDecl* node. For method calls, the type is the declared return type of the matching method declaration. For class instance expressions the type is the same as the constructed class. The type attribute is in some cases straight-forwardly implemented, but there are also challenges that occur due to type inference.

Another important task of type analysis is *type checking*: ensuring that all expressions are correctly typed and match the expected type from the expression context. Type checking relies heavily on the attribute

syn boolean *TypeDecl.subtype*(*TypeDecl type*)

which determines if the receiver type is a subtype of the argument type. The subtype attribute is implemented with double dispatch to handle all combinations of different kinds of types [EH07b]. For example, the subtype attribute for classes and interfaces looks like this:

```
syn boolean     TypeDecl.subtype(TypeDecl type) circular(true)
eq              ClassDecl.subtype(TypeDecl type) =
                        type.supertypeClassDecl(this)
eq              InterfaceDecl.subtype(TypeDecl type) =
                        type.supertypeInterfaceDecl(this)
```

For each combination of two kinds of types X and Y, a corresponding set of attributes *TypeDecl.supertypeX(X)* and *TypeDecl.supertypeY(Y)* are needed. As an example, the equation for testing if a class declaration is a supertype of an interface declaration looks like this:

$$\textbf{\textit{eq}}\ \textit{ClassDecl.supertypeInterfaceDecl(InterfaceDecl type)} = \textit{isObject}$$

This attribute equation follows directly from the Java specification: the `Object` class (from package `java.lang`) is the only class which is a supertype to any interface, and it is a supertype to all interfaces.

While double dispatch enables us to extend the type system with new kinds of types without editing all *subtype* equations, it still can lead to a lot of work due to the need for many additional equations. In principle, the double dispatch pattern requires a pair of equations for each pair in the Cartesian product of all kinds of types in the language.

4.9 Method Call Resolution

One of the most demanding parts of the Java specification, in terms of implementation effort required, is method call resolution [JLS7, §15.12.2]. Method call resolution is used for finding which method declaration a method call refers to. If we gloss over the details, we can illustrate the method resolution process as the following algorithm:

1. Determine the receiver type for instance method calls. For qualified method calls, the receiver type is the type of the qualifying expression. For unqualified method calls, the receiver type is the enclosing class at the call site.
2. Find matching method declarations based on the called method name. For instance-method calls, matching methods are searched for in the receiver type, otherwise the members of the enclosing class and imported static methods are searched for matching declarations.
3. Filter candidate method declarations based on declaration visibility rules.
4. Filter out overridden method declarations from candidate methods.
5. Select the most specific method declaration (if one exists) based on the actual argument types used in the call. Take into account variable arity, type parameters, and type inference.

A key attribute in the method resolution algorithm is

$$\textbf{\textit{syn}}\ \textit{Set}\langle\textit{MethodDecl}\rangle\ \textit{MethodAccess.maxSpecific}(\textit{candidates})$$

which implements the algorithm for determining the most specific method, a very precisely defined concept from the Java specification [JLS7, §15.12.2.5]. The equation for this attribute was not too complicated in the Java 1.4 version of ExtendJ, but

in the Java 5 extension it indirectly uses the complicated type inference system via the attribute

$$\textbf{\textit{syn}}\ \mathit{Set}\langle \mathit{MethodDecl}\rangle\ \mathit{MethodAccess.potentiallyApplicable}(\mathit{candidates})$$

For generic candidate methods, *potentiallyApplicable* uses a utility attribute to infer the type arguments for the current method call based on its context.

4.10 Control Flow and Dataflow Analysis

The Java specification requires several control flow and dataflow analyses, including the following:

- exception handling checks,
- unreachable statements,
- missing returns,
- definite assignment
- finally handlers (code generation).

The following text gives some examples of how these analyses work in ExtendJ.

Exception Handling Checks

All *checked* exceptions must be caught by an enclosing `try`-statement, or else declared to be thrown [JLS7, §11.2]. This requirement is handled in ExtendJ by *handlesException*, an inherited parameterized attribute that determines if the argument exception type is handled by the surrounding context (an enclosing `try`-statement, for example).

A separate exception handling check ensures that `try`-statements with a `catch` clause enclose a statement that can actually throw the caught exception type. For this analysis, ExtendJ uses the synthesized parameterized attribute *reachedException*. The attribute determines if the argument exception type can be thrown from the receiver statement (e.g., a block or method call).

Unreachable Statements and Missing Returns

The aforementioned exception handling check for `catch` clauses implements a small part of the more general requirements for unreachable statements analysis in the Java specification [JLS7, §14.21]. The purpose of the specification is to disallow many, but not all, kinds of unreachable code. In ExtendJ, most of the unreachable statement analysis is done with an inherited attribute named *reachable*. As the name implies, the attribute determines if the receiver node is reachable in its context (this is necessarily an imprecise analysis).

The unreachable statement analysis is used in another kind of analysis: checking that all paths through a non-`void` method end with a `return` statement (or throw an exception). This analysis uses an attribute on statements named *canCompleteNormally*. The equations for this attribute, for the most part, rely on the *reachable* attribute. For example, here is the equation for *canCompleteNormally* on `if`-statements:

$$
\begin{aligned}
&\textbf{\textit{eq}}\ \mathit{IfStmt.canCompleteNormally} = \\
&\quad (\mathit{reachable} \wedge \mathit{Else} = \textbf{\textit{nil}}) \\
&\quad \vee\ \mathit{Then.canCompleteNormally} \\
&\quad \vee\ (\mathit{Else} \neq \textbf{\textit{nil}} \wedge \mathit{Else.canCompleteNormally})
\end{aligned}
$$

This attribute equation is derived directly from the Java specification, which intentionally treats `if`-statements with constant `true` conditions as if they can be `false`.[27]

Definite Assignment

Definite assignment ensures that all local variables are initialized before use [JLS7, §16]. The central attributes for definite assignment analysis in ExtendJ are the parameterized attributes *assignedAfter*, *assignedBefore*, *unassignedAfter*, and *unassignedBefore*. These attributes determine if the argument variable is definitely (un)assigned before/after the receiver statement or expression.

The Java specification defines the meaning of definitely assigned and definitely unassigned by many rules for all kinds of expressions and statements. For example, one rule in the Java 7 specification states that a variable v is definitely assigned after a variable declaration statement that contains no initializer if v is definitely assigned before the declaration (paraphrased). This rule, combined with a few other rules, is implemented in ExtendJ by the following attribute equation:

$$
\begin{aligned}
&\textbf{\textit{eq}}\ \mathit{Declarator.assignedAfter}(\mathit{Variable}\ v) = \\
&\begin{cases}
\mathit{Init} \neq \textbf{\textit{nil}}, & \text{if } v = \textbf{\textit{this}} \\
\begin{cases}
\mathit{assignedBefore}(v), & \text{if } \mathit{Init} = \textbf{\textit{nil}} \\
\mathit{Init.assignedAfter}(v), & \text{if } \mathit{Init} \neq \textbf{\textit{nil}}
\end{cases}, & \text{otherwise}
\end{cases}
\end{aligned}
$$

In the first case, the *Declarator* declares a variable with the same name as v, and v is only definitely assigned after the declaration if there is an initializer ($\mathit{Init} \neq \textbf{\textit{nil}}$).

[27] See the examples at the end of §14.21 in the Java 7 specification.

The *assignedBefore*(v) case implements the rule described above. The remaining case comes from another rule in the specification.

The equation above is one of the smaller definite assignment equations; some are much larger, although they follow the Java specification closely. The definite assignment attributes can circularly depend on themselves when loop statements are involved. This is discussed briefly in the Java specification [JLS7, §16]. Interestingly, the definite assignment attributes have remained mostly untouched since the Java 1.4 version of ExtendJ, with few additions for later Java versions.

Finally Handlers

Finally handlers are needed in the generated bytecode for all control-flow paths out of a `try`-statement. Even though there is at most one `finally` block for each `try`-statement, the statement may require multiple copies of the `finally` block to handle if the exception is re-thrown, or if the `try`-block executed a return statement. To illustrate, the following two pieces of code are equivalent:

```
try {
  if (m()) {

    return;
  }
} catch(Exception e) {
  print("y");

  throw e;
} finally {
  doLast();
}
```

```
try {
  if (m()) {
    doLast();
    return;
  }
} catch(Exception e) {
  print("y");
  doLast();
  throw e;
}
doLast();
```

Notice that the body of the `finally` block (highlighted gray) is implicitly duplicated to three places in the code on the right. The implicit `finally` blocks are reified with a higher-order attribute named *ntaFinallyBlock*, which duplicates the code from the `finally` block.

4.11 Representation of Polymorphic Types

Java has parametric polymorphism in the form of generic classes and methods with *type parameters*. Type arguments can be specified at the use-site of a parametric type or method. Alternatively, type arguments can be inferred in certain contexts.

In ExtendJ, each generic type is represented by a *GenericTypeDecl*. Because Java is nominally typed, each instantiation of a generic type (a class or interface) needs to be reified in the compiler. In ExtendJ, this reification is done by constructing a type declaration node in the AST by using a higher-order attribute. This higher-order

attribute is a parameterized attribute where the parameter is the type argument list for the specific parameterized type to be reified.

The higher-order attribute for reifying a parameterized type has the following declaration:[28]

syn nta *TypeDecl GenericTypeDecl.lookupParTypeDecl*(*typeArgs*)

The declaration constructed by the attribute is a shallow copy of the generic class, containing only the externally visible API in the form of member signatures. Member signatures are needed for type checking any use of the class, but the specific parameterization of the class is not used in code generation and so the code in the methods (and field initializers, instance initializers, etc.) can be discarded.

Previously, type variables were substituted for their corresponding type argument when building a parameterized type in the higher-order attribute [EH07b]. However, this can lead to problems. For generic members, this process prevents recursive type substitutions. During the implementation for Paper IV, I refactored this so that the original type variables are kept unmodified and substitution is instead handled by adding new equations for the type lookup attributes on the parameterized types. This solved the problem of recursive type substitutions in generic methods, and also seems to have sped up parallel compilation (not specifically evaluated).

4.12 Type Inference and Generic Types

Type inference was added to the Java language in Java 5, together with generic types and methods. Type inference can be used to compute the type parameters for generic method invocations if they are omitted. For example, the static method `Collections.emptyList()` is generic and we can call it with explicit type parameters like this:

```
List<String> list = Collections.<String>emptyList();
```

In this case it is also possible to omit the type parameter `String` and instead rely on type inference to compute the right type:

```
List<String> list = Collections.emptyList();
```

Type inference was further extended in Java 7, with the diamond expression, and in Java 8 to make anonymous functions easier to use. In Java 8, we can write a lambda expression (anonymous function) without specifying the types of the formal parameters, e.g.

```
x -> 3*x
```

[28]The type of the parameter has been left out to make it fit in one line here. The ***nta*** keyword comes from Non-Terminal Attribute, another name for higher-order attributes.

Type inference can be implemented in several ways. For example, by using the well-known unification algorithm [Pie02, p. 326; MS18, p. 19; Ses17, p. 102]. In ExtendJ, type inference works a bit differently: we gather a set of subtype and supertype constraints from the context. The constraints are solved by finding the greatest lower bound or least upper bound of constraint types in the type hierarchy. Constraints are solved one at a time without backtracking, and at the end there is either a most general solution or no solution. With improved type inference, introduced in Java 8 [JLS8, §18], the constraint systems become more complicated, by introducing new circular dependencies in type inference that were not previously possible. Additionally, type inference can occur simultaneously at different parts of a single expression, whereas previously sub-expression types were inferred one at a time and bottom-up.

4.13 Code Generation

The final pass in ExtendJ generates bytecode for the Java Virtual Machine (JVM) to run. The generated bytecode is unoptimized. Like OpenJDK, ExtendJ relies on the JVM to optimize the bytecode during runtime. The JVM usually does a good job of runtime optimization, resulting in high performance.

The Java bytecode consists of instructions for a stack-based virtual machine. Most of the bytecode generation is straightforward, except for one part: since Java 7, the JVM requires so-called *stack map frames* in the bytecode. Stack map frames describe the possible types of stack and local variables at each point where control flow merges in the bytecode instructions. The stack map frames are used for type checking the bytecode during runtime. Previous to Java 7, the JVM automatically inferred all stack map frames. However, inferring these stack map frames has a cost. To avoid that cost the Java language designers decided that stack map frames should instead be computed by the compiler and output alongside the Java bytecode. When the stack map frames are included with bytecode, the JVM just has to perform the computationally simpler task of type checking the bytecode against the provided stack map frames.

During my work for this thesis I implemented the stack map frames generation in ExtendJ, so that it can output Java 7+ bytecode.

5 Contributions

In this section I describe my key contributions in this dissertation. To give a quick overview, the main contributions are:

- An extension of the ExtendJ compiler to Java 7 (Paper I), with two new methods for extending a programming language with higher-order attributes.

  The resulting extended compiler remains comparatively fast and the implementation was much smaller than the reference compiler for Java.

- The multiplicities Java language extension implemented as an ExtendJ extension (Paper II).

The main contribution in this paper is the case study of a new language mechanism, multiplicities. The implementation is an extension of the Java type system with new code generation for handling multiplicities.

- An automated algorithm for incremental regression testing based on program extractions (Paper III). The implementation is an efficient dependency graph extraction method based on the ExtendJ compiler.

- Algorithms supporting concurrent RAGs. In particular, a new algorithm for concurrent fixpoint attribute evaluation (Paper IV).

  These algorithms enable automatic parallelization of static analyses built with RAGs. By parallelizing the ExtendJ compiler, Java error checking was sped up by about a factor of two. Additionally, our evaluation showed reduced attribute response time in an incremental evaluation benchmark, from seconds to below a millisecond.

- Correctness proofs for the concurrent RAG algorithms (Paper IV).

  Correctness is of paramount importance in concurrent settings, not least due to the well-known difficulties of debugging and reproducibly testing flaws in concurrent code. The correctness proofs are thus an essential contribution for the new concurrent RAG algorithms. The implementation of the concurrent RAG algorithms in JastAdd could of course still contain errors, irrespective of the correctness of the algorithms themselves. However, the implementation follows the algorithms closely so that the implementation is easier to manually verify.

- Simplification of circular attributes in RAGs (Paper IV).

  This is an important relaxation of the requirements for specifying circular attributes which I discovered while working on the concurrent attribute algorithms.

- ExtendJ improvements.

  One of the most important improvements I have made to the ExtendJ compiler was to remove side effects in the frontend of ExtendJ, to enable parallel error checking for the evaluation of Paper IV. I have implemented several additional redesigns in the compiler in order to improve correctness and/or to simplify the design and make the compiler more usable by others.

The following sections describe each contribution in more detail.

5.1 Extension of ExtendJ to Java 7

In Paper I, we describe the design of the Java 7 extension to ExtendJ. The Java 7 extension includes the following main additions to the compiler: try-with-resources, diamond access (type inference), and strings in switch.

Try-With Resources

Try-With Resources (TWR) was the largest language change in Java 7, adding resource declarations in try-statements. Each resource declaration contains an initializing expression which opens a resource. At the end of the TWR statement, the resource is closed. For example:

```
try (OutputStream fout = new FileOutputStream("x");
     PrintStream out = new PrintStream(fout)) {
  out.println("Solving old problems in new ways.");
  ...
}
```

This TWR statement uses two resources: a `FileInputStream` and a `PrintStream`. When control leaves the try-statement, both resources are automatically closed.

The main challenge in implementing TWR statements in ExtendJ was code generation. There are several special cases that must be handled depending on how many resources are used inside the resource declaration part of the statement. Each resource declaration should be initialized in order, and each initialization may be interrupted by an exception. If an initialization is interrupted, then all previously initialized resources must be closed.

The implementation described in Paper I elegantly solves code generation for TWR resources: we use higher-order attributes to unfold a TWR statement into simpler statements that each have only a single resource declaration. This greatly reduces the number of different cases that need to be handled with regard to handling exceptions during resource initialization. The unfolding of TWR statements into simpler statements is a kind of desugaring, but instead of desugaring to elementary language features we desugar to a new language feature which is just a simplified form of the full construct.

Diamond Access

The *Diamond Access* is a new way of using type inference to create instances of a generic class. For example, the following statement

```
List<String> list = new ArrayList<String>();
```

can be replaced by

```
List<String> list = new ArrayList<>();
```

The ExtendJ implementation of diamond access reuses generic method type inference which existed in the compiler for Java 5. Using higher-order attributes, a synthetic method invocation is created which corresponds to the class instance expression.

Then, for each accessible constructor for the current class, a synthetic method is created with type parameters matching the class type parameters. The synthetic method call is then used to infer type arguments for the method call, which gives directly the type arguments for the diamond access.

Strings in Switch

The implementation of strings in switch was straightforward. I extended the bytecode generation for the case when the `switch` argument is a string by using the `refine` mechanism described in Section 3.3. Type analysis for strings in switch was similarly extended by refining the attribute *SwitchStmt.type*.

Evaluation

The empirical evaluation of the Java 7 extension showed that the Java 7 version of ExtendJ was only 52% the size of the corresponding OpenJDK compiler, the reference Java compiler. ExtendJ has always been slower than OpenJDK, but the compile time remained reasonably close in comparison.

The compile time evaluation was done by measuring total compilation time across seven Java applications of varying sizes, between 5 and 87 kilo lines of code. Compile time was measured both for cold-start and steady-state compilation. In the cold-start case, ExtendJ compile time was within a factor 1.6 from that of OpenJDK. In the steady-state case, ExtendJ was at most 3.3 times slower than OpenJDK.

5.2 Multiplicities Implementation

Paper II is a case study in programming with *Multiplicities* as a Java language extension. The concept of multiplicities was invented by Friedrich Steimann, the main author. I implemented these concepts as an extension to the ExtendJ compiler.

Multiplicities allow the programmer to easily change how many objects a reference can relate to, either to-one, or to-many. For example, a regular Java program may contain the following code fragment to keep track of a single account:

```
Account acc = new Account(name);
acc.export();
```

With multiplicities, the programmer may easily track multiple account objects by changing adding a new modifier to change the multiplicity type of `acc` to **any**:

```
any Account acc = new Account(name);
acc += otherAccount();
acc.export();
```

This code will export all accounts added to the `acc` reference, which may be many.

The most novel part of the multiplicities implementation is the extension of the type system in ExtendJ to support the new multiplicity types. I developed the new typing rules in collaboration with Steimann, based on the original multiplicities concept.

The implementation of multiplicities as an ExtendJ extension is straightforward. The type system was extended by adding new kinds of types for the new multiplicities, with new equations extending the double dispatch framework discussed in Section 4.8. Code generation was extended by adding new specialized bytecode generation for some statements and expressions involving non-bare multiplicities.

We evaluated the performance of code compiled with the multiplicities extension, to see if code that used multiplicities was slower than the corresponding code without multiplicities. To this end, we modified a version of JUnit, a popular Java unit testing framework, to use multiplicities in many places. The modified version of JUnit was compiled and compared against the unmodified version of JUnit. The performance evaluation showed a very small run-time overhead for the multiplicities-modified version of JUnit compared to the unmodified version. We also evaluated the correctness of the extended version of ExtendJ by verifying that the modified version of JUnit passed all of its own unit tests.

5.3 Safe Regression Test Selection

Regression testing is the practice of running tests after each change to an application in order to ensure that previously implemented, and tested, features do not break (which would cause a regression). Paper III presents a new algorithm for *safe regression test selection*. The goal of safe regression test selection is to reduce the number of tests that are run after each modification to the code while still guaranteeing that no regression will go undetected.

In Paper III we also describe the implementation of our algorithm in a tool based on ExtendJ. The tool is a small and efficient extension for extracting a program dependency graph from Java programs. Our test selection algorithm is then run on the dependency graph to quickly select tests to run.

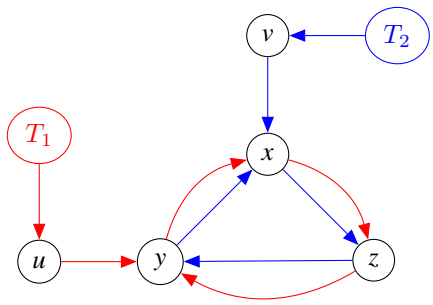
We evaluated the implementation by using real-world Java programs. The commit history of the programs was replayed and the test selection tool was run for each change. We ran all tests selected by our tool and verified that the tool selected all tests that changed result from the previous commit.

The implementation of the test selection tool itself was quite straightforward, but it showed that it was very easy to develop a tool using class dependency graphs based on ExtendJ.

5.4 Concurrent Evaluation of Reference Attribute Grammars

Paper IV describes new algorithms that I developed for concurrent evaluation of RAGs, with correctness proofs (in the extended version of the paper), and implementation in JastAdd. Importantly, the algorithms support circular (fixpoint) attributes, which are needed for many types of static analyses like dataflow analyses and type inference.

The following diagram illustrates two threads, T_1 and T_2, concurrently evaluating mutually dependent attributes:



Arrows illustrate the evaluation control flow, which follows the attribute dependency graph. Thread T_1 starts evaluating attribute u, and T_2 starts at attribute v. The start attributes are independent, but the other attributes are in a dependency cycle. A locking implementation that tries to lock individual attributes would cause a *deadlock* in the current scenario, getting stuck forever due to circular hold-and-wait. The algorithms in Paper IV are lock-free, which ensures that they never deadlock. Furthermore, different evaluation threads can cooperate by sharing partial results. The algorithms enable threads to share results even in fixpoint iterations like the one that arises in the illustrated scenario.

The evaluation in Paper IV is based on ExtendJ. I measured attribute evaluation latency for short-running attributes while long-running attributes were computed in parallel. Latency was reduced from seconds to less than a millisecond. I also measured total error checking time for Java programs and found an approximate twofold speedup with parallel error checking.

The parallelized version of ExtendJ is publicly available on the main ExtendJ source code repository.[29] Parallelization is done by evenly distributing source files between threads. Each thread then performs error checking for its allocated source file, in parallel with other threads. This division of work between threads was implemented by hand.

An alternative way of dividing work between threads in JastAdd compilers is to use automatic parallelization through concurrent collection attributes. Concurrent collection attributes were only briefly mentioned in Paper IV, so I will describe them in more detail here.

JastAdd collection attributes work in two phases [MEH07]. First, in the *survey phase*, the attribute evaluator traverses the AST and looks for contribution statements matching the collection attribute. Second, the *collection phase* collects values from contribution statements. An ordinary collection attribute can be annotated with two annotations to activate parallel evaluation of the attribute: the `@Parallel` annotation makes the value collection phase run in parallel, and the `@ParallelSurvey` annotation makes the survey phase parallelized. The workload is split among worker threads by using fixed-size thread pools.[30]

[29] `https://bitbucket.org/extendj/extendj`

[30] The number of worker threads used is controlled by the `numThreads` option to JastAdd. The number of worker threads can also be changed at runtime via the *ASTState.numThreads* field.

Parallel Performance

Improved run-time performance is possible, but not guaranteed, when parallelizing JastAdd code. The performance results in ExtendJ are especially pleasing given that the Java language has many interdependencies between classes. These dependencies mean that when evaluating an attribute in one source file, it is likely to have several dependencies on attributes in other source files. This leads to redundant computations in parallel attribute evaluation when two or more threads evaluate the same attribute at the same time: at least one thread is then wasting time computing an attribute when it would ideally be left to a single thread.

While it would be more ideal to split attribute evaluation evenly among threads, so that each thread separately works only on attributes from its own source file, this is not practically possible as Java code tends to be highly interconnected.

The degree of parallelization that can be achieved probably varies largely based on language, but we have not specifically evaluated other languages to compare how parallel performance differs with JastAdd for different languages.

5.5 Simplification of Circular Attributes

Previous work on circular attributes defined them with the assumption that all attributes in a dependency cycle are treated equally, with fixpoint iteration [Far86; Jon90]. This led to the requirement for circular attributes in JastAdd to be annotated with the *circular* keyword, and that all attributes which could be in a dependency cycle must be annotated as such [MH07, p. 27]. While working on the concurrent circular attribute algorithm, I realized that this condition is needlessly strict. It is possible to loosen the requirement to the following: *at least one* of the attributes in *each possible* dependency cycle is annotated as circular (and thus evaluated with fixpoint iteration).[31]

This relaxed requirement for circular attributes makes the task of writing attribute grammars much simpler. Static analysis extensions and language extensions could often introduce circularity by linking previously non-circular attributes. With the relaxed circular evaluation, such extensions only need to annotate their circularity-causing attributes for fixpoint iteration. Additionally, this saves some extra memory overhead needed in the fixpoint iteration mechanism for circular attributes.

5.6 ExtendJ Improvements

This section describes some of the larger improvements to ExtendJ that I implemented during my thesis work. These improvements are not described in the included papers. The purpose of the changes was either to fix correctness problems, or to simplify the compiler design.

The most important change to ExtendJ was to remove all extant side effects from the frontend so that error checking could be parallelized for the evaluation in Paper IV. The side effects that I removed were of three different kinds:

[31] The relaxed requirements for circular attributes do require that memoization is delayed for attributes that are part of a circular attribute evaluation, but which are not annotated as circular.

Imperative tree transformations Modifying the AST is not safe after any attribute has been evaluated. This is because attribute values are derived from the AST, and any change in the AST can affect previously memoized attribute values. Modifications to the AST are safe if they are done before any attribute is evaluated, for example during parsing. In ExtendJ, some transformations were done after parsing was finished, between attribute evaluations, and this caused errors in concurrent compilation. These side effects were replaced by using higher-order attributes to safely compute the transformation.

Non-pure attributes As previously discussed, attributes must be observationally pure. JastAdd does not yet have a mechanism for checking that attributes are pure, however, and side effects can be introduced easily by accident. In ExtendJ, there were several attributes which had unintentional side effects.

A common example of an unintentional side effects in ExtendJ were attributes that modified a mutable data structure returned by another attribute. Mutable data structures should be used carefully: it is not safe to modify a data structure returned by some attribute because that same data structure may have been memoized. Modifying the data structure is then equivalent to modifying the memoized value of an attribute, which is not safe.

Non-fresh higher-order attributes Higher-order attributes must build fresh subtrees. However, there is currently no check for this requirement and it is surprisingly easy to accidentally break the requirement. Non-fresh higher-order attributes were used in a few places in ExtendJ, unintentionally.

The following sub-sections describe some of the other redesigns that I implemented in ExtendJ.

Desugaring Multiple Declarations

Java allows variable (and field) declarations that declare multiple names at once (individual name declarations are referred to as *declarators*). For example, this statement:

```
int a, b[2] = { 1, 2 };
```

is equivalent to

```
int a;
int b[2] = { 1, 2 };
```

Previously, ExtendJ transformed programs using the former pattern into the equivalent desugared form with single-variable declarations. This was done by using a deprecated JastAdd feature called *list rewrite*. List rewrites were problematic for a few reasons. There were doubts about the correctness of the list rewrite implementation in JastAdd, and they caused large runtime overhead when accessing list items in certain ways.

The reason a multi-declaration transformation was used in ExtendJ is because the different variables in a multi-declaration can have different types. To simplify the

analysis of multi-declarations in the compiler, we need an easy way to access the type of a single variable inside a multi-declaration. Without transforming the AST, this can be accomplished in a clean way by using higher-order attributes (HOAs). Here is the relevant part of the abstract grammar for variable declarations:

$$\begin{aligned} \mathit{VarDeclStmt} : \mathit{Stmt} \quad &::= \mathit{Modifiers}\ \mathit{TypeAccess{:}Access} \\ &\quad\ \ \mathit{Declarator{:}VariableDeclarator}* \\ \texttt{abstract}\ \mathit{Declarator} : \mathit{ASTNode} \quad &::= \langle \mathit{ID} \rangle\ \mathit{Dims}*\ [\mathit{Init{:}Expr}] \\ \mathit{VariableDeclarator} : \mathit{Declarator} & \\ \mathit{FieldDeclarator} : \mathit{Declarator} & \end{aligned}$$

The two subclasses of *Declarator* are used to distinguish local variables from fields. The context of the declaration also makes the difference clear, but having a subclass for each makes it slightly easier to specialize some attributes for each case.

I added the following HOA to compute the type of each variable declarator:

$$\begin{aligned} &\textbf{\textit{syn nta}}\ \mathit{Access}\ \mathit{Declarator.TypeAccess} = \\ &\quad \mathit{treeCopy}(\mathit{declarationType}).\mathit{addArrayDims}(\mathit{Dims}) \end{aligned}$$

The *declarationType* attribute is a helper attribute which gives a reference to the enclosing variable declaration type (*TypeAccess:Access* in *VarDeclStmt*). The *treeCopy* function is used to clone the original type access, which is necessary in order to build a fresh tree for the HOA. The *addArrayDims* attribute is a helper attribute to include any necessary array dimensions.

An advantage of using HOAs to compute single variable types instead of using a rewrite to transform the AST is that we retain the source AST, making it easier to pretty-print the original code or to report errors with precise source locations.

Reifying Implicit Constructs

Java compilers use many implicit constructs which the programmer never sees, but which are necessary for generating correct Java bytecode. Examples of implicit constructs include, among others,

Enum switch maps Implicit classes are generated with a field `$SwitchMap$` that is used in bytecode for switch statements with enum type arguments.

Accessor methods Implicit accessor methods are needed for all cases where a class accesses a non-public field of an inner class. These are needed to bridge a gap between the Java source language and the Java bytecode language which does not have inner classes.

Bridge methods When generics are erased in bytecode generation, it may lead to overriding methods that no longer override the method of a superclass. A bridge method is implicitly generated to recover the intended overriding behaviour.

Enclosing and super references Constructors are automatically augmented with an implicit parameter for the superclass reference. Inner classes receive an additional implicit parameter for the enclosing class reference.

Previously, ExtendJ reified the above implicit constructs by using imperative AST modifications during a transformation pass. The transformation pass was not cleanly separated from static analyses (due in part to uses of attributes in the transformations), which caused problems when evaluating attributes which depended on transformed AST structures. I replaced these imperative transformations by higher-order attributes. All implicit constructs in ExtendJ are now reified by higher-order attributes.

Type Variable Substitution

Generic classes and methods are parameterized by one or more *type variables*. Type variables are replaced by the corresponding *type arguments* in each parameterization of a generic class or method. For instance, consider the following generic class:

```
public class Container<T> {
  private T value;
  public void set(T v) {
    value = v;
  }
  public T get() {
    return value;
  }
}
```

This class has one type parameter: the type variable `T`. We can instantiate the class by writing, e.g., `Container<String>`, which means the type of `Container` where all occurrences of `T` are substituted by `String`. Here, `Container<String>` is called a *parameterization* of `Container`. In ExtendJ, each parameterization of a generic class must be reified as a type declaration, that is, an AST subtree that represents the full type with all its public member declarations. For example, ExtendJ represents the type of `Container<String>` by the following structure:

```
public class Container<String> {
  public void set(String v);
  public String get();
}
```

Notice that the private field and the method bodies were removed. Only the public interface of `Container<String>` is needed to reify the parameterization. As described in Section 4.11, the parameterization is implicitly created by a higher-order attribute. However, we have not yet discussed how the type variable is replaced by `String`. The replacement process is called *type variable substitution*.

Previously, ExtendJ performed type variable substitution by replacing all occurrences of `T` by `String` when creating the parameterization `Container<String>`.

ExtendJ created an AST matching the reified parameterization code shown above. For various reasons, this process did not work perfectly: it led to some low-level problems in type inference with generic methods, and it made parallel evaluation of parameterized types run slowly.

I redesigned type variable substitution by delaying the substitution process until type lookup. Type lookup is the latest possible time we can substitute type variables, and it turns out to work very well for enabling correct type inference in some cases that previously failed because of the too-eager type variable substitution. Additionally, this change improved parallel evaluation performance because it made the higher-order attribute for parameterized types much faster to evaluate and led to fewer threads trying to compute the same attribute instance at the same time.

The central attribute equation which performs type variable substitution in the redesigned implementation looks like this:

$$
\begin{aligned}
&\textbf{\textit{eq}}\ \textit{ParTypeDecl.BodyDecl.lookupType(String name)} = \\
&\quad \textbf{\textit{let}}\ t \coloneqq \textit{Parameterization.substitute(name)}\ \textbf{\textit{in}} \\
&\qquad \begin{cases} t, & \text{if } t \neq \textbf{\textit{nil}} \\ \textit{localLookupType}(\textit{name}), & \text{otherwise} \end{cases}
\end{aligned}
$$

The equation above is slightly abbreviated. The *localLookupType* attribute is used for searching the local scope of the type declaration.

5.7 Related Work

AGs have been an active research topic in the programming language community for many years. Classical AGs, with only synthesized and inherited attributes, were never really used for implementing real-world programming languages.[32] On the other hand, various extended AGs have been used to develop practical implementations of real-world programming languages, for example: Pascal [KHZ82], Ada [Uhl+82], VHDL [FS89], and Oberon2 [Boy96]. More recently, RAGs, another AG extension, have been successful for implementing several languages like Java [EH07b; Wyk+07], Modelica [Åke+10], PROMELA [MW11], Grafchart [TÅJ12], Bloqqi [FH16], and C [Kam+17].

ExtendJ is a Java compiler built with the JastAdd metacompiler [HM03a], and as such has quite different compiler architecture compared to most other conventional compilers, which use tree visitors for most analyses. The following table gives a quick overview of related open source Java compilers and source code analysis frameworks:

[32] That is, languages other than the kind of "toy" languages that are often used in teaching and research articles to demonstrate a handful of programming language concepts.

Name	*Bytecode*	*Analysis*	*Transformation*	*Java Version*	*Active Devel.*	*Implementation*
ableJ	○	●	●	1.4	○	RAG (Silver)
Eclipse JDT	●	●	●	11	●	Java
Error Prone	○	●	◔	11	●	OpenJDK
ExtendJ	●	●	●	8	●	RAG (JastAdd)
OpenJDK	●	●	●	11	●	Java
JavaParser	○	◔	○	11	●	Java
Polyglot	○	●	●	7	●	Java
Spoon	○	●	●	11	●	Eclipse JDT
SugarJ	○	○	●	5	○	SDF/Stratego

Bytecode: bytecode generation is implemented.
Analysis: full static analysis for Java (following the specification).
Transformation: allows modifying the program AST to affect other analyses.
Java Version: latest supported Java version.
Active Devel.: public implementation with functional changes in the past two years.

OpenJDK is the reference implementation of Java, including a compiler, standard library, and virtual machine. The OpenJDK compiler, *javac*, can either be extended by forking and editing the compiler code, or by developing plugins for the compiler (including annotation processors). Plugins are used for compile-time program transformation. It is possible to do many useful things with plugins like annotation processors, but in order to extend the language with new syntax it is necessary to modify javac itself. An advantage of extending javac is that it has near-perfect conformance to the Java specification.

Error Prone uses the Java reflection API for javac, adding static analysis for common bug patterns in Java code. Error Prone supports the addition of custom checks via plugins [Aft+12]. AST transformation with analysis is not supported, instead transformed code is printed as code patches.

The Eclipse Java Development Tools (JDT) contain an incremental Java compiler and a collection of static analysis tools used in the Eclipse editor [JDT]. Eclipse JDT uses a conventional visitor-based compiler architecture.

Spoon is a static analysis framework for Java that uses the Eclipse JDT internally [Paw+16].[33] Spoon provides a transformation framework which can be used to easily construct type-safe syntax transformations.

Polyglot is an extensible compiler framework for Java based on an extensible visitor pattern [NCM03a]. Polyglot provides similar functionality as ExtendJ, and a detailed comparison of the two was given by [AET08a]. In Polyglot, the visitor pattern results in large amounts of boilerplate code and monolithic data structures for some problems which are solved more succinctly in ExtendJ.

[33] The connection to Eclipse JDT is neither mentioned in the documentation or in the paper about Spoon, but can be seen its implementation.

SugarJ is a library-based syntax language extension framework for Java [Erd+11]. SugarJ is implemented in Java, SDF [Hee+89], and Stratego [Vis01]. SugarJ enables language extensions to be imported as libraries in user code [Erd+11].

JavaParser is a Java library for parsing and building ASTs of Java programs, with name analysis provided through an additional library [JavaP].

ableJ is an extensible compiler supporting Java 1.4 and built with RAGs in the Silver metacompiler [Wyk+07]. There seems to be no active work on the project as the latest changes in the past few years appear to be non-functional.

6 Conclusions

This thesis presents my contributions to declarative specification of static program analysis with RAGs. My contributions include new language extensions, static analyses, and tools for the Java language and based on the Java compiler ExtendJ. In developing these extensions, I found new design principles for developing declarative language and analysis extensions with RAGs. For example, I developed a partial desugaring technique with higher-order attributes which I used in the implementation of try-with-resources for Java 7.

My more fundamental contributions to RAGs themselves include concurrent attribute evaluation algorithms with support for circular (fixpoint) attributes. The concurrent evaluation algorithms are presented in Paper IV, with correctness proofs included in the extended technical report version of the paper. Another important contribution to RAGs is a relaxation of the requirements for circular attribute specifications to be well-defined. Previous work on circular attributes required all attributes to be evaluated with fixpoint iteration. In Paper IV, I show how this requirement can be relaxed with a modification of the evaluation algorithm.

Static program analysis with RAGs has many benefits: declarative specification of analyses improves their composability and readability, enabling code reuse and extensibility for evolving programming languages. Some of the benefits of using RAGs, like structure-shy programming, are demonstrated in Section 3. In Section 3.7, I present a pattern for specifying generic tree traversals in a RAG.

With static analyses specified in RAGs, there is an opportunity for optimizing the static analysis to improve run-time performance, by memoizing attributes and parallelizing evaluation. Compilers specified with RAGs can be automatically parallelized using parallel collection attributes which I implemented for the JastAdd metacompiler (see Section 5.4, and briefly mentioned in Paper IV).

The ExtendJ compiler is a main focus in this thesis. ExtendJ was used for developing implementations for the included papers, as well as empirically evaluating the results of the included papers. A contribution in this dissertation is the development of a fully declarative and side effect free version of ExtendJ which was possible to parallelize. This work benefits other programming language research which use ExtendJ to develop and evaluate language extensions for Java.

References

[Aft+12] Edward Aftandilian et al. “Building Useful Program Analysis Tools Using an Extensible Java Compiler”. In: *Source Code Analysis and Manipulation*. IEEE Computer Society, 2012, pp. 14–23.

[Åke+10] Johan Åkesson et al. “Modeling and optimization with Optimica and JModelica.org - Languages and tools for solving large-scale dynamic optimization problems”. In: *Computers & Chemical Engineering* 34.11 (2010), pp. 1737–1749.

[AET08a] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. “Modularity first: a case for mixing AOP and attribute grammars”. In: *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD 2008)*. ACM, 2008, pp. 25–35.

[AH17] Mohammad Reza Azadmanesh and Matthias Hauswirth. “Concept-Driven Generation of Intuitive Explanations of Program Execution for a Visual Tutor”. In: *VISSOFT*. IEEE, 2017, pp. 64–73.

[Bac+60] John W. Backus et al. “Report on the algorithmic language ALGOL 60”. In: *Commun. ACM* 3.5 (1960), pp. 299–314.

[BS96] David F. Bacon and Peter F. Sweeney. “Fast Static Analysis of C++ Virtual Function Calls”. In: *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA ’96), San Jose, California, USA, October 6-10, 1996.* ACM, 1996, pp. 324–341.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. “A Validation of Object-Oriented Design Metrics as Quality Indicators”. In: *IEEE Trans. Software Eng.* 22.10 (1996), pp. 751–761.

[Bin07] David Binkley. “Source Code Analysis: A Road Map”. In: *2007 Future of Software Engineering*. FOSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 104–119.

[Bla+06] S. M. Blackburn et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[Boy96] John T Boyland. *Descriptional composition of compiler components*. Tech. rep. University of California at Berkley, Department of Electrical Engineering and Computer Sciences, 1996.

[BS17] Scott J. H. Buckley and Anthony M. Sloane. “A Formalisation of Parameterised Reference Attribute Grammars”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 139–150.

[Bür15] Christoff Bürger. “Reference attribute grammar controlled graph rewriting: motivation and overview”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2015. ACM, 2015, pp. 89–100.

[Cal+15] Cristiano Calcagno et al. “Moving Fast with Software Verification”. In: *NFM*. Vol. 9058. Lecture Notes in Computer Science. Springer, 2015, pp. 3–11.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. “A Metrics Suite for Object Oriented Design”. In: *IEEE Trans. Software Eng.* 20.6 (1994), pp. 476–493.

[CV11] Alcino Cunha and Joost Visser. “Transformation of structure-shy programs with application to XPath queries and strategic functions”. In: *Sci. Comput. Program.* 76.6 (2011), pp. 516–539.

[CVE] *Common Vulnerabilities and Exposures*. Nov. 2018. URL: `http://cve.mitre.org/about/`.

[DK18] Jan C. Dageförde and Herbert Kuchen. “A constraint-logic object-oriented language”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. ACM, 2018, pp. 1185–1194.

[DC90] G. D. P. Dueck and Gordon V. Cormack. “Modular Attribute Grammars”. In: *Comput. J.* 33.2 (1990), pp. 164–172.

[DMG07] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[JDT] *Eclipse Java development tools (JDT)*. Dec. 2018. URL: `https://projects.eclipse.org/projects/eclipse.jdt`.

[EH04] Torbjörn Ekman and Görel Hedin. “Rewritable Reference Attributed Grammars”. In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 144–169.

[EH07a] Torbjörn Ekman and Görel Hedin. “Pluggable checking and inferencing of nonnull types for Java”. In: *Journal of Object Technology* 6.9 (2007), pp. 455–475.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 1–18.

[EH07c] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26.

[Erd+11] Sebastian Erdweg et al. “SugarJ: library-based syntactic language extensibility”. In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 2011, pp. 391–406.

[ExJorg] *ExtendJ Website*. Feb. 2018. URL: `https://extendj.org`.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. Palo Alto, CA, USA: ACM, 1986, pp. 85–98.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. “Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation”. In: *POPL*. ACM Press, 1992, pp. 223–234.

[FS89] Rodney Farrow and Alec G. Stanculescu. “A VHDL Compiler Based on Attribute Grammar Methodology”. In: *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. ACM, 1989, pp. 120–130.

[FCH15] Niklas Fors, Gustav Cedersjö, and Görel Hedin. “JavaRAG: a Java library for reference attribute grammars”. In: *MODULARITY*. ACM, 2015, pp. 55–67.

[FH16] Niklas Fors and Görel Hedin. “Bloqqi: modular feature-based block diagram programming”. In: *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*. ACM, 2016, pp. 57–73.

[GBE07a] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 57–76.

[HO93] William H. Harrison and Harold Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings.* ACM, 1993, pp. 411–428.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[Hed11] Görel Hedin. “An Introductory Tutorial on JastAdd Attribute Grammars”. In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009. Revised Papers*. Vol. 6491. Lecture Notes in Computer Science. Springer, 2011, pp. 166–200.

[HM03a] Görel Hedin and Eva Magnusson. “JastAdd–an aspect-oriented compiler construction system”. In: *Sci. Comput. Program.* 47.1 (2003), pp. 37–58.

[Hee+89] Jan Heering et al. “The syntax definition formalism SDF - reference manual”. In: *SIGPLAN Notices* 24.11 (1989), pp. 43–75.

[Hog14] Erik Hogeman. “Extending JastAddJ to Java 8”. Master Thesis. Report LU-CS-EX:2014-14. Sweden: Dept. of Computer Science, Lund University, 2014.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1969.

[JavaP] *JavaParser - For processing Java code*. Dec. 2018. URL: `https://javaparser.org/`.

[JLS7] James Gosling et al. *The Java™ Language Specification, Java SE 7 Edition*. Oracle America, Inc., Feb. 2013.

[JLS8] Tim Lindholm et al. *The Java™ Language Specification, Java SE 8 Edition*. Oracle America, Inc., Feb. 2015.

[Jon90] Larry G. Jones. “Efficient Evaluation of Circular Attribute Grammars”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 429–462.

[Jou84] Martin Jourdan. “An Optimal-time Recursive Evaluator for Attribute Grammars”. In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[KA18] Tetsuo Kamina and Tomoyuki Aotani. “Harmonizing Signals and Events with a Lightweight Extension to Java”. In: *Programming Journal* 2.3 (2018), p. 5.

[Kam+17] Ted Kaminski et al. “Reliable and automatic composition of language extensions to C: the ableC extensible language framework”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), 98:1–98:29.

[KHZ82] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator*. Vol. 141. Lecture Notes in Computer Science. Springer, 1982.

[KW94] Uwe Kastens and William M. Waite. “Modularity and Reusability in Attribute Grammars”. In: *Acta Inf.* 31.7 (1994), pp. 601–627.

[KV10] Lennart C.L. Kats and Eelco Visser. “The spoofax language workbench”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA ’10*. ACM Press, 2010.

[Knu68b] Donald E. Knuth. “Semantics of Context-Free Languages”. In: *Mathematical Systems Theory* 2.2 (1968), pp. 127–145.

[KV15] Shriram Krishnamurthi and Jan Vitek. “The Real Software Crisis: Repeatability As a Core Value”. In: *Commun. ACM* 58.3 (Feb. 2015), pp. 34–36.

[Lan74] Bernard Lang. “Deterministic techniques for efficient non-deterministic parsers”. In: *Automata, Languages and Programming*. Springer, 1974, pp. 255–269.

[Lie96] Karl J. Lieberherr. *Adaptive object-oriented software: The demeter method with propagation patterns*. Boston: PWS Publishing Company, 1996.

[MEH07] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. “Extending Attribute Grammars with Collection Attributes–Evaluation and Applications”. In: *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (Source Code Analysis and Manipulation 2007), September 30 - October 1, 2007, Paris, France*. IEEE Computer Society, 2007, pp. 69–80.

[MH07] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars - their evaluation and applications”. In: *Sci. Comput. Program.* 68.1 (2007), pp. 21–37.

[MW11] Yogesh Mali and Eric Van Wyk. “Building Extensible Specifications and Implementations of Promela with AbleP”. In: *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*. Vol. 6823. Lecture Notes in Computer Science. Springer, 2011, pp. 108–125.

[Mar02] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[MS18] Anders Møller and Michael I Schwartzbach. *Static program analysis*. `http://users-cs.au.dk/amoeller/spa/`. 2018.

[NR15] Krishna Narasimhan and Christoph Reichenbach. “Copy and Paste Redeemed”. In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 630–640.

[Nau05] David A. Naumann. “Observational Purity and Encapsulation”. In: *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Vol. 3442. Lecture Notes in Computer Science. Springer, 2005, pp. 190–204.

[NCM03a] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. "Polyglot: An Extensible Compiler Framework for Java". In: *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Vol. 2622. Lecture Notes in Computer Science. Springer, 2003, pp. 138–152.

[Par72] David Lorge Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (1972), pp. 1053–1058.

[Paw+16] Renaud Pawlak et al. "SPOON: A library for implementing analyses and transformations of Java source code". In: *Software: Practice and Experience* 46.9 (2016), pp. 1155–1179.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Sci. Comput. Program.* 74.7 (May 2009), pp. 470–495.

[Ryu16] Sukyoung Ryu. "ThisType for Object-Oriented Languages: From Theory to Practice". In: *ACM Trans. Program. Lang. Syst.* 38.3 (2016), 8:1–8:66.

[Sad+18] Caitlin Sadowski et al. "Lessons from Building Static Analysis Tools at Google". In: *Commun. ACM* 61.4 (Mar. 2018), pp. 58–66.

[SS99] Joao Saraiva and Doaitse Swierstra. "Generic attribute grammars". In: *2nd Workshop on Attribute Grammars and their Applications*. Amsterdam, The Netherlands: INRIA Rocquencourt, 1999, pp. 185–204.

[Ses17] Peter Sestoft. *Programming Language Concepts, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2017.

[Slo09] Anthony M. Sloane. "Lightweight Language Processing in Kiama". In: *GTTSE*. Vol. 6491. Lecture Notes in Computer Science. Springer, 2009, pp. 408–425.

[SH10] Emma Söderberg and Görel Hedin. "Automated Selective Caching for Reference Attribute Grammars". In: *Software Language Engineering - Third International Conference, SLE 2010*. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 2–21.

[SH15] Emma Söderberg and Görel Hedin. "Declarative rewriting through circular nonterminal attributes". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 3–23.

[Söd+13] Emma Söderberg et al. "Extensible intraprocedural flow analysis at the abstract syntax tree level". In: *Science of Computer Programming* 78.10 (2013), pp. 1809–1827.

[Ste06] Friedrich Steimann. “The paradoxical success of aspect-oriented programming”. In: *OOPSLA*. ACM, 2006, pp. 481–497.

[SHU16] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. “Computing repair alternatives for malformed programs using constraint attribute grammars”. In: *OOPSLA*. ACM, 2016, pp. 711–730.

[Tem+10] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345.

[BSD] *The Modified BSD License*. Regents of the University of California, 1999.

[TÅJ12] Alfred Theorin, Karl-Erik Årzén, and Charlotta Johnsson. “Rewriting JGrafchart with Rewritable Reference Attribute Grammars”. In: *Industrial Track of Software Language Engineering* (2012).

[Tur37] Alan M Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[Uhl+82] Jürgen Uhl et al. *An Attribute Grammar for the Semantic Analysis of Ada*. Vol. 139. Lecture Notes in Computer Science. Springer, 1982.

[Vis01] Eelco Visser. “Stratego: A Language for Program Transformation Based on Rewriting Strategies”. In: *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 357–362.

[VSK89a] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI ’89*. Portland, Oregon, USA: ACM, 1989, pp. 131–145.

[Wad98] Philip Wadler. *The Expression Problem*. Posted on the Java Genericity mailing list. 1998.

[Wyk+02] Eric Van Wyk et al. “Forwarding in Attribute Grammars for Modular Language Design”. In: *CC*. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 128–142.

[Wyk+07] Eric Van Wyk et al. “Attribute Grammar-Based Language Extensions for Java”. In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 575–599.

[Wyk+10a] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Sci. Comput. Program.* 75.1-2 (2010), pp. 39–54.

[XPath] “W3C XML Path Language”. In: *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014, p. 2337.

[ZC16] YungYu Zhuang and Shigeru Chiba. “Expanding Event Systems to Support Signals by Enabling the Automation of Handler Bindings”. In: *Journal of Information Processing* 24.4 (2016), pp. 620–634.

[Zob14] Justin Zobel. *Writing for computer science*. 3rd ed. Springer, 2014.

Included Papers

EXTENDING THE JASTADD EXTENSIBLE JAVA COMPILER TO JAVA 7

Abstract

JastAddJ is an extensible Java compiler, implemented using reference attribute grammars. It has been shown previously how the language constructs of Java 5, like generics, could be modularly added to the original JastAddJ compiler that supported Java 1.4.

In this paper we discuss our experiences from extending JastAddJ to support Java 7. In particular, we discuss how the Try-With-Resources statement and the Diamond operator could be implemented, and how efficient the resulting Java 7 compiler is regarding code size, compilation time, and memory usage.

1 Introduction

The Java language has gone through several updates, including versions 1.4, 5, 6, 7, and currently 8, each adding new constructs or standard class libraries to the language. Each new language version requires compiler support. While a state-of-the art compiler, like OpenJDK, implements such updates by modifying the source code of the previous version of the compiler, it has been shown how *reference attribute grammars* [Hed00] (RAGs) can be used to build extensible compilers, where new language constructs can be added modularly, without changing the previous source

Jesper Öqvist and Görel Hedin. "Extending the JastAdd Extensible Java Compiler to Java 7". In *Proceedings of the 10th International Conference on Principles and Practicies of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13), ACM, pp. 147–152.* Stuttgart, Germany, 2013.

modules. In particular, *JastAddJ* is an extensible compiler for Java, implemented using *JastAdd*, a metacompilation system that supports RAGs [HM03b]. JastAddJ originally supported Java 1.4, but was extended modularly to support all Java 5 features – enums, the enhanced for-statement, autoboxing, varargs, static imports, generics with wildcards, and annotations [EH07b].

The ability to modularly add language constructs has many advantages. For example, several versions of a language can be supported simultaneously without duplicate code, and researchers can reuse a compiler with relative ease, to construct new language extensions or tools. There are many examples of research languages that have been implemented on top of JastAddJ, e.g., *abc* (the AspectBench Compiler) [AET08b], *JCop* (a context-oriented programming extension to Java) [App+10], and *Fuji* (an extensible compiler for feature-oriented programming in Java) [Ape+12].

In this paper, we describe how we have modularly extended JastAddJ to support Java 7. We focus on the implementation of the two constructs we found the most challenging to implement, *Try With Resources* (TWR), and the *Diamond* operator. We were particularly aided by the use of *higher-order attributes* [VSK89b], i.e., computed abstract syntax tree values.

We evaluate the resulting compiler by investigating code size, compilation time, and memory usage, both as compared to the Java 6 version of JastAddJ and to the Java 7 version of OpenJDK.

The remainder of this paper is organized as follows. Section 2 gives background information on the JastAddJ compiler. Sections 3 and 4 describe how the TWR and Diamond constructs were implemented. Section 5 evaluates the implementation and section 6 discusses related work. The paper is concluded in section 7.

2 The JastAddJ Compiler

The *JastAddJ* compiler, [EH07b] is a Java compiler developed to demonstrate extensible compiler development using reference attribute grammars (RAGs) [Hed00].

2.1 Overall architecture

The compiler was initially developed for Java 1.4, and later extended to Java 5, 6, and 7. An extension to Java 8 is ongoing work. The development has recently been moved to bitbucket, at `https://bitbucket.org/jastadd/jastaddj`, and there is one source directory for each Java version: `java4`, `java5`, `java6`, and `java7`. Each such directory contains subdirectories containing the specifications for `scanner`, `parser`, `grammar` (for the abstract syntax tree), `frontend` (static-semantic analysis), and `backend` (byte code generation). The scanners are implemented in JFlex [JFl13], the parsers in Beaver [Bea13], and the grammars, frontends, and backends are implemented in the JastAdd metacompilation tool, using RAGs [HM03b]. Figure 1 shows the number of source lines of code (excluding whitespace and comments) in these directories.

We can note that the major part of the total code is in the frontend (65%), and the next largest part is the backend (27%). The Java 6 addition is very small: most of the

	java4	*java5*	*java6*	*java7*	**total**	%
scanner	308	17		109	**437**	2
parser	763	537		102	**1402**	5
grammar	167	57		21	**245**	1
frontend	9409	6178	27	1725	**17573**	65
backend	5505	1321		443	**7250**	27
total	**16150**	**8110**	**27**	**2400**	**26687**	100

Figure 1: Lines of code for JastAddJ modules. Excludes whitespace and comments.

new features in Java 6 concerned libraries, and the only change to the language was a change in the semantics of the `Override` annotation.

To generate the Java 7 compiler, the scanner modules are combined and then processed by JFlex, the parser modules are combined and then processed by Beaver, and the grammar, frontend, and backend modules are passed to JastAdd. The resulting generated Java source files are compiled together with around 1800 lines of driver code, written in Java. The driver code includes a Unicode scanner, Beaver runtime classes, and entry points for the Java compiler, static semantic checker, pretty printer and corresponding Ant tasks. The driver code is reused for all versions of the compiler.

2.2 Reference attribute grammars

It is the use of reference attribute grammars (RAGs) that makes it possible to modularize the different Java support levels in JastAddJ. A RAG specifies abstract syntax trees (ASTs), decorated with attributes that are defined using equations. The specification is *declarative* in the sense that in any correctly attributed AST all attributes will have values such that all equations are satisfied. The specification order of attributes and equations is thus irrelevant. The specification is also *executable*: an evaluator that computes the correct attribution can be automatically generated from the RAG. RAGs differ from Knuth's original attribute grammars [Knu68a] in that attributes may have *reference* values, i.e., they may refer to other nodes in the AST, and they may be *parameterized*, i.e., they may take arguments. Reference attributes are useful for representing graph structures on top of the AST, e.g., for linking a use of a variable to its declaration node. In JastAdd, there are also additional attribution mechanisms, in particular, higher-order attributes [VSK89b], also known as *non-terminal attributes* (NTAs). An NTA is an attribute whose value is an AST subtree, and which can itself have attributes. NTAs are useful for computing AST structures during compilation, for example for macro expansions or similar problems. In JastAdd, NTAs can be parameterized, just like ordinary attributes.

Like in Knuth's attribute grammars, an attribute can be either synthesized or inherited. For *synthesized* attributes, the defining equation must be located in the same node as the attribute. In contrast, if an attribute is declared as *inherited*, the responsibility to define the attribute is delegated to the context, i.e., to the parent of the node. In JastAdd, this responsibility is automatically delegated transitively through the parent chain until a node is found with an equation defining the attribute.

Feature	*scanner*	*parser*	*grammar*	*frontend*	*backend*	*Total*
Try-With-Resources		54	4	181	150	389
Strings in Switch				41	229	270
Diamond		6	2	327		335
Improved Numeric Literals	111	20	11	932		1074
Multi-catch		22	4	190	60	276
More Precise Rethrow				174	4	178
Safe Varargs				86		86
Miscellaneous				69		69
Total	**111**	**102**	**21**	**2000**	**443**	**2677**

Figure 2: Lines of code needed for the different Java 7 features (excluding whitespace and comments).

The attributed AST is the main data structure used in the JastAddJ compiler. For example, instead of using traditional symbol tables, declarations in a program are simply represented by the corresponding AST nodes through reference attributes that can locate the declaration from a use site. If the AST constructed by the parser is not sufficient for representing some information conveniently, additional structure can be added through NTAs. The advantage of this approach is that all information can easily be extended or overridden by other modules, through the RAG mechanisms.

To add a new language construct, one or more new RAG modules are added with new node types, attributes and equations. Through aspect-oriented inter-type declarations [Kic+97], attributes and equations in the new modules can apply to the new types as well as to types in the existing modules. A RAG specification is in many ways similar to an object-oriented framework: the AST node types are actually an object-oriented (Java) class hierarchy. The attributes can be called as methods, and equations defining attribute values can be overridden in subclasses, similar to Java methods.

We have implemented the Java 7 features as a JastAddJ extension. Figure 2 shows which parts of JastAddJ were extended by each new Java 7 feature, and how many lines of code were needed.

The Improved Numeric Literals feature required relatively many lines of code due to a refactoring made to the parsing of numeric literals that replaced old non-attribute grammar code.

3 Try With Resources

The *try-with-resources* (TWR) statement was added to the Java language to reduce the code clutter from closing a resource, particularly resources that can throw exceptions when closed.

Before Java 7 the following code would be required to ensure that a resource was closed properly:

```
Resource resource = null;
try {
    resource = new Resource(); // may throw
    // use resource here
} finally {
    if (resource != null) {
        try {
            resource.close(); // may throw
        } catch (IOException e) {
        }
    }
}
```

The TWR statement is similar to a regular `try` statement – it can have catch clauses and a finally clause – but with an extra resource declaration part where one or more resources are declared and initialized. Using TWR we can rewrite the above example to the following:

```
try (Resource resource = new Resource()) {
    // use resource here
}
```

When control passes out of a TWR statement all resources opened in the resource declaration part will be auto-closed. This occurs even if the TWR completes abruptly (e.g., due to an exception).

3.1 TWR static semantics

The JastAddJ framework for Java 1.4 contains a class modeling the regular `try` statement (`TryStmt`), containing a main code block, a list of catch clauses, and an optional `finally` clause. Figure 3 outlines how the framework was extended in the Java 7 frontend to support static semantics for TWR. A new node class `TryWithResources` was introduced that extends `TryStmt` with a list of resource declarations, modeled as a new subclass of the existing class `VariableDeclaration`. Most of the behavior in `TryStmt` is reused, but a few declarations are added in the extension, in order to adapt the behavior for TWR. For example, name analysis and analysis of reachability, exception handling, and definite assignedness is adapted.

The name analysis is adapted to make the resource declarations visible in the main block of the TWR. This is accomplished through the addition of an equation in `TryWithResources` that redefines the inherited attribute `lookupVariable` of the block. This attribute is used to look up declarations for variable uses inside the block. In a similar way, the reachability analysis (of TWR catch clauses) and exception handling checking (of resource initialization and closing exceptions) are handled by adding new equations that define or redefine inherited attributes of children of the TWR.

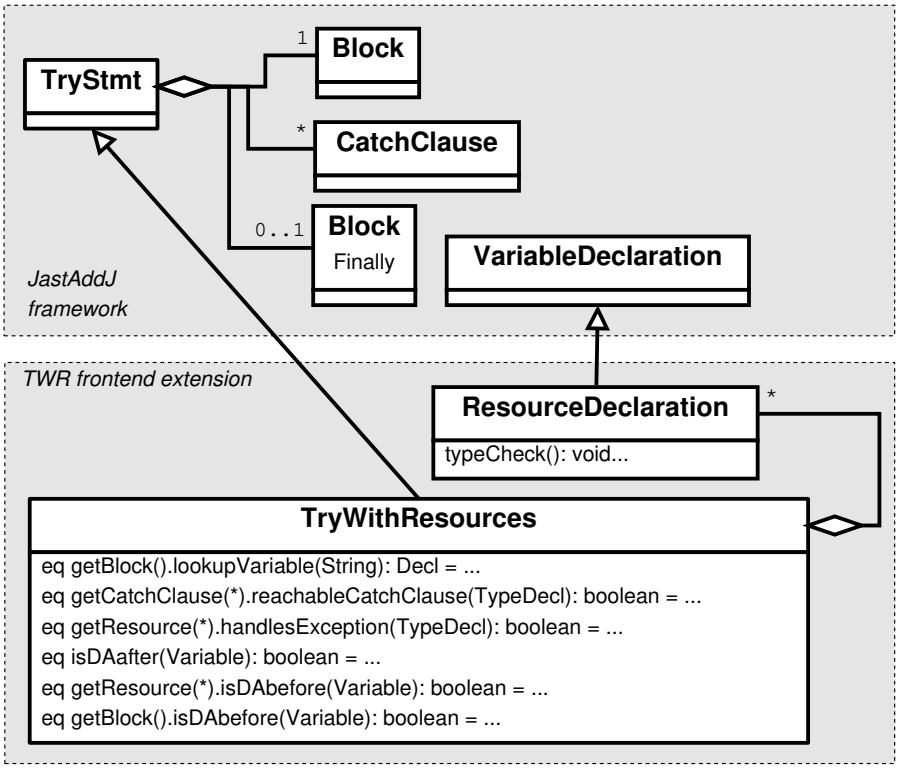


Figure 3: `TryWithResources` and `ResourceDeclaration` are two new node types added to support TWR. The static semantics of TWR is adapted through equations and other declarations.

The definite assignedness analysis in JastAddJ uses a synthesized attribute `s.isDAbefore(v)` and an inherited attribute `s.isDAafter(v)` to define if a variable `v` is definitely assigned before/after a statement `s`. The TWR extension includes equations that (re-)define `isDAafter` for the TWR statement, and `isDAbefore` for the children, in order to take the resource declarations into account.

There are new kinds of possible compile-time errors for TWR statements. An example is declaring a resource of a type that does not implement `AutoCloseable` (a new interface that declares the `close` method used to close a resource). JastAddJ collects compile-time error messages through calling the method `typeCheck` on all nodes. This method is implemented for `ResourceDeclaration` in order to check for these errors.

3.2 TWR code generation

While the `TryWithResources` node type conveniently describes the static semantics of the TWR statement, implementing code generation was trickier. The TWR state-

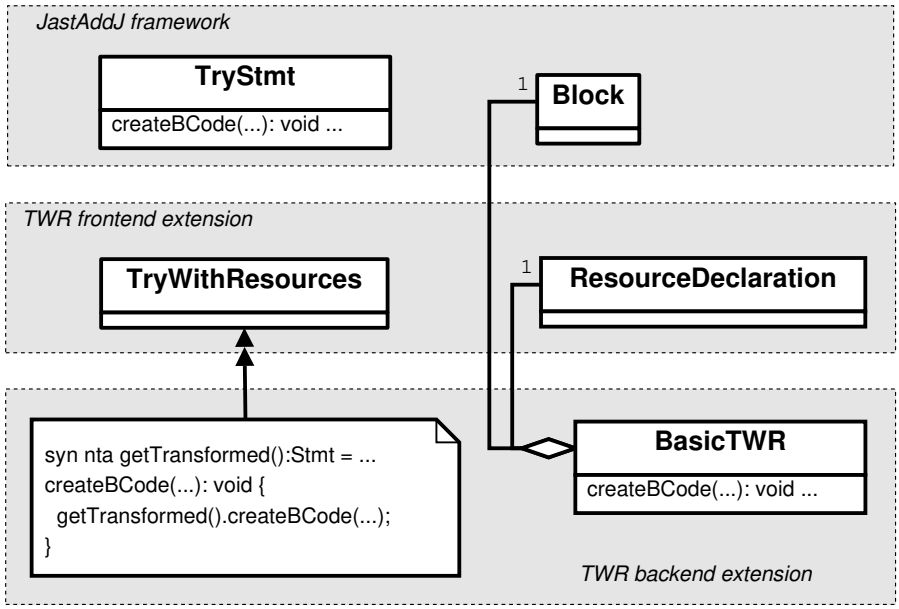


Figure 4: To implement code generation for TWR, a `BasicTWR` node type is introduced, whose code generation handles autoclosure. The code generation method for the TWR statement delegates to the NTA `getTransformed`, which will be either a `TryStmt` or a `BasicTWR` statement. The double-arrowed relation indicates features added to an existing class using inter-type declarations.

ment does many things implicitly that need explicit code generation support. The Java bytecode does not have built-in support for auto-closing resources, so all the steps needed for managing resources correctly need to be generated into bytecode.

The approach we used in JastAddJ builds on the fact that any TWR statement can be transformed into a regular `try` statement with a series of basic TWR statements nested inside it, as described in the Java Language Specification [JLS7, p. 408]. A *basic TWR* statement has no catch clauses or finally clause–it has only resource declarations and a block. By transforming a TWR statement this way, the bytecode generation for the `try` statement can be reused, and the generation of the autoclosing support can be implemented in a simple way for the basic TWR nodes.

We implemented this approach by introducing the transformed version of the TWR statement as a *non-terminal attribute* (NTA), i.e., an attribute whose value is a new AST subtree. The original TWR is used for static-semantic analysis, allowing any compile-time errors to be more closely related to the original source code. The code generation for a TWR is delegated to the transformed version of the statement, i.e., to the NTA. Figure 4 outlines the extensions done in the Java 7 backend. Note that the NTA and the code generation method are added to the `TryWithResources` node type using *inter-type declarations*. Inter-type declarations allow features of classes to be added in separate modules [Kic+97].

The transformed version of a TWR may contain a new (simpler) TWR which in turn has an NTA containing its transformed version, so the transformation is carried out in several steps. Furthermore, we defined the `BasicTWR` to include only a single resource declaration, so if the original TWR contains several resource declarations, this will result in a transformed AST with several nested `BasicTWR` nodes. Figure 5 shows an example. A TWR (1) with catch clauses and/or a finally block is transformed to a regular `try` statement (2), including a new simpler TWR statement (3) with only resource declarations and a block. This simpler TWR statement is transformed into a `BasicTWR` (4) with one resource declaration, and any remaining resource declarations are included in a yet simpler TWR statement (5). This expansion terminates when the last resource declaration is handled by a `BasicTWR` (6).

4 The Diamond Operator

In Java 7 the *diamond operator* was added to reduce some of the redundantly verbose nature of Java generics.

The diamond operator, <>, allows omission of type arguments for generic class instance creations in some contexts (assignments, variable/field initializers, method-/constructor arguments). For example, instead of writing

```
List<Integer> myList = new LinkedList<Integer>();
```

Java 7 allows us to write the following shorter and more readable code:

```
List<Integer> myList = new LinkedList<>();
```

The compiler needs to infer omitted type parameters, i.e., `Integer` in this case. To accomplish this, we added a new node type, `DiamondAccess`, to represent diamond expressions such as `LinkedList<>` above. It is defined as a subtype of the existing type `Access`, allowing it to occur in a class instance expression, see Figure 6.

The parent `ClassInstanceExpr` will query its `Access` child for its `type` attribute. The `DiamondAccess` node provides an equation that computes the type attribute by performing type inference: first a set of candidate constructors is computed, and then the most applicable constructor and type argument set is found.

It turns out that this problem can be transformed into a similar problem of inferring method type arguments for generic method invocations, a problem that was solved already for Java 5. We could implement the diamond inference quite easily, reusing the existing inference solution by creating a stand-in method declaration for each candidate constructor.

Figures 6 and 7 illustrate the extension. To represent candidate constructors, a new type, `StandInMethodDecl` was added, inheriting from the `GenericMethodDecl` type used by the existing type inference algorithm.

A candidate method set is created using all accessible constructors for the class to be instantiated. Let C be the generic class to be instantiated, with $T_{1...n}$ as its type parameters. For each accessible constructor K_i of C we create a stand-in method M_i with the same parameter list as K_i and return type $C < T_1, T_2, \cdots, T_n >$. The stand-in methods $\{M_{1...m}\}$ are passed to the existing inference algorithm which computes

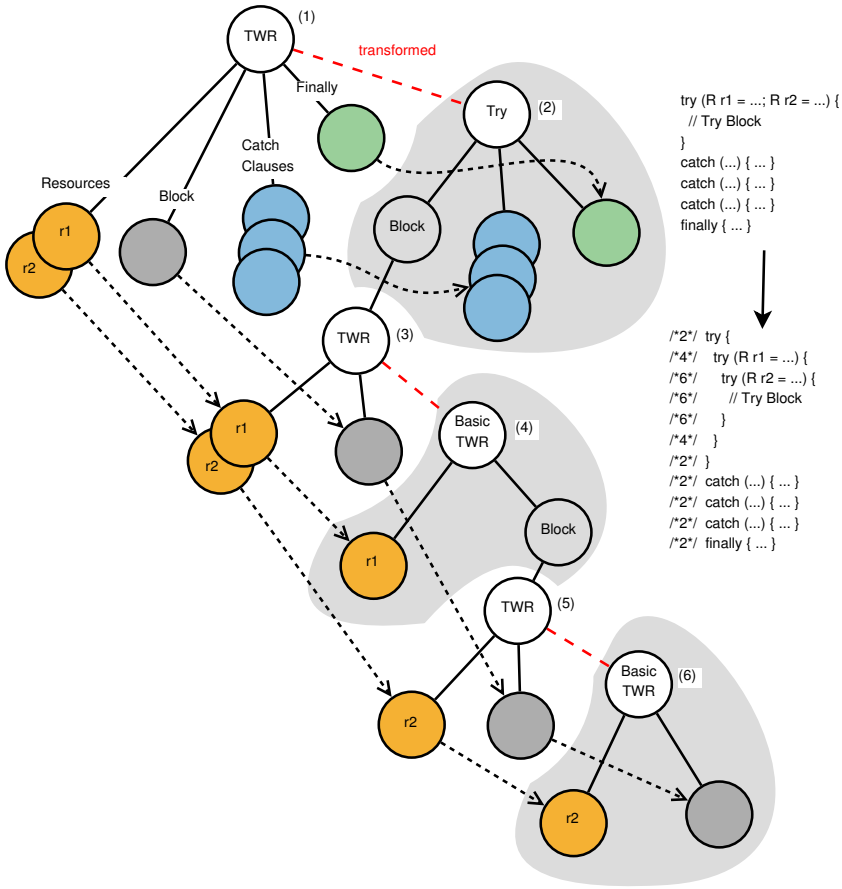


Figure 5: Example of stepwise transformation of a TWR statement using NTAs. An NTA `transformed` contains the new AST for each step. Lines without arrowheads indicate child-parent relationships, where the red dashed lines indicate NTA children. Dotted black arrows indicate AST nodes that are copied into an NTA. Grey shaded areas indicate transformed nodes to which code generation is delegated.

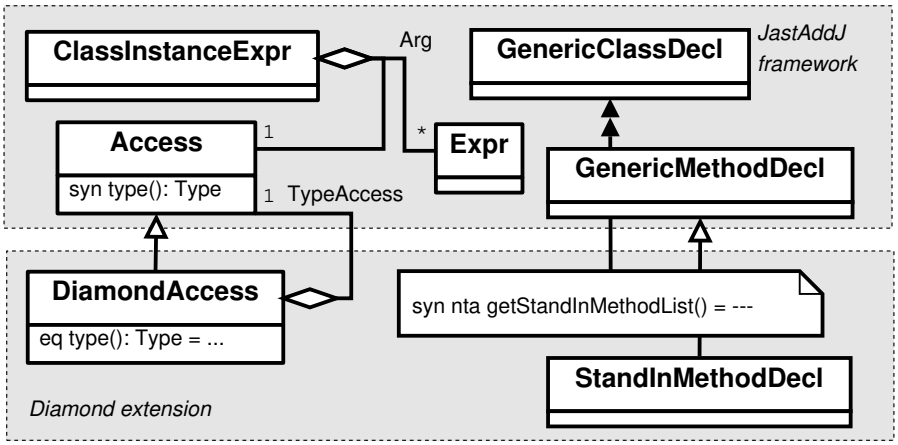


Figure 6: To support the diamond operator, the JastAddJ frontend is extended with two new node types, `DiamondAccess` and `StandInMethodDecl`, and an NTA in the existing type `GenericClassDecl`.

the most applicable method M_j and inferred type arguments. This gives us the most applicable constructor (K_i) and the full type argument list for the instance creation. The candidate method set is represented by a new NTA `getStandInMethodList` added to the type `GenericClassDecl`.

5 Evaluation

In 2007, JastAddJ was compared to several other Java compilers, including Sun's *javac* compiler [EH07b]. JastAddJ was then found to be less than three times slower than javac, and with an implementation size of 66% of javac (counted in source lines of code). The execution speed of the generated code was roughly the same for most programs.

Since 2007, both JastAddJ and javac have evolved, so it is interesting to do a new comparison, and to include the new Java 7 implementations. We have compared the OpenJDK Java 6 and Java 7 versions of JastAddJ and javac on a number of different open source benchmark programs. We compare source code size, compilation speed, memory consumption. We also tested the execution speed of the generated bytecode, and found that it was almost the same for JastAddJ and javac, with the JastAddJ-generated code being around 2% slower on the average.

The particular versions tested were builds 7.1.1-49 for JastAddJ (jastaddj-6, jastaddj-7), OpenJDK 6 b24 for javac-6, and OpenJDK 7 b146 for javac-7.

All tests were carried out on a quad core Intel Core i7-3820 CPU clocked at 3.60GHz with 64 GiB of memory, running Linux Mint with Linux kernel version 3.5.0-17-generic. All measurements were taken using the 64-bit Server editions of the IcedTea6 1.12.5 (javac-6, jastaddj-6) and IcedTea 2.3.9 (javac-7, jastaddj-7) runtime

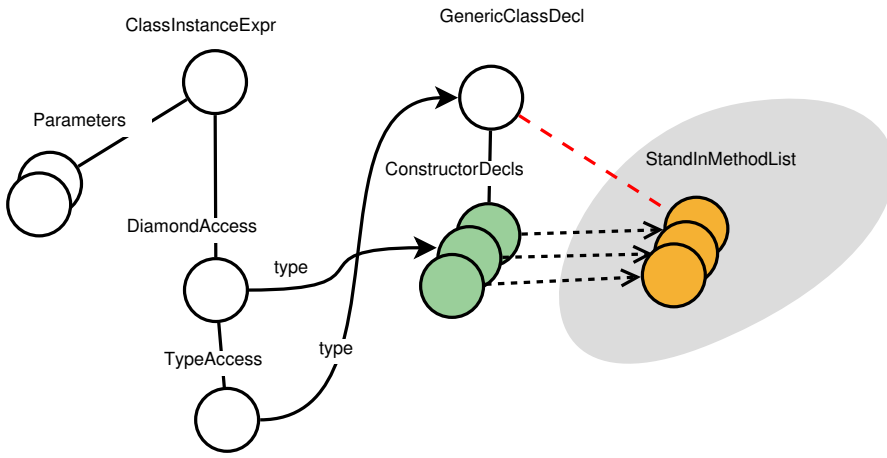


Figure 7: The Diamond type analysis reuses the Java 5 method type inference by computing stand-in methods using an NTA (the red dashed line). Solid arrows represent the `type` attributes. Dotted black arrows indicate construction of stand-in method declarations from constructor declarations.

environments. IcedTea is a compatible Java implementation based directly on the freely available OpenJDK source code.

5.1 Implementation size

As an estimate of implementation effort, we have measured the total source code sizes of JastAddJ and javac, excluding comments and whitespace, using the tool SLOC-Count [Whe13]. Figure 8 shows the results. We see that JastAddJ is just slightly more than half the size of javac: 55% for Java 6, and 52% for Java 7.

Compiler	*kSLOC (% of javac)*
javac OpenJDK 6 b24	47.4 (100%)
JastAddJ 7.1.1-49 Java 6	26.0 (55%)
javac OpenJDK 7 b146	55.1 (100%)
JastAddJ 7.1.1-49 Java 7	28.7 (52%)

Figure 8: The source code size of the compilers, in thousands of source lines of code, and in percentage of javac.

5.2 Compilation speed

We have measured the mean time to compile a number of different benchmark programs using Java 6 and 7 versions of javac and JastAddJ. We measured both the compile time when the compilers are invoked in a fresh Java VM (see section 5.2), as well as the steady-state compile time (section 5.2).

All time measurements were run with 2 GiB heap space, with JIT-compilation enabled to allow each compiler to run at its optimal performance. The 2 GiB heap space is well above the minimum required heap space for each benchmark program (see figure 11).

The benchmark programs we have used and their corresponding SLOC counts are listed in the table in figure 9.

Name	*Version*	*kSLOC*	*Description*
junit	4.5	5.2	Java unit testing framework
jsilver	1.0.1-SS	30.1	HTML template system
clojure	1.3.0-RC0	35.8	Clojure compiler
lucene	3.0.1	45.3	Text search engine
javac	jdk7-b146	55.1	OpenJDK 7 javac
jython	2.2alpha1	76.4	A Python environment in Java
jastaddj	R20111208	87.3	JastAddJ (generated Java code)

Figure 9: Benchmark programs

Fresh JVM compilation time

For each benchmark-compiler pair the benchmark program was compiled 25 times – each time in a new JVM instance. The total time to run each invocation of the compiler was measured (including JVM startup time). The arithmetic mean of the measured compile time is presented in figure 10 (labeled without the `ss-` prefix). Since JVM startup time was measured as part of the total compile time we can expect a smaller relative difference in execution time for the smaller benchmark programs (e.g. junit).

Steady-state compile time

We measured steady-state compile time using the method described in [GBE07b]. A brief summary of the benchmarking procedure:

- For each benchmark-compiler pair we run 10 new JVM invocations.
- For each JVM invocation the benchmark program is compiled until the last 15 measured execution times have a coefficient of variance no greater than 0.05.
- The mean of the last 15 compile times for the JVM invocation is calculated and stored.

The mean steady-state execution time of the 10 JVM invocations for each benchmark-compiler pair is plotted in figure 10 (labeled with the `ss-` prefix).

Using a Student's T distribution we calculated the 90% confidence intervals for each benchmark and found that they lie within 3% of the mean execution time (both for steady-state and fresh JVM).

Much of the time increase from Java 6 to Java 7 compilation is probably due to the extra parsing time required to parse the larger Java 7 class library.

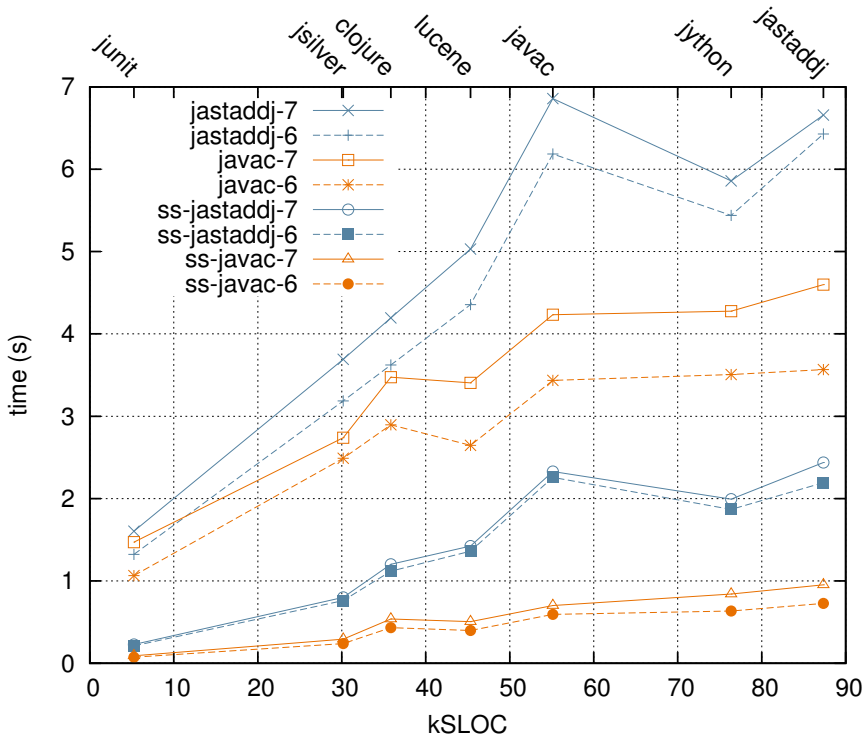


Figure 10: Compilation time. Includes times measured for fresh JVM invocations (top four curves) and steady-state execution (bottom four curves). Benchmarks are ordered from left to right by increasing SLOC count.

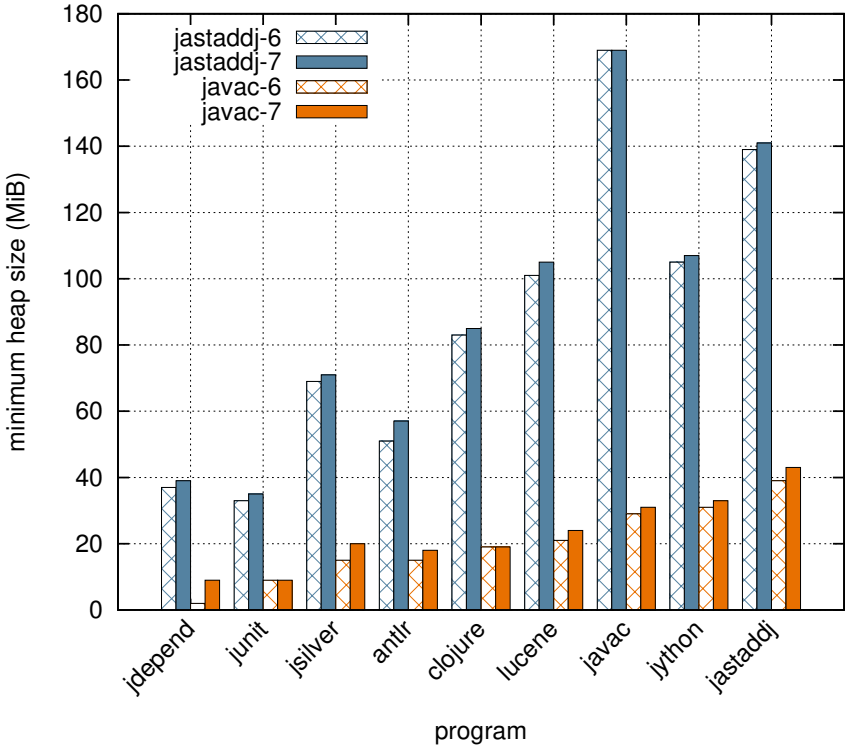


Figure 11: Minimum heap space. Benchmarks ordered from left to right by increasing SLOC count.

5.3 Memory consumption

We also compared the minimum required JVM heap space to run each benchmark-compiler combination. We found the minimum heap space by doing a binary search from an initial 2048 MiB heap size to the lowest heap size at which it was still possible to compile the benchmark program. The minimum measured heap sizes are illustrated in figure 11.

6 Related Work

OpenJDK is the reference implementation of the Java language, and is implemented in Java. It is based on a traditional compiler pipeline with a series of phases that are called one after the other. It is not specifically designed for language extension, and different language versions like Java 6 and Java 7 are kept in different parallel repositories [Ope13]. Language extensions of OpenJDK need to copy and modify source files. For example, OpenJML is an implementation of JML (Java Modeling

Language), and is built as a language extension to OpenJDK by modifying 41 of its 683 source files [Cok11].

Polyglot [NCM03b] and AbleJ [Van+07] are Java compiler frontends, built specifically to be extensible. Being frontends, they do scanning, parsing, and static semantic checking, but they do not generate bytecode.

Polyglot is implemented as a Java framework, and performs the frontend analysis as a series of passes, most of which are implemented using a variant of the Visitor design pattern. The original version supported Java 1.4, and recently, in 2012, support for Java 5 was released, implemented as a modular extension of the Java 1.4 version [Pol13]. So far, there is no support for Java 7 released.

AbleJ is implemented using the attribute grammar system Silver [Wyk+10b], and supports most of Java 1.4. Modular extensions have been defined for supporting parts of Java 5 and SQL queries from within Java code [Van+07].

The Java 7 extensions of JastAddJ were implemented by Jesper Öqvist as a master's thesis project, and a more detailed account of the implementation is available in the report [Öqv12].

7 Conclusion

We have extended the JastAddJ Java 6 compiler to support Java 7. The extension could be done modularly and concisely: the resulting source code for the compiler is only 55 % of that of OpenJDK javac for Java 6, and 52% for Java 7. Thus, JastAddJ has grown less than javac since the Java 6 version.

We discussed some details of our implementation and how we could extend the Java 6 implementation by adding new node types, equations and attributes. In particular, the use of nonterminal attributes allowed us to reuse code generation and type inferencing from Java 6 to implement the Try-With-Resources statement and the Diamond operator.

We have measured the performance of our compiler to show that it is practical, despite being a research compiler that is generated from a specification. When run on a fresh JVM, our compiler runs within a factor of 1.6 compared to javac. When run in steady state (like it would be run in an IDE), it runs within a factor of 3.3 of javac for our benchmarks. Our compiler uses substantially more memory than javac: 3.2 to 5.5 times as much memory as javac for our benchmarks. However, with the memory capacity on today's computers, large programs can still be compiled without problems.

Acknowledgements

This work was in part financed by the Swedish Research Council under grant 621-2012-4727.

References

[Ape+12] Sven Apel et al. “Access control in feature-oriented programming”. In: *Science of Computer Programming* 77.3 (2012), pp. 174–187.

[App+10] Malte Appeltauer et al. “Event-specific software composition in context-oriented programming”. In: *Software Composition*. Springer. 2010, pp. 50–65.

[AET08b] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. “Modularity first: a case for mixing AOP and attribute grammars”. In: *Proceedings of the 7th international conference on Aspect-oriented software development*. ACM. 2008, pp. 25–35.

[Bea13] Beaver. *Homepage*. `http://beaver.sourceforge.net/`. May 2013.

[Cok11] David R Cok. “OpenJML: JML for Java 7 by extending OpenJDK”. In: *NASA Formal Methods*. Vol. 6617. LNCS. Springer, 2011, pp. 472–479.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 1–18.

[GBE07b] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 57–76.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HM03b] Görel Hedin and Eva Magnusson. “JastAdd–an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[JFl13] JFlex. *Homepage*. `http://jflex.de/`. May 2013.

[JLS7] James Gosling et al. *The Java™ Language Specification, Java SE 7 Edition*. Oracle America, Inc., Feb. 2013.

[Kic+97] G. Kiczales et al. “Aspect-oriented programming”. In: *ECOOP 1997 - Object-Oriented Programming* (1997), pp. 220–242.

[Knu68a] D.E. Knuth. “Semantics of context-free languages”. In: *Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[NCM03b] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. “Polyglot: An extensible compiler framework for Java”. In: *Compiler Construction*. Springer. 2003, pp. 138–152.

[Ope13] OpenJDK. *JDK 6 Project*. `http://openjdk.java.net/projects/jdk6/`. May 2013.

[Öqv12] Jesper Öqvist. “Implementation of Java 7 Features in an Extensible Compiler”. Master Thesis. Report LU-CS-EX:2012-13. Sweden: Dept. of Computer Science, Lund University, 2012.

[Pol13] Polyglot. *A compiler front end framework for building Java language extensions*. `http://www.cs.cornell.edu/Projects/polyglot/`. May 2013.

[Van+07] Eric Van Wyk et al. “Attribute grammar-based language extensions for Java”. In: *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 575–599.

[VSK89b] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. “Higher order attribute grammars”. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. ACM. 1989, pp. 131–145.

[Whe13] David A. Wheeler. *SLOCCount*. `http://www.dwheeler.com/sloccount/`. 2013.

[Wyk+10b] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54.

Multitudes of Objects

First Implementation and Case Study for Java

Abstract

In object-oriented programs, the relationship of an object to many objects is usually implemented using indirection through a collection. This is in contrast to a relationship to one object, which is usually implemented directly. However, using collections for relationships to many objects does not only mean that accessing the related objects always requires accessing the collection first, it also presents a lurking maintenance problem that manifests itself when a relationship needs to be changed from to-one to to-many or vice versa. Continuing our prior work on fixing this problem, we show how we have extended the Java 7 programming language with multiplicities, that is, with expressions that evaluate to a number of objects *not* wrapped in a container, and report on the experience we have gathered using these multiplicities in a case study.

ein Vieles, welches kein Eines ist
(*a multitude which is not a one*)
— inspired by Georg Cantor's conception of a set as "jedes Viele, welches sich als Eines denken läßt", i.e., any multitude which can be thought of as a one

1 Introduction

Just like English grammar distinguishes singular and plural, object-oriented programming languages distinguish one object and many objects. However, unlike with English utterances, for which the syntactic difference between the singular and the plural of a noun phrase is usually small, the difference between program fragments

Friedrich Steimann, Jesper Öqvist, and Görel Hedin. "Multitudes of Objects: First Implementation and Case Study for Java". In *Journal of Object Technology, Vol. 13, no. 5 (November 2014), pp. 1:1–33*.

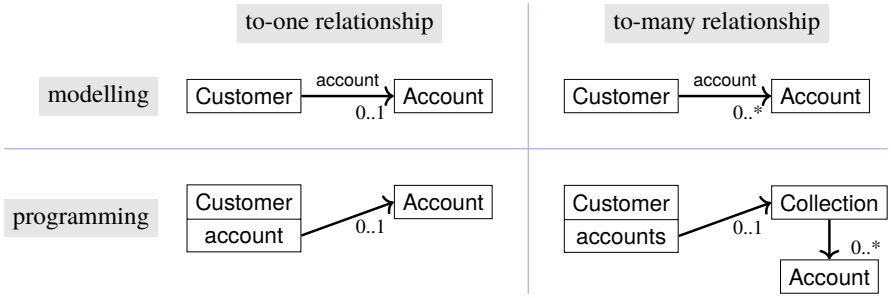


Figure 1: Relationships to one and to many objects in object-oriented modelling and programming languages: differences

dealing with one object and dealing with many objects is often substantial. For instance, while the English utterances "I go to work" and "we go to work" differ only in the pronoun used, in an object-oriented program, the difference would be that between i.goto(work) and for (each : we) each.goto(work), which is cumbersome not only by comparison. The problem, here, is that in object-oriented programming, the multitude denoted by we is reified as a one (usually a collection object), and this one has different properties (responds to a different protocol) than the objects it comprises. In particular, the object denoted by we cannot go to work.

Similar to the English language, relational and object-oriented modelling languages make only a small distinction between singular and plural or, more specifically, between one object being associated with one other, or any number of other objects [8; 9; 10; 28; 36]. In these languages, *multiplicity*, also known as *cardinality*, constrains the number of times an object, or entity, may occur in a relationship or association. Hence, a change from singular to plural (or vice versa) requires little more than a corresponding change in multiplicity, as the top half of Figure 1 suggests. By contrast, in object-oriented programming languages multiplicities are commonly coded in the declared type of a variable (which is the type of the related object if it is only one, or the type of a sequence, stream, or collection object if there are more). Here, a change of multiplicity may require a major redesign of the program, as the bottom half of Figure 1 suggests (several more untoward consequences of such a change will be presented below).

In previous work [37], we advocated the introduction of *multiplicities* as annotations of expressions indicating whether an expression is singular or plural, i.e., whether it is expected to evaluate to at most one object, or to any number of objects (not reified). This is to grant the programmer a more uniform treatment of relationships to one object and relationships to many objects in object-oriented programs. In this paper, we present an implementation of our ideas as an extension of the Java programming language using the JastAddJ extensible Java compiler [13], and report on a case study we have conducted.

The remainder of this paper is organized as follows. To motivate our work, we present in Section 2 the peculiarities we observe when implementing multitudes of objects using collections. In Section 3, we briefly describe how enhancing

object-oriented programming with multiplicities can generally alleviate the associated problems, with Section 4 specializing our proposal for Java. Section 5 describes our implementation of multiplicities as an extension of the JastAddJ compiler for Java 7. In Section 6, we present qualitative and quantitative findings from a case study extending JUnit with multiplicities. Notes on related and future work conclude.

2 Using Collections for Representing Multitudes of Objects

Undoubtedly, collections are among the most useful abstractions in object-oriented programming: they not only liberate the programmer from manually implementing multitudes of objects as (static) arrays or dynamic data structures (such as linked lists or trees), they also offer a uniform protocol for bulk processing of these object using internal iterators (foreach, select, collect, etc.). And yet, the use of collections for representing many (rather than one) objects comes with a number of peculiarities which make dealing with multitudes of objects very different from dealing with single objects.

2.1 Multiplicity Determines Type

In a program in which every customer can have only a single account, we may see code like

```
Account account;
account = new Account();
account.check();
```

If however a customer can have several accounts, adjustment of just the declaration to reflect this leads to an ill-typed program (faulty expressions underlined):

```
Set<Account> accounts;
accounts = new Account();

accounts.check();
```

Both errors result from the fact that accounts (with a plural "s" appended to express that there can be more than one) now has type Set<Account>, reflecting the changed multiplicity. However, intuitively, what is expressed by the ill-typed program is rather clear: initialize accounts to hold just one account, and then check all accounts (which happens to be only one here). To translate this to standard Java, we would have to write

```
Set<Account> accounts = new HashSet<>();
accounts.add(new Account());
for (Account account : accounts) account.check();
```

which means quite a change to the original program.

2.2 Multiplicity Determines Meaning of `null`

When a variable represents an optional relationship to one object, the value null usually means that there is no relationship (but may also mean failure to initialize):

```
if (account != null) print(account);
else print("no account");
```

For a relationship to many accounts, relating to no account is usually represented using an empty collection:

```
if (accounts != null)
  if (! accounts.isEmpty())
    for (Account account : accounts) account.print();
  else print("no account");
else throw new Error("accounts not initialized");
```

Here, the value null means failure to initialize. Note that having null as an element of a collection makes no sense if the collection is to represent a relationship.

2.3 Multiplicity Determines Subtyping Conditions

If SavingsAccount is a subtype of Account, writing

```
SavingsAccount saving = new SavingsAccount();
Account account = saving;
```

is type-correct. However, when we change to many accounts, the analogue

```
Set<SavingsAccount> savings = new HashSet<>();
Set<Account> accounts = savings;
```

is ill-typed. Instead, we would have to write something like

```
Set<? extends Account> accounts = savings;
```

[24] which does however preclude write access to the set through the variable accounts, greatly limiting its use (especially when considering that the singular account can be used freely).

2.4 Multiplicity Determines Encapsulation Strategy

It is considered good practice in object-oriented programming that the fields of an object are encapsulated and, if necessary, made accessible for clients using setter and getter methods. For collection-valued fields, however, this is different [16]: they are to be updated using add...(...) and remove...(...) methods offered by the encapsulating object (where the ellipses are replaced by the field's name), and if the collection as a whole is to be retrieved, the getter should return a copy or an immutable wrapper [16]. This is so because the collection is considered a representation object which clients should not be able to manipulate directly and of which they should possess no aliases [25]. This brings us directly to the next point.

2.5 Multiplicity Determines Availability of Relationship Aliasing

While assigning an object to a variable with reference semantics always means creating an alias for the object, the semantics differ when the variables are uniformly viewed as implementing relationships to objects, as the following example demonstrates:

```
Account backup = account;
account = null;
if (mistaken) account = backup;
```

Here, backup is an alias for the to-one relationship implemented by account. This is different for

```
Collection<Account> backups = accounts;
accounts.clear();
if (mistaken) accounts = backups;
```

where backups is an alias for the collection denoted, and not for the to-many relationship that is logically established, by accounts. Surely, the problem can be solved by keeping a copy of the collection as backup, but copying is not needed for the to-one case.

2.6 Multiplicity Determines Call Semantics

Continuing the previous example, it may seem awkward that the method

```
void clear(Collection<Account> accounts) {
  accounts.clear();
}
```

performs as intended (i.e., sets the relationship represented by an actual parameter to "no accounts"), while the analogous method for the to-one case

```
void clear(Account account) {
  account = null;
}
```

has no effect on actual parameters. While this may look like a newbie's mistake to the seasoned programmer, it is still indicative of a conceptual chasm, which culminates in the fact that in Java, it is impossible to implement

```
void swap(Object o1, Object o2)
```

with the suggested semantics, while implementing

```
void sort(ArrayList<Object> os)
```

is not a problem. Note that escaping to call-by-reference for swap(. . ., . . .) does not bridge the chasm — not having to do so for collections is just another peculiarity of using them for representing multitudes of objects.

2.7 Multiplicity Determines Meaning of the `final` Modifier

When a variable is declared as final, it means that its value cannot be changed after its initialization. For a variable representing a relationship to a single object this means that the owner of the variable is stuck with the related object for its whole lifetime. For a variable representing a relationship to many objects implemented using a collection, final means that the holder of the relationship is stuck with the collection — its elements, and thus the conceptually related objects, may change freely:

```
final Account forLife = new Account();
forLife = null; // compile error

final Set<Account> allForLife = Arrays.asSet(forLife);
allForLife.clear(); // no problem
```

3 Programming with Multiplicities

The core idea of object-oriented programming with multiplicities as put forward in [37] is that expressions may evaluate directly to any number, or *a multitude*, of objects. This is in contrast to standard object-oriented programming, in which every expression evaluates to either one object or to null and in which multitudes of objects are reified using special container objects (collections, sequences, iterators, etc.). Note that, since multitudes are not reified in our approach, they are always flat, i.e., there is no multitude of multitudes (though it is possible to create a multitude of collections).

Terminological Note We use "multitude of objects" to denote many objects; the term is to be distinguished from "collection of objects" or "set of objects", which each denote an entity in its own right. Note that for this reason it makes no sense to speak of "the elements" or "the members of a multitude", or even of "the objects of a multitude" (since the objects of the multitude *are* the multitude) — if we want to refer to one of many, we say just that, or "one object among a multitude".

3.1 Dynamic and Static Multiplicity

With expressions evaluating to any number of objects, the *dynamic multiplicity* of an expression is defined as the number of objects it evaluates to. In the general case, the dynamic multiplicity of an expression can only be determined at runtime. Therefore, we complement dynamic multiplicity with *static multiplicity*, which can be declared and inferred at compile-time. In the following, the term multiplicity refers to static multiplicity unless stated otherwise.

While dynamic multiplicities are cardinals, we distinguish mainly two (symbolic) static multiplicities, which we call *option* and *any*. *Option* stands for no or one object, while *any* stands for any number of objects. Other static multiplicities are also

conceivable (in particular, multiplicity *one*, for precisely one, will be useful; see below); however, since our focus here is on eliminating as much as possible the differences between relating to zero or one and to any number of objects, *option* and *any* suffice.

3.2 Separation of Multiplicity and Type

As long as multitudes of objects are reified, the multiplicity of an expression (i.e., whether it evaluates to one or many objects) is coded in its type: for multiplicity *any*, this type is a collection type (commonly parameterized with the member type, i.e., the type of the elements of the collection), whereas for multiplicity *option*, the type is the type of the optional object (see Section 2.1). Object-oriented programming with multiplicities as put forward in [37] separates multiplicity from type in the declaration of variables and methods: for instance, it allows one to write

```
any Account accounts;
```

instead of

```
Collection<Account> accounts;
```

for declaring that accounts can hold any number of Account objects (note that it cannot hold a collection!), whereas

```
option Account account;
```

which differs only in the multiplicity, is roughly equivalent to

```
Account account;
```

meaning that account can hold either no or one account (see Section 3.4 for the important difference). Note that using multiplicities, both account and accounts have the same type Account; they differ only in their declared multiplicities.

3.3 Assignment Compatibility

While the types of account and accounts are the same and, therefore, do not oppose their mutual assignment compatibility, their multiplicities differ — since *option* is subsumed by *any*, account can be assigned to accounts, and

```
any Account accounts = new Account();
```

is a legal assignment (cf. Section 2.1). An assignment from *any* to *option* is illegal, however; here, a multiplicity downcast (from *any* to *option*) as in

```
account = (option) accounts;
```

is required, but may fail at runtime (namely when accounts holds more than one object).

For variables with multiplicity *any*, assignment is complemented with adding to (+=) and subtracting from (−=) a multitude of objects, where the right-hand side of the update operations can have multiplicity *any* or *option*.

null remains assignment compatible with every reference type; also, it is assignment compatible with both multiplicity *option* and *any* (and means "related to no object" in both cases; cf. Section 2.2).

3.4 Member Access

That account and accounts have the same type means that they respond to the same protocol, i.e., that the same set of methods can be invoked and the same set of fields can be accessed on them. For instance, if class Account defines a method check(), both account.check() and accounts.check() are well-typed; the latter simply means that check() is separately invoked on all objects accounts holds. If Account declares an *option* field bank, accounts.bank returns a multitude of bank objects, namely the banks each account among the multitude of accounts held by accounts is related to. Note that if accounts holds no object, or no account referred to by accounts has a bank associated with it, accounts.bank will evaluate to no object. Since *option* is subsumed by *any*, account.bank will also evaluate to no object if account does not hold an account; note in particular that no null pointer exceptions can arise from dereferencing expressions whose multiplicity is *option* or *any*.[1]

3.5 Aliasing

In object-oriented programming with multiplicities, multitudes of objects are not reified, so multitudes cannot be aliased. This retires the problems noted in Sections 2.3–2.6. In particular,

```
any SavingsAccount savings;
any Account accounts = savings;
```

does not cause a covariance problem, since the assignment does not create an alias for a container, but instead assigns accounts the same multitude of objects that savings refers to (by copying pointers just like in the *option* case). It follows that

```
accounts += new Account();
```

does not also add an account to savings (cf. Section 2.3). Likewise, returning accounts as in

```
any Account getAccounts() { return accounts; }
```

does not expose representation to clients (there is no representation object representing the multitude held by accounts) and, in particular,

```
any Account temp = getAccounts();
temp += new Account();
```

does not update the field accounts returned by the getter (Section 2.4). Similarly, after the assignment

[1] For the relationship of the multiplicity *option* with the type Option of some functional programming languages (including Scala), see the related work in Section 7

```
any Account backups = accounts;
```

(Section 2.5), clearing accounts (by assigning it null; cf. Section 3.3) does not also clear backups, which is therefore still available for restoration. Also, passing a variable into the method clear(...) of Section 2.6, now defined as

```
void clear(any Account accounts) {
  accounts = null;
}
```

does not affect the number of objects that this variable holds, thereby unifying the behaviour for one and many objects. Lastly, the fact that multitudes of objects are not reified unifies the meaning of the final modifier (Section 2.7), which now pertains to variables holding single object and multitudes of objects alike.

3.6 When to and When Not to Use Multiplicities

Our motivation of introducing multiplicities to object-oriented programming is to allow the programmer

- the implementation of relationships (or, more precisely, directed associations [28]) to many objects in a more direct way, and further
- the implementation of relationships to one and to many objects in as much the same way as possible.

This raises the question of what is a relationship, or when multiplicities are to be used.

Experience teaches that programmers will use a construct wherever they deem its use advantageous, so we attempt no dogmatism here. We still make one exception, though: value types, like int, float, boolean (including their wrapper types), or String (whose instances are usually immutable) cannot be the target of a relationship (note that they cannot act as entity types in the entity-relationship model [8]) and hence cannot be used in combination with *option* or *any* multiplicity annotations. While there are conceptual justifications for this (e.g., people do not relate to their age, an integer value), the main technical reason is that this saves us from defining special semantics for operations on value types (such as +) for operands with *option* or *any* multiplicity (which both include "no object" as a possible value), and also from introducing a ternary logic for handling the case that a boolean expression used in a conditional evaluates to no object. For instance, for a variable declared as option boolean error, it is unclear what if (error) ... means if error has dynamic multiplicity 0. For the same reason, we must exclude that value-typed members are accessed via receiver expressions with *option* or *any* multiplicity, since this can likewise result in dynamic multiplicity 0 (namely when the receiver evaluates to "no object").

4 Multiplicities for Java

While the idea of object-oriented programming with multiplicities as presented in the previous section is language-independent, its adoption in any concrete language invariably requires an individualized integration with existing language constructs. In the following, we present our extension of Java 7 with multiplicities, whose design was driven by our objective to allow a smooth transition between Java programming without and Java programming with multiplicities. Figure 2 has the extended syntax; Figure 4 shows some sample code using it.

```
Modifier ::= ...
 | MultiplicityAnnotation;

MultiplicityAnnotation ::=
   "@any" "(" ReferenceType ")"
 | "@any"
 | "@option"
 | "@bare";

Primary ::=
 | "[[" UnaryExpressionNotPlusMinus "]]"
 | "|[" UnaryExpressionNotPlusMinus "]|";

CastExpression ::= ...
 | MultiplicityCastExpr;

MultiplicityCastExpr ::=
   "(" MultiplicityAnnotation ")" UnaryExpression
 | "(" MultiplicityAnnotation TypeName ")"
   UnaryExpression;
```

Figure 2: Extension of concrete syntax.

4.1 Multiplicity Annotations

For compatibility with existing Java code, we implemented four static multiplicities:

- *none*, the multiplicity of `null` (representing no object);
- *bare*, the default multiplicity (and, in particular, the multiplicity of all standard, or legacy, declarations);
- *option*, the multiplicity of entities and expressions representing relationships to no or to one object; and
- *any*, the multiplicity of entities and expressions representing relationships to any number of objects.

Multiplicities determine assignment compatibility according to the order

$$none < bare < option < any$$

i.e., every multiplicity is assignment compatible with itself and all greater ones.

Hidden Collections To give the programmer control over the nature of multitudes, and also for interfacing with legacy Java code that uses collections (see below), *any* multiplicities may be parameterized with a collection type *C* whose definition has a single type parameter (e.g., List‹E›). This type will be used to instantiate a *hidden collection* holding the multitude of objects. To acknowledge the widespread use of abstract collection types in Java programs, *C* may be an abstract class or an interface.

Syntactic Integration The multiplicity *none* does not occur in program texts; the other multiplicities appear as annotations @bare, @option, and @any, respectively (see Figure 2). Since *bare* is the default multiplicity, it occurs only in multiplicity downcasts from *option* or *any* to *bare* (see below).

original (without using multiplicities)

```
class Subject implements Observable {
  Set<Observer> obs = new HashSet<>();

  public void addObserver(Observer o) {
    if (o == null)
      throw new NullPointerException();
    obs.add(o);
  }

  public void deleteObserver(Observer o) {
    obs.remove(o);
  }

  public void notifyObservers(Object arg) {
    for (Observer o : obs)
      o.update(this, arg);
  }

  public void deleteObservers() {
    obs.clear();
  }

  public int countObservers() {
    return obs.size();
  }
}
```

using multiplicities

```
class Subject implements Observable {
  @any(HashSet) Observer obs;

  public void addObserver(@option Observer o) {


    obs += o;
  }

  public void deleteObserver(@option Observer o) {
    obs -= o;
  }

  public void notifyObservers(Object arg) {

    obs.update(this, arg);
  }

  public void deleteObservers() {
    obs = null;
  }

  public int countObservers() {
    return |[obs]|;
  }
}
```

Figure 3: Example of implementing class Subject of the Observer Pattern, with and without using multiplicities (differences highlighted)

4.2 Declarations

The multiplicity annotations @option and @any may appear (in the position of modifiers; cf. Figure 2) in all declarations of reference-typed entities, except where the type is a wrapper type (such as Boolean) or String (see Section 3.6 for the reasons). If an entity is annotated with @any(C) and C is a concrete collection class, the compiler uses C as the class of the hidden collection that holds the multitude of objects the annotated entity denotes; if C is abstract or not provided, the compiler automatically picks a suitable concrete class (note that the single type parameter of C is instantiated with the type of the declaration). Thus, multitudes of objects *are* implemented using collections; however, this implementation is strictly under the hood and, in particular, these collections cannot be accessed from the program as objects (they cannot be aliased). Entities annotated with @option are not implemented using collections; however, the compiler gives their value null a special meaning (see below).

Final Declarations A declaration final @option T v means that, after its initialization, the value of v cannot change (i.e., v always refers to the same object). That v is not assigned new values after initialization is ensured (by the compiler) as usual. A declaration final @any T v likewise means that v cannot be updated after its initialization (i.e., it always refers to the same objects), where updating includes assignment (=), adding objects (+=), and removing objects (−=); this is ensured by the very same means. Hence, no immutable collections are required to express the immutability of a multitude.

4.3 Interfacing with Collections: Wrapping and Unwrapping

To interface multiplicity *any* with code that uses (bare) collections for representing multitudes of objects, we must be able to wrap a multitude in a collection object, and to unwrap it from a collection object. We use double square brackets ("[[...]]") for both purposes (see Figure 2). If the argument expression has static multiplicity *any(C)*, the result is a (fresh) collection of type *C* holding the objects the expression evaluates to; if it has multiplicity *option*, the result is a new instance of class ArrayList which either holds the object the expression evaluates to, or is empty if it evaluates to null. We call this *wrapping*. If the argument expression has static multiplicity *bare* and is a collection, the result is the multitude of objects that the collection holds (which is internally represented using a fresh collection of the same type). We call this *unwrapping*. Unwrapping is particularly useful for initializing final variables with multiplicity *any*:

```
final @any Account accounts =
  [[Arrays.asList(new Account(), new Account())]];
```

The expression [[null]] is not allowed.

4.4 Number of Objects

The dynamic multiplicity, or the number of objects an expression evaluates to, is computed using "|[...]|" (see Figure 2). In case the argument expression has multiplicity *any*, it returns the size of the underlying collection; if the expression has multiplicity *option*, it returns 0 if it evaluates to null, and 1 else.

4.5 Casts

As for types, multiplicity upcasts (e.g., from *option* to *any*) are always safe and therefore may remain implicit. Multiplicity downcasts from *any* to *option* or *bare*, however, may fail, namely when the *any* expression being cast evaluates to more than one object. Therefore, the compiler inserts a runtime multiplicity check for all such casts which, upon failure, throws a multiplicity cast exception. Casts from *option* to *bare* are also always safe; since unlike for *option* receivers, accessing members on *bare* receivers can lead to null pointer exceptions, we will require explicit downcasts from *option* to *bare* (see below for examples of where this is needed).

4.6 Expressions

With new syntax given meaning as above, we now turn to the impact multiplicities have on standard Java expressions.

Update Assignment (=) makes the variable on the left-hand side refer to the objects the right-hand side refers to. Given the arbitrariness of the definition of "identity of multitudes of objects" for multiplicity *any* (see below), we defined assignment pragmatically: the hidden collection holding the multitude of the left-hand side is first emptied (cleared), and then all objects of the hidden collection representing the right-hand side are copied into it using its addAll(...) method. If the right-hand side has multiplicity *option*, its object (if any) is added to the collection using add(...). If the left-hand side has multiplicity *option*, the object that the right-hand side evaluates to is assigned to it.

Adding (+=) is only allowed for left-hand sides with *any* multiplicity and adds the object(s) of the right-hand side (if any) to it, again using add(...) or addAll(...). Removing (−=) works accordingly, using the corresponding remove methods. Note that the programmer can override the meaning of += and −= by supplying her own collection implementations to the *any* annotations in the declarations.

Member Access Accessing a member m on a receiver r with multiplicity *any* unfolds to accessing m on every object among the multitude r evaluates to, in the order provided by the iterator of the hidden collection holding the multitude. If m is a field or a non-void method, r.m evaluates to a multitude of objects, independently of whether m has multiplicity *option* or *any* (recall that multiplicities are always flat). As argued in Section 3.6, m must not be *bare*; if it is, the receiver must be cast to *bare* first (cf. Section 4.5 and 6.2).

Since *option* is subsumed by *any*, accessing `m` on `r` having multiplicity *option* behaves exactly as if `r` had static multiplicity *any* and dynamic multiplicity 0 or 1. In particular, if `r` is `null`, evaluating `r.m` does not raise a null pointer exception — it simply evaluates to `null` (for "no object"). However, deviating from receiver multiplicity *any*, `r.m` has multiplicity *option* for *option* members (see Table 1).

If `m` is a method, parameter passing works according to the rules of assignment (see under "Update" above). In particular, the formal parameters do not receive aliases to the hidden collections holding the objects of formal parameters having multiplicity *any*. Similarly, `m` does not return aliases to the hidden collection representing the returned expression. Note that, with respect to multiplicity, overriding methods must be contravariant in the formal parameter multiplicities (i.e., *option* can be overridden with *any* etc.) and covariant in the return multiplicities (i.e., *any* can be overridden with *option* etc.). Figure 4 has an example of a covariantly overridden method (`getLeaves()`).

r \ m	*bare*	*option*	*any*
bare	*bare*	*option*	*any*
option	N/A	*option*	*any*
any	N/A	*any*	*any*

Table 1: Multiplicity of member access expressions `r.m`

Test for Identity Strictly speaking, a test for identity ("==") does not make sense for multitudes of objects: if multitudes are not reified, how can they be identical? On the other hand, if the dynamic multiplicities of the left-hand side and the right-hand side of such a test are 0 or 1, there seems little choice in defining the meaning of `==`: it is true if and only if either both evaluate to the same object, or both evaluate to `null`. For greater numbers of objects, it would seem reasonable to require that both sides have the same dynamic multiplicities; yet, this means that even immediately after

original (without using multiplicities)

```
abstract class Composite {
  abstract List<Leaf> getLeaves();
}

class Component extends Composite {
  List<Composite> children = new ArrayList<>();
  List<Leaf> getLeaves() {
    List<Leaf leaves = new ArrayList<>();
    for (Composite child : children)
      leaves.addAll(child.getLeaves());
    return leaves;
  }
}

class Leaf extends Composite {
  List<Leaf> getLeaves() {
    return Arrays.asList(this);
  }
}
```

using multiplicities

```
abstract class Composite {
  abstract @any Leaf getLeaves();
}

class Component extends Composite {
  @any Composite children;
  @any Leaf getLeaves() {


    return children.getLeaves();
  }
}

class Leaf extends Composite {
  @option Leaf getLeaves() {
    return this;
  }
}
```

Figure 4: Example of implementing a composite structure with and without using multiplicities (differences highlighted)

an assignment of an expression having multiplicity @any(List) to a variable having multiplicity @any(Set), identity may not be given (due to the dropping of duplicate objects). In practice, what it means for two multitudes to be identical (or only equal) is at least as variable as what it means for two collections to be equal, so that eventually, the programmer must be given control over this question (by letting her implement her own tests). Therefore, we made an arbitrary choice for == and implemented it as each object from each multitude occurring exactly the same number of times in both multitudes. Note that for a test of equality using the equals(...) methods provided for collections, the multitudes must be wrapped first (see Section 4.3).

Iteration over Multitudes While member access on a multitude results in an implicit (hidden) iteration over its objects (see "Member Access" above), there are iterations that require explicit access to the individual objects of the multitude, for instance to apply a filter, because there are case analyses to be made, or because the objects are to be used as arguments to operations or method calls (see Section 6.1 for examples). In these cases, wrapping a multitude in a collection (see Section 4.3) allows us to iterate over its objects as usual, i.e., using for, while, and do. For the special (and presumably most frequent) case of using the enhanced for loop, multitudes can be used in place of an iterable object without prior wrapping in a collection. E.g., we can write

```
for (Account account : accounts) ...
```

if accounts is declared as @any Account accounts. Note that, if the type of accounts was a subtype of Iterable, the for-loop would still iterate over the multitude, and not the elements of the iterable(s). This is also true if the (declared) multiplicity of accounts is @option (in which case the for-loop behaves more like an if-statement).

While the Java 8 Stream API adds another abstraction over collections which makes them more convenient to use by removing the need for external iteration in many cases, a stream is just another container — and hence another reification — of a multitude of objects. However, using the wrapping mechanism (see above and Section 4.3), the full repertoire of stream operations can be invoked on multitudes; in case of the above accounts example, one simply needs to write [[accounts]].stream()....

5 Implementation

We implemented multiplicities for Java as described above as an extension to JastAddJ [13], an extensible Java compiler implemented using reference attribute grammars [12; 20], and which currently supports Java 7 [30]. The extension comprises 44 source lines of JastAdd code for the syntax, 672 lines for the static semantics, and 1,180 lines for code generation. The multiplicity compiler can be downloaded from `https://bitbucket.org/joqvist/multiplicities`.

Abstract Syntax Abstract syntax is added to support multiplicity modifiers and expressions for wrapping/unwrapping, cardinality, and multiplicity casts, as shown in Figure 5. Each rule corresponds to a class representing an abstract syntax tree

(AST) node, extending and reusing existing classes in the JastAddJ compiler, like Modifier, Access, and Expr. Much of the static semantics behaviour, like name analysis, is reused as is from JastAddJ, but type analysis is refined in the extension, supplying new attribute grammar equations that define appropriate attribute values to handle multiplicities.

```
abstract MultiplicityModifier extends Modifier;
AnyModifier extends MultiplicityModifier ::=
  ContainerType:Access;
AnyDefaultModifier extends MultiplicityModifier;
OptionModifier extends MultiplicityModifier;
BareModifier extends MultiplicityModifier;
MultiplicityWrap extends Expr ::= Expr;
MultiplicityCardinality extends Expr ::= Expr;
MultiplicityCast extends Expr ::=
  Modifier:MultiplicityModifier
  [TypeAccess:Access]
  Expr;
```

Figure 5: Abstract syntax of extension with multiplicities

Type Analysis In JastAddJ, each type is represented by a unique AST node. Type checking, as used in assignment, parameter passing, etc., relies on the binary property of assignment compatibility which is implemented by comparing two type nodes, using double dispatch to encode the type lattice in an extensible way [13]. To handle types with multiplicities (other than *bare*), we construct synthetic multiplicity nodes that decorate ordinary type nodes. This allows us to compare different multiplicities with each other and with *bare* (non-decorated) types, again using the double dispatch pattern.

The synthetic nodes are constructed using the attribute grammar mechanism of *non-terminal attributes* (NTAs) [40], i.e., attributes whose values are new AST children. In JastAddJ, all attributes are computed automatically by the attribute grammar evaluator, and on demand, constructing only the synthetic decorating nodes that are needed for a particular program.

As an example, consider the following code fragment:

```
@any Account accounts;
...
accounts += new Account();
```

Figure 6 shows parts of the corresponding attributed AST. While the new expression is bound to the (*bare*) Account type, the declaration and access of accounts are bound to the AnyMult node that decorates the Account type.

Member Access Multiplicities affect the analysis of member accesses (qualified expressions). In regular Java code, the type of any qualified expression is the type of

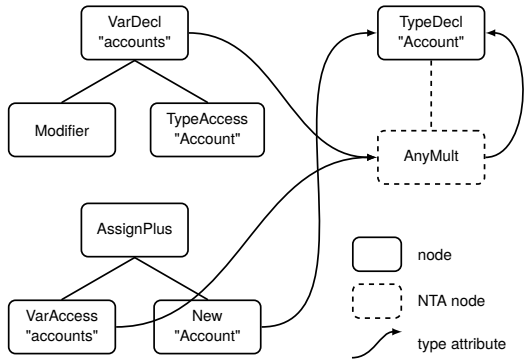


Figure 6: AST with reference attributes (see text)

the object on the right-hand side of the rightmost dot. The qualifiers are only used for looking up the declaration of the rightmost part. However, with multiplicities it is not sufficient to only look at the multiplicity of the rightmost part – the qualifying expression multiplicities may affect the multiplicity of the whole expression. For example, as discussed in Section 3.4, if the Account class has a field @option Bank bank, the expression accounts.bank will have the multiplicity *any*, although bank has the multiplicity *option*. This is handled by extending the type analysis with attributes to find the multiplicity of the left-hand part of a dot expression. The multiplicity of the entire dot expression is computed using both the multiplicity of the left and right parts, following Table 1.

Code Generation The code generation constitutes the bulk of the multiplicities implementation, translating from the higher-level operations on multitudes to corresponding lower-level for-loops, and handling all the different combinations of multiplicities in assignments and expressions. The implementation largely follows the translation scheme to a multiplicity-free program proposed in [37]; it is not repeated here for space reasons (Section 4 provided an outline, however).

Copying Hidden Collections In the bytecode, multiplicity *any* is represented by a collection object (the hidden collection), and care must be taken to not create aliases of these objects when multiplicity values are copied. For this reason, we create a copy of the collection object

- when passing an *any* as an argument to a method or constructor,
- when returning an *any* from a method,
- at an explicit (@any) cast, and
- when the multiplicity wrap expression either wraps or unwraps an *any*.

The assignment to an *any* does not need to create a copy of the hidden collection of the right-hand side — instead, the objects of this collection are added to the cleared hidden collection of the left-hand side.

Copying of collections can in many cases be avoided through static analysis and/or lazy copying: In certain cases, we can deduce statically that the original collection cannot be used anymore, for instance when returning the value of a local variable. In these cases, copying is not needed. Another optimization strategy is to represent a multitude by a wrapper object that contains an internal collection. Copying can then be implemented lazily by doing a shallow copy of the wrapper object, delaying the copying of the internal object until it is modified. By letting the wrapper keep a reference count, copying of the internal collection can be avoided for wrappers whose internal collection is not shared by other wrappers. We implemented such lazy collections as a library that we then benchmarked against the regular, non-lazy collections in our case study (see Section 6.3 for the performance discussion).

6 Case Study

To assess the impact of using multiplicities in a representative case study, we looked for a subject program

1. that uses a wide array of accepted object-oriented coding idioms so that multiplicities can be evaluated in a spectrum of constructions typically encountered in object-oriented programming,

2. that is tightly covered by test cases so that accidental changes of functional behaviour induced by the use of multiplicities would quickly be discovered, and

3. that can be run using file-based input that is openly available for reproduction of our performance observations (replication).

Given these criteria, we selected the widely known regression testing framework JUnit 4.0

1. since it is renowned for its consistent use of design patterns and, generally, for its exemplary object-oriented style,

2. since its own test suite is comprehensive, and

3. since open-source programs are available whose test suites execute it, giving us the standardized program runs we wanted.

Note in particular that every JUnit test suite tests not only the program under test, but also JUnit itself against the suite's oracle: all and only the tests of a program that pass using the original version of JUnit should also pass using our modified version using multiplicities. We selected JUnit 4.0 rather than one of its successors since it is manageable in size and since it contains fewer features that are not used by the majority of available JUnit test suites.

To manually obtain a version of JUnit that utilizes multiplicities in a way that is both reproducible and that we deem to be representative of how multiplicities will be used in practice, we changed the multiplicity of every field having a collection type whose element type (type parameter) is not a value type to @any (and removed the collection type from the type declaration), and every other field not having a value type to @option. We then changed the multiplicity of every other variable and method return as required by the assignment compatibility and overriding rules of multiplicities (see Section 4.6), unless where use of APIs (method invocation and subclassing) required *bare* parameters (in which case a cast to *bare* was introduced). The results of this procedure are summarized in Table 2.

	any	*option*	*bare*	total	value-typed[†]
fields	11	44	121	176	29
returns	9	34	249	292	108
formals	3	83	642	728	403
locals	7	34	482	523	380
casts to	2[§]	21[$]	92	115	
total	32	216	1586	1834	920

[†] included in *bare*
[§] implicit upcasts
[$] all downcasts from *any*

Table 2: Multiplicities in declarations and casts

6.1 How Introduction of @any Changes Program Source

There were 11 collection-typed fields whose type parameter (element type) was not a value type (cf. Section 3.6), and which we changed to multiplicity *any*. Of these, 5 were originally declared final ; in 4 of these cases, the final modifier had to be dropped since the multitudes were actually modified after initialization (cf. Section 2.7). Introducing the multiplicity @any (and changing the types; cf. Section 3.2) for these 11 fields required the subsequent change of 3 formal parameters (one of which involved the removal of a wildcard; cf. Section 2.3), of 9 method returns, and of 7 local variables (giving us a total of 30 introduced @any annotations; see Table 2). Of the remaining 27 uses of collections, 9 were required by APIs, 5 held instances of value types, 3 were required by concurrent modification (iter/remove; see Section 6.1), and the rest was used by local variables that never got a field assigned to it (they could also have been changed to multiplicity *any*).

Together with the introduction of @any annotations, we replaced 12 invocations of add(. . .) and 1 of addAll(. . .) with +=, and 2 invocations of remove(. . .) with −=; at the same time, 10 invocations of size() where replace with |[. . .]| and 5 invocations of isEmpty() were replaced with a test for null (cf. Section 3.2). There were 27 indexed accesses to list elements (using get(. . .)) in the original program where the list was

replaced by an *any* multitude; all but 5 of these could be removed using multiplicity downcasting (see Section 6.1); the remaining 5 required wrapping (see Section 6.1).

Loop Elimination

One of the supposed benefits of introducing *any* multiplicity is the elimination of loops (see Sections 2.1 and 3.4). And indeed, 5 for-loops over the elements of collections could be replaced by plain member access on a corresponding multitude of objects. For instance, we replaced the loop

```
for (Runner each : fRunners)
  each.run(notifier);
```

(from CompositeRunner.run(RunNotifier)) with

```
fRunners.run(notifier);
```

In addition, even where the iteration variable is not used as the left-most receiver in the loop expression (as above), it may still be possible to eliminate the loop. For instance,

```
for (Runner runner : fRunners)
  spec.addChild(runner.getDescription());
```

(from CompositeRunner.getDescription()) was rewritten to

```
spec.addChild(fRunners.getDescription());
```

after the multiplicity of the formal parameter description in

```
public void addChild(Description description) {
  fChildren += description;
}
```

had been changed to @any (note how this does not affect the implementation of addChild(...)).[2] However, because *bare* members may not be accessed on *option* or *any* receivers (Section 4.6), this required the declaration of @option as the returned multiplicity of getDescription(...) which, since *option* is not assignment compatible with *bare* (Section 4.1), required the subsequent introduction of 32 more @option annotations throughout the program. Yet, given that @any and @option annotations are designed to be used together, this does not appear to be counterproductive.

With a little redesign of programs, loop elimination can be pushed even further. For instance, we found (in method createTest(...) from class JUnit4TestAdapterCache) the loop

```
for (Description child : description.getChildren())
  suite.addTest(asTest(child));
```

[2] In fact, increasing parameter multiplicity of add...(@option) methods like the above allowed us to drop two addAll ...(@any) methods from the JUnit 4.0 source (they are now subsumed by the add ...(@any) methods).

Here, the loop variable child is the argument of another method so that introducing multiplicity @any for the parameter of addTest(...) as above is not sufficient — asTest(...) would need to be changed to accept and return *any* multiplicity as well, which would require a major reworking of its implementation. However, as it turned out, asTest(...) can straightforwardly be moved to class Description (the class of its formal parameter, using the refactoring Move Method [16]), so that the loop can be replaced by

```
suite.addTest(description.getChildren().asTest(this));
```

which is not only more succinct, but also more fluent[3] than the original phrasing. In fact, this minor refactoring even allowed us to remove a loop that was designed to fill a collection: we turned

```
List<Test> returnThis = new ArrayList<Test>();
for (Description child : description.getChildren())
  returnThis.add(child.asTest(this));
return returnThis;
```

(from JUnit4TestAdapterCache.asTestList(...)) into

```
return new ArrayList<Test>(
  [[description.getChildren().asTest(this)]]
);
```

in which the *any* multitude returned by asTest(...) is wrapped in a collection (note that API calls explicitly expect the collection here; hence the name of the method, "asTestList"!).

Of the 11 loops on the elements from a multitude that could not be removed, 2 contained accesses of value-typed members (methods for counting the number of leaves in a composite structure; see Figure 4 for how this can be simplified using collections), 3 used explicit iterators for removing elements from the originally underlying collection (cf. Section 6.1), and 6 had complex loop bodies that would have required major refactorings to cast them to member access on *any* expressions.

Wrapping and Unwrapping Multitudes

We required a total of 15 wrappings or unwrappings:

- In 5 cases, wrapping a multitude in a list was necessary because indexed access to individual objects was required and the dynamic multiplicity was not known to be 0 or 1 (in which case a downcast to *option* would have been sufficient; cf. Section 4.5).
- In 3 cases, wrapping a multitude into and subsequently unwrapping it from a local list-typed variable was necessary because of iter .remove() loops.
- The remaining 4 wrappings and unwrappings were due to API calls.

[3] "fluent" in the sense of a "fluent API":
see http://www.martinfowler.com/bliki/FluentInterface.html

Multiplicity Casting

In Java without multiplicities, a multiplicity upcast (from *option* to *any*) requires the wrapping of a single object in a collection. For instance, the method whose header is declared as

```
List<Throwable> getCauses(Throwable cause)
```

(from class ErrorReportingRequest) returns the expression Arrays.asList(cause) as a special case. In case cause was null, it would need to return an empty list, involving yet another clumsy construction (see Section 6.1). Using multiplicities, the same method is declared as

```
@any(List) Throwable getCauses(@option Throwable cause)
```

for which cause is a type-correct and multiplicity-correct return expression (the multiplicity upcast is implicit here).

Multiplicity downcasts (from *any* to *option*) are somewhat more involved. In standard Java, this would require the test of the size of a collection and, in case it is 1, the extraction of the sole element of the collection (the cast would result in null if size is 0, or else raise an exception). Indeed, we found 24 of constructions such as

```
Failure failure= result.getFailures().get(0);
assertEquals(expected, failure.getDescription());
```

in JUnit, which silently assumes that there is at least one failure and ignores possible failures beyond the first (actually, it leaves unstated whether there may be additional failures). Using multiplicities, we can rewrite the first line to

```
@option Failure failure= (@option) result.getFailures();
```

which makes the cast explicit and states that there should be at most one failure (which proved to be the correct assumption in 17 out of the 24 occurrences of this pattern).

Enforce Proper Encapsulation of Multitudes

As noted in Section 2.4, fields holding collections should be encapsulated and not be passed to clients via getters. Nevertheless, we find in JUnit's class Description the method

```
public ArrayList<Description> getChildren() {
  return fChildren;
}
```

allowing clients to bypass the public method addChild(...) supplied by the same class for directly manipulating the children of Description objects. After replacing the declaration of the field fChildren in Description with

```
@any(ArrayList) Description fChildren;
```

and adjusting the above declaration of getChildren() accordingly, an invocation of getChildren().add(...) will have no effect on fChildren, since getChildren() no longer returns an alias of it (Section 3.5). As it turns out, however, the sole occurrence of a manipulation of fChildren in JUnit via getChildren() is in the body of addChild(...) itself:

```
public void addChild(Description description) {
  getChildren().add(description);
}
```

Here, the idea of Self Encapsulate Field [16] clearly conflicts with how collections should be encapsulated (see Section 2.4). Using multiplicities, the body of addChild(...) is rewritten to

```
fChildren += description;
```

while that of getChildren() can remain as is, without granting true clients access to fChildren.

While the use of multiplicities enforces proper encapsulation as shown above, it can also help avoid explicit cloning, as found in class TestResult:

```
synchronized List<TestListener> cloneListeners() {
  List<TestListener> result= new ArrayList<>();
  result.addAll(fListeners);
  return result;
}
```

Here, using multiplicity *any* it suffices to return fListeners in the body of the method (which needs to remain synchronized — all multiplicity operations are non-synchronized by default).

Uniform Use of `null`

The fact that relating to no object in a to-many relationship is commonly represented by an empty collection (cf. Section 2.2) has led to the introduction of special collection classes (e.g., Collections.EmptyList, Collections.EmptySet, both from java.util) whose sole instances represent an empty collection. For instance, the method declared as

```
List<Throwable> validateAllMethods(Class<?> clazz)
```

(from class ParameterizedTestMethodTest) returns Collections.emptyList() as a special case (an upcast from multiplicity *none* to *any*). Replacing the declaration of the method with

```
@any(List) Throwable validateAllMethods(Class<?> clazz)
```

allows the method to return null instead, which has the same meaning as null for @option, i.e., is subsequently interpreted as no object (and, unless it is cast to @bare, cannot cause a null pointer exception). Note that the fact that, unlike Collections.emptyList(), the returned multitude is mutable maintains behavioural subtyping [22]: while

```
@any(List) Throwable result = null;
result += new Throwable("it's OK!");
```

is indeed OK, the seeming equivalent

```
List<Throwable> result = Collections.emptyList();
result.add(new Throwable("not OK!!"));
```

causes an "unsupported operation" exception.

Uniform Call Semantics

The fact that method calls are by value effectively (i.e., a method cannot modify the multitude that it gets passed; cf. Section 3.5) means that methods such as void Collections.sort(List<T>) (which would need to be rewritten to void Collections.sort(@any(List) T)) no longer work, simply since sorting has no effect on the multitude that is passed into the method. To fix this, we wrote our own sort method that returns a sorted multitude which can be assigned back to the variable holding the original multitude. Specifically, we changed 2 invocations of the kind

```
Collections.sort(fRunners, ...)
```

(here from class CompositeRunner) to

```
fRunners = Multiplicities.sort(fRunners, ...)
```

where Multiplicities is a helper class analogous to Collections. Note how this makes clear why fRunners cannot be declared final, since the multitude is in fact changed (even though the collection secretly holding it has remained the same object; cf. Section 2.7).

6.2 How Introduction of `@option` Changes Program Source

Changing the remaining fields that did not have value types to multiplicity *option*, and subsequently also formal parameters, method returns, and local variables as required by the rules of Section 4.1, gave us a total of 44 fields, 34 returns, 83 formal parameters, and 34 locals, all with multiplicity *option* (see Table 2).

Elimination of Tests for Not Null

Just like the use of *any* can eliminate loops, the use of *option* can eliminate tests for not null (Section 4.6). As it turns out, however, JUnit does not make much use of the value null representing "no object": in fact, in the whole of JUnit there is no test for not null on a field that could be declared with @option, and only a single test for null (which is however only used for lazy initialization of the field). However, there are some tests for not null on local variables, one of which,

```
Runner childRunner= Request.aClass(each).getRunner();
if (childRunner != null)
  runner.add(childRunner);
```

(from method ClassesRequest.getRunner()), we could rewrite to

```
runner.add(Request.aClass(each).getRunner());
```

This was possible since method add(...) accepts *any* multiplicity and getRunner() returns *option* multiplicity, and since null uniformly means "no object" for *option* and *any* multiplicities (see Sections 3.2 and 4.6).

Multiplicity Casting

While explicit and implicit multiplicity casts to *option* and *any* avoid clumsy coding idioms (Section 6.1), the current well-formedness rules of multiplicities may also require explicit downcasts to *bare* (cf. Section 4.5), which can be a nuisance. Specifically, the fact that value-typed members (which must be *bare*) may not be accessed on receiver expressions with multiplicity *option* (Sections 3.6 and 4.6) can require annoying casts. For instance, in

```
public int countTestCases() {
  return ((@bare) fRunner).testCount();
}
```

(from class JUnit4TestAdapter) the cast (@bare) is required since fRunner has multiplicity *option* (meaning that it may evaluate to no object) and testCount() returns an integer. Even though the cast (@bare) can be read as a warning that a null pointer exception may occur here (which can never occur when dereferencing *option* or *any* receivers; see Section 3.4), given that we needed to insert 62 such casts in JUnit (cf. Table 2; the remaining 30 casts to *bare* were needed for interfacing the JDK and assertions), not all programmers will regard this aspect of our language design as ideal. An elegant solution to this problem seems to be the introduction of *one* as an additional multiplicity annotation (for relating to precisely 1 object) and to allow access of *bare* members on *one* receivers with resulting multiplicity *bare*. However, since this would require our notion of multiplicities to be integrated with existing not-null annotations and checks, we have left this to future work (Section 8.1).

6.3 Performance Observations

To check the correctness of our multiplicity compiler, we ran JUnit's own test suite on our multiplicity-enhanced version of JUnit (named "JUnit-M"), and also on the test suites of three additional multiplicity-free benchmark programs listed in Table 3. All tests gave the same results, suggesting that the modified and the original version of JUnit are functionally equivalent. There were 25 test cases that failed in both versions (for AC Lang), because they require a newer runtime version of JUnit. We decided to keep these tests since they exercise failing behaviour in JUnit. All other tests that could be compiled with JUnit 4.0 (cf. Table 3) passed.

To check how multiplicities affected the execution time of JUnit-M, we compared running the following different compiled versions of JUnit:

- *ju4jc*: original JUnit 4.0 compiled using javac from OpenJDK 7, i.e., the reference compiler for Java.

- *ju4jj*: original JUnit 4.0 compiled using JastAddJ for Java 7.
- *ju4m*: JUnit-M 4.0 compiled with our multiplicity-enhanced compiler
- *ju4l*: JUnit-M 4.0 compiled with a variant of our multiplicity-enhanced compiler that uses lazy copying of collections, as described at the end of Section 5.

We used these four different versions of compiled JUnit to run the test suites in Table 3, all of which were compiled using javac.

Subject program and Version	*Number of Test Cases*
Apache Commons Codec 1.3	191
Apache Commons Lang 3.0	1923[†§]
Jaxen 1.1.6	716[$]
JUnit 4.0	255

[†] excluding 10 that we had to remove because they could not be compiled with JUnit 4.0
[§] 25 of these tests fail both with and without multiplicities because they should normally be run with JUnit 4.7 (see text)
[$] excluding 2 that were removed because they contain an infinite loop (see text)

Table 3: Subject programs used in the evaluation.

Steady-state performance We measured execution time in steady state, i.e., after running for a while so that the optimizing JIT compiler has *warmed up*, and reached a stable state. This is a relevant test scenario for long running applications. However, to use this method on the JUnit test suite, we had to remove two test cases that include an infinite loop that can be stopped only by a call to System.exit() (thereby terminating the host JVM).

To measure on steady state, we used the *multi-iteration determinism* method for benchmarking from Blackburn et al. [5], which includes the following steps:

1. The benchmark is iterated $N - 1$ times in the same JVM to achieve steady-state for the JIT.
2. JIT optimization is then turned off to not further affect the measurements.
3. One more iteration of the benchmark is made, but is not measured.
4. Finally, K iterations are made, measuring the execution time of each.

During different runs (consisting of $N + K$ iterations of the benchmark), the JIT may stabilize on different states, due to the non-determinism of the JIT optimization. For this reason, we make R runs for each benchmark, and compute the arithmetic mean of the means of the K iterations in each run, and the 95% confidence interval, as

	ju4jc	ju4jj	ju4m	ju4l	ju4m/ju4jj loss
AC Codec					
mean	172	169	178	204	0.0530
conf. int.	±16.5	±15.2	±15.5	±2.7	
AC Lang					
mean	6747	6748	6750	6758	0.0003
conf. int.	±2.6	±2.6	±2.2	±2.3	
Jaxen					
mean	249	248	249	258	0.0033
conf. int.	±3.1	±1.0	±1.3	±1.3	
JUnit					
mean	845	848	853	867	0.0053
conf. int.	±2.2	±2.3	±2.6	±3.1	

Table 4: Execution times (in msecs).

suggested by Georges et al [17]. For our measurements we chose $R = 15$, $N = 30$, and $K = 20$. Table 4 shows the results of our steady-state experiments.

In comparing *ju4jj* and *ju4m*, we anticipated there to be a performance loss due to copied collections and extra null checks. We can see that there is a tendency to a slight performance loss when using multiplicities for all four benchmarks. However, the confidence intervals overlap, and the difference between the means is only 5% for AC Codec, and less than 1% for the other benchmarks. We therefore regard the performance loss as negligible. We can also note that the performance of the javac compiler and the JastAddJ compiler (*ju4jc* and *ju4jj*) are almost the same, indicating that the results should transfer to javac, should one wish to implement multiplicities there.

Using Lazy Copying Table 4 also shows the results from running *ju4l*, i.e., JUnit-M compiled with a variant of our multiplicity-enhanced compiler that implements lazy copying of collections, as discussed at the end of Section 5. Unfortunately, the results show that the use of lazy copying degrades the performance, rather than improving it. We measured the number of copied collections (Table 5) and their sizes, and found that around half of the collections had size 0 and that less than 1% had a size larger than 10. Apparently, because the collections are so small, the cost of copying the hidden collection is lower than the cost of delegating all method calls through an intermediate lazy collection. Further investigation is needed to see if the implementation of the lazy copying can be improved, and if it can be useful for other benchmarks.

6.4 Discussion

As the examples of Figure 4 and Figure 4 suggest, savings in terms of the number of tokens used in a program fragment can be considerable. Also, Section 6.1 pre-

	AC Codec	*AC Lang*	*Jaxen*	*JUnit*
collection copies	1007	20326	8038	8105
avoided using lazy	778	16516	6554	6884
checking not null	1045	8417	3112	3310

Table 5: Instruction overhead

sented several interesting examples of loop elimination enabled by member access on multitudes of objects. In a complete program, however, savings are diluted, and the total number of tokens can even increase because of the additional annotations required in declarations. In fact, in our case study the multiplicity-enhanced version has 151 more tokens than its original. However, this increase is explained by the additional annotations used in declarations, whereas the number of tokens in the other statements (instructions) are reduced. The possible reduction of tokens in the instructions is currently diminished by the casts to *bare* that we had to introduce for interfacing with API code and accessing *bare* members (Section 6.2). We expect these numbers to improve with the introduction of *one* as an additional multiplicity, and of course with the migration of APIs.

7 Related Work

Smalltalk not only comes with a powerful collections library, with its indexed instance variables it also offers a way of directly associating one object with a multitude of other objects, without reifying this association [19]. However, since indexed integer variables are unnamed (they are similar to the so-called indexers of the .NET languages [23]), there can be only one set of indexed instance variables per object, limiting their use for implementing relationships (of which an object may have many). And yet, indexed instance variables share with our *any* fields that two objects cannot share the same set of indexed instance variables (i.e., there is no aliasing of multitudes).

The object constraint language (OCL) [6; 27], which is used to express conditions of well-formedness of UML models, allows the dereferencing ("navigation") of attributes and associations with arbitrary multiplicities using the dot notation. However, OCL still reifies multitudes of objects using collections; the difference between one and many objects (singular and plural) is mitigated only slightly by allowing collection operations to be applied to single objects also. This is different for Alloy [21], a textual modelling language which maps object-orientation to relational logic and in which the notion of multiplicity is also prominent. Unlike OCL, Alloy does not distinguish between scalars and sets, and treats scalars as singletons. This largely removes the differences between one and many objects from Alloy expressions (which we strive for also); however, like OCL, Alloy is not a programming language.

The programming languages JavaFX™ [38] and Cω [2] offer sequences, or streams, as array-like type constructors for variables with multiplicities greater than 1. Like arrays, sequences are reified multitudes of objects; however, unlike arrays,

they are immutable and have value semantics. Sequences cannot be nested — any attempt to do so results in a flat sequence. null in the context of a sequence means the empty sequence and a scalar value means a singleton sequence, so that both can be assigned to a sequence-typed variable. In Cω, a stream can occur as the receiver of a member access; this access is then mapped over the elements of the stream, yielding a stream of the member type (so that chained member accesses on streams are possible). While this generalized member access has the same semantics as corresponding expressions in OCL and Alloy, the suitability of streams (which have been subsumed by iterators in C# 3.0 [4]) for implementing relationships to many objects is limited by their immutability.

The semantics of our static multiplicity *option* is somewhat similar to using the Option class in Scala [29]: a receiver of type Option can be None, in which case applying a function (using map or flatMap) produces None. Similarly, a function can be applied to a collection (again using map or flatMap), resulting in a collection of the same type, containing the return values. The main difference to object-oriented programming with multiplicities as put forward here is that we use no container types, but instead separate type from multiplicity, avoiding the awkward dominance of the container type over the content type [37] imposed by wrappers such as Option and collections. Another difference is that in object-oriented programming with multiplicities as we implemented it, the use of flatMap to apply functions to *option* and *any* multiplicities is implicit.

Ungar and Adams have recently presented a parallel programming language Ly that offers so-called *ensembles* as an alternative to collections [39]. Ensembles accommodate member objects that, when the ensemble is sent a message, all respond in parallel. However, unlike our multitudes of objects, an ensemble in Ly is a first-class object, and a singleton ensemble is different from the object that it contains. Since Ly is untyped, runtime checks are required to avoid that an ensemble contains itself (which may lead to infinite recursion when a message received by an ensemble is forwarded to itself). Also, empty ensembles are currently not integrated seamlessly, and demand further dynamic checks. It seems that the multiplicities described in this paper would solve at least some of the problems incurred by ensembles (but notably not those related to parallelism).

While implementing relationships to many objects using collections (or similar reifications of multitudes) is by far the most commonly used pattern [26], automatic mappings from object-oriented models to programs may introduce other, more sophisticated patterns [18]. Both are however challenged by integrating relationships in object-oriented programming as a native concept.

As far back as 25 years ago, Rumbaugh argued for the lifting of the field-and-collection based relationship encodings of object-oriented programs to the level of a first class language construct [34]. For this purpose he introduced relations as instances of a special class Relation that has fields holding a relation declaration (i.e., the types of the participants, role names, cardinalities, etc.), as well as a field holding the extension of the relation (i.e., its tuples). Unlike in many other approaches that followed, an instance of Relation represents a relation, not a tuple; standard operations Rumbaugh defined on these instances included the adding and removal of tuples, indexed access to tuples of the relation, and scanning of the relation (iterating

over its tuples). Later, Rumbaugh also added propagation attributes to relations which allowed the controlled recursive propagation of certain method invocations through object graphs [35]; however, this is not to be confused with our lifting of method invocations from single objects to multitudes of objects. While Rumbaugh's proposals amount to embedding a native implementation of (parts of) a relational database system in object-oriented programs, our approach of implementing to-many references is lightweight. Also, our relationships (represented by multitudes of objects) are not first-class.

Østerbye picked up Rumbaugh's proposals and presented a Smalltalk-based association compiler that can choose between internal and external implementations of relationships [32]. An internal implementation keeps the information which other objects an object is related to local to the object, whereas an external implementation uses first class relationship objects for this purpose. Independent of the implementation choice, Østerbye, like Rumbaugh before him, offers role-based and association-based access to relationships. However, in his role-based access protocol, he distinguishes between to-one and to-many relationships, continuing the discontinuity we want to rid programming of. This discontinuity is preserved in Østerbye's subsequent work [31], in which he leaves the untyped realm of Smalltalk to present a library-based approach for C#. In his library, association classes are complemented by role classes providing for internal implementation of relationships. However, given the fundamental meaning relationships have in most problem domains, we argue for a native, rather than a library-based, integration of relationships.

Bierman and Wren's RelJ is based on a formalized notion of relationships as first class types whose instances, called relationship instances, are tuples [3]. These tuples, which — like objects — can have state and behaviour, are created and returned by adding a pair of objects to a relationship. Navigation of a relationship always results in a set having value semantics, making the result of navigation covariant with the target type of the navigation [3]. However, sets cannot be the source of navigation, so that navigation cannot be chained as in our approach. Bierman and Wren also suggest how multiplicities can be restricted statically, using *one* (for $[0, 1]$, analogous to our *option*) and *many* (for $[0, *]$, analogous to our *any*) annotations; the invariant imposed by *one* is then enforced by changing the semantics of adding to a relationship with that of replacing an instance of a relationship (destructive update, or assignment). By contrast, we have restricted the additive update (+=) to *any* multiplicities, and require a downcast from *any* to *option* for an assignment to *option*, protecting us from a silent change of behaviour when a multiplicity is changed from *any* to *option*.

The relationship aspects of Pearce and Noble use the intertype declarations of AspectJ to shift the bookkeeping necessary for maintaining relationships between objects from the objects to relationships [33]. The relationships are coded as aspects which can carry additional, relationship-specific behaviour. Class definitions remain ignorant of the relationships for which they supply the participants, which is considered an increase in the separation of concerns. This separation goes too far, however, when an object needs access to others it is related to — in that case, it has to query the relationship it was to be kept unaware of.

In the language Rumer, references to objects are completely expelled from so-called entity types (conventional classes), and objects are related exclusively through

relationship types [1]. It follows that, analogous to the relationship aspects of Pearce and Noble [33], only relationships know which entities are related (referred to as stratification in [1]). Entity and relationship types have associated extent types which are instantiated and populated explicitly by the programmer. Relationships can be nested, and relationship extents can be owned by relationships, so that they cannot escape the owning relationship. While owned relationship extents bear some resemblance to our multiplicities (which likewise cannot be aliased), the whole approach seems rather heavy weight — in particular, with all knowledge about relationships fully encapsulated in relationships (so that objects are ignorant of whether and how they are related), much of an application's logic (including that captured in most methods) has to be moved to relationships, with objects being degraded mostly to passive data containers with identity. This means a fundamental paradigm shift for object-oriented programming, and migrating an existing application to the concepts embodied in Rumer will amount to a major redesign effort.

8 Future Work

8.1 Integrating NonNull

As noted in Section 6.2, introduction of multiplicity *one* would help avoid an unpleasant restriction concerning the access of *bare* members via *option* receivers. The multiplicity *one* is equivalent to annotating a type use as being *NonNull* like in, for example, the Checker framework [11]. Additionally, (@one) can be used as a cast on an expression. Fähndrich and Leino showed how *NonNull* can be implemented to handle initialization correctly, introducing the notion of raw types [15]. This solution has been implemented for a previous version of JastAddJ [14]. A natural next step for us is thus to extend our implementation of multiplicities with this solution, supporting multiplicity *one*. We expect this to allow us to replace the multiplicity of most *bare* variables with *one*, and hence to reduce the number of casts substantially, as *one* expressions can safely be used as arguments to library methods requiring bares, and *bare* members (value types!) can safely be accessed on *one* receivers. Additionally, by adding type annotations, as introduced in Java 8, @NonNull annotations can be represented by *one* multiplicities, and be typechecked by the compiler.

8.2 Qualified Access

As noted in Section 4.1, a collection C used in an @any(C) annotation must have a single type parameter representing the type of the elements of the collection. This requirement excludes maps from a key type to a value type (such as HashMap<K, V>). Not excluded, but not especially supported are indexed collections (like ArrayList<E>), which are special maps (with positive integers as keys): read access of the i^{th} object to which an expression e with multiplicity *any(List)* evaluates currently requires the clumsy workaround [[e]].get(i); write access is even clumsier (not shown here). For qualified access of the objects among a multitude, lists and maps can be generalized to associative arrays, effectively implementing the quali-

fied associations of UML [28]. However, we have not yet investigated the language extensions this would require.

8.3 Case Studies on Modelling and Grammar Frameworks

Our current case study focuses on making use of multiplicities for ordinary Java code. Another interesting focus for case studies would be to focus on modelling frameworks such as EMF, where an API is generated from a metamodel expressing relations with cardinalities. It would be interesting to investigate how multiplicities could be used to simplify both the API and its usage. Furthermore, an interesting avenue of research would be to investigate to what extent metamodels can be automatically computed from code using multiplicities, reducing the gap between models and code.

In a similar manner, it would be interesting to investigate how multiplicities can simplify abstract syntax tree APIs, as generated by many compiler tools from EBNF-like formalisms. Here, there is a natural match between the typical *child*, *optional* and *list* constructs and the *one*, *option* and *any* multiplicities.

8.4 Refactoring to Multiplicities

In our experiment described in Section 6 we refactored JUnit manually from ordinary Java code to code using multiplicities. An interesting opportunity for further research is to design automated refactorings for this purpose. Based on the current case study we can see that most of these refactoring cases are fairly simple (they are related to a Change Declared Type refactoring), but also that there are a number of challenges that need to be addressed to find a general refactoring approach.

9 Conclusion

Letting expressions evaluate to any number of objects (rather than just one), and handling multitudes of objects that are not reified as one object, means a departure from object-oriented programming as we know it. In this paper, we have picked up a proposal for implementing object-oriented programming with multiplicities presented at last year's Onward! conference [37], and turned it into a fully functional compiler of Java 7 that can handle multitudes of objects as proposed. We tested this compiler by changing the source code of JUnit 4.0 so that it utilizes multiplicities, and by using the binaries produced by the compiler in place of the original binaries for running a number of different open source test suites on their programs under test. Functionally, both binaries are equivalent; furthermore, observed performance measures suggest that using multiplicities in JUnit 4.0 imposes only minor penalties. At the same time, a detailed analysis of the changes performed on the JUnit sources suggests that programs can indeed be simplified using multiplicities, avoiding many of the peculiarities imposed by using collections as containers of multitudes.

Acknowledgments

This work was in part financed by the Swedish Research Council under grant 621-2012-4727.

References

[1] Stephanie Balzer and Thomas R. Gross. “Verifying Multi-object Invariants with Relationships”. In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Ed. by Mira Mezini. Springer, 2011, pp. 358–382.

[2] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. “The Essence of Data Access in Cω”. In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. Ed. by Andrew P. Black. Springer, 2005, pp. 287–311.

[3] Gavin M. Bierman and Alisdair Wren. “First-Class Relationships in an Object-Oriented Language”. In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. Ed. by Andrew P. Black. Springer, 2005, pp. 262–286.

[4] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. “Lost in translation: formalizing proposed extensions to C#”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 479–498.

[5] Stephen M. Blackburn et al. “Wake up and smell the coffee: evaluation methodology for the 21st century”. In: *Commun. ACM* 51.8 (2008), pp. 83–89.

[6] Jordi Cabot and Martin Gogolla. “Object Constraint Language (OCL): A Definitive Guide”. In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Springer, 2012, pp. 58–90.

[8] Peter P. Chen. “The Entity-Relationship Model - Toward a Unified View of Data”. In: *ACM Trans. Database Syst.* 1.1 (1976), pp. 9–36.

[9] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[10] Steve Cook and John Daniels. *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice Hall, 1994.

[11] Werner Dietl et al. “Building and using pluggable type-checkers”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald Gall, and Nenad Medvidovic. ACM, 2011, pp. 681–690.

[12] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26.

[13] Torbjörn Ekman and Görel Hedin. “The JastAdd extensible Java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[14] Torbjörn Ekman and Görel Hedin. “Pluggable checking and inferencing of nonnull types for Java”. In: *Journal of Object Technology* 6.9 (2007), pp. 455–475.

[15] Manuel Fähndrich and K. Rustan M. Leino. “Declaring and checking non-null types in an object-oriented language”. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. Ed. by Ron Crocker and Guy L. Steele Jr. ACM, 2003, pp. 302–312.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 57–76.

[18] Dominik Gessenharter. “Implementing UML Associations in Java: A Slim Code Pattern for a Complex Modeling Concept”. In: *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*. RAOOL ’09. Genova, Italy: ACM, 2009, pp. 17–24.

[19] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[20] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[21] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2011.

[22] Barbara Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841.

[23] Microsoft Corporation. *C# Language Specification v1.2*. `http://download.microsoft.com`.

[24] Maurice Naftalin and Philip Wadler. *Java generics and collections*. O'Reilly, 2006.

[25] James Noble, Jan Vitek, and John Potter. "Flexible Alias Protection". In: *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*. Ed. by Eric Jul. Springer, 1998, pp. 158–185.

[26] James Noble. "Basic relationship patterns". In: *Pattern Languages of Program Design 4*. Addison-Wesley, 2000, pp. 73–89.

[27] Object Management Group. *Object Constraint Language Version 2.2*. `http://www.omg.org/spec/OCL/2.2`.

[28] Object Management Group. *UML Superstructure V2.2*. `http://www.omg.org/spec/UML/2.2/Superstructure`.

[29] Martin Odersky. *The Scala Language Specification*. 2009.

[30] Jesper Öqvist and Görel Hedin. "Extending the JastAdd extensible Java compiler to Java 7". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. Ed. by Martin Plümicke and Walter Binder. ACM, 2013, pp. 147–152.

[31] Kasper Østerbye. "Design of a Class Library for Association Relationships". In: *Proceedings of the 2007 Symposium on Library-Centric Software Design*. LCSD '07. Montreal, Canada: ACM, 2007, pp. 67–75.

[32] Kasper Østerbye. "Associations as a Language Construct". In: *TOOLS Europe 1999: 29th International Conference on Technology of Object-Oriented Languages and Systems, 7-10 June 1999, Nancy, France*. IEEE Computer Society, 1999, pp. 224–235.

[33] David J. Pearce and James Noble. "Relationship aspects". In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*. Ed. by Robert E. Filman. ACM, 2006, pp. 75–86.

[34] James E. Rumbaugh. “Relations as Semantic Constructs in an Object-Oriented Language”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’87), Orlando, Florida, USA, October 4-8, 1987, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, 1987, pp. 466–481.

[35] James E. Rumbaugh. “Controlling Propagation of Operations Using Attributes on Relations”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’88), San Diego, California, USA, September 25-30, 1988, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, 1988, pp. 285–296.

[36] James E. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1990.

[37] Friedrich Steimann. “Content over container: object-oriented programming with multiplicities”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH ’13, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld. ACM, 2013, pp. 173–186.

[38] K. Topley. *JavaFX Developer’s Guide*. Developer’s Library. Pearson Education, 2010.

[39] David Ungar and Sam S. Adams. “Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance”. In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 19–26.

[40] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. Ed. by Richard L. Wexelblat. ACM, 1989, pp. 131–145.

Extraction-Based Regression Test Selection

Abstract

Frequent regression testing is a core activity in agile software development, but large test suites can lead to long test running times, hampering agility. By safe RTS (Regression Test Selection) techniques, a subset of the tests can be identified that cover all tests that can change result since the last run. To pay off in practice, the RTS overhead must be low. Most existing RTS techniques are based on dynamic coverage analysis, making the overhead related to the tests run. We present Extraction-Based RTS, a new safe RTS technique which uses a fast static analysis with very low overhead, related to the size of the modification rather than to the tests run. The method is suitable for program-driven testing, commonly used in agile development, where each test is a piece of code that uses parts of the system under test. We have implemented the method for Java, and benchmarked it on a number of open source projects, showing that it pays off substantially in practice.

1 Introduction

Frequent automated regression testing is an essential part of agile software development, whether it is done on a continuous integration server, or by the developers as part of the development cycle [Bec99]. As a project grows, the time for running regression tests increases, which can hinder agile developement by increasing the development iteration time.

Jesper Öqvist, Görel Hedin, and Boris Magnusson. "Extraction-Based Regression Test Selection" In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'16), ACM, pp. 5:1–5:10.* Lugano, Switzerland, 2016.

During agile development, it is common to make use of automated *program-driven testing*, where each test case is implemented as code, typically using *xUnit*, i.e., a testing framework based on Beck's *sUnit* framework for Smalltalk [Bec94]. While the name xUnit alludes to unit testing at the class level, an xUnit test case can call any code, and can thus be used for testing at any granularity, including subsystem testing, integration testing, and complete system testing. Another commonly used approach to automated testing is to use *input-driven testing*, where a program is run with different sets of input data, and where each data set corresponds to a test case.

An agile developer typically writes the program-driven tests along with production code, and runs the complete test suite frequently. In the most extreme case, using the Test-Driven Development methodology, the developer runs the test suite after creating each new test, after completing each piece of new or changed functionality, and after each refactoring [Bec03]. It is thus of great importance to keep the test running time short, in order to reduce the work interruption of running tests. In addition, the tests are usually run regularly on a build server, where test runs consume valuable time and power.

To reduce testing time, Regression Test Selection (RTS) methods can be used, selecting a subset of the test suite to re-run after a program modification. A multitude of RTS methods have been published, see, for example, the following surveys [RH96; Bis+11; YH12]. An RTS method is said to be *safe* if it will run all tests that have changed result after a program modification, under certain conditions [RH96]. Running a safe RTS method is thus equivalent to running the complete test suite. An RTS algorithm is more precise the fewer tests it runs in addition to the safe subset.

To pay off, the overhead of running the RTS algorithm must be lower than the time it would take to run the tests not selected [LW91]. To achieve this, the right balance between precision and time spent on analysis should be found: a very precise analysis could be so slow that it does not make up for the time saved by the reduced testing time, thus defeating its purpose. To be useful in practice, it is not necessary that RTS pays off for each run, but it should pay off on the average. Additionally, to use RTS in agile development, it is desirable that the overhead is low enough to not be noticed for runs when the RTS does not pay off, for example when all tests are selected.

In this paper, we present *Extraction-Based RTS*, a new safe RTS method for program-driven testing. Our method is static, coarse-grained, and incremental, computing a dependency graph over code files, including both production files and test files. The dependency graph represents an *extraction* of each test program, i.e., a subset of all program files, sufficient for running the test program. The dependency graph is incrementally updated after changes to the project. This makes the RTS overhead related to the size of the modification rather than to the project size or to the number of tests run. Most previous work is instead based on dynamic coverage analysis, giving an overhead related to the time for running the selected tests.

Extraction-Based RTS is related to methods for computing program extractions, i.e., methods for reducing the code footprint for applications [AU94; Tip+02]. However, since one of our main goals is to have a low RTS overhead, we prioritize quick extraction computation over minimizing extraction size, so we have chosen to use a coarse-grained extraction algorithm.

To evaluate our new method, we have implemented a tool, AutoRTS[1], that supports extraction-based RTS for Java projects with JUnit tests, and measured its performance on several Open Source projects.

Our main contributions are the following:

- A new safe regression test selection method for program-driven testing, Extraction-Based RTS (Section 2)
- An incremental reverse dependency algorithm that makes the RTS overhead scalable to large projects (Section 3).
- A concrete instantiation of the algorithm for Java (Section 4).
- An open source tool, AutoRTS, that implements Extraction-based RTS for Java and JUnit (Section 5).
- Evaluation of the method on open source Java projects, showing that the method pays off substantially in practice, and that the overhead is very low (Section 6).

Section 7 compares to related work, and Section 8 provides a concluding discussion.

2 Extraction-Based Test Selection

We model a project (the system under test) as a set of *code files*, F, of which a subset $T \subseteq F$ are *test files*. The remaining files $P = F \setminus T$ are called *production files*. Each test file $t \in T$ serves as a main program that needs a subset of F for its execution, termed its *minimal extraction*, $e_{min}(t)$.

We assume that everything outside of F that could affect the program behavior is stable, e.g., library code, data files, runtime system, operating system, etc. Reading data from the environment, like the current system time, can affect the control flow of a test, and while we allow this type of nondeterministic behavior we assume, for simplicity of presentation, that the test is not *flaky* [Luo+14], i.e., we assume that it is written so that it produces the same result regardless of the path taken. In the terminology of Rothermel and Harrold, the test is *deterministically fault revealing* [RH96]. Under these circumstances, a test t will produce the same result for every run, given that no files in its minimal extraction, $e_{min}(t)$, are modified. Consequently, if only code files outside of the minimal extraction are modified, we do not need to rerun the test. Our method trivially generalizes to handle flaky tests too: there is no reason to rerun a flaky test if we know that its minimal extraction is unchanged.

Code files could be either in source code form, or in a processed form, like bytecode or binary object code. We will primarily consider the case of source code, but see 4.1 for a discussion of bytecode and binary code.

[1] AutoRTS is available under an Open Source license at `https://bitbucket.org/joqvist/autorts`

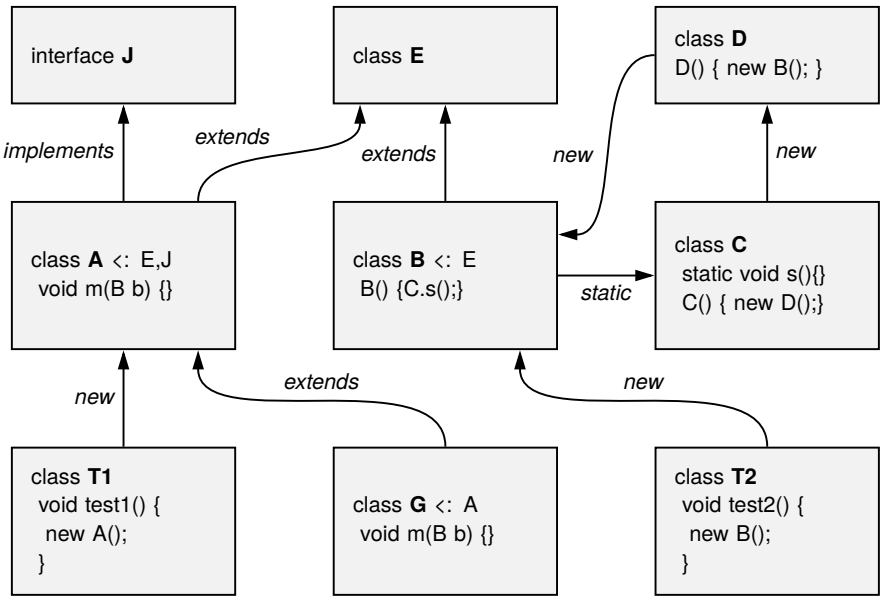


Figure 1: An example project with files (boxes) and dependency edges (arrows with labels indicating reason for the edge). Note that there is no edge from A to B although A mentions B in its code: the file B is only needed for programs that create B objects, and thus have another dependency on B.

Computing the minimal extraction is, in general, undecidable.[2] Instead, we compute the *extraction* $e(t)$, a conservative approximation of the minimal extraction. I.e., $e_{min}(t) \subseteq e(t) \subseteq F$. Before giving the algorithm for computing extractions, we provide some examples.

2.1 Examples

The extraction is computed by maintaining dependency edges between files, representing which other files a given file needs for its execution. As we will see, a file A that mentions an entity in another file B, does not necessarily give rise to a dependency, which may at first seem counter-intuitive.

Figure 1 shows an example Java project with two test files, T1, T2, and seven production files, J, E, D, A, B, C, G. The notation X <: Y, Z means that X is a subtype of Y and Z. The dependency edges are labelled to illustrate the different reasons for introducing them (*extends*, *implements*, *static*, and *new*).

Consider the test case T1. Its extraction, i.e., the files needed to execute it, is its transitive closure with respect to the dependency graph:

$$e(\mathsf{T1}) = \{\mathsf{T1}, \mathsf{A}, \mathsf{E}, \mathsf{J}\}$$

[2]To know if a code line is executed requires knowing if the line is reachable, equivalent to the halting problem.

The type `A` is needed in the extraction since `T1` creates a new instance of it. The types `E` and `J` are needed since they are supertypes of `A`, and are thus needed to initialize the new `A` object.

Note that although `A` refers to the type `B`, there is no edge from `A` to `B`. If `A.m(b)` is called with some `B`-object `b`, there must be a path from the main program to `B`, otherwise, it would not have been possible to create the `B` object. For `T1` there is no such path, and `B` is not needed for the execution of `T1`.

Similarly, we can note that `A`'s subclass `G` is not part of `T1`'s extraction. Although `G` overrides `A`'s method `m`, that overriding method could only be invoked if there is an instance of `G` on the heap, in which case there must be a path from the main program to `G`.

Now, consider test case `T2` with the following extraction:

$$e(\texttt{T2}) = \{\texttt{T2}, \texttt{B}, \texttt{E}, \texttt{C}, \texttt{D}\}$$

This example illustrates that static methods also need to be taken into account, and that there can be loops in the dependency graph.

`T2` also illustrates a case of imprecision, i.e., where the extraction of a test will be greater than the minimal extraction: In running `T2`, the statement `new D()` will never be executed, and class `D` is not part of `T2`'s minimal extraction. Through a slower method-level analysis we would have been able to exclude `D`, but due to the fast coarse file-level analysis, we get this imprecision. A modification of `D` will thus result in our method selecting `T2`, although its execution result will be the same.

Libraries

Recall that in our model, libraries are considered to be part of a stable environment that is not changed. Yet, there can be execution paths from the project code to a library, and back again via callbacks. How does this affect analysis and extractions? With our method, it is not necessary to analyze libraries, even if there are callbacks to the project code. The reason is that a library does (by definition) not know anything statically about the project code, and it can therefore only do callbacks to objects that the project code passes to it, and that the project code therefore already is dependent on.

This is illustrated in Figure 2 where test case `T3` creates objects of subclasses to library classes, and calls the library method `m1` which in turn calls another library method `m2` that is overridden in the project code (i.e., a callback). The callback to `m2` can go to `Q`'s `m2` method, which `T3` is already dependent on. However, there can be no callback to `R`'s `m2` method, since the `T3` program does not create any `R` objects.

If anything that is part of the stable environment actually is modified, for example, if a library is updated to a new version, all tests are considered to be affected, and have to be rerun. It would be possible to improve this by including libraries, e.g., jar-files, as nodes in the dependency graph. By keeping track of which project files depend on a particular library, only test programs dependent on those files would need to be rerun when that library is updated. The library itself would still not need to be analyzed.

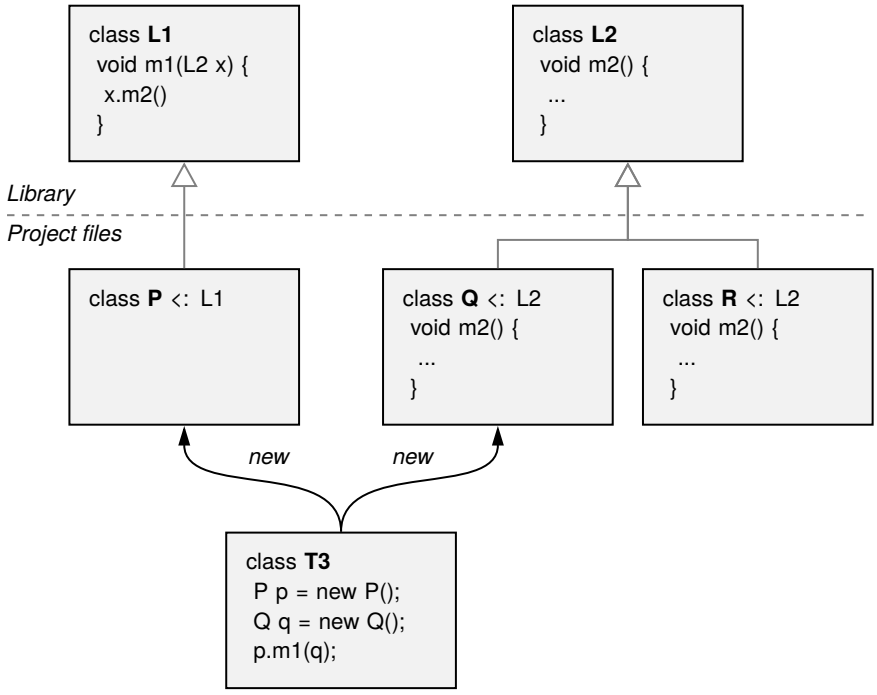


Figure 2: The library does not need to be analyzed even if there are callbacks: it can only do callbacks to entities that the test program is already dependent on. Grey arrows indicate inheritance.

If a library has only made internal changes betweeen two specific versions, with no externally visible effect on the project code, then switching between either of those versions of the library can be ignored with respect to test selection.

2.2 Extractions

To compute extractions, we construct a file dependency graph $G = \langle F, D \rangle$ where the code files F are the vertices, and the edges D are file dependencies of the form $(f \rightarrow g)$. The extraction of a test t is then simply the file t and all files transitively reachable from t, i.e., the transitive closure of t:

$$e(t) = \{t\} \cup \bigcup_{(t \rightarrow f) \in D} e(f)$$

A dependency $(f \rightarrow g)$ means that the execution of code in f directly depends on static code elements in g. Examples of this includes the access of static variables, calls to static methods, and calls to object initializers. As discussed in the examples, the mere mentioning of a type does *not* induce a dependency. For example, if f contains a method call `r.m()`, where `r` is a receiver of a type declared in g, this will *not* induce a dependency $(f \rightarrow g)$. The reason is that at the time the call is made, the object `r` can only exist if there is another (transitive) dependency from the main program to g (due to the code that created the object). In the context of a dynamically loading language, like Java, a dependency $(f \rightarrow g)$ means that execution of code in f may cause code in g to be dynamically loaded. We will discuss this in detail in Section 4.

This way of treating dependencies is fundamentally different from many other methods for analyzing test dependencies, and relies on the fact that we trace dependencies from *main programs* (test cases), and use the dependencies to compute *extractions*, i.e., a subset of the total set of files that is guaranteed to include the files needed to run the program. Other code analyzing RTS methods, e.g., DejaVOO [OSH04], work in a different way (see Section 7), and insert dependencies for all types used.

As mentioned previously, the analysis needs only to take the code files F in the project into account, and not the library code. This is because library code is (by definition) compile-time independent of the project, i.e., the library does not directly use static code elements in F. Note that this does not hinder the library to call code in F indirectly, e.g., through callbacks on function pointers, or by calling virtual methods declared in the library but implemented in classes in F, since such calls do not induce dependencies.

Algorithm 1 Build the complete dependency graph for a file set F

```
procedure BuildGraph(F)
    D ← ∅                                          ▷ Edge set D
    for all f ∈ F do
        G ← GetDeps(f)
        add {(f → g) | g ∈ G} to D
    end for
    return ⟨F, D⟩                                  ▷ The dependency graph
end procedure
```

Our analysis works for introspective reflection, where only the state of the heap is inspected, since such code does not access static code elements. However, to handle general reflection, which may cause new code to be dynamically loaded at runtime, our method needs to be complemented, for example with manually added dependencies.

We have chosen to build the dependency graph at the file level, to make the graph small and the analysis fast. Naturally, this gives imprecision as compared to finer-grained analysis on the class level, or on the method level. Since many files contain only a single class, we do not believe that class-based analysis would make a big difference in practice. Doing analysis at the method level would lead to a much more complex, and time-consuming, analysis that might not pay off, although future investigations in this direction would be interesting.

2.3 Building the dependency graph

Building the dependency graph at the file level allows each file to be analyzed locally. Assume we have a function GetDeps(f) that analyzes a file f and returns the set of other files it uses static code elements from (i.e., a local analysis of f). Building the graph from scratch for a project is then done by adding the local dependencies for each file to the graph, as shown in algorithm BuildGraph.

2.4 Selecting tests

To find the dependent tests after modifications to files, it is actually not the extraction sets that are interesting, but the reverse information, that we call *reached tests*. The reached tests $r(f)$ of a file f is the set of test cases that have f in their extraction. I.e.,

$$r(f) = \{t \in T | f \in e(t)\}$$

For example, consider the project in Figure 1. Here, the reached tests for `A` is $r(\texttt{A}) = \{\texttt{T1}\}$, and for `E` it is $r(\texttt{E}) = \{\texttt{T1}, \texttt{T2}\}$.

After modifications to a subset $M \subseteq F$ of the code files, the selected tests $S \subseteq T$ is then simply the union of all the reached tests sets for all the modified files:

$$S = \bigcup_{f \in M} r(f)$$

Algorithm 2 Compute the set of tests to select, given the dependency graph $\langle F, D \rangle$, the set of modified files $M \in F$, added files $A \notin F$, deleted files $X \in F$, and test files $T \in (F \cup A)$.

```
procedure SelectTests(⟨F, D⟩, M, A, X, T)
    S ← ∅                                              ▷ Selected tests S
    unmark all files in F
    for all m ∈ M do
        SelectReached(m)
    end for
    add A ∩ T to S                                     ▷ Select new tests
    return S                                           ▷ Return the selected tests

    where
    procedure SelectReached(g)
        if g is not marked then
            mark g
            if g ∈ T then
                add g to S                             ▷ Select reached test
            end if
            for all f such that (f → g) ∈ D do
                SelectReached(f)
            end for
        end if
    end procedure
end procedure
```

This set of selected tests can be found by marking the modified files, recursively traversing the dependencies backwards from each marked file, marking all files found on the way, stopping the traversal at already marked files, and collecting the resulting marked test files, as shown in the algorithm SelectTests.

Algorithm 3 Update the dependency graph $\langle F, D \rangle$, given a set of modified files M, added files A, and deleted files X.

```
procedure UpdateGraph(⟨F, D⟩, M, A, X)
    add A to F
    for all f ∈ (M ∪ X) do                          ▷ Remove outdated edges
        remove all edges (f → ...) from D
    end for
    remove X from F
    for all f ∈ (M ∪ A) do                          ▷ Add new edges
        G ← GetDeps(f)
        add {(f → g) | g ∈ G} to D
    end for
    return ⟨F, D⟩                                   ▷ The updated dependency graph
end procedure
```

3 Updating the dependency graph

After modifications and rerunning of the selected tests, the dependency graph needs to be updated. Instead of recomputing it from scratch, it can be updated incrementally by removing the outgoing dependencies from each modified file, reanalyzing those files, and adding the dependencies corresponding to the new content of the files. For added files the corresponding node needs to be added together with outgoing dependencies. For deleted files the corresponding node is removed from the dependency graph. The algorithm UpdateGraph describes how to update the graph.

If the analysis is done at the source level, additional files might need to be analyzed, because due to name shadowing, a modification may change the meaning of names in other files. We discuss this for Java in Section 4.1.

Note that extraction-based RTS is not dependent on having to build and run all the tests in order to do the first test selection. A user can thus check out a project, run the batch analysis to get the first dependency graph for the code, do modifications, then run the incremental analysis to select and run affected tests. Running the batch analysis is typically faster than running all tests. Other test selection methods that rely on running instrumented tests cannot avoid the initial run. Avoiding the initial test run can be an important advantage, allowing developers to quickly start to work after checking out a project. Furthermore, avoiding instrumented tests speeds up testing altogether.

4 Dependencies for Java

As a concrete example of extraction-based RTS, we will consider the Java language. Please refer back to Figure 1 for illustration of the different cases.

When running a Java program, the main program is loaded and executed, and additional code modules are loaded on demand, depending on the executed code. The constructs that lead to code loading are *extends*, *implements*, *static*, and *new*:

- *Extends.* If a class `A` extends another class `E`, running the code for instantiating `A` will cause the loading of `E`. The code of `E` is needed both for initializing the new `A` object with any fields declared in `E`, for initializing any static fields in `E`, and for allowing methods in `E` to be called on `A` objects. We therefore add a dependency from the file containing `A` to the file containing `E`.
- *Implements.* If a class `A` implements an interface `J`, running the code for instantiating `A` will cause the loading of `J`. The code for `J` is needed both for initialization of static fields in `J`, and, since Java 8, to allow calls to default methods in `J`. We therefore add a dependency from the file containing `A` to the file containing `J`.
- *Static.* If there is code accessing a static field or calling a static method of a class or interface `T`, running that code will cause loading of `T`, to be able to access the field, or call the method. We therefore add a dependency from the file containing that code to the file containing `T`.
- *New.* If there is code that instantiates a class `C`, using the `new` construct, this will cause loading of `C`. The code for `C` is needed for instantiating the new `C` object, and for allowing access to its methods or fields. We therefore add a dependency from the file containing the `new` construct to the file containing `C`.

Based on these observations, the function GetDep(f) (used in BuildGraph and UpdateGraph) is implemented simply by traversing the file, and computing the accessed files in the *extends*, *implements*, *static*, and *new* constructs.

A key observation is that the mere reference of a type in the code, for example in a method signature, does not cause that type to be loaded. For example, consider the example in Figure 1 again, where class `A` contains a method `m` which takes an argument of type `B`. Neither loading of the bytecode of `A`, nor calling its method `m` will trigger any loading of `B`. The code for `B` is not needed until methods on `B` are called, and at that point in time, the code for `B` must already have been loaded in order to create the `B` object. Thus, we do not need to include any dependency from `A` to `B`: any program that will call `m` with a `B` object argument, will have another dependency on `B`.

4.1 Source versus bytecode analysis

Extraction-Based test selection is a general technique that can be applied by analyzing source code as well as compiled formats of the code, e.g., bytecode or binary code. If the analysis is done on source code rather than on bytecode, adding or deleting a type may cause type accesses in other types in the same package to change meaning. Consider the example in Figure 3. Here, class `A` in package `p` has a wildcard import to a package `q`, and accesses a class `B`. In version 1, this access is bound to `q.B`. However, in version 2, a new class `B` is added to the package `p`. This class will shadow `q.B`, causing the `B` access to be bound to `p.B`, and thereby change the meaning of the code in `A`. The test `T` that depends on `A` will thus need to be rerun if `p.B` is added.

Such changes of the semantics, due to adding or removing types, can only occur within a package in Java. This is because Java does not allow accessing a type via a

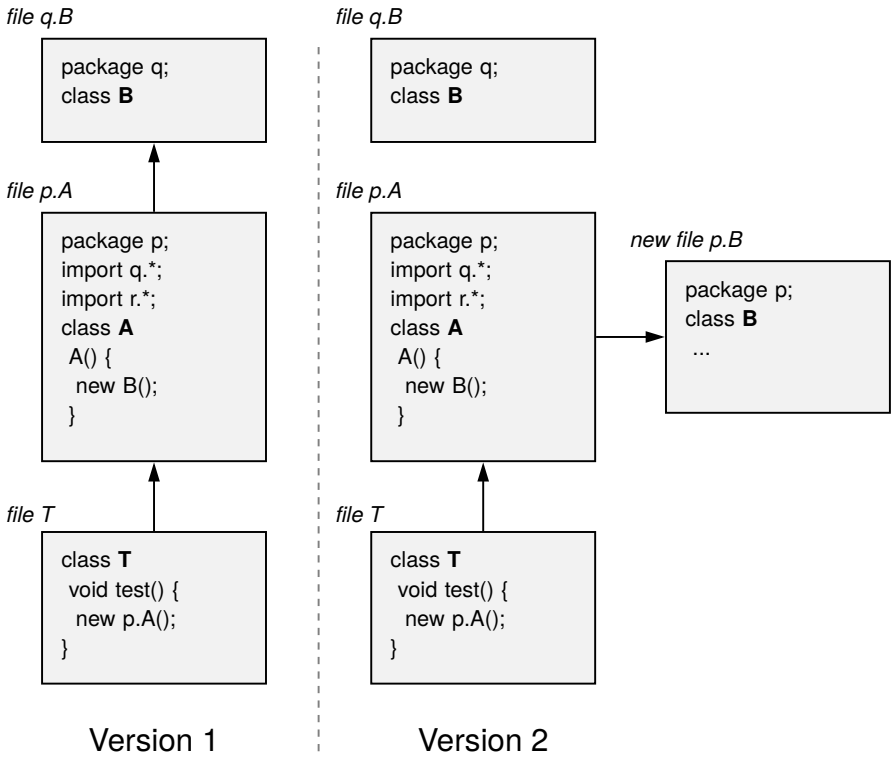


Figure 3: In version 1, `A`'s access `new(B)` is bound to `q.B`. In version 2, the file `p.B` has been added. This causes the access `new(B)` to change meaning because of shadowing, binding it to `p.B`.

wildcard import if there is another wildcard import that would also match the access. Thus, in version 1, adding a class `B` to package `r` would cause a compile-time error for class `A`. For this reason, if a type is added or deleted from a package, it is sufficient to recompute the dependencies for all the files in that package.

An analysis at the bytecode level does not have this issue, since the bytecode only contains fully qualified type names, and no wildcard imports.

In our tool, AutoRTS, we have chosen to implement the method at the source level. While the implementation is slightly more involved due to handling shadowing, it has the advantage that it is not necessary to compile the complete project in order to do the test selection.

For a language like C, the technique could be used on the binary object code. The dependency analysis could then be done on the symbols referenced between different object files, i.e., in principle corresponding to the work a static linker like `ld` does.

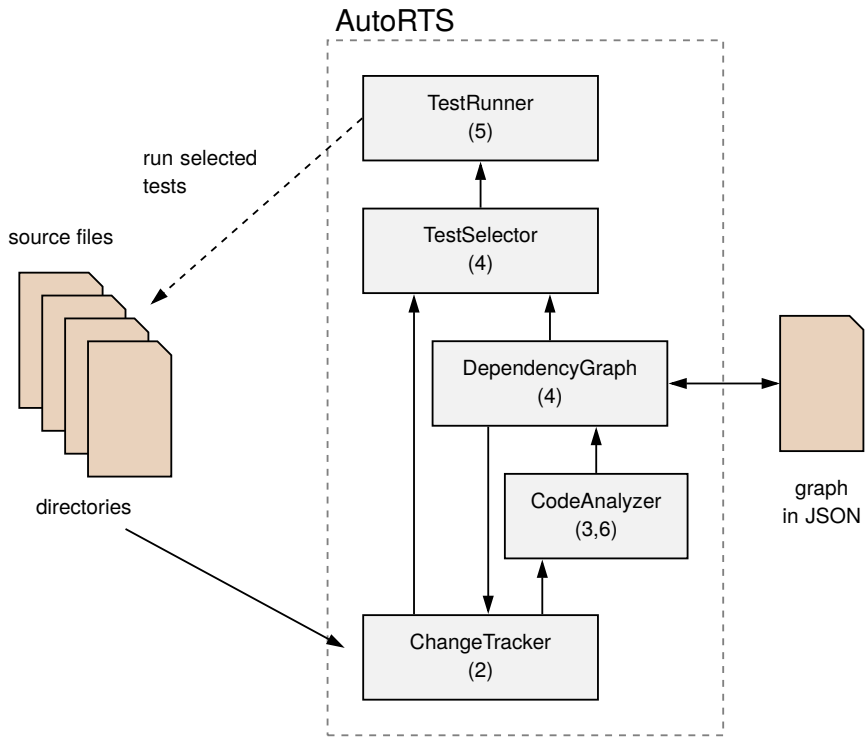


Figure 4: The AutoRTS tool. Solid arrows indicate information flow.

To find out which source files are modified after a change, the file system can be scanned for files with new timestamps, which is very fast. A tool working at the bytecode or binary level could also use timestamps provided that an incremental compiler is used that does not update the files unnecessarily. For a batch compiler that updates all files, a file differencing technique would need to be used, for example computing a sufficiently long hash code and comparing to the hash code for the previously analyzed file. Our method could be made more precise by using semantic differencing between files to only detect modifications that had a semantic effect. This would increase the modification checking time, but the extra precision might make it worthwhile. The Java bytecode format itself is resilient to some non-semantic changes such as indentation and comments. Implementation using bytecode analysis would be interesting future work.

5 Tool implementation

We have implemented a standalone tool, AutoRTS, that supports extraction-based test selection for Java and JUnit, using analysis on the source code. Instead of running the complete test suite using JUnit, the user runs AutoRTS which selects a safe subset

of the tests and runs them using JUnit. Figure 4 shows the different parts of the tool, and how they interact with the file system.

1. The *dependency graph* is stored in a JSON file between runs of AutoRTS.
2. The *change tracker* scans the source directories to identify modified, added and deleted source files since the last test run, by comparing file time stamps to those stored in the dependency graph.
3. The *code analyzer* parses changed or added files, collecting outgoing dependencies, and identifying if the file contains JUnit tests.
4. The *test selector* selects tests to run, based on the changed files and the dependency graph.
5. The *test runner* runs the selected tests.
6. The *code analyzer* analyzes the changed files to update the dependency graph to be ready for the next run of AutoRTS.

The code analyzer is implemented as an extension to the extensible Java compiler ExtendJ [EH07b; ÖH13][3]. The parsing, abstract syntax tree construction, and type lookup is reused from ExtendJ, and extended with dependency analysis, as explained in Section 4.

6 Evaluation

To evaluate the effectiveness of Extraction-Based RTS we have conducted an empirical study attempting to simulate Java application development, by replaying the commit histories of five Open Source Java projects and using AutoRTS to perform test selection.

In the following sections we describe the research questions addressed in the study, objects of study, variables, threats to validity, experiment process, and results and analysis.

6.1 Research Questions

To help us investigate how efficient Extraction-Based RTS is for Java applications, we posed the following research questions for our empirical study:

- *How does using our RTS technique compare to running all tests for a Java application?*
- *How large is the overhead for using AutoRTS on real-world Java projects?*

[3]ExtendJ was previously named JastAddJ.

6.2 Objects of Study

For our experiment we selected five Java applications ranging in size from 20-254K lines of code. The applications we selected are: Apache Commons Lang 3.0 (ACLang), Closure Compiler, Functor, Jaxen, and JUnit.[4] Each of the selected projects has a public repository with a long history of commits that we could replay to simulate development. Table 1 shows some metrics for the projects.

The projects we selected are well-known in the Open Source community, and in particular JUnit could be expected to use a rigorous unit testing methodology for development. We found that testing regimes vary between projects and can even change significantly during the commit history of a single project.

Project	*SLOC*	$\|P\|$	$\|T\|$	*Revision*
ACLang	65K	280	122	e1ad4b1
Closure	254K	706	271	8edc042
Functor	21K	226	170	3da1a4b
Jaxen	20K	300	77	1405
JUnit	26K	394	152	47707e8

Table 1: Sizes for the last version measured on each benchmarked project. *SLOC* is the number of source lines of code, excluding whitespace and comments for all files (both production and test code), measured using the tool *cloc*. $|P|$ is the number of production files. $|T|$ is the number of test files. *Revision* is the Git commit hash or Svn Revision number for the last version measured.

6.3 Variables

The test selection technique measured is an independent variable of the empirical study. We measured two different test selection techniques: *Select-All*, and Extraction-Based RTS. Select-All runs all tests without any analysis, while Extraction-Based RTS is performed using AutoRTS which first analyzes the source code and then selects a subset of tests to execute.

The total running time for each test selection technique is a dependent variable.

6.4 Threats to Validity

External threats to validity include our choice of Java projects to study and the commits measured. We tried to avoid bias and selected well-known and easily available projects with JUnit tests and at least 500 commits in the commit history. We did not change the selection or measured commits based on how Extraction-Based RTS performed on the selected projects.

[4] git://git.apache.org/commons-lang.git
https://github.com/google/closure-compiler
https://github.com/apache/commons-functor
https://svn.codehaus.org/jaxen/trunk/jaxen/
https://github.com/junit-team/junit

The size and number of projects measured limits the general applicability of our results. Given that Java coding styles and program architecture can vary greatly between projects, our results should be regarded only as informative examples. To draw more general conclusions, a larger study would be needed.

Internal threats to validity include possible bugs in our implementation. To mitigate this risk, we wrote tests for our tool during implementation of the analysis, we also checked that our tool detected all available tests for a subset of the commits we measured. Furthermore, the extensible compiler ExtendJ, that AutoRTS builds on, has its own large test set helping to ensure the Java source analysis works well.

6.5 Experiment Process

We simulated development on each of the selected Java projects by using the commit log to replay commits in chronological order. For each project a series of commits were checked out in order, and for each commit we measured the total testing time using Select-All and AutoRTS. When using AutoRTS the analysis was based on the changes introduced in the current commit. We recorded statistics about the test selection, such as the time spent running selected tests, how long it took to analyze dependencies and update the dependency graph for each commit, and how long time it took to run all available tests.

Commits that failed to compile or did not change any source files were skipped during the experiment process. Compilation failures were caused by, e.g., missing libraries, dependencies on old Java versions, and changes to source folder locations. Each of the benchmark projects has changed build system, source folder locations, and library dependencies multiple times over the course of their development. We could not account for every such change, but we tried to fix build errors where possible. The number of skipped commits, due to errors or lack of source changes, is listed per project in Table 2.

Project	Total commits	Compile errors	No changes	Measured
ACLang	1524	44	364	1116
Closure	1500	131	307	1062
Functor	819	116	155	548
Jaxen	1009	92	196	721
JUnit	1314	28	336	950

Table 2: Classification of commits used. Commits with compile errors and commits where no source files were changed were excluded from the measurements.

6.6 Benchmark Environment

All measurements were taken on a 4-core 3.6GHz Intel i7-3820 CPU, with 64 GiB RAM, running Linux Mint 17.0.

To compile the benchmark projects the following Java versions were used:

- Java SE2 1.4 (Sun Java `1.4.2_19-b04`)
- Java 7 (Oracle Java `1.7.0_45-b18`)
- Java 8 (Oracle Java `1.8.0_71-b15`)

6.7 Results and Analysis

Figure 5 shows mean execution times for groups of 20 consecutive commits for each Java project. In each figure, the dots show the mean Select-All running time, and the stacked bars illustrate the AutoRTS running time, where the upper green bar represents test running time, and the lower grey bar represents the analysis time. The analysis time includes time spent to identify modified, added, and deleted files, and it includes time spent selecting which tests to run (algorithm SELECTTESTS), and the time used to update the dependency graph for the project (algorithm UPDATEGRAPH).

The test running time using AutoRTS is lower than running all tests with Select-All, on average, for ACLang and JUnit. For Functor and Jaxen the mean AutoRTS running time drops below the Select-All running time in the later parts of their commit histories. Closure is the only project where the Select-All running time remains slightly lower than AutoRTS (8% lower for the last 40 commits) during the entire measured commit history.

For all projects except Closure, if testing time continues to be similar to the latest commits, using AutoRTS would save time for continued development. If testing and analysis times continue to follow a similar trend the gap would widen for future commits. To summarize the results for the most recent commits from each project, Table 3 shows the AutoRTS analysis and test running time as a percentage of Select-All time for the last 40 measured commits in each project. For the last 40 commits, average total testing time is reduced by by 37-87% for all projects except Closure, and increased by 8% for Closure. We thus conclude that Extraction-Based RTS pays off for ACLang, Functor, Jaxen, and JUnit.

Project	*Analysis (%)*	*Run Tests (%)*	*Total (%)*
ACLang	3	40	43
Closure	24	84	108
Functor	5	8	13
Jaxen	16	47	63
JUnit	6	44	50

Table 3: Average analysis, test running, and analysis plus test running time (*Total*) using AutoRTS, as percentage of Select-All running time, for the last 40 commits.

Figure 5: Testing time, means for bins of 20 commits. Test selection pays off when the red dot is above the stacked bars.

Project Differences

As can be seen by test and analysis running times each project behaves differently. These differences are due to a number of factors. Below are our hypothesis of some of the important differences between the projects.

Jaxen and Closure use system-level tests where most tests exercise large parts of the whole system. Jaxen is an XPath library where most tests send inputs to the core XPath parser, and Closure is a JavaScript compiler where many tests use JavaScript code as input and the test validates the compiler output against an expected output. For many commits in Jaxen, and even more so in Closure, most of the tests needed to be run – even after commits that only changed one source file. This is due to the tests using a central parsing part of the system under test which has a large set of transitive dependencies.

JUnit and ACLang in general have tests that depend on fewer parts of the system under test, thus small changes are less likely to trigger many tests to be run. In these projects commits that changed a single file triggered a smaller percentage of the total test set.

ACLang has a notable difference from JUnit in that a few small commits triggered nearly all tests to run. Upon inspecting a subset of such commits we saw that they were changing central utility classes that were used as helpers to do various simple string and collection manipulation in very many places throughout the code.

All projects except Closure show some large changes for test running times during the commit history of the project. In some cases this is caused by single large refactorings as can be seen in JUnit, or gradual changes where more tests are introduced as seen in Jaxen and JUnit. ACLang and Functor show some large spikes where many tests were added during a few commits. The trend for ACLang, Functor, and Jaxen, seems to be an increasing test set over time.

Analysis Overhead

The overhead incurred by checking file modifications, re-analyzing dependencies, and selecting tests, did not vary much based on project sizes. In fact, the analysis time per line of code was lower for the projects with larger code sizes. This is not surprising since the cost is dominated by the analysis of the modified files, i.e., the size of the commit, and not by the total size of the project. Table 4 shows average analysis time, in milliseconds, per thousand lines of code for each project.

The AutoRTS analysis is incremental in that only modified files (including all files in the packages if files are added or deleted) are analyzed. The analysis time thus depends mostly on the size of the change rather than the size of the project. The method should therefore scale to large projects, which our measurements confirm.

Although our method is very coarse-grained, and thus imprecise, it pays off substantially. Naturally, it will pay off more for projects that use clearly separated tests and that test different code paths in the program.

Project	*kSLOC*	*time (s)*	*ms/kSLOC*	*Mod/ci*
ACLang	65	0.491	7	3.7
Closure	254	0.983	4	6.6
Functor	21	0.357	16	10.9
Jaxen	20	0.339	17	3.0
JUnit	26	0.406	15	12.5

Table 4: kSLOC = thousand lines of source code. Time (s) = average analysis time in seconds. ms/kSLOC = analysis time in milliseconds per thousand lines of code. Mod/ci = mean of the number of file modifications per measured commit with at least one modification.

7 Related Work

Regression Test Selection (RTS) is a well researched area, and there are several surveys of different methods [RH96; Bis+11; YH12]. To our knowledge, Extraction-Based RTS is the first safe RTS method that only relies on static analysis of source code, and not dynamic analysis using instrumentation of code. Early RTS methods include TestTube [CRV94] and DejaVu [RH97], both for C. TestTube relies on running instrumented tests to associate each test with a set of coarse program units, such as function definitions and global variables. DejaVu uses a more fine-grained method, and computes a statement level control-flow graph from the system under test, and relies on running instrumented tests to associate each test case with a set of edges in the control-flow graph. Variants of these methods were later developed for Java [OSH04; SR07].

Early studies [RH97; BRR01] divided development into a *preliminary* phase where data could be gathered about the initial version of the program and its tests, and a *critical* phase where modified programs were regression tested. Typically, the cost for the preliminary phase has been ignored in empirical investigations [RH97; BRR01; OSH04; SR07]. However, in agile development, there are no such phases, and in our view, the time for running instrumented tests should be considered part of the overhead. It might well be the case that the time for running instrumented tests outweighs what is gained by the test selection.

Despite the large amount of research on safe RTS, there are few practical tools available [Gli+14]. One recent practical tool is Ekstazi [GEM15] that performs RTS for Java. Like the previously mentioned methods, Ekstazi performs dynamic analysis by instrumenting the code. The analysis is performed at the file level, by dynamically instrumenting the bytecode, and monitoring the execution to identify accessed class files as well as files explicitly accessed from the user program. In contrast to much earlier work, their empirical studies does include the overhead for running the instrumented tests, and they report average time savings on many projects.

Discussion

In comparing Extraction-Based RTS to instrumentation-based methods, we identify the following main advantages:

- Extraction-Based RTS is safe also for non-deterministic programs. In contrast, instrumentation-based methods are safe only provided that the system under test is deterministic.
- The overhead for Extraction-Based RTS is very small, making it negligible in practice. This is partly because the analysis is coarse-grained and incremental, and partly because it is proportional to the size of the modification, which is usually small. This is important for the worst-case scenario, when all or most tests are selected, such as when a central module is modified. In our experience, these scenarios are fairly common. In contrast, an instrumentation-based method will have an overhead proportional to the number of tests run, which can be substantial. For example, Ekstazi reports a couple of data points where an individual run increases from roughly 15 seconds for Select-All to roughly 27 seconds for their test selection [GEM15].
- With Extraction-Based RTS, a project can be checked out and test selection can start without having to first run all tests. This can be important when running all tests takes a long time.

There are also potential disadvantages of Extraction-Based RTS. First, because the method supports program-driven rather than input-driven testing, it works only when the tests are structured as separate programs (e.g., like JUnit tests), and not when the tests are structured as separate input data files. Future work could address the problem of automatically converting input-driven tests to program-driven tests. Second, because the analysis is static and coarse-grained, sacrificing precision but not safety for speed, it is more conservative than dynamic and finer-grained instrumentation-based methods, and will therefore select more tests. We think that depending on the structure of the code in a project, it can either be more advantageous to use Extraction-Based RTS, or more advantageous to use instrumentation-based RTS. More research is needed to investigate this in detail.

Extraction-Based RTS is a safe method, with the goal of conceptually running all tests, but saving time by not having to run tests whose outcome is guaranteed to be unchanged. Even if a safe RTS algorithm pays off, it might not reduce the testing time sufficiently. Safe RTS can then be combined with unsafe RTS methods to further reduce the time used, using heuristics to select the tests deemed most important. These methods can also be combined with test prioritization which uses heuristics to run tests in some order of importance, to find failing tests faster, or to run until a fixed testing time budget has been used up. For examples of such heuristics, see, for instance, the recent survey by Yoo and Harman [YH12], and the recent work by Elbaum, Rothermel, and Penix [ERP14]. While many such techniques are instrumentation-based, there are also static methods that are reported to give similar performance [Mei+12].

8 Conclusion

We have presented Extraction-Based RTS, a new safe regression test selection method, suited for program-driven testing, i.e., where tests are formulated as code, using a

framework like JUnit. In contrast to other safe RTS methods, it is based on static analysis rather than dynamic instrumentation-based analysis. This, in combination with a coarse-grained incremental analysis, gives a low overhead, which is dominated by the size of the change rather than by the number of tests selected. The overhead consists of the time for scanning changed files, selecting tests to run, and updating the dependency graph. A low overhead is important in the rather common situation when all tests are selected, like when changing a key class, since the penalty for using the method then is low. For our subject programs, ranging from 20-254 kLOC, the overhead was typically less than half a second, corresponding to a small fraction of the time for running all tests, and thus negligible from a practical point of view.

On the average, the method pays off well, though it did not not pay off for one out of five benchmarked programs, in which case the extra testing time was around 8% higher. For subject programs where our method payed off, the average running time, including overhead, ranged between 13-63% of the time for running all tests. We could also see that the average payoff changes during the development of a program. For a couple of the subjects, the method did not pay off initially, but did so when the project started to grow. Since we measured individual commits, which often consist of multiple individual changes, the payoff would be even larger during development if tests are run for smaller intermediate changes.

In comparison to dynamic methods that are based on code instrumentation, our method is safe also for non-deterministic programs that do not necessarily take the same path in each run.

It is important to note that Extraction-Based RTS works for program-driven tests rather than for input-driven tests. An interesting avenue for further research would be to develop methods for generating program-driven tests from input-driven tests, thereby making the method applicable also in these cases. It would also be interesting to combine the method with unsafe RTS methods to be able to skip tests also in that setting.

Acknowledgments

We thank the anonymous reviewers. This work was partly financed by the Swedish Research Council under grant 621-2012-4727.

References

[AU94] Ole Agesen and David Ungar. “Sifting Out the Gold: Delivering Compact Applications from an Exploratory Object-oriented Programming Environment”. In: *OOPSLA’94, Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, USA, October 23-27, 1994*. ACM, 1994, pp. 355–370.

[Bec94] Kent Beck. “Simple Smalltalk testing: with patterns”. In: *The Smalltalk Report* 4.2 (1994), pp. 16–18.

[Bec99] Kent Beck. “Embracing change with extreme programming”. In: *Computer* 32.10 (1999), pp. 70–77.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BRR01] John Bible, Gregg Rothermel, and David S Rosenblum. “A comparative study of coarse-and fine-grained safe regression test-selection techniques”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10.2 (2001), pp. 149–183.

[Bis+11] Swarnendu Biswas et al. “Regression test selection techniques: A survey”. In: *Informatica: An International Journal of Computing and Informatics* 35.3 (2011), pp. 289–321.

[CRV94] Yih-Farn Chen, David S Rosenblum, and Kiem-Phong Vo. “TestTube: A system for selective regression testing”. In: *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press. 1994, pp. 211–220.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 1–18.

[ERP14] Sebastian Elbaum, Gregg Rothermel, and John Penix. “Techniques for improving regression testing in continuous integration development environments”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 235–245.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. “Practical regression test selection with dynamic file dependencies”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 211–222.

[Gli+14] Milos Gligoric et al. “An empirical evaluation and comparison of manual and automated test selection”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014, pp. 361–372.

[LW91] Hareton KN Leung and Lee White. “A cost model to compare regression test strategies”. In: *Proceedings. Conference on Software Maintenance 1991*. IEEE. 1991, pp. 201–208.

[Luo+14] Qingzhou Luo et al. “An empirical analysis of flaky tests”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 643–653.

[Mei+12] Hong Mei et al. “A static approach to prioritizing junit test cases”. In: *Software Engineering, IEEE Transactions on* 38.6 (2012), pp. 1258–1275.

[ÖH13] Jesper Öqvist and Görel Hedin. “Extending the JastAdd extensible Java compiler to Java 7”. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM. 2013, pp. 147–152.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. “Scaling regression testing to large software systems”. In: *ACM SIGSOFT Software Engineering Notes* 29.6 (2004), pp. 241–251.

[RH96] Gregg Rothermel and Mary Jean Harrold. “Analyzing regression test selection techniques”. In: *Software Engineering, IEEE Transactions on* 22.8 (1996), pp. 529–551.

[RH97] Gregg Rothermel and Mary Jean Harrold. “A safe, efficient regression test selection technique”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.2 (1997), pp. 173–210.

[SR07] Mats Skoglund and Per Runeson. “Improving class firewall regression test selection by removing the class firewall”. In: *International journal of software engineering and knowledge engineering* 17.03 (2007), pp. 359–378.

[Tip+02] Frank Tip et al. “Practical extraction techniques for Java”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24.6 (2002), pp. 625–666.

[YH12] Shin Yoo and Mark Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

Concurrent Circular Reference Attribute Grammars

Abstract

Reference Attribute Grammars (RAGs) is a declarative executable formalism used for constructing compilers. Previous work has extended RAGs with circular (fixed-point) attributes, higher-order attributes, and collection attributes. In this paper we present wait-free concurrent attribute evaluation algorithms for Circular RAGs. These algorithms enable interactive queries to be performed with low latency while heavier computations are running.

We design and evaluate a lock-free implementation of our algorithms in Java, for the JastAdd metacompiler. Our implementation can be used without further changes to existing JastAdd-specified compilers, provided they fulfill well-formedness conditions like observationally pure semantic functions. Our evaluation on a JastAdd-specified compiler for the Java programming language shows that our approach is useful for reducing latency, and can give a slight overall speedup.

1 Introduction

Reference Attribute Grammars (RAGs) [Hed00] have proven useful for generating extensible compilers for languages like Java [Wyk+07; EH07b] and Modelica [Åke+10]. They are supported in several attribute grammar systems, for example JastAdd [Hed00], Silver [Wyk+08], Kiama [SKV10], JavaRAG [FCH15], and RACR[Bür15].

Extended version of
Jesper Öqvist and Görel Hedin. "Concurrent Circular Reference Attribute Grammars". In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017), pp. 151–162*. Vancouver, BC, Canada, 2017.

A RAG is a declarative formalism defining attributes in an Abstract Syntax Tree (AST), through directed equations attached to the rules of an abstract grammar. In a generated compiler, an attribute *evaluator* computes attribute instance values in an AST in order to compile the program.

Typically, attributes are evaluated sequentially, in a single thread. Concurrent evaluation could provide several advantages, like lower evaluation latency. In interactive systems, for example IDEs, it is typically desired to keep the response time below 0.1 seconds, to ensure that users perceive the tool as reacting instantaneously [Nie93]. By using concurrency, an interactive task can be performed within this time limit even while longer-running analysis tasks run in the background. Another advantage of concurrent evaluation is to run threads in parallel, to speed up compilation.

RAG systems typically support many kinds of attributes, and providing concurrent evaluation algorithms for them is non-trivial. In particular, *circular* attributes [Far86; Jon90; MH03], i.e., attributes evaluated using fixed-point iteration, are tricky to handle. If a RAG contains circular attributes, it is not possible to use locks on individual attributes, since two threads entering the same dependency cycle leads to deadlock. Circular attributes are useful in handling many complex problems in compilers. Examples include definite assignment (a dataflow problem), and type inference.

In this paper, we present concurrent evaluation algorithms for RAGs including circular attributes. We have implemented the algorithms in the JastAdd metacompiler, supporting all the different attribute kinds in JastAdd. To avoid deadlock problems, all our algorithms are lock-free. In fact, the algorithms are also wait-free, but the current implementation is only lock-free as it uses a map implementation that is lock-free but not wait-free. For a correctly specified JastAdd project, our implementation can be used without further modification.

Our focus has been to support low latency in interactive applications, but we also report on initial speedup results using parallelization. We have validated the implementation on a full-fledged Java compiler built using JastAdd. To evaluate latency, we have run the concurrent evaluation in an interactive tool for inspecting Java programs, running the interactive thread in parallel with a long-running task computing compile-time errors.

Our contributions are:

- Sound and wait-free concurrent algorithms for Circular RAGs, in Sections 4 and 5.
- Correctness proofs for the attribute evalution algorithms.
- Generalization of Circular RAGs to allow combinations of circular and non-circular attributes on the same cycle (Section 6).
- Validation of the algorithms in an interactive tool for exploring properties of Java programs (Section 7).
- Empirical evaluation of latency of interactive tasks, comparing our concurrent implementation to a sequential implementation (Section 7).

Soundness and wait-freedom proofs for the algorithms are available in a separate technical report. Section 2 briefly introduces Circular RAGs. Section 8 discusses related work, and Section 9 concludes the paper and outlines future work.

2 Circular Reference Attribute Grammars

In a RAG [Hed00], an abstract grammar is viewed as a set of node classes representing the nonterminals of the grammar. Attributes are specified for the nodes by attribution rules. An Abstract Syntax Tree (AST) defined by the grammar will have attribute instances attached to its nodes. We refer to attribute instances as simply attributes, unless otherwise noted.

An attribute is defined by a *semantic function* of an AST node. For example, an attribute x with semantic function f can be written as $x = f(n)$, where n is an AST node. The attribute x belongs to either n or one of its children. If x is an attribute of n, we say that x is *synthesized*, and if it is an attribute of one of n's children, we say that x is *inherited*.[1]

Unlike the original definition of attribute grammars by Knuth [Knu68b], RAGs allow attributes to be references to nodes in the AST. This means that a semantic function can access remote attributes and nodes via reference attributes in the node it is computed on.

The typical way to evaluate a Knuth AG is to statically analyze attribute dependencies, and use a static schedule to evaluate all attributes in dependency order, for example using Ordered AGs [Kas80]. For RAGs, this does not work, since attribute dependencies are not statically known due to the use of reference attributes. Instead, RAGs use recursive dynamic attribute evaluation.

Extensions to RAGs supported in the JastAdd system include circular attributes, higher-order attributes, collection attributes and rewrites. Circular attributes may depend upon themselves, and are evaluated using a fixed-point iteration algorithm [Far86]. For RAGs, the fixed-point algorithm is recursive [MH03]. Higher-order attributes are attributes whose value is an AST subtree [VSK89a]. In RAGs, it is important that they create a fresh subtree on each computation. However, only one result reference should become visible to the rest of the program. Collection attributes allow compound values to be defined by a combination of contributions in an AST [Boy05; MEH07]. Rewrites allow AST nodes to be conditionally rewritten, and have been shown to be equivalent to circular higher-order attributes [SH15].

Attributes in a RAG can be memoized to make subsequent accesses fast [Jou84; Hed00]. For memoization to work, the semantic function must be an observationally pure function, i.e., without visible side-effects. Because JastAdd attributes are declared with regular Java code, it is up to the JastAdd user to write only attributes that follow well-formedness conditions like having a pure semantic function. For concurrent evaluation, we will in particular need the following well-formedness conditions:

[1] It can be noted that the attribute grammar concept of *inherited* is independent of the object-oriented concept with the same name.

WF1: Pure semantic functions Semantic functions must be *observationally pure* [Nau05], meaning that they always compute the same value, do not modify the AST, or rely on any external mutable data.

WF2: Terminating semantic functions Each semantic function terminates, given that access to other attributes terminates.

WF3: Circular attributes are computable To guarantee a computable least fixed point, we require the semantic function of circular attributes to be monotonic and yield values in a lattice of finite height. This is the condition used by Jones [Jon90].

3 Correctness

There are two important correctness conditions that are required for a concurrent circular RAG evaluator: soundness and lock-freedom. The algorithms we present will have to be sound, meaning that they compute the correct attribute value, and lock-free so that they do not cause deadlocks when used in circular attribute evaluation. To prove lock-freedom we will instead prove the stronger progress guarantee of wait-freedom. To prove wait-freedom we use the fact that an algorithm is wait-free if it terminates in a finite number of steps [Her88].

Soundness for higher-order attributes works a little differently. A higher-order attribute creates a new AST node object each time it is computed, but only one such result must be attached to the AST and become visible to the rest of the program. A higher-order attribute thus requires memoization to be sound.

4 Non-Circular Attribute Implementation

An attribute evaluator consists of a mechanism for computing the attribute value, and optional memoization of the attribute value. A generic attribute evaluator is shown in Algorithm 4. The EVAL procedure takes as parameter an attribute instance to be evaluated. Attribute computation and memoization have been abstracted out of EVAL as procedures with the following purpose:

COMPUTE Compute the value of an attribute.

MEMOIZED Test if an attribute has been memoized.

STORE Memoize a value for an attribute.

LOAD Retrieve a previously memoized value of an attribute.

The EVAL procedure can be trivially translated to Java as a method of an AST node class that the attribute it evaluates was declared on [Hed00]. By proving that the called procedures fulfill certain requirements we can show that the resulting EVAL implementation is sound and wait-free.

The memoization procedures used in Algorithm 4 must fulfill the following soundness requirement for EVAL to be sound for concurrent evaluation:

Algorithm 4 Template attribute evaluation algorithm for memoized non-circular attributes.

```
procedure EVAL(x)                                  ▷ Evaluate attribute x.
    if MEMOIZED(x) then                            ▷ Test if already memoized.
        return LOAD(x)                             ▷ Return memoized value.
    else
        u ← COMPUTE(x)                             ▷ Compute attribute value.
        return STORE(x, u)                         ▷ Memoize and return result.
    end if
end procedure
```

Memoization Requirement In one thread, if MEMOIZED(x) returns TRUE before LOAD(x), then LOAD(x) returns a value v stored by some execution, in any thread, of STORE(x, v).

Store Requirement Executing STORE(x, _) returns some value v stored by some execution, in any thread, of STORE(x, v).

The following theorem states that if these requirements hold, EVAL will compute the right value for any attribute.

Theorem 1 (Eval Sound)**.** *If MEMOIZED, STORE and LOAD fulfill the Memoization Requirement and Store Requirement, then, for some attribute x, EVAL(x) (Algorithm 4) computes the value of x.*

Proof. Consider a thread that executes EVAL(x). The **if**-statement in EVAL has two branches:

- If MEMOIZED(x) returned TRUE, then by the Memoization Requirement the returned value, v, was stored by some call STORE(x, v). Because all calls to STORE(x, _) store a computed value of x, and because x is well-formed (WF1), the returned value is the value of x.
- Otherwise, MEMOIZED(x) returned FALSE. The returned value is the result of STORE(x, u). According to the Store Requirement, the result v was stored by some call STORE(x, v). Because all calls to STORE(x, _) store a computed value of x, the returned value is the value of x.

□

For a non-higher-order attribute the semantic function always computes an identical value. However, for a higher-order attribute this is not the case, as the attribute returns a freshly created AST node object. It is important that only one result of a higher-order attribute becomes visible to the rest of the program. For higher-order attributes we add the following soundness requirement:

Higher-Order Memoization Requirement For a higher-order attribute instance x, each call to STORE(x, _) returns a single entity object. If MEMOIZED(x) returns TRUE, LOAD(x) returns the same entity object as STORE(x, _).

The last requirement for EVAL to be correct is that it is wait-free. We will require that COMPUTE, MEMOIZED, STORE, and VALUE are wait-free, to ensure that EVAL is wait-free (when evaluating non-circular attributes).

Theorem 2 (Eval Wait-Free)**.** *Consider a non-circular attribute instance x. If COMPUTE, MEMOIZED, STORE and LOAD are wait-free, then EVAL(x) (Algorithm 4) is wait-free.*

Proof. EVAL itself uses no iteration and no self-recursion (because x is not circular). Thus, EVAL is wait-free because all called procedures are wait-free by assumption. □

In the following sections we discuss implementations of COMPUTE, MEMOIZED, STORE, and VALUE. The procedures are implemented by Java methods and fields declared on the AST node class that the corresponding attribute is declared on. The attribute instance is not explicitly passed to the methods, as it is not a concrete Java object. Instead, the implicit `this` parameter of the Java methods separates the evaluation of different attribute instances.

4.1 Synthesized and Inherited Attributes

For synthesized attributes, the COMPUTE procedure is a direct translation of the semantic function into an executable form, where other attribute uses are replaced by calls to EVAL. Because JastAdd attributes are specified with Java code, the translation of the COMPUTE procedure is just a Java method containing the code of the semantic function. Because well-formed attributes terminate (well-formedness condition WF2), the resulting COMPUTE procedure is wait-free.

For inherited attributes, the COMPUTE procedure that computes an inherited attribute on a node n must find the semantic function for the inherited attribute. This is done by accessing the parent of n, and determining which semantic function should be computed for the child node n. Locating the semantic function is thread-safe because the AST is immutable after construction. Locating the semantic function is not recursive or iterative, so it is wait-free. Computing the semantic function is wait-free for the same reason as for synthesized attributes.

Concurrent memoization for synthesized and inherited attributes can be implemented using a simple cache field and volatile flag in Java, as shown in Listing 1. Well-formed synthesized and inherited attributes always compute the same value, so concurrent calls to `store()` are safe because they store the same value.

Theorem 3. *The methods in Listing 1 fulfill the Memoization Requirement.*

Proof. The Memoization Requirement entails that if `memoized()` returns TRUE before `load()`, then `load()` returns a value `v` stored by some execution of `store(v)`.

The `cached` flag starts out as FALSE. Hence, `memoized()` returns TRUE only after some write has set `cached` to TRUE. The `cached` flag is declared as `volatile`. According to the Java Memory Model, a previous write to `value` must therefore be visible to the thread that observed `cached` having value TRUE, so a following call to `load()` will return this value or some other value of x stored by `store(_)`. □

Listing 1: Memoization in Java for simple attributes.

```
T value;
volatile boolean cached = false;

boolean memoized() {
  return cached;
}

T store(T v) {
  value = v;
  cached = true;
  return v;
}

T load() {
  return value;
}
```

Theorem 4. *The methods in Listing 1 fulfill the Store Requirement.*

Proof. The Store Requirement entails that `store(u)` returns a value `v` stored by some execution of `store(v)`. The requirement is fulfilled because `store(u)` always returns `u`. □

Theorem 5. *The methods in Listing 1 are wait-free.*

Proof. All of the operations used are wait-free according to the Java specification. There exists no iteration or recursion, hence the methods are wait-free. □

4.2 Parameterized Attributes

Synthesized and inherited attributes can be optionally parameterized. In either case, the only difference in the implementation is that additional parameters are passed through the Eval procedure to the Compute procedure. Adding parameters does not affect the wait-freedom of Compute, and parameterized Compute is wait-free for the same reason as synthesized Compute.

Parameterized attributes are memoized by mapping parameter values to result values. Our implementation of concurrent memoization for parameterized attributes relies on a thread-safe map to memoize attribute results. For unary attributes, the single parameter value is used as map key, and for 2+ arity attributes a list of the parameter values is used as map key.

For the memoization procedures to be wait-free, we need a wait-free map, for example a Java version of the wait-free hash map by Lange et al. [LWZ16]. In our implementation, seen in Listing 2, we use the `ConcurrentMap` interface from Java's standard library, and a helper method to create an object implementing the interface.

Listing 2: Memoization in Java for parameterized attributes.

```
ConcurrentMap map = buildMap();

boolean memoized(Object params) {
  return map.containsKey(params);
}

T store(Object params, T v) {
  map.putIfAbsent(params, v);
  return map.get(params);
}

T load(Object params) {
  return (T) map.get(params);
}
```

Listing 3: Memoization in Java for non-parameterized higher-order attributes.

```
Ref value = new AtomicRef(nil);

boolean memoized() {
  return value.get() != nil;
}

T store(T v) {
  value.compareAndSet(nil, v);
  return value.get();
}

T load() {
  return (T) value.get();
}
```

The memoization map can be initialized either when the AST is created, or on first attribute access. The `ConcurrentMap` interface does not require implementations to be wait-free, however a wait-free map could straightforwardly implement the interface. We chose to use the Java standard library's `ConcurrentHashMap`, which is only lock-free. This makes our full implementation of Concurrent RAGs in JastAdd lock-free, instead of wait-free.

The `ConcurrentMap` interface extends Java's standard `Map` interface, with an additional `putIfAbsent` method to atomically insert a key-value pair in the map if the key was not already associated with a value. We assume that all `ConcurrentMap` methods are linearizable.

For parameterized attributes, the memoization implementation in Listing 2 is used.

Theorem 6. *The methods in Listing 2 fulfill the Memoization Requirement.*

Proof. The Memoization Requirement entails that if `memoized(p)` returns TRUE before `load(p)`, then `load(p)` returns a value `v` stored by some execution of `store(p,v)`.

The `map` is initially empty, with no key associated to a value. A call to `map.containsKey(p)` then only returns TRUE if some call to `putIfAbsent(p,_)` inserted a value for the given key previously. Keys are never disassociated in the map, so `map.get(p)` is guaranteed to return an inserted value `v`, which was inserted by `store(p,v)`. □

Theorem 7. *The methods in Listing 2 fulfill the Store Requirement.*

Proof. The Store Requirement entails that `store(p,u)` returns a value `v` stored by some execution of `store(p,v)`.

Keys are never disassociated in the map, and because `store(p,_)` either inserts a value for the key `p`, or does not because the key was already associated, and because

the call to `map.get(p)` occurs after that in program order, `map.get(p)` returns an inserted value `v`, which was inserted by `store(p,v)`. □

Theorem 8. *The methods in Listing 2 are wait-free if the* `ConcurrentMap` *implementation is wait-free.*

Proof. There exists no iteration or recursion, so if the methods implemented by the `ConcurrentMap` object are wait-free (`containsKey`, `putIfAbsent`, and `get`), then the methods in Listing 2 are wait-free. □

4.3 Higher-Order Attributes

A higher-order attribute can be either synthesized or inherited, and optionally parameterized. In either case, the only difference in computing the attribute is that, at the end of the COMPUTE procedure, the result value is attached to the parent AST node by setting the parent reference of the result node. Setting the parent pointer at the end of COMPUTE does not affect the wait-freedom of COMPUTE.

For non-parameterized higher-order attributes we can not reuse the synthesized attribute memoization implemented in Listing 1 because it is possible that two threads race to write a value with `store()`, making it possible for two separate entity objects to be shared with the rest of the program. This is a consensus problem: concurrent threads calling `store()` must agree on a single value. A standard solution for n-thread consensus is to use *Compare-And-Set* (CAS) [Her06]. CAS is a wait-free, linearizable, operation that atomically tests the value of a variable and updates it to a new value if it had an expected value. Java provides library classes implementing CAS, for example `AtomicReference` which provides CAS via the `compareAndSet()` method - the first argument is the expected value, and the second is the new value.

We use `AtomicReference` to implement memoization for higher-order attributes, as shown in Listing 3. We use `nil` to represent an illegal attribute value (not equal to `null`). This value is used to indicate that the attribute has not yet been computed and thus replaces the `cached` flag from the synthesized memoization algorithm. The expression `value.get()` reads the value of the atomic variable.

For higher-order attributes, Algorithm 4 is used under the Higher-Order Memoization Requirement.

Theorem 9. *Consider a higher-order attribute instance x. If* MEMOIZED, STORE *and* LOAD *fulfill the Higher-Order Memoization Requirement, then* EVAL(x) *(Algorithm 4) returns a single entity object.*

Proof. Consider a thread that executes EVAL(x). The **if**-statement in EVAL has two branches that return either the result of STORE$(x, _)$ or LOAD(x). The result of LOAD(x) is returned if MEMOIZED(x) returned TRUE, so according to the Higher-Order Memoization Requirement, LOAD returns a single entity object. Additionally, the Higher-Order Memoization Requirement specifies that LOAD(x) returns the same entity as STORE$(x, _)$, so only one entity object can be returned from EVAL(x). □

For higher-order attribute memoization, we must show the that the implementation fulfills the Higher-Order Memoization Requirement. For non-parameterized higher-order attributes, the implementation in Listing 3 is used.

Theorem 10. *The methods in Listing 3 fulfill the Higher-Order Memoization Requirement.*

Proof. The `value` field is only updated by the CAS in `store()`, with the expected value NIL. Only one CAS is able to succeed, because the attribute value is never equal to NIL. Because `store()` returns the single successful CAS value, it always returns the same value for a single attribute instance.

Note that `memoized()` returns TRUE only if a previous CAS has succeeded, and then `load()` must return the stored value of the single successful CAS. □

Theorem 11. *The methods in Listing 3 are wait-free.*

Proof. All methods of `AtomicReference` are wait-free. No other method calls are used, and no iteration or recursion is used, so the methods in Listing 3 are wait-free. □

For parameterized higher-order attributes, the parameterized memoization implementation in Listing 2 is used. The following theorem states that it fulfills the Higher-Order Memoization Requirement.

Theorem 12. *The methods in Listing 2 fulfill the Higher-Order Memoization Requirement.*

Proof. The associated value for some key `p` is only updated by `putIfAbsent(p,_)` in `store(p,_)`. Because keys are never disassociated, only one `putIfAbsent(p,_)` is able to succeed. Because `store(p,_)` returns the single successful `putIfAbsent(p,_)` value, it always returns the same value, or entity object, for a single attribute instance.

Note that `memoized(p)` returns TRUE only if a previous `putIfAbsent(p,v)` call has succeeded, and then `load(p)` must return the value `v` stored by the single successful `putIfAbsent(p,v)` call. □

4.4 Collection Attributes

Collection attributes collect values from nodes in a subtree of the AST, where Each node can be marked as a contributor to the collection with a semantic function to compute the contributed value. Each semantic function for a node may also be given a boolean expression to restrict it to contribute its value to the collection only if some condition holds.

Collection attribute computation is divided into two phases [MEH07]:

Survey phase A subtree of the AST is traversed, starting from some predetermined *collection root*. All nodes that are potential contributors to the collection attribute are added to a worklist for the next phase.

Collection phase For each node in the worklist from the previous phase, the contribution condition is checked to determine if the node actually should contribute a value. If the node is contributing to the collection then its semantic function is computed and its value is added to the result collection.

A simple method of computing collection attributes is to perform a depth-first traversal for the survey phase, and then use a loop to iterate over the resulting list of contributors in the collection phase.

Collection attributes are only computed using the base AST, excluding higher-order attributes. Computing a non-parallelized collection attribute is wait-free because each contribution is computed by a semantic function that must terminate in a finite number of steps, and there are a finite number of contributions because the base AST has a bounded height and each AST node object has a finite number of children.

A simple non-parallelized collection attribute evaluator is safe for concurrent evaluation if it does not memoize its result. If memoization is needed the same memoization scheme used for concurrent synthesized attributes is sound for collection attributes.

4.5 Rewrites

JastAdd provides automatic AST rewriting controlled by attributes. This is a powerful tool for transforming the AST, but it is problematic because the AST should not be modified after construction. Söderberg and Hedin [SH15] show how rewrites can be defined in terms of circular higher-order attributes, and this avoids any modification to the base AST. As long as the base AST is immutable, and the rewrite is implemented using attributes that are concurrent, the resulting rewrite is safe for concurrent evaluation.

5 Circular Attribute Implementation

In fixed-point evaluation, a function f can be computed by repeated application, starting with some bottom value $\bot$. The fixed-point is reached when the result x satisfies $x = f(x)$. There may be several values that satisfy this equation. However, if we start from the bottom value we will always reach the same, least, fixed point.

A circular attribute can be seen as a fixed-point function f. However, there may be multiple mutually dependent attributes. Therefore, f does not necessarily correspond to a single semantic function, rather it represents multiple simultaneously applied semantic functions. Furthermore, it is possible to apply the individual semantic functions one at a time, in any order, and reach the same simultaneous least fixed-point. This is true because each attribute takes values from a lattice, so a combination of attribute approximations, for example a vector of approximations, is also a value in a lattice. Since each semantic function is monotonic, according to well-formedness condition WF3 in Section 2, updating one approximation is a monotonic operation on the combined approximation vector.

We will now illustrate how a circular attribute can be evaluated in practice. Let x be some circular attribute (instance), with $D(x)$ being the set of attribute (instances) that x transitively depends on. For now, we assume that all attributes in $D(x)$ are circular and mutually transitively dependent. We discuss how to loosen these requirements later, in Section 6.

Let S be a vector of attribute approximations for the attributes $D(x)$. The S vector forms the state of a fixed-point computation of the attributes $D(x)$. A successor state S' is found by updating one approximation $S'_y = f_y(S)$ where y is an attribute in $D(x)$. If the new approximation of y is not equal to the previous approximation, i.e., $S'_y \neq S_y$, then since f_y is monotonic, S' is greater than S.

Consider a starting state $S_\perp$, where each approximation is equal to the bottom value of the corresponding attribute. By repeatedly updating approximations of attributes in $D(x)$ as above, in any order, starting in state $S_\perp$, the approximations will eventually reach a simultaneous fixed point in which all approximations are equal to the fixed-point value of the corresponding attribute.

A state S^{fp} is a simultaneous fixed point of the attributes in $D(x)$ if, for all $y \in D(x)$, $S^{fp}_y = f_y(S^{fp})$.

5.1 Concurrent Circular Attribute Algorithm

Our algorithm for concurrent evaluation of circular attributes is shown in Algorithm 5. The basic structure of the algorithm is similar to the sequential algorithm of Magnusson and Hedin [MH03]. The main idea for supporting concurrency is to let each thread keep track of thread-local approximations for any ongoing fixed-point loop, and to synchronize with the global approximations only at specific points during the loop. The main CEval procedure uses three Case subroutines similarly to the formulation of the sequential algorithm of Söderberg and Hedin [SH15].

To compute a circular attribute x, our algorithm uses successive approximation of all attributes that x transitively depends on, until no approximation changes value. This works through recursive calls to the three Case procedures. Each thread starts in Case1, which starts a new fixed-point loop. During the loop, Case2 is used to update the approximation of any attribute on the dependency cycle. Whenever an attribute is recursively revisited during a particular iteration in the loop, Case3 is used to return the previous approximation.

In concurrent execution, separate threads can individually compute new approximations of attributes. The computations of each thread is shared with other threads via global approximation variables.

To illustrate, assume there are two threads T_1 and T_2 computing mutually dependent attributes x and y respectively. The control flow then looks like this:

T_1: CEval$(x) \rightarrow$ Case1$_x \rightarrow$ Case2$_x \rightarrow$ Case2$_y \rightarrow$ Case3$_x$

T_2: CEval$(y) \rightarrow$ Case1$_y \rightarrow$ Case2$_y \rightarrow$ Case2$_x \rightarrow$ Case3$_y$

5.2 The CEval Procedure

The CEval procedure takes two arguments: an attribute to be evaluated, x, and an iteration index, i. The iteration index identifies uniquely which iteration of the fixed-point loop an attribute value was computed during, and is used to determine if Case2 or Case3 should be called in a recursive computation.

CEVAL(x, i) is called with $i = 0$, when there is no ongoing fixed-point computation, and CEVAL will then return the fixed-point value of x. If $i \neq 0$, then CEVAL returns an approximation of x. Importantly, $i \neq 0$ only when CEVAL is called recursively from CASE1, i.e., during an ongoing fixed-point computation.

CEVAL(x, i) computes a new approximation of x via CASE2, by calling COMPUTE(x, i). The COMPUTE procedure is an executable translation of the semantic function of an attribute, where each access to some other attribute y is translated as a call to CEVAL(y, i). Consequently, calling COMPUTE(x, i), leads to recursive calls to CEVAL(y, i) for each attribute y that x directly depends on.

The execution of CEVAL starts by testing if x has already been memoized, in which case the memoized value is returned. Otherwise, if the global approximation was not initialized (equal to NIL), the global approximation is updated to the bottom value of x. Next, the execution continues to either CASE 1, 2, or 3:

- if $i = 0$, CASE1(x) is called to start a new fixed-point computation, otherwise,
- if a thread-local approximation of x has not been recorded during the current iteration i, CASE2(x, i) is called to compute a new approximation of x, otherwise,
- CASE3(x) is called to reuse the previous thread-local approximation of x.

5.3 Thread-Local State

Each thread stores thread-local state (TLS), that is not visible to other threads, in the *tls* field. The purpose of each member of the *tls* field is described below:

tls.change A flag indicating if, in the current thread, any local attribute approximation has changed value during the current iteration of the fixed-point loop in CASE1.

tls.value A map from attributes to local attribute approximations. Unassociated keys are mapped to the value NIL.

tls.iter A map from attributes to iteration indices (described below). Unassociated keys are mapped to an unused non-zero iteration index.

In each thread, the iterations of the fixed-point loop in CASE1 are assigned unique indices. The iteration index is used to determine if an attribute approximation has already been computed by the current thread during the current CASE1 iteration.

Each time a thread records a new approximation for an attribute x, it updates $tls.iter(x)$ to the current iteration index. Thus, $tls.iter(x)$ is equal to the current iteration index if an approximation for attribute x has already been computed during the current iteration of CASE1.

Updating the iteration index at the start of the CASE1 loop ensures that each attribute that can change approximation is computed during the CASE1 iteration. This is required so that the *tls.change* flag reflects if any local approximation could be updated to a new value.

Algorithm 5 Concurrent evaluation algorithm for circular attributes.

```
▷ Shared global value of attribute x:
gv_x : Value × Boolean ← (NIL, FALSE)

▷ Thread-local state:
tls.change : Boolean
tls.value : (Instance → Value)
tls.iter : (Instance → Integer)

procedure CEVAL(x, i)
    (value, done) ← read(gv_x)
    if done then
        return value
    else if value = NIL then
        ▷ Initialize gv_x by Compare-And-Set:
        CAS(gv_x, (NIL, FALSE), (⊥_x, FALSE))
    end if
    if i = 0 then
        return CASE1(x)
    else if tls.iter(x) ≠ i then
        return CASE2(x, i)
    else
        return CASE3()
    end if
end procedure

▷ Get current thread-local approximation of x.
procedure CASE3(X)
    return tls.value(x)
end procedure
```

```
▷ Run a fixed-point computation of attribute x.
procedure CASE1(x)
    repeat
        i ← uniqueId()
        tls.change ← FALSE
        CASE2(x, i)
    until ¬tls.change
    ▷ Memoize x by marking gv_x as done:
    (result, _) ← read(gv_x)
    CAS(gv_x, (result, FALSE), (result, TRUE))
    return result
end procedure

▷ Compute a new approximation of attribute x.
procedure CASE2(x, i)
    (prev, _) ← read(gv_x)
    last ← tls.value(x)
    if prev ≠ last then tls.change ← TRUE end if
    tls.value(x) ← prev
    tls.iter(x) ← i
    next ← COMPUTE(x, i)
    if prev ≠ next then
        tls.change ← TRUE
        CAS(gv_x, (prev, FALSE), (next, FALSE))
        tls.value(x) ← next
        tls.iter(x) ← i
    end if
    return next
end procedure
```

Note also that the iteration index is always updated to a unique value, to ensure that iteration indices are unique across all CASE1 loop invocations, not only across iterations of a single CASE1 invocation.

5.4 Shared State

All threads share a global approximation for each attribute x, stored in the atomic variable gv_x. The atomic variable is updated using Compare-And-Set (*CAS*) and read using $read(gv_x)$. The *CAS* and read operations are wait-free and atomic.

The value of gv_x is a tuple of an attribute value, and a *done* flag indicating if the value is the fixed-point result for the attribute, i.e. if the attribute is memoized. If the flag is FALSE, the value is either uninitialized (NIL), or an approximation of the attribute x.

The notation used for updating gv_x is $CAS(gv_x, p, n)$, where p is the expected previous value and n is the value to update to. When *CAS* is linearized, if the value of gv_x is indeed p then it is atomically updated to n. In Java, the gv_x field can be implemented by `AtomicReference` as in the higher-order attribute memoization from Section 4.3.

Each thread evaluating a circular attribute x can detect if the global approximation, gv_x, has changed since the thread last read it by comparing to its own thread-local approximation of x.

Case1

In CASE1, a new fixed-point computation for an attribute x is started. The computation is performed by a loop, and an iteration index i is used to identify each iteration of the loop.

Each iteration of the loop starts by updating i to a new unique, non-zero value, and clearing the *tls.change* flag. Next, CASE2(x, i) is called to compute a new approximation of x. The loop is exited at the end of an iteration i if *tls.change* remains unset. The *tls.change* flag remains unset only if, during an iteration of the CASE1 loop, no attribute approximation was updated to a new value via CASE2(x, i).

After the loop is exited, the stored global value of x is equal to the fixed point value of x, so the current thread attempts to memoize the attribute by updating the *done* flag in the global value of x.

Case2

When CASE2(x, i) is called, it computes a new approximation of the attribute x, during an ongoing iteration i of the fixed-point loop in CASE1.

First, the shared global value of x is read and compared to this thread's local approximation. If they are not equal, we have two cases:

- this thread had no local approximation stored for x, or,
- another thread has updated the global approximation to a new value.

In either case, the *tls.change* flag is set to indicate that there was an approximation update during the current iteration i.

Before computing a new approximation, the current thread's local approximation of x is updated to the current global value, and the iteration index for the local approximation is set to i. This causes recursive CEVAL(x, i) calls in the current thread to enter CASE3 rather than CASE2, thereby avoiding unbounded recursion.

A new local approximation of x is computed by COMPUTE(x). If the new value is different from the previous global approximation then the *tls.change* flag is set. Next, the global approximation is updated using compare-and-set (*CAS*).

The value returned by CASE2 is equal to the current thread's local approximation for x. This is not ensured by the dataflow in CASE2 alone, but it is a consequence of the fact that COMPUTE(x, i) does not update the current thread's local approximation for x, as recursive calls to CEVAL(x, i) enter CASE3.

Case3

CASE3 returns the previous local approximation computed by the current thread for the attribute x. When called after CASE2(x, i) during the same iteration i, CASE3(x) returns the same value as CASE2(x, i) did.

5.5 Correctness

We will here show informal outlines for proofs of soundness and wait-freedom of CEVAL. The full proofs are in Appendix A.

Soundness CEVAL is sound if, for a well-formed circular attribute x, CEVAL$(x, 0)$ computes the fixed-point value of x. Well-formedness is defined in Section 2. Specifically, the semantic function of x must be monotonic (WF3).

Proof sketch. Consider a single-threaded execution of CEVAL. It will always enter CASE1 initially, then perform iterations until the *tls.change* flag remains unset. For this to work, CASE2 should be called for each attribute that x depends on, that can change value, in each iteration of CASE1. It can be shown that in each iteration of CASE1, either all attributes that x transitively depends on have reached their fixed-point value, or CASE2 is executed for all attributes that x transitively depends on.

It is important that a single thread only advances the global state of an attribute to a monotonically increasing value. This is both ensured by the well-formedness of the attributes being evaluated, since their semantic functions are monotonic, and in concurrent execution the fact that the global approximation is read before computing a new approximation, and used as the expected value before updating to a new approximation.

Wait-freedom It is mostly straight-forward to prove that CEVAL is wait-free. The tricky part is to show that the loop in CASE1 performs a finite number of iterations, and that each one performs a finite amount of work. This relies on the fact that the attributes are well-formed and thus have terminating semantic functions (WF2), and a finite greatest possible value (WF3) which is eventually reached by successive approximation in CASE1.

5.6 Parameterized Circular Attributes

Like most other types of attributes, circular attributes can be parameterized. Algorithm 5 can be used to evaluate parameterized circular attributes, with a few modifications. A new parameter p is added to the CEVAL procedure. The parameter p is a tuple of the attribute parameter values, and it is is passed to CASE1, CASE2, CASE3, and COMPUTE.

The global value of a non-parameterized circular attribute is stored in an atomic variable with a CAS operation and atomic read. To store global values for a parameterized circular attribute we instead use a concurrent map object that maps attributes to atomic variables. The global value map is indexed by p, i.e. $gv_x(p)$ gives the atomic variable for the global value of x with parameters p.

A parameterized circular attribute x is initialized by using `putIfAbsent()` to insert a new atomic variable containing the bottom value of x in the global value map. The other uses of gv_x from the non-parameterized algorithm are replaced by map lookups $gv_x(p)$. Because we use `putIfAbsent()`, we ensure that an attribute is only initialized once. The rest of the uses of gv_x will all act as before just on different atomic variables for different parameter combinations.

The local approximation map and iteration index map, *tls.value* and *tls.iter*, need to be indexed by both attribute and parameter values. We implement this by using tuple objects containing the attribute and parameter value tuple as map key.

6 Mixed Circular Evaluation

We have used some simplifying assumptions about the structure of circular attributes. In this section we review these assumptions, and we show why some of them are not necessary for correctness, and how others can be relaxed by simple additions to our algorithms.

By relaxing assumption 1 below, we allow more general combinations of circular and non-circular attributes than were previously allowed in Circular RAGs according to Magnusson and Hedin [MH03]. This loosened requirement is useful in practice since it is common that attributes are on a cycle only for a small fraction of typical ASTs. Requiring these attributes to be declared as circular would start expensive fixed-point computations also for those ASTs where there is actually no cycle.

To concisely discuss these assumptions we will first need two auxiliary definitions:

Circularly evaluated attribute An attribute instance x is *circularly evaluated* if it has a bottom value, and CEVAL(x, i) is used to compute its value.

Effectively circular attribute An attribute instance is *effectively circular* if it depends transitively on itself. Otherwise it is said to be *effectively non-circular*.

An attribute declaration can have both effectively circular and effectively non-circular instances.[2]

The assumptions we have used so far, are:

[2]Magnusson and Hedin [MH03] refer to circularly evaluated attributes as *potentially circular* and to effectively circular attributes as *actually circular*.

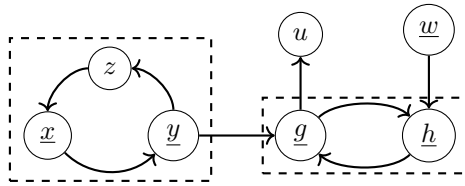


Figure 1: An attribute dependency graph. Each circle is an attribute instance. Attributes with underlined names are circularly evaluated (with bottom value). Attributes inside the dashed rectangles are strongly connected and effectively circular. The attributes u and $\underline{w}$ are effectively non-circular.

1. a circularly evaluated attribute instance depends only on circularly evaluated attribute instances,
2. all circularly evaluated attribute instances are effectively circular,
3. all effectively circular attribute instances are circularly evaluated,
4. if an effectively circular attribute instance x transitively depends on an attribute instance y, then y transitively depends on x.

Assumption (1) can be relaxed. Figure 1 shows two attribute instances breaking this assumption: z and u both have dependents that are circularly evaluated, namely $\underline{g}$ and $\underline{y}$. Evaluating u with Algorithm 4 works as it should, because u is not effectively circular and always computes the same value. For z, however, Algorithm 4 does not work correctly, because z can compute different values based on an approximation of $\underline{x}$. Algorithm 1 memoizes attributes on the first computation, but during a fixed-point computation, circularly evaluated attributes return approximations, which are not safe to memoize. An attribute depending (transitively) on a circularly evaluated attribute should thus not memoize its result during a fixed-point computation.

Assumption (1) can be relaxed by using Algorithm 6 for non-circular attributes, replacing Algorithm 4. Algorithm 6 works by making the STORE call conditional, so that EVAL only memoizes an attribute if a fixed-point computation is not currently ongoing. We need to add a new field to the thread-local state: *tls.i*, to track the CASE1 iteration index. At the start of each iteration of the CASE1 loop, *tls.i* is updated to the current iteration index, and at the end of CASE1, *tls.i* is set to 0. A memoized attribute must check that $tls.i = 0$ before memoizing a result. The updated algorithm works even for higher-order attributes, because the result node is not memoized by any other attribute that depends on the higher-order attribute before the circular evaluation has reached the fixed point, thereby different AST nodes do not become visible to the rest of the program.

This updated algorithm is the one that we implemented in JastAdd.

Assumption (2) is not necessary for correctness. In Figure 1, $\underline{w}$ is circularly evaluated, but it is not effectively circular. In general, if some attribute a is not effectively circular, but circularly evaluated with CEVAL$(a, 0)$, then there are two

Algorithm 6 Evaluation algorithm for memoized non-circular attributes with circular dependees.

```
procedure EVAL(x)
    if MEMOIZED(x) then
        return LOAD(x)
    else
        u ← COMPUTE(x)
        ▷ Test if called in circular evaluation.
        if tls.i ≠ 0 then
            ▷ In circular evaluation: not safe to memoize.
            return u
        else
            ▷ Memoize the computed value as usual.
            return STORE(x, u)
        end if
    end if
end procedure
```

cases: a transitively depends on some circularly evaluated attribute instance, or it does not.

If a depends on some circularly evaluated attribute instance, then as long as that attribute changes approximation, the CASE1 loop for a will not terminate, thereby the circular attribute reaches its fixed-point.

If a does not depend on a circular attribute, then in the first CASE1 iteration, there is no local approximation of a so the change flag is set in CASE2. Since a is not circular it does not compute a new approximation in the second CASE1 iteration, so the fixed-point loop completes after the second iteration, and the value of a is memoized.

Assumption (3) is not necessary for correctness of Algorithm 5. It is sufficient that at least one distinguished attribute in each dependency cycle is circularly evaluated. The bottom values of other attributes are then computed in terms of the bottom values of the distinguished circular attributes.

Assumption (4) is not necessary for correctness. It implies that the dependency graph of each circular attribute is strongly connected. If it is not strongly connected, our algorithm works without modification. However, the algorithm could potentially be modified to improve performance by separately evaluating the connected components in topological order and memoizing each component separately, similar to the method used by Magnusson and Hedin [MH03]. Future work could investigate extending the concurrent circular evaluation algorithm to improve evaluation performance for separate component evaluation.

7 Empirical Evaluation

The research questions we want to answer in the evaluation are:

RQ1 Does our implementation of the concurrent algorithms work on existing well-formed JastAdd projects?

RQ2 Can the implementation be used for interactive tools with both interactive and long-running tasks?

RQ3 Does our concurrent implementation give sufficiently low latency for interactive tasks?

Section 7.1 addresses the applicability of the approach (RQ1 and RQ2). Latency (RQ3) is addressed in section 7.2. Threats to validity are discussed in section 7.3.

7.1 Concurrent ExtendJ and Interactive Applications

We applied our concurrent implementation on ExtendJ, a full-featured Java compiler [EH07b]. The ExtendJ specification is complex, with 3473 attributes, and it uses all attribute kinds discussed in this paper, including attribute-controlled rewrites, using the circular higher-order attribute mapping supported by JastAdd [SH15].

Initially, running ExtendJ concurrently did not work because its specification was not completely well-formed, with some semantic functions being non-pure (WF1). Most of these problems happened to be masked in sequential evaluation, but resulted in errors when running concurrently. In one case there was also an error when running sequentially caused by purity issues.

Substantial work was required to find and fix attribute purity problems, but the result benefits the sequential compiler by removing cases where it could compute incorrect results when attributes were evaluated in a certain order.

After fixing the identified well-formedness problems, we successfully ran both the sequential and the concurrent implementations on all regression tests for ExtendJ using the same JastAdd specification. Based on this, we can answer RQ1 affirmatively.

To address RQ2 we implemented an extension of an interactive AST debugging tool named DrAST [LTH16]. DrAST has a Graphical User Interface in which the user can explore a JastAdd AST for a program and interactively inspect/compute attribute values of nodes in the AST. In our extension to DrAST, we integrated ExtendJ and added a few features. We added a source editor for the program, and changes to the program are reflected in the AST view. A screenshot of our version of the tool is showin in Figure 3 in Appendix B We also added a computation of ExtendJ's `problems()` attribute containing compile-time error and warning messages so that these messages are displayed by DrAST. The user can interactively inspect/compute attribute values while the long-running `problems()` attribute is computed. Any interactive tasks are run concurrently with error-checking tasks using our concurrent attribute evaluator. The tool thus works similarly to a typical Integrated Development Environment, and we can thereby answer RQ2 affirmatively.

7.2 Latency and Performance Evaluation

The independent variable in studying latency is the attribute evaluator implementation. We measure two different attribute evaluators: the sequential implementation

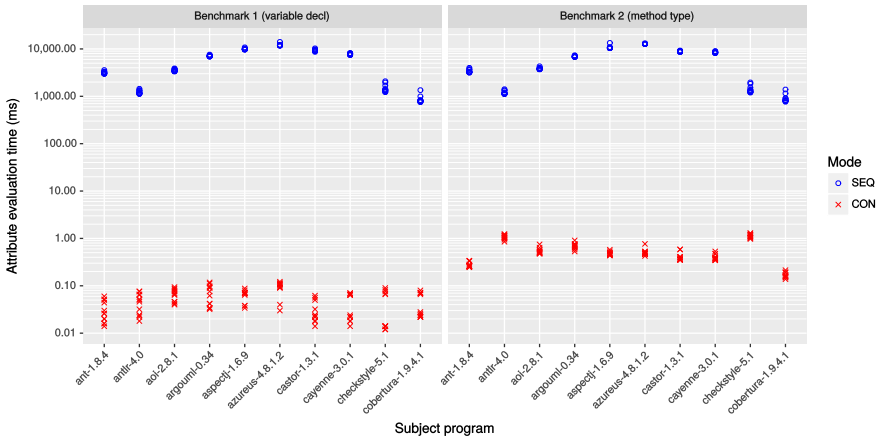


Figure 2: Latency results from Benchmark 1 and 2. Red ×:s show the latency for interactive tasks when using the concurrent implementation. Each × shows the average time for computing a variable declaration (left) or a method type (right) attribute, when running concurrently with the long-running `problems` attribute. Blue ○:s show the time it took to complete the long-running task of computing the `problems` attribute when using the sequential algorithms with locking. This is the minimum latency for interactive tasks that would occur when the interactive task is started right after starting the long-running task.

from JastAdd, and our concurrent implementation presented in this paper (Algorithms 5 and 6). Attribute evaluation time is the measured dependent variable. Confounding variables are the compiler (ExtendJ) on which we measure, and the attributes measured.

Setup

We designed benchmarks to measure attribute evaluation latency and overall overhead and speedup of concurrent attribute evaluation. For evaluation latency we measure two relatively short-running attributes that are evaluated concurrently with a long-running attribute. For overhead and speedup we measure the evaluation time of the long-running attribute when evaluated sequentially and in parallel.

We use four benchmark configurations, as shown in Table 1. Each benchmark runs some combination of three tasks executed in separate concurrent threads. The three tasks are listed below.

Task P Evaluates the long-running attribute `problems()` on all `CompilationUnit` nodes.

Task VD Evaluates the short-running attribute `decl()` (variable declaration) on 500 stochastically selected `VarAccess` nodes.

Benchmark	Thread			
	1	2	3	4
1	Task P	Task VD	–	–
2	Task P	Task MT	–	–
3	Task P	–	–	–
4	Task P	Task P	Task P	Task P

Table 1: Benchmark configurations: each benchmark runs up to four threads, with each thread running one of three tasks (P, VD, or MT). The table shows the thread-task mapping for each benchmark.

Task MT Evaluates the short-running attribute `type()` (method type) on 500 stochastically selected `MethodDecl` nodes from classes and interfaces.

The first two benchmarks are used to measure attribute evaluation latency in an interactive setting. They run many short-running attributes in concurrently with a long-running attribute, in two separate threads. Benchmark 3 is used to measure sequential performance by running a long-running attribute in a single thread. Benchmark 4 is used to measure parallelization performance by running long-running attributes in parallel in four threads.

All benchmarks are run both with the sequential and concurrent implementation. In the concurrent mode, task threads are allowed to evaluate attributes concurrently, but in the sequential mode, we use a lock to ensure that only one thread at a time is evaluating any attribute.

Each benchmark configuration is executed 15 times in a single Java process. The results of the first three iterations are discarded to reduce the impact of warm-up effects in the Java environment.

Before Benchmark 1 and 2 are executed we first search the AST of the subject program to find all `VarAccess` or `MethodAccess` nodes, then the list of nodes is shuffled and the first 500 nodes are used in the benchmark.

Subject programs we used for the benchmarks are taken from the Qualitas Corpus, Version 20130901 [Tem+10]. We measured the first 10, in alphabetical order, of the subject programs in the Qualitas Corpus that were written for Java 5 or higher.

The benchmark suite was run on an Intel Core i7-3820 CPU at 3.60GHz, running 64-bit Linux Mint, with Java version 1.8.0_112 (Oracle JDK). A relatively large Java heap size of 32Gb was used, more than 10× the minimum requirement to compile each subject program in sequential mode, in order to limit runtime garbage collection.

Results

RQ3 asks whether our concurrent implementation gives sufficiently low latency for interactive tasks. Benchmark 1 and 2 address this question by measuring the time it took to evaluate 500 instances of two kinds of attributes: a variable declaration attribute (Benchmark 1) and a method type attribute (Benchmark 2). In both benchmarks, the attributes are evaluated while concurrently computing compile-time errors and warnings for the whole subject program via a long-running attribute.

Our results show that when running the concurrent implementation, for any of the 10 programs, the highest average latency for finding a variable declaration is 0.5 ms, and the worst average latency for computing a method type is 5 ms. This is far below the acceptable threshold of 100 ms, so this answers RQ3 affirmatively.

If the sequential implementation with locks is used instead, the lower bound for the latency of an interactive task that starts right after the start of a long-running task will be the time it takes to complete the long-running task. This could in principle be a very long time. In our experiments, we used the computation of the `problems()` attribute as a typical representative of a long-running task. Our experiments show that the average time for this computation is between 500 ms and 11 seconds for the 10 different programs. The latency in the sequential case is thus clearly too high for interactive tasks. Figure 2 shows the average latency of the short-running attributes in Benchmark 1 and 2 compared to the long-running attribute.

Although it is not one of our research questions, we were interested to investigate the overhead and speedup of concurrent evaluation. We make some observations here based on the results of Benchmark 3 and 4. However, more performance evaluation is needed to draw definitive conclusions.

For overhead, we use the time for evaluating a long-running attribute in a single thread by using both the concurrent and sequential implementation. The overhead is computed as the concurrent time divided by the sequential time. Speedup of parallelization is estimated by taking the time to find all compile-time errors in a program using four parallel threads, divided by the time of doing the same computation in a single thread, using the concurrent implementation. Average overhead and speedup is shown in Table 2. If the speedup estimate is greater than the overhead, then finding the compile-time errors in four parallel threads using the concurrent implementation was on average faster than performing the same computation using the sequential implementation. This was the case for each subject program.

Program	NCLOC	Overhead	Speedup
ant-1.8.4	105,007	1.10	1.95
antlr-4.0	21,919	1.16	2.04
aoi-2.8.1	111,725	1.22	2.07
argouml-0.34	192,410	1.33	2.24
aspectj-1.6.9	412,394	1.17	2.43
azureus-4.8.1.2	484,739	1.22	2.32
castor-1.3.1	115,543	1.18	2.03
cayenne-3.0.1	127,529	1.16	1.98
checkstyle-5.1	23,316	1.07	2.27
cobertura-1.9.4.1	51,860	1.19	1.52

Table 2: Overhead of one thread running the concurrent algorithms, compared to running the sequential algorithms. Speedup on a 4-core processor when running four concurrent threads in parallel, as compared to running only one thread, all running the concurrent algorithms. NCLOC is the number of non-comment lines of code of the subject programs.

7.3 Threats to Validity

The general applicability of our results is limited by the fact that we have measured only three attributes in a single JastAdd-specified compiler, ExtendJ. However, in our opinion, ExtendJ is representative of a typical JastAdd compiler. Also, ExtendJ is one of the largest JastAdd projects freely available, and it uses all different attribute kinds discussed in this paper.

One alternative we looked at is the JModelica compiler for the Modelica language. The specification of this compiler is even larger than that of ExtendJ. However, JModelica currently uses several difficult to remove side-effects in the specification that would need to be fixed in order to run it concurrently.

Our results of course depend on the subject programs that were used. We selected these programs in a systematic manner from a well-known corpus in order to avoid bias.

8 Related Work

There are many algorithms for concurrent and parallel evaluation of Knuth attribute grammars, see the surveys by Jourdan [Jou91] and Paakki [Paa95]. However, that work is based on tree-walking evaluators which are not applicable to RAGs. First, tree-walking evaluators take only local dependencies into account, and can therefore not deal with the non-local dependencies arising from the use of reference attributes. Second, the tree-walking algorithms evaluate *all* attributes in an AST, whereas in RAGs, the only attributes that are evaluated are those needed for the computation of some goal attribute. Third, the tree-walking algorithms do not work for circularly dependent attributes. We have not found any previous attempts to parallelize the demand-driven evaluation algorithms used in RAGs, neither for circular nor for non-circular attributes.

In dynamic programming, results to subproblems are memoized, typically in a hash table, so that they only need to be computed once. In top-down dynamic programming, subproblems are computed and memoized recursively, similar to demand-driven evaluation of RAGs. Stivala et al. [Sti+10] have developed lock-free parallel algorithms for top-down dynamic programming. The basic idea is to let several threads solve the complete problem in parallel, and let them store and share the memoized subproblems through a global lock-free hash table. Randomization is used to encourage different threads to work on different subproblems. This approach is not sufficient for concurrent evaluation of RAGs, with their different kinds of attributes and fixed point computations. However, the idea of using randomization is interesting to investigate in future work for RAGs in order to gain better speed-up when running threads in parallel.

Ditter et al. [DCL12] develop a method for evaluating fixed-points in parallel, with the goal of speeding up software verification using boolean equation systems. They observe that in a fixed point iteration, the order of evaluating the different equations does not matter, and the equations can therefore be evaluated in parallel. We also make use of this observation in order to let several threads cooperatively evaluate a

circular attribute. Our demand-driven fixed point algorithm is, however, substantially different from the traditional fixed-point algorithm used by Ditter. In the traditional algorithm, it is assumed that both the equations and the variables to be solved are known a-priori, and it is therefore straight-forward to view this as a homogeneous data-parallel problem. For RAGs, neither the equations nor the variables are known a-priori, but are discovered during the recursive evaluation algorithm, and the fixed-point problem is heterogeneous, involving attributes associated with many different node types and which are defined by many different equations.

Similarly, traditional graph algorithms do not work for RAGs, as the attributes (graph nodes) are not represented by concrete objects with explicit dependencies.

9 Conclusions

The goal of this work was to develop safe concurrent algorithms for Circular Reference Attribute Grammars, in order to reduce latency in interactive tools.

We have designed new algorithms for circular attributes that make it practical to parallelize existing RAGs, and we generalized the algorithms for non-circular attributes to work more generally with circular attribute dependencies. Our algorithms support synthesized, inherited, parameterized, higher-order, collection, and circular attributes. Attribute-controlled rewrites are supported by representing rewrites as circular higher-order attributes [SH15].

Through empirical evaluation we showed that our algorithms can be used to reduce attribute evaluation latency. The implementation works well with existing tools, like an interactive language-based tools for exploring ASTs.

We implemented our algorithms in the JastAdd metacompiler and the implementation can be used directly for any well-formed JastAdd project. Using our implementation, it is straight-forward to parallelize a JastAdd project. Performance evaluation showed that concurrent attribute evaluation is on average slower than sequential evaluation, but the speedup when using four parallel threads outweighed that overhead.

Interesting future work includes exploring how the overhead of the concurrent implementation can be reduced, and how more performance can be gained by parallelization.

We have identified some possibilities of reducing concurrency overhead, and through more measurements, we expect to find additional ones. One possibility is to tune which attributes are memoized. The memoization configuration we used for our measurements is tuned for the sequential case, but the trade-offs will be different in the concurrent case: the cost of memoization is higher in the concurrent algorithms, and therefore it may pay off to avoid memoizing certain attributes when running concurrently. The data structures used in the concurrent implementation might also be improved.

Collection attributes present an opportunity to introduce parallelism, for example by dividing the work in the tree traversal between threads. Techniques like randomization and work stealing could be used to improve work distribution between threads. Refactoring attributes to be more long/short-running would also affect parallel per-

formance: short-running attributes reduce the risk of duplicate work when running in parallel, while long-running attributes reduce the relative concurrent memoization overhead.

References

[Åke+10] Johan Åkesson et al. “Modeling and optimization with Optimica and JModelica.org - Languages and tools for solving large-scale dynamic optimization problems”. In: *Computers & Chemical Engineering* 34.11 (2010), pp. 1737–1749.

[Boy05] John Tang Boyland. “Remote attribute grammars”. In: *J. ACM* 52.4 (2005), pp. 627–687.

[Bür15] Christoff Bürger. “Reference attribute grammar controlled graph rewriting: motivation and overview”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2015. ACM, 2015, pp. 89–100.

[DCL12] Alexander Ditter, Milan Ceska, and Gerald Lüttgen. “On Parallel Software Verification Using Boolean Equation Systems”. In: *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. Ed. by Alastair F. Donaldson and David Parker. Vol. 7385. Lecture Notes in Computer Science. Springer, 2012, pp. 80–97.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 1–18.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. Palo Alto, CA, USA: ACM, 1986, pp. 85–98.

[FCH15] Niklas Fors, Gustav Cedersjö, and Görel Hedin. “JavaRAG: a Java library for reference attribute grammars”. In: *MODULARITY*. ACM, 2015, pp. 55–67.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[Her88] Maurice Herlihy. “Impossibility and Universality Results for Wait-Free Synchronization”. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*. Ed. by Danny Dolev. ACM, 1988, pp. 276–290.

[Her06] Maurice Herlihy. “The art of multiprocessor programming”. In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*. Ed. by Eric Ruppert and Dahlia Malkhi. ACM, 2006, pp. 1–2.

[Jon90] Larry G. Jones. “Efficient Evaluation of Circular Attribute Grammars”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 429–462.

[Jou84] Martin Jourdan. “An Optimal-time Recursive Evaluator for Attribute Grammars”. In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[Jou91] Martin Jourdan. “A Survey of Parallel Attribute Evaluation Methods”. In: *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings*. Ed. by Henk Alblas and Borivoj Melichar. Vol. 545. Lecture Notes in Computer Science. Springer, 1991, pp. 234–255.

[Kas80] Uwe Kastens. “Ordered Attributed Grammars”. In: *Acta Inf.* 13 (1980), pp. 229–256.

[Knu68b] Donald E. Knuth. “Semantics of Context-Free Languages”. In: *Mathematical Systems Theory* 2.2 (1968), pp. 127–145.

[LWZ16] Patrick Lange, René Weller, and Gabriel Zachmann. “Wait-free hash maps in the entity-component-system pattern for realtime interactive systems”. In: *9th IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS 2016, Greenville, SC, USA, March 20, 2016*. IEEE Computer Society, 2016, pp. 1–8.

[LTH16] Joel Lindholm, Johan Thorsberg, and Görel Hedin. “DrAST: An Inspection Tool for Attributed Syntax Trees (Tool Demo)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 176–180.

[MEH07] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. “Extending Attribute Grammars with Collection Attributes–Evaluation and Applications”. In: *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (Source Code Analysis and Manipulation 2007), September 30 - October 1, 2007, Paris, France*. IEEE Computer Society, 2007, pp. 69–80.

[MH03] Eva Magnusson and Görel Hedin. “Circular Reference Attributed Grammars - Their Evaluation and Applications”. In: *Electr. Notes Theor. Comput. Sci.* 82.3 (2003), pp. 532–554.

[Nau05] David A. Naumann. “Observational Purity and Encapsulation”. In: *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Vol. 3442. Lecture Notes in Computer Science. Springer, 2005, pp. 190–204.

[Nie93] Jakob Nielsen. *Usability engineering*. Academic Press, 1993.

[Paa95] Jukka Paakki. “Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation”. In: *ACM Comput. Surv.* 27.2 (1995), pp. 196–255.

[SKV10] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 205–219.

[SH15] Emma Söderberg and Görel Hedin. “Declarative rewriting through circular nonterminal attributes”. In: *Computer Languages, Systems & Structures* 44 (2015), pp. 3–23.

[Sti+10] Alex D. Stivala et al. “Lock-free parallel dynamic programming”. In: *J. Parallel Distrib. Comput.* 70.8 (2010), pp. 839–848.

[Tem+10] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345.

[VSK89a] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI ’89*. Portland, Oregon, USA: ACM, 1989, pp. 131–145.

[Wyk+07] Eric Van Wyk et al. “Attribute Grammar-Based Language Extensions for Java”. In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 575–599.

[Wyk+08] Eric Van Wyk et al. “Silver: an Extensible Attribute Grammar System”. In: *Electr. Notes Theor. Comput. Sci.* 203.2 (2008), pp. 103–116.

Appendices

A Circular Attribute Correctness Proofs

This section gives the proofs of soundness and termination for CEval, i.e., the concurrent algorithm for circular attributes (Algorithm 5). We start by giving several technical lemmas with proofs. The main theorems and proofs appear at the end of this section.

The lemmas below state properties about iterations of CASE1 as executed in one single thread. There may be concurrently executing threads, running their own CASE1 iterations, but threads never share CASE1 executions. The only interaction between threads happens through reading and writing global shared attribute value approximations (gv_x, gv_z, etc.). Iteration indices can, without loss of generality, be thought of as being globally unique between all threads.

We will often talk about properties such as the *done* flag being set before some point in execution. This means that given some linearization of several threads executing the algorithm concurrently, the linearization point of a write to gv_x, setting *done* to TRUE, was linearized as happening before the given point in the current thread being discussed.

Also note that the lemmas only deal with attribute instances, though we sometimes refer to them as just attributes.

First, we will need a few definitions.

Definition 1 (Attribute Set)**.** *$\mathcal{A}$ is the set of attribute instances in some attributed AST.*

Definition 2 (Direct Dependencies)**.** *For an attribute $x \in \mathcal{A}$, $d(x)$ is the set of attributes that x directly depends on.*

Definition 3 (Transitive Dependencies)**.** *For an attribute $x \in \mathcal{A}$, $D(x)$ is the set of attributes that x transitively depends on, including x.*

We assume here that $D(x)$ is strongly connected, in other words, for all $y \in D(x)$, $D(y) = D(x)$.

The following observation restates a consequence of how semantic functions are translated into COMPUTE procedures:

Observation 1. *For an attribute $x \in \mathcal{A}$ and an iteration i of the loop in CASE1(x), executing CASE2(x, i) leads to executing CEVAL(y, i) for all $y \in d(x)$.*

We will need to reason about what happens during an iteration i of the loop in CASE1. The following definitions introduce boolean functions to succinctly reason about this.

Definition 4 (Execution of Case2)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in CASE1(x). Then, $case2(x, i)$ is true iff CASE2(x, i) was executed during iteration i.*

Definition 5 (Fixed-Point Value)**.** *Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in CASE1(x). Then, $fix(x, i)$ is true iff, before the end of iteration i, the global approximation for x is equal to the fixed-point value of x.*

Definition 6 (Memoized Value)**.** *Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in CASE1(x). Then, $done(x, i)$ is true iff the* done *flag in the tuple gv_x is TRUE before the end of iteration i.*

In other words, $done(x, i)$ implies that x was memoized before or during iteration i.

Note that $\mathit{fix}(x, i)$ is not equivalent to $done(x, i)$, though if the algorithm is correct, $done(x, i)$ should imply $\mathit{fix}(x, i)$. We prove this in Lemma 3.

Lemma 1 (Case2 on Direct Dependency)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in CASE1(x), and let $y \in d(x)$ be a direct dependency of x. If CASE2(x, i) is executed during iteration i, and y is not marked as done before the end of the iteration, then CASE2(y, i) is executed at some point during the iteration.*

Proof. According to Observation 1, CEVAL(y, i) is executed as a direct consequence of executing CASE2(x, i).

When CEVAL(y, i) is executed, there are three cases:

- y is already marked as done, contradicting the premise of the lemma, or,
- no previous approximation has been stored for y during iteration i, so then CASE2(y, i) is executed, or,
- a local approximation of y has been previously stored during iteration i. Note that local approximations are only stored in CASE2, so then CASE2(y, i) was executed at some point during i.

□

We now define another helper function to reason about paths in the dependency graph of an attribute:

Definition 7 (Dependency Paths)**.** *Let $x \in \mathcal{A}$ be an attribute, with a transitive dependency on $y \in D(x)$. The function $paths(x, y)$ gives all acyclic paths from x to y following the attribute dependency graph.*

In other words, for each $p = (a_1, a_2, \cdots, a_n)$ in $paths(x, y)$, the following holds:

- $x = a_1$,
- $y = a_n$,
- *and $a_{i+1} \in d(a_i)$, where $1 \leq i < n$.*

Lemma 2 (Case2 on All if None Done)**.** *Let $x \in \mathcal{A}$ be an attribute, transitively depending on $y \in D(x)$, and let i be an iteration of the loop in CASE1(x). For each path $p \in paths(x, y)$, from x to y, where none of the attributes in p are marked as done by the end of i, CASE2(z, i) is executed for each attribute z in p during i.*

Proof. By induction on prefixes of p. The one-length prefix of p is equal to x, and since i is an iteration of CASE1(x), CASE2(x, i) is directly executed in the loop body.

Assuming that the lemma holds for an n-length prefix of p, we must show that it holds for a prefix of length $n + 1$. Let a_n be the n^{th} element of p, then it follows from the induction hypothesis that $\neg done(a_n)$, and by the definition of $paths(x, y)$ it follows that $a_{n+1} \in d(a_n)$. By the induction hypothesis it also follows that $case2(a_n, i)$, and together with the conclusion that $\neg done(a_n)$, Lemma 1 gives the goal: $case2(a_{n+1}, i)$.

By induction the lemma holds for any length prefix of $p \geq 0$, so the lemma holds for p. □

Lemma 3 (Done $\implies$ Fix)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in CASE1(x). If x is marked as* done *before the end of i, then for each transitive dependency $y \in D(x)$, the global approximation of y is equal to the fixed-point value of y before the end of i.*

Proof. Note that the *done* flag for an attribute can only be set to TRUE by the CAS after the CASE1 loop. CAS is a linearizable operation, so the effect of several (concurrent) CAS calls is identical to the effect of some sequential ordering of the CAS operations. Consequently, among the CAS calls that mark attributes in $D(x)$ as *done*, there exists a first one.

Consider the first attribute $y \in D(x)$ that is marked as $done$ by the CAS at the end of CASE1. The loop always takes at least one iteration, so let k denote the last iteration before y was marked as $done$.

From Lemma 2 and the assumption that y is the first attribute in $D(y)$ which is set to computed, it follows that CASE2 was executed for all $z \in D(y)$. The loop condition implies that none of the attributes in $D(y)$ changed approximation, thus a simultaneous fixed-point has been reached and $fix(z, k)$ is true for all $z \in D(y)$.

Because $D(x)$ is strongly connected, and $y \in D(x)$, then $D(x) = D(y)$. Substituting $D(y)$ for $D(x)$ we get the conclusion: for all $z \in D(x)$ it holds that $fix(z, k)$ is true. □

Lemma 4 (All Fix Or Case2)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in CASE1(x). Then one of the following properties hold:*

- *all attributes in $D(x)$ have reached their fixed-point value before the end of iteration i, or,*
- *CASE2 is executed for all attributes in $D(x)$ during iteration i.*

Proof. There are two cases:

- some attribute $z \in D(x)$ was marked as done before the end of iteration i, or,
- none of the attributes in $D(x)$ were marked as done before the end of iteration i.

In the first case, there exists some $z \in D(x)$ such that $done(z, i)$ is true, and by Lemma 3 we have the fact that all attributes $w \in D(z)$ have reached their fixed-point values before the end of iteration i. Additionally, $D(x)$ is strongly connected, and $z \in D(x)$ means that $D(x) = D(z)$. Substituting $D(z)$ for $D(x)$ gives us the goal: for all $w \in D(x)$, $fix(w, i)$ is true.

In the second case, there does not exist an attribute $z \in D(x)$ such that $done(z, i)$ is true. Consequently, for each path $p \in paths(x, y)$ it most hold that $done(z, i)$ is false for each element z of p. By Lemma 2 it follows that CASE2(y, i) is executed during iteration i for all $y \in D(x)$. □

Lemma 5 (Case1 Sound)**.** *Let $x \in \mathcal{A}$ be an attribute, with a transitive dependency on some attribute $y \in D(x)$, and let i be the last iteration of an execution of CASE1(x). Then, the global approximation of y is equal to y's fixed-point value before the end of iteration i, i.e. $fix(y, i)$ is true.*

Proof. By Lemma 4 there are two cases:

- all attributes in $D(x)$ have reached their fixed-point values before the end of iteration i, or,
- CASE2 is executed for all attributes in $D(x)$ during iteration i.

In the first case, we get the conclusion directly from the premise $y \in D(x)$.

In the second case, CASE2(z, i) is executed during iteration i for all $z \in D(x)$. Since i was the last iteration, and the loop is only exited if $tls.change =$ FALSE, it must be that all attributes $z \in D(x)$ were computed to the same value as their previous approximations. According to our definition of simultaneous fixed-point, all attributes in $D(x)$ must then have reached their fixed-point value. Again, according to the premise, $y \in D(x)$ leads to the conclusion that y has reached its fixed-point value. □

Lemma 6 (Case2 is Monotonic)**.** *Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in CASE1(x). Then, executing CASE2(x, i) does not update the global approximation for x to a new value that is lower in the value lattice of x.*

Proof. First note that the global approximations of attributes are updated only by using CAS. The CAS operation is linearizable, so the calls take effect as if executed in some sequential order. Consequently, there exists a first global approximation update among any set of global approximation updates for any set of attributes.

Proof by contradiction. Assume that there exists an attribute $z \in \mathcal{A}$ which is the first attribute whose approximation gv_z is updated to a lower value in the value lattice of z by CASE2(z, k) during some iteration k.

At the start of CASE2(z, k), the value v_0 was read from gv_z. Note that the approximation v_0 was computed by applying the semantic function of z to some state S_0. Later in CASE2, computing z gives a value v_1 applying the semantic function to a state S_1. Because the semantic function is monotone, according to well-formedness condition WF3, the value v_1 can only be lower than v_0 in the attributes value lattice if S_1 is lower than S_0 in the state lattice. However, since S_1 is read after S_0, there must exist some other attribute y whose approximation has been updated to a lower value, but this contradicts the assumption that z was the first attribute to update approximation to a lower value. □

Now we can finally present the main correctness theorems and proofs using the above lemmas. There are two things we must prove: that CEVAL always terminates, and that it returns the correct value.

Theorem 13 (Termination)**.** *For a well-formed circular attribute x, CEVAL$(x, 0)$ terminates.*

Proof. We must show that all individual operations terminate, and that only a finite number of those operations are performed.

We can distinguish the following kinds of operations in the algorithm:

- writes and reads to $tls.change$,
- queries and updates of $tls.value$ and $tls.iter$,
- updating a global approximation with CAS,
- reading a global approximation,
- unpacking a tuple

All of the above listed operations are assumed to terminate. Some of them terminate due to Java semantics, and the remaining can be ensured to terminate using appropriate library implementations. The $tls.value$ and $tls.iter$ maps can use non-concurrent implementations since they are thread-local.

It remains to show that a finite number of these kinds of operations are used. For this, it suffices to show that CEVAL, CASE1, CASE2, and CASE3 are executed a finite number of times. Except the initial call to CEVAL$(x, 0)$, all calls to CEVAL, CASE2, and CASE3 are executed via CASE1. Additionally, CASE2 causes recursion. So, we must show that CASE1 executes a finite number of iterations and CASE2 never leads to unbounded recursion.

First, we show that CASE1 always has a finite number of iterations. For each iteration i of CASE1, by Lemma 4, there are two cases:

- all attributes in $D(x)$ have reached their fixed-point values before the end of iteration i, or,
- CASE2 is executed for all attributes in $D(x)$ during iteration i.

If the first case holds for some iteration i, then i is either the last or penultimate iteration. There can not be more than one additional iteration after i because the next iteration will not be able to update any attribute approximation to a new value, causing *tls.change* to remain FALSE after the assignment at the start of the next iteration, and then leading to the loop exiting after that iteration.

Now, assume that there is an unbounded number of iterations. This implies that the second case must hold for all iterations: CASE2 is executed for all attributes in $D(x)$ in each iteration k. However, executing CASE2 for all attributes means that *tls.change* is set to TRUE only if at least one attribute changed approximation. Since attribute values are in a lattice, there are only a finite number of possible value updates until no value can be further updated. Additionally, Lemma 6 shows that all approximation updates are monotonic. Thus, after a finite number of iterations it will not be possible to update any approximation and *tls.change* remains FALSE and the loop ends.

Finally, we show that CASE2 never leads to unbounded recursion. Note that CASE2 is only called during some iteration k of CASE1. If there exists an unbounded recursion for some attribute z, then by definition executing CASE2(z, k) leads to a call to CASE2(z, k). However, the condition of the **if**-statement for calling CASE2 in CEVAL tests if z has already been computed during the iteration k, by reading $tls.iter(z)$ and comparing against k. In the first execution of CASE2(z, k), $tls.iter(z)$ is assigned k before the COMPUTE call, which is the only control flow path that could lead to recursion. Thus, CASE2 can not lead to unbounded recursion for any z and k. □

Theorem 14 (Fixpoint Sound)**.** *For a well-formed circular attribute x, CEVAL$(x, 0)$ computes the least fixed-point value of x.*

Proof. Lemma 5 shows that CASE1(x) only terminates after the global approximations of all $y \in D(x)$ have reached their fixed-point values. The approximations of all attributes in $D(x)$ then form a simultaneous fixed point. Because $x \in D(x)$, this means that x has reached a fixed-point value. To show that the value of x is the *least* fixed-point value, we must show that the initial approximations of all attributes in $D(x)$ were their respective bottom values. This is ensured by the first **if**-statement at the start of CEVAL. Because the approximation of each attribute is initially set to (NIL, FALSE), any thread executing CEVAL will attempt to initialize gv_x to $(\bot_x, \text{FALSE})$ if it sees the initial state. This happens before using the global approximation, because all uses of the global approximation are in CASE2 which only happens after the first **if**-statement of CEVAL.

It remains to show that multiple threads can concurrently execute the algorithm without affecting the correctness of each other's results. For this we only need to look at the points in the algorithm where threads communicate - that is, via $read(gv_x)$ and $CAS(gv_x, \cdots)$.

The CAS used to initialize the global value of x to the bottom value, $\bot_x$, only has an effect for a single invocation of the CAS, and it ensures that all threads read the bottom value of x as the first approximation of x.

In CASE1, the global value is accessed when the fixed-point computation is finished and a thread tries to mark the global approximation as the final result. The

CAS fails if another thread has marked the same result as final. Since the returned value is read from the global approximation two separate threads will return the same result regardless of which thread performed the successful CAS.

In CASE2, the global approximation is read and used as the local approximation in case it was different from the local approximation. A fundamental property of CASE2 is that it should only be able to update the global approximation to a higher value, which is proven in Lemma 6. □

B DrAST Screenshot

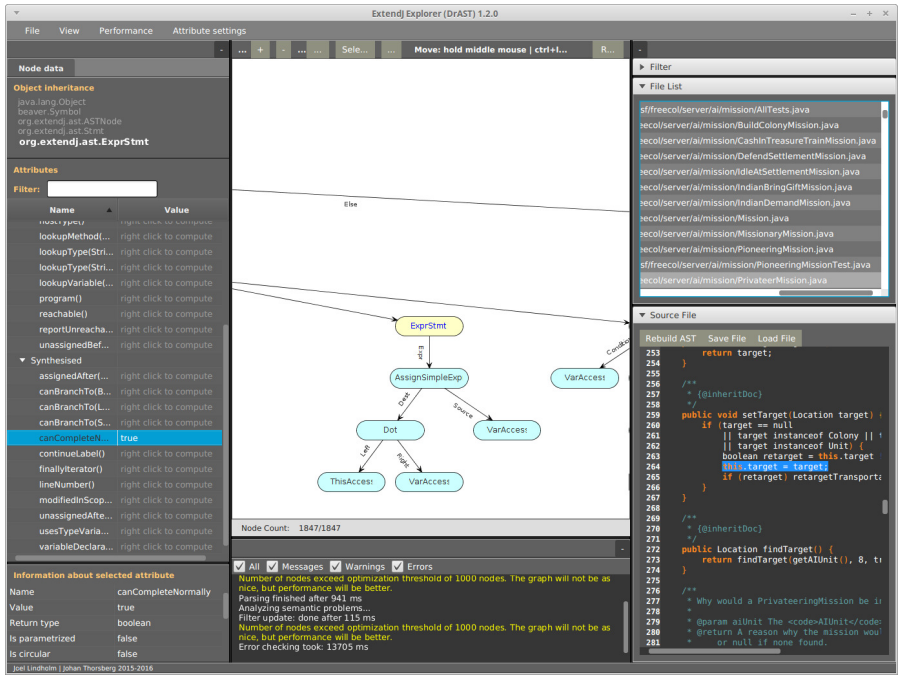
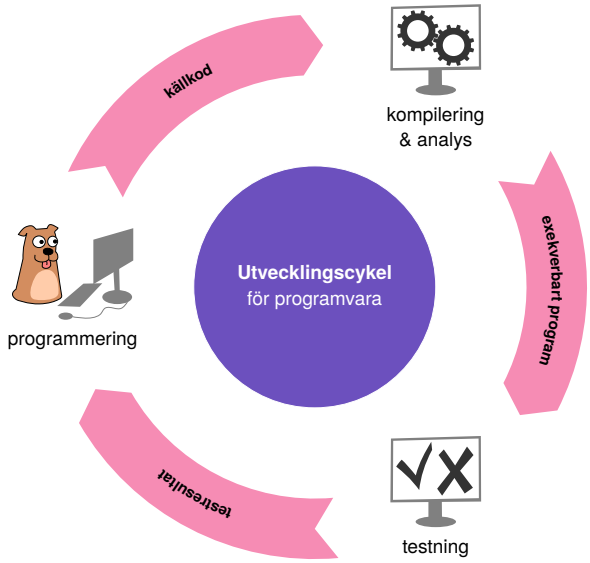


Figure 3: Screenshot of DrAST extended with ExtendJ. The center part of the window contains a graph of the AST of one source file in freecol-0.10.7. The left part of the window contains a list of all attributes in the currently selected node. The `canCompleteNormally()` attribute has been selected and manually computed by the user. The right side of the window contains a list of all files in freecol-0.10.7, and below that is a source file view of the currently selected file. The bottom center part of the window shows status messages from DrAST. Note the message about compile-time error checking at the end. No compile-time errors or warnings were present in freecol-0.10.7.

Popular Science Summary in Swedish

Utveckling av statisk analys

Statisk programanalys, eller kort och gott statisk analys, är av stor betydelse inom mjukvaruutveckling. Statisk analys är en samlingsterm för olika automatiserade analyser av datorprogram. Framför allt behövs statisk analys för att kompilera program till exekverbar maskinkod, så att de kan köras på en dator (eller mobiltelefon, till exempel). Statisk analys används också för att hitta och förhindra fel i program, samt för att optimera prestandan hos program. Med statisk analys kan man bland annat hitta säkerhetshål och förhindra att känslig data läcker ut ur ett program. Dessutom används statisk analys i *programmeringsverktyg*, alltså de verktyg en programmerare använder för att konstruera sina program. Till exempel kan statisk analys användas för att föreslå ändringar i koden (kodkomplettering och korrigering av enkla fel), eller för att låta programmeraren snabbt hoppa mellan definitioner och användningar av namn i koden.



Figur: Den vanliga utvecklingsprocessen för datorprogram. Under programmering och kompilering används statisk analys av programmeraren och kompilatorn.

Statisk analys är en form av mjukvara som ofta är komplicerad och tidskrävande att utveckla. Dessutom finns det ofta ett behov av att kunna bygga vidare på en befintlig statisk analys med nya egenskaper, till exempel om programmeringsspråket som analyseras uppdateras. Således är det önskvärt att utveckla statisk analys med en flexibel mjukvaruarkitektur, det vill säga på ett sätt som gör det möjligt att bygga på analysen utan allt för stor ansträngning.

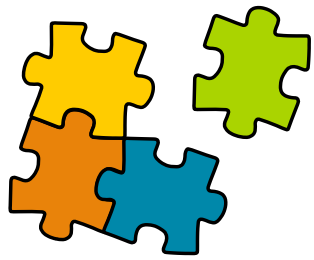
Givet att det finns några grundläggande analyser som är utvecklade med flexibel arkitektur så kan vi utveckla många nya viktiga analyser som använder de grundläggande analyserna som byggstenar. Detta sparar mycket tid och pengar.

Deklarativ programmering

För att uppnå en flexibel mjukvaruarkitektur kan vi använda oss av *deklarativ programmering*. Det innebär att programmeraren beskriver *vad* som skall beräknas, snarare än att exakt beskriva stegen som datorn tar för att beräkna det som behövs. Den främsta fördelen med deklarativ programmering, när det gäller att bygga flexibla program, är att det blir enkelt att dela upp ett deklarativt program i moduler. Programmoduler kan utvecklas separat och sedan kombineras på olika sätt, vilket sparar tid för programmerarna i det långa loppet. En deklarativ statisk analys kan ganska enkelt användas som en modul inuti en större, mer komplicerad, analys.

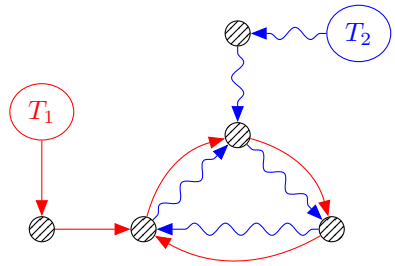
En statisk analys kan delas upp i små deklarativa delar som kallas *attribut*. Attributen kan enkelt kombineras till moduler som sedan används för att utveckla nya analyser.

Ett sätt att tänka på modulerna i en statisk analys är som pusselbitar. En pusselbit (modul) kan läggas till i ett pussel (en statisk analys) så länge den har rätt form (programgränssnitt):



Ett viktigt forskningsresultat i den här avhandlingen är en ny algoritm för att kunna beräkna flera attribut samtidigt. Detta kan förkorta beräkningstiden: i ett experiment visade mina mätningar att kompileringstiden kunde kortas med hälften. En annan fördel är att svarstiden kan minskas i interaktiva programmeringsverktyg: med samtidiga beräkningar måste man inte vänta på att föregående beräkningar är klara innan man börjar på den nästa. Mina experimentella resultat visar att svarstiden kan minskas från sekunder till under en millisekund.

För att kunna utföra beräkningar samtidigt har de flesta datorer idag flera *kärnor*, där varje kärna exekverar varsin *beräkningstråd* jämnlöpande med andra kärnor i datorn. Med min nya algoritm kan den sammanlagda beräkningstiden för attribut minskas genom att dela upp beräkningsarbetet på flera (beräknings)trådar. När en tråd beräknat ett attribut sparas värdet och kan direkt användas av en efterföljande tråd som då slipper göra om beräkningsarbetet. Uppdelning av arbetet för fem attribut med två jämnlöpande trådar illustreras i figuren nedan.



Figur: Illustration av två trådar, T_1 och T_2, som samtidigt beräknar fem attribut (randiga cirklar). Beräkningsflödet för T_1 visas med röda pilar, och för T_2 med vågiga blåa pilar. Beräkningen är i det här fallet *iterativ*, och går runt i en cirkel tills det rätta värdet har beräknats. Med min algoritm kan varje tråd utföra sin beräkning i varje attribut utan att behöva vänta på den andra tråden. Dessutom hjälper trådarna varandra genom att en tråd använder resultatet från den andra tråden om den andra hann före.

Kompilatorer

Statiska analyser är centrala i kompilatorer, det vill säga program som översätter programmerarens kod till maskinkod som en dator kan köra. I mitt arbete har jag arbetat främst med en kompilator som heter ExtendJ, som kompilerar program skrivna i programmeringsspråket Java. ExtendJ är fullt deklarativt programmerad och använder attribut, vilket gör det förhållandevis enkelt att använda ExtendJ för att bygga nya statiska analyser för Java.

För min avhandling har jag dels utvecklat nya analyser som bygger på ExtendJ, dels förbättrat ExtendJ självt.

Bland de statiska analyser som jag utvecklat i ExtendJ finns en analys som kan användas för att minska testningstiden för Java-program. Idén är att endast köra om de test som kan påverkas av den senaste ändringen i programmet som testas. Med denna teknik lyckades jag halvera den totala tiden för testning av flera olika program.

En av förbättringarna jag gjort i ExtendJ var att uppdatera kompilatorn till att stödja en ny version av Java. Sammanlagt har förbättringarna som jag gjort i ExtendJ gjort det enklare för forskargrupper runt hela världen att bygga nya statiska analyser och språktillägg till Java.