Fully Declarative Specification of Static Code Checkers

Alexandru Dura

ABSTRACT

Static code checkers are tools that help software engineers by automatically finding defects without executing the programs. These tools contain a set of detectors that rely on static program analyses to find common programming defects or to enforce coding guidelines.

While existing code checker frameworks package a rich collection of detectors, aimed at common bug defects, the effort to adapt these detectors to specific project needs is not trivial. In their definition, the detectors rely on checker-specific program representations and auxiliary data structures, which incurs a high up-front cost for customization attempts.

Being encoded in a general-purpose programming language, the detectors inherit the evaluation model of the host programming language. This precludes the adoption of evaluation schemes that fit the dynamics of the project, such as incremental evaluation or caching of partial results.

In this thesis we address the hindrances to adaptability in current checker frameworks by combining two declarative techniques: syntactic pattern matching and logic programming in Datalog. We use syntactic pattern matching to identify the program fragments that are of interest for a detector and Datalog logical rules to enable non-local reasoning between these fragments of interest. Syntactic patterns allow us to decouple the detector specification from the internal representations of programs, while the use of Datalog-style rules enables the adoption of alternative evaluation modes, such as incremental evaluation.

We materialize our techniques in declarative specification languages and runtimes that facilitate the construction of declarative static code checking frameworks for the C and Java languages. We show that these declarative specification languages enable concise description of bug detectors, while achieving analysis quality and runtime performance that is comparable with established frameworks.

CONTRIBUTION STATEMENT

This thesis is a compilation consisting of an introduction and three papers:

Paper I Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*. SOAP 2019. URL: https://doi.org/10.1145/3315568.3329970

Paper II Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection". In: *Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA* (2021). URL: https://doi.org/10.1145/3485542

Paper III Alexandru Dura and Christoph Reichenbach. "Clog: A Declarative Language for C Static Code Checkers". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. URL: https://doi.org/10.1145/3640537.3641579

The artifacts related to this thesis are:

Artifact I Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. *JavaDL: Automatically Incrementalizing Java Bug Pattern Detection*. 2021. URL: https://doi.org/10.5281/zenodo.5090141

Artifact II Alexandru Dura and Christoph Reichenbach. *Clog: A Declarative Language for C Static Code Checkers*. 2024. URL: https://doi.org/10.5281/zenodo.10525151

The table below indicates the author's contributions to each of the papers. The dark portion of the circle represents the contribution of the author.

Paper	Writing	Concepts	Implementation	Evaluation	Artifact
I	◑	◑	◕	◕	
II	◕	◕	●	●	●
III	◕	◕	●	●	●

ACKNOWLEDGEMENTS

This dissertation owes its existence to all the help I have received.

My deepest gratitude goes towards my supervisors. With insight and kindness, they guided me towards completing this thesis — in spite of my resistance.

Christoph Reichenbach, my main supervisor, taught me almost everything I know about research in general and program analysis in particular. He was invested in the research and always open to an in-depth technical discussion.

Emma Söderberg, my co-supervisor, asked the challenging questions that enabled me to see my research from a different perspective. She stepped in and helped me when I needed the most.

I am grateful to the members of the Software Development and Environments reading group for everything I have learned from them: Alfred Åkesson, Anton Risberg Alaküla, prof. Boris Magnusson, Hampus Balldin, Idriss Riouak, Jesper Öqvist, Mattias Nordahl and Niklas Fors. I thank prof. Görel Hedin for her thoughtful feedback and for handling my never-ending stream of parental leave requests.

I thank Patrik Åberg and Magnus Templing for showing interest in my work and for organizing my internship at Ericsson.

I was lucky to make good friends during my studies. I enjoyed the camaraderie of Noric Couderc and our wandering conversations on everything. Matthias Mayr walked with me during the pandemic and afterwards. Momina Rizwan listened to my parenting stories and shared hers. Flavius Gruian introduced me to bouldering.

At the foundation of this work lies the unbounded support of my family, their love and their trust in me. For her moral support, Gabriela, my better half, deserves to be a co-author — at least for the work done during the pandemic. Our children, Jacob and David, brought me joy and the necessary distraction from the downs of graduate student life.

I am grateful to my parents for many things. For the sake of brevity, I thank my mother, Valerica, for showing me how to be persistent and pursue my goals; I thank my father, Octavian, for instilling in me the interest in science and engineering.

CONTENTS

INTRODUCTION

Software is pervasive in our world. Almost any object that has a power source is also running a piece of software. Objects that, two decades ago, were purely analog are now software-controlled for improved usability and efficiency: light bulbs, stoves, heating thermostats, etc. Other objects, such as medical journals, train tickets and even currency have been almost completely replaced by their digital equivalents — all controlled by software. This puts software in a central but invisible role in our everyday lives and also in our society. To ensure that our everyday lives and our society function smoothly we have to ensure that the software is reliable and that it behaves according to our intentions.

The study of the behaviour of computer programs is called *program analysis*. We study the behaviour of programs in two major ways. The first approach, *dynamic program analysis*, is to run the program on some input and observe the runtime behaviour of the program. While this approach can reveal some defects in the program, it cannot guarantee the lack of defects on every possible input since it is infeasible to run any non-trivial program on all possible inputs.

The other approach, *static program analysis*, is concerned with automatic reasoning about a program's behaviour without running the program. Static program analyses infer the possible behaviours of a program from its source code or binary representation. While dynamic program analyses report only programs that have defects, static analyses may either under-approximate the behaviour of the program and let some defects go unreported, or be conservative and safely over-approximate a program's behaviour, and thus report on all programs containing defects, but also on some defect-free programs. The latter, conservative static analyses have been embedded in compilers. Two prominent uses of static analyses in compilers are type checking and optimization. Type checkers ensure that any operation encountered during program execution is well-defined (i.e. the program does not get stuck). In optimizers, we use program analysis to prove that transforming the program does not change its observable behaviour.

Complementary to compilers, engineers have built stand-alone tools around program analyses. These tools extract information from the program and combine it according to a set of rules to detect common bug patterns. The choice for not embedding

bug pattern detection in compilers has pragmatic reasons: some analyses are too slow to run within the development loop, while others are domain-specific and would not be relevant for other users of the same programming language. We call these tools *(static) program analyzers* or, mundanely, *(static) code checkers*. Static code checkers typically package hundreds of detectors, each aimed at finding a common bug pattern for a given language or incorrect usage of a widely-adopted API.

While code checkers are gaining traction in companies and open-source projects, their users still find pain-points that are not yet addressed. In a study from Microsoft [CB16], more than 80% of developers complain about the selection of detectors that are enabled by default and about 15% complain about the lack of support for custom rules. This hints that customizability of code checkers is a concern for software developers. Other issues with code checkers reported in the same study are the high number of false-positive warnings, the lack of ranking between warnings and the speed of the checker. Another report from Google [Sad+18] witnesses that customizability of their Tricorder tool is a valuable feature: users contribute checks for generic bug patterns and they also build project-specific detectors.

Traditionally, code checkers are built as opaque black boxes that parse the source of the analyzed program or its binary encoding, build an internal representation in the form of an abstract syntax tree or three address code, and then derive their results using this representation. The internal representation is tool-specific and the reasoning is encoded in imperative code, where all the data structures are exposed and their consistency is explicitly managed. This design of code checkers is at odds with the need for customizability: there is a high up-front effort that the user of the checker needs to make in order to customize the checker.

A majority of software engineers prefer that static code checkers present their results in the IDE or during the build process [CB16; Joh+13], which means that code checkers should compute their results with low latency. This is hard to achieve for complex checks that require precise heap modeling or inter-procedural flow information, unless they reuse analyses results between the runs. Thus, being able to run a code checker incrementally is a desirable feature. However, current code checkers manage their internal state explicitly and their rules are encoded in imperative code, which makes their incrementalization an impractical task.

Fortunately, declarative approaches have made strides in the space of program analysis. These approaches focus on *what* the rules of the analysis are, rather then *how* they are implemented, delegating the implementation details to an underlying evaluation framework. In particular, researchers have been successfully employing Datalog, a declarative logic programming language, to express sophisticated point-to analyses [BS09; BS16], decompilation and analysis of smart contracts [Gre+20], inter-procedural data-flow analyses [MYL16]. Besides being adopted as a language for describing program analyses, researchers have been developing novel approaches for evaluating Datalog programs incrementally [Sza+18], reducing false-positive rates by ranking reports based on user feedback [Rag+18] or explaining the results of a Datalog program [ZSS20]. This evolution has been supported by the advent of high-

performance Datalog engines such as LogicBlox [Are+15a] and Soufflé [Sch+16].

In spite of significant progress and adoption of Datalog as a language for describing program analyses, these analyses are still dependent on an additional software component, the *fact extractor*, that parses the analyzed program, builds an internal representation and extracts the relevant facts for client analyses. These fact extractors are usually derived from traditional compiler front-ends and are written in an imperative style, which is opaque to incrementalization or explainability.

Moreover, fact extractors are specific to the analysis that uses the facts, but different analyses refer to different fragments of the analyzed program, and thus they require different facts. For example, when the client analysis is a pointer analysis, the fact extractor may only provide information about the memory operations of the program, discarding other information available in the source, so it is not possible to reuse the same fact extractor for checking for unnecessary parentheses in arithmetic expressions. Fact extractors identify relevant program fragments by *pattern matching* on the internal representation, which requires that the patterns are expressed in terms of this representation, making the fact extractors dependent on it. This dependency raises the threshold for analysis customization, since modifying the set of available facts requires familiarity with the internals of the fact extractor. But this dependency is avoidable. By using *syntactic patterns* in the concrete syntax of the analyzed language, we are able to decouple the fact extractors, and consequently the client analyses, from any internal representation.

The aim of this thesis is to enable the construction of *fully declarative program checkers*. We achieve this by combining two existing declarative approaches: syntactic pattern matching and logic programming in Datalog. It is the combination of these declarative approaches that enables us to build *fully declarative* code checkers that are explainable, can be evaluated incrementally and with high performance by state-of-the-art Datalog engines.

Thesis Statement Syntactic patterns and logic programming enable the construction of fully declarative static code checkers that are independent of any program representation other than its source code. Our experiments show that these fully declarative code checkers are comparable with well-established systems in regard to analysis quality and runtime performance.

Methodology The research in this thesis follows the *systems development* methodology, as proposed by Nunamaker et al. [NCP90]. To demonstrate that the combination of syntactic patterns and logic programming is a feasible approach for constructing static code checkers from the perspective of analysis quality and performance, we build three static code checker systems aimed at different programming languages: METADL for Datalog, JAVADL for Java and CLOG for C.

In Figure 1, we present a design science view of the activities that contributed to this thesis, using the framework described by Engström et al. [Eng+20]. The development of the core of this work, the combination of Datalog and syntactic patterns, followed
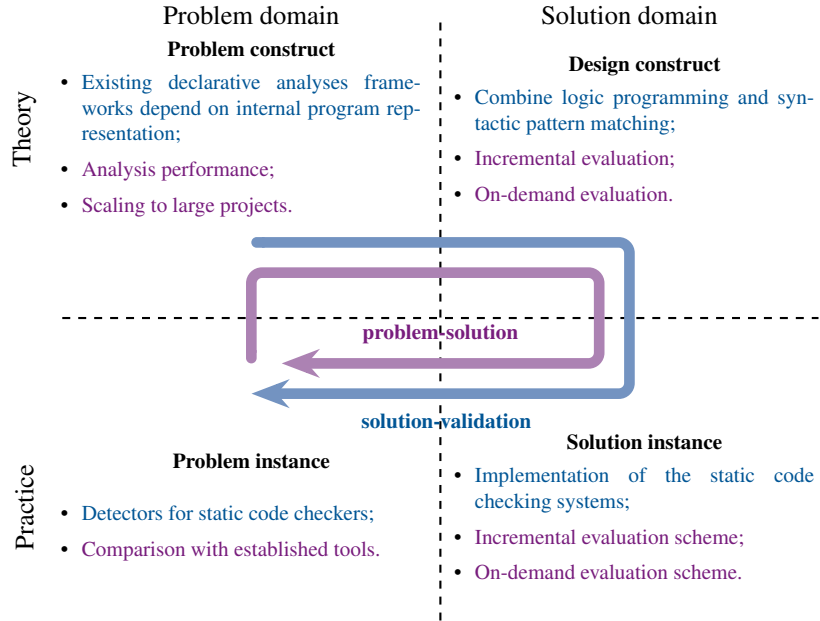
Figure 1: A map of the activities that contributed to this thesis from a design science perspective

the *solution-validation* trajectory. In the problem construction phase, we identified a limitation of existing approaches to declarative program analysis: they depend on the internal representation of programs. In the design phase, we chose to combine Datalog and syntactic patterns to solve the problem. We produced a solution, METADL and an early variant of JAVADL, which we applied to a concrete problem: defining detectors for static code checkers.

We evaluated the early variant of JAVADL on realistic workloads and established the need of improved performance and scaling to large projects. These became, in turn, new problems. Instead of deriving from limitations of previous work, we empirically identified these problems, thus their development followed the *problem-solution* trajectory. We have iterated this trajectory twice, once for developing the incremental evaluation scheme demonstrated by JAVADL and once for the on-demand evaluation scheme for CLOG.

Following good research practice, we submitted the system prototypes and the accompanying evaluation frameworks for peer review, through the artifact evaluation process [KV15]. We released the evaluated artifacts under permissive open-source licenses.

BACKGROUND AND RELATED WORK

1 Static Program Analysis

Static program analysis is the branch of computer science concerned with automatic reasoning about the behavior of computer programs without executing them. Given that it is impossible to exhaustively test most programs on all possible inputs, the motivation behind static program analysis is to ensure that programs are free of defects, in the sense that the programs behave according to a given specification.

A program's specification is the set of properties it must satisfy. All programs have an extrinsic specification, which describes how the program should interact with the environment, via hardware and software interfaces. This specification may be formal, in natural language or just a thought in the mind of the developer. Departing from this extrinsic specification leads to application-specific defects, such as API-misuse, resource leaks or violation of timing constraints in real-time systems.

However, each program must also satisfy an intrinsic specification, stemming from the programming language the program is written in, in the sense that the program should not reach a state from which further execution is either impossible (the program is *stuck*) or is nondeterministic (undefined behavior). Language-specific defects arise when a program does not obey the specification of its language. An example of language-specific defects are null dereferences.

To prevent defects, static analyses have been embedded in compilers and in static code checkers. We refer to the analyses that compute general properties of programs as *(static) (program) analyses*, while we use the terms *(static) checks* or *(bug) detectors* for the analyses that aim at detecting concrete defects. Often, the latter are implemented as clients of general program analyses.

1.1 Families of Program Analyses

From the perspective of their formal definition, the multitude of program analyses used by the static code checkers discussed in this thesis can be cast in two broad classes: type systems and abstract interpretations [CC77].[1] We do not aim to further refine this classification, but rather describe the families of analyses that occur in code checkers from a functionality perspective, based on the information they compute. Thus we identify four families of analyses, common to the code checkers we discuss: type checking, data-flow analysis, control-flow analysis and pointer analysis. We describe these analysis families in the following paragraphs.

Type Checking   Type systems categorize program terms based on the values they may take during evaluation, possibly including the side-effects of evaluating the term. The type of a term can be computed either statically, at compile-time, or dynamically, at run-time. Many compiled languages opt for a mixed strategy: they perform most of type checks at compile-time, while leaving for run-time the cases for which type checking is expensive or would otherwise reduce the modularity of the programming language.[2] Knowing the types of program terms is relevant for code optimization and machine code generation, thus static type checking is typically integrated in compilers, but stand-alone code checking tools also make use of type information.

Data-Flow Analysis   For programs in imperative languages, it is natural to encode the problems of finding the properties of variables and expressions in monotone data flow analysis frameworks [KU76]. In these frameworks, the analysis defines the properties of the variable that are of interest and the effect that each of the constructs in the language has on them. Such properties may be the location where a variable was last assigned (reaching definitions analysis), the location where it was last used (dead code analysis), whether the variable always holds a constant value or not (constant propagation), etc. Conservative results for monotone data flow problems can be computed efficiently using fixpoint algorithms. These analyses, restricted to single procedures, have been integrated into optimizing compilers for the purposes of register allocation, dead code elimination, constant folding, etc.

Static code checkers employ inter-procedural variants of data-flow analyses [RHS95; SRH96] for finding application-specific defects, such as taint analyses, or programming language-specific defects, such as checking for nullness of references.

Control-Flow Analysis   Control-flow analysis aims at establishing the order in which the expressions or statements are evaluated. The control-flow graph (CFG) is a conservative approximation of the control-flow of a procedure, which can be derived

---

[1] The classification is rather superficial, based on whether the formal definition of the analysis uses Gentzen-style deduction rules or semantic brackets. The latter class even subsumes the former, as abstract interpretations can express type systems [Cou97].

[2] An example is the run-time checking of down-casts in object-oriented languages such as C++ and Java.

exclusively from the syntactic structure. The information from the CFG can be further refined by using data-flow facts. In the example in Figure 1 we use a data-flow parity analysis to prove that the variable m in the condition of the while statement is odd, rendering the final print statement unreachable.

	$parity(\mathtt{m})$	$parity(\mathtt{n})$
`n = 2 * n` ;	$\{even, odd\}$	$\{even\}$
`m = n + 1` ;	$\{odd\}$	$\{even\}$
while `(m != 0`) {	$\{odd\}$	$\{even\}$
`m = m - 2` ;	$\{odd\}$	$\{even\}$
`print(m)` ; }	$\{odd\}$	$\{even\}$
`print("done")` ;	$\{odd\}$	$\{even\}$
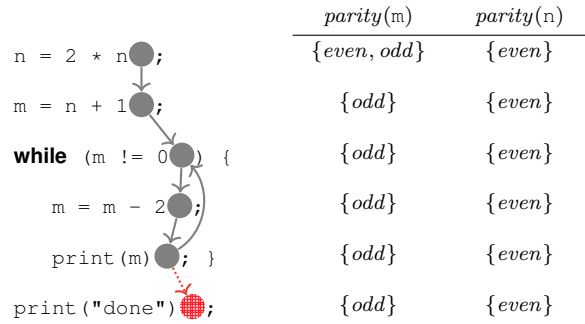
Figure 1: Simple program and its CFG. Using parity information, the control-flow analysis can refine the CFG by removing unreachable nodes and edges (marked in red).

The call graph represents the flow of control between the procedures of the program. For languages without indirect calls (e.g. the OpenGL Shading Language), the call graph is an accurate representation of the inter-procedural control-flow. However, for most other languages, which allow indirect calls (function pointers) or dynamic dispatch, the precision of the call graph may be improved by accounting for data-flow, types and pointer aliasing information.

Pointer Analysis   Programs written in object-oriented languages such as Java and C# hold most of their state in heap-allocated objects. Consequently, data-flow analyses need to track the flow of values through memory locations and a single memory location may be referenced by different program variables or fields of an object. The analyses that map variables or fields to the (abstract) memory locations they may (or must) refer to are collectively named pointer analyses.

Precise pointer analyses depend on tracking flow of pointer values between functions, thus on control-flow and data-flow analyses. In turn, to resolve the call targets of indirect function calls or dynamically-dispatched method calls and thus compute a precise call graph, pointer information is required.

Common Themes   All of the above analyses transform the program from a concrete domain to an abstract domain, where the changes to the program state by an expression or statement are modeled by a transfer function between abstract domain elements. When the analyses need to merge contradictory information from two paths,

they choose a conservative approximation which is closest to the values being merged, which makes a lattice structure well-suited for the values in the abstract domain. Iterating the application of transfer functions and the merging operations until a fixpoint is reached results in solution to the analysis; if the transfer functions are distributive over the lattice join operation, then the solution computed by fixpoint iteration is also the most precise [KU76]. Thus, computation of fixpoints is also a central element of these analyses.

1.2 Limits of Program Analysis

Fundamental results [Ric53], stemming from the undecidability of the halting problem [Tur36], prevent the construction of a program analysis that reliably answers a non-trivial semantic question for all possible programs.[3] Hence, program analysis must derive approximations of all possible behaviours of a program and to decide whether the approximation satisfies a given property.

Since program analyses must approximate the behavior of a program, properties that hold for the approximation may not necessarily hold for the actual program. A *sound* analysis proves a given property only for programs that actually satisfy the given property, while a *complete* analysis is able to prove a given property for all programs that satisfy the property. Unfortunately, no useful analysis is both sound and complete.

1.3 Declarative Static Program Analysis

Program analyses are traditionally defined using a formal description, which is accompanied by proofs of their properties, such as soundness. Therefore, it is appealing to use declarative programming languages to encode these formal definitions, such that they stay close to their formal description rather than deal with low-level details, such as the worklists or the data structures used for storing the partial results of an analysis. Using declarative languages for specifying analyses leads to a decoupling between the analysis specification and the supporting data structures and evaluation strategy. Three strands of declarative approaches to program analysis are close to this thesis: attribute grammars, code property graphs and logic programming.

Attribute Grammars Attribute grammars [Knu68] are a formalism for connecting the semantics of a programming language to its syntax definition. Attribute grammar associate to each terminal or non-terminal in the grammar a set of attributes. These attributes are defined in terms of equations which use attributes of either descendant nodes or ancestor nodes. [4]

---

[3]Assume that there exists an analysis that is able to compute, for all C programs, whether it triggers a segmentation fault. Then, to decide the halting problem about any program with entry point `q(void)`, we need to ask the analysis whether the program `p(void){ q(); *(void*)0; }` triggers a segmentation fault. This means at least that it is impossible to build a null pointer analysis for C that detects all null pointer dereferences and only them.

[4]For a concrete discussion on attribute grammars, see Section 4.

Relevant extensions for program analysis are the introduction of circular attribute grammars [Jon90], which allow for computing a fixpoint for the attribute values, reference attribute grammars [Hed00], which significantly increase the expressivity of the attribute grammar formalism by allowing attribute values to be references to AST nodes, and circular reference attribute grammars [MH07], which combines the benefits of both circular and reference grammars. Attribute grammars are materialized in meta-compiler systems, such as JastAdd [HM03], Silver [VW+10], Kiama [SKV13] and RACR [Bür15].

Circular reference attribute grammars have been used to develop static code checkers, as demonstrated by IntraJ [Rio+21], a framework for intra-procedural analyses for Java. Improvements in the underlying evaluation strategy for circular attributes [Rio+24], lead to reduced running times for IntraJ, without modifying the analyses, which witnesses the benefits of using a declarative approach to analysis specification. In our JAVADL system, we extract semantic information from ExtendJ [EH07], a compiler built using attribute grammars.

Code Property Graphs   A code property graph [Yam+14], as first introduced in the Joern code analysis platform [Joe], is a program representation in which the control-flow graph and the program dependence graph [FOW87] are overlaid on the AST, resulting in a multi-graph with the same nodes as the AST, but with three sorts of edges. In a similar fashion, in the CLOG system, we overlay control-flow edges over the AST.

In addition, each node of a code property graph contains a set of key-value pairs, which are specific to its syntactic category. Code property graphs are stored as graph databases and thus they can be queried using graph query languages. Unlike attribute grammars, which uniformly encode all the properties of a node as attributes, code property graphs make a clear distinction between properties that are intrinsic to an AST node, encoded as key-value pairs, and computed properties, which they express as graph database queries. Program analyses are expressed as graph traversals. For example, a taint analysis is encoded as a traversal from sinks towards sources.

Logic Programming   Logic programming languages such as Prolog are a convenient choice for defining recursive predicates and queries. This makes them a suitable choice for exploring the deeply recursive structures used to represent programs, such as the AST and the CFG. Often, these languages also allow for a straight-forward translation of the formal analysis specification. Consequently, the use of logic programming for program analysis is well established [Rep95].

The goal oriented, top-down, evaluation model of Prolog limited the performance of the analyses and did not allow for scaling beyond small programs. The adoption of tabled Prolog [SW12] enabled scalability [BF07], by reusing goals and their computed results between evaluations. This caching of intermediate results is an intrinsic feature of the evaluation model of Datalog, a restricted version of Prolog. In addition to being a syntactic subset of Prolog, Datalog further excludes negation through recursion and

complex terms as predicate arguments. These restrictions allow for efficient bottom-up evaluation of Datalog programs, where intermediate results are cached and reused.

The efficient evaluation model of Datalog enabled its adoption for expressing a wide range of program analysis: context-sensitive pointer analyses [BS09], interprocedural data-flow analyses [MYL16], or general program analyses [DM+07]. We follow these systems and also build our code checker specification languages on top of Datalog.

2 Datalog

Datalog [CGT89] is a declarative logic programming language, originally aimed at expressing recursive queries over databases. It emerged in the late 1970s as a restriction of Prolog [AHV95]. While initially a database language, Datalog was adopted by the program analysis community to describe analyses with complex interdependencies. Being a declarative language, it separates the analysis description from its evaluation strategy.

2.1 The Datalog Language

A Datalog program contains a set of rules that describe how to derive new relations from existing relations. The rules have the shape of Horn clauses:

$$\forall \overline{v_0 v_1 \ldots v_n} \left(P_1 \left(\overline{v_1} \right) \wedge \ldots \wedge P_n \left(\overline{v_n} \right) \Rightarrow P_0 \left(\overline{v_0} \right) \right)$$

where $\overline{v_i}$ represents a sequence of variables and constants. These rules must be *range-restricted*, in the sense that any variable occurring in the sequence $\overline{v_0}$, must occur at least once in the sequence $\overline{v_1 \ldots v_n}$.

In the textual form of a Datalog program, we denote these rules as:

$$P_0(\overline{v_0}) \text{ :- } P_1(\overline{v_1}), \ldots, P_n(\overline{v_n}).$$

P_i are named *predicates* or *relations* and we will use these two names interchangeably, while the inhabitants of these relations are called *tuples*. Any $P_i(\overline{v_i})$ is an *atom*. We call $P_0(\overline{v_0})$ the *head* of the rule, and the conjunction $P_1(\overline{v_1}), \ldots, P_n(\overline{v_n})$, the *body* of the rule. A *fact* is a rule with an empty body, and it follows from the range-restriction requirement that its head must contain only constants.

A relation is *intensional* if it appears in the head of at least one rule. Conversely, an *extensional* relation appears only in the rules' body. The set of all intensional relations is called the intensional database (IDB), while the set of the extensional relations is called the extensional database (EDB). An instance of the EDB acts as the program's input, while some, but not necessarily all, relations from the IDB are the program's output.

To illustrate, consider the relation Author(*name*, *paper*) which contains pairs of authors and their papers. To compute whether two persons have

```
1 COAUTHOR(name1, name2) :- AUTHOR(name1, paper), AUTHOR(name2, paper).
2
3 AUTHORCHAIN(name1, name2) :- COAUTHOR(name1, name2).
4 AUTHORCHAIN(name1, name2) :- AUTHORCHAIN(name1, name),
5                                      COAUTHOR(name, name2).
6
7 FINITEERDŐSNUMBER(name) :- AUTHORCHAIN(name, "Erdős").
```

Figure 2: Datalog program for computing author relations

co-authored a paper, we introduce the rule in Figure 2, line 1, which defines the COAUTHOR relation. Two authors are in the COAUTHOR relation if there exists a paper for which both are authors. For example, if the AUTHOR relation contains the tuples ⟨"Djikstra","On-the-Fly Garbage Collection"⟩ and ⟨"Lamport","On-the-Fly Garbage Collection"⟩, then the program in Figure 2 computes the tuples ⟨"Djikstra","Lamport"⟩, ⟨"Lamport","Djikstra"⟩, ⟨"Djikstra","Djikstra"⟩ and ⟨"Lamport","Lamport"⟩ for the relation COAUTHOR (so this relation is reflexive and symmetric by definition).

Assume now that we want to compute whether there exists a co-authorship chain between two authors. To achieve this, we introduce in line 3 and line 4 the AUTHORCHAIN relation, defined as the transitive closure of the COAUTHOR relation. Finally, we may ask the question of whether an author has a finite Erdős number. To answer this question, we introduce another relation, FINITEERDŐSNUMBER, defined in line 7 as containing all the authors for which there exists a co-authorship chain between them and the mathematician Paul Erdős.

Semantics We can view the Datalog program as a set of constraints that the relations referred by the program must satisfy. Thus, the result of evaluating the Datalog program is the smallest relations that satisfy these constraints, also called the *minimal model* of the program.

While the *minimal model* approach gives a precise definition of what the result is, it is not constructive and it does not hint at how to compute such a result. We thus resort to an alternative, but equivalent semantics: fixpoint.

The intuition behind the fixpoint semantics of a Datalog program is that the evaluation proceeds with the tuples from the EDB, and then repeatedly applies the program rules until no new tuples can be derived. Since any tuple contains either constants appearing in the EDB or in the program text, the number of tuples that can be derived is bounded, thus the evaluation of a Datalog program always terminates.

The minimal model semantics and the fixpoint semantics of Datalog, as well as proofs of their equivalence are well covered in the literature, thus we refer the reader to these sources [AHV95].

$$\text{AUTHOR} \rightarrow \text{COAUTHOR} \rightarrow \text{AUTHORCHAIN} \rightarrow \text{FINITEERDŐSNUMBER}$$

Stratum 0	Stratum 1	Stratum 2	Stratum 3
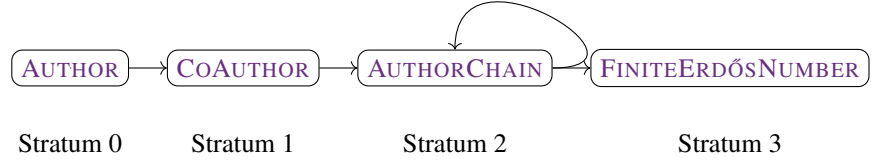
Figure 3: Predicate dependency graph

Evaluation strategies Datalog programs are usually evaluated *bottom-up*, which means that the evaluation derives the full relations of the program, starting from a given EDB. This strategy is employed by state of the art Datalog engines such as Souf-flé [Sch+16] or LogicBlox [Are+15b].

An alternative strategy is *top-down* evaluation which can answer whether a given tuple exists in the result of the program. This is the same strategy used for evaluating Prolog programs.

Stratification For the program in Figure 2, deriving new tuples to the relation AU-THORCHAIN, does not influence the contents of the COAUTHOR relations. However, deriving new tuples for the COAUTHOR relation, produces new tuples in the AUTHOR-CHAIN relation. We capture this dependency relation between predicates by introduc-ing a predicate dependency graph, as in Figure 3. We observe that the program contains four strongly connected components and that we can split the original program into 4 sub-programs, each corresponding to a strongly connected component, and then use the output of a sub-program as the EDB for the sub-programs that depend on it.

The process of splitting the original program into sub-programs that match the strongly components of the predicate dependency graph is called *stratification*. The resulting sub-programs are named *strata* (sg. *stratum*). Since all the cycles in the predicate dependency graph are contained by their respective stratum, the dependencies between strata form a directed acyclic graph (DAG). An evaluator can thus traverse this DAG in topological order and evaluate each stratum once the strata it depends on have been fully evaluated.

Negation Assume that in the earlier example in Figure 2, we would like to find unre-lated authors. The immediate solution is to define a predicate UNRELATEDAUTHORS as

UNRELATEDAUTHORS(`name1`, `name2`) `:-` ¬AUTHORCHAIN(`name1`, `name2`).

But the variables `name1` and `name2` are universally quantified, so the UNRELATEDAU-THORS relation must contain all the objects in the universe, except the related author

names. This is unlikely to be the intention of the programmer, so we require that variables used in a negated atom appear in positive atoms in the body of the same clause. Thus, the query for unrelated authors becomes:

```
UNRELATEDAUTHORS(name1, name2) :- AUTHOR(name1, paper1),
                                   AUTHOR(name2, paper2),
                                   ¬AUTHORCHAIN(name1, name2).
```

To correspond to our intuitions about proofs, we want to avoid rules such as

```
P(0) :- ¬P(0).
```

where we derive P(0) assuming ¬P(0). We achieve this by introducing a syntactic restriction. If two predicates P and Q are from the same stratum, then P is not allowed to appear negated in the body of any definition of Q. That is, we disallow rules of the shape

```
P(...) :- ... , ¬Q(...), ... .
```

This restriction is named *stratified negation* and is one of the approaches to give sensible semantics to Datalog programs in the presence of negation. All Datalog engines used in this work implement stratified negation.

Arithmetic A common extension to Datalog is the introduction of arithmetic operations and arithmetic constraints. Arithmetic operations allow the construction of new values, not existing in the EDB, which invalidates the termination property of plain Datalog programs. The program in Figure 4 uses this extension to compute the distance between two points on the number line.

```
1 DISTANCE(x, y, d) :- POINT(x), POINT(y), x > y, d = x - y.
2 DISTANCE(x, y, d) :- POINT(x), POINT(y), x <= y, d = y - x.
```

Figure 4: Datalog program for computing the distance between two points

Order Going back to the authorship example in Figure 2, we may aim to compute the distance between two authors. The program in Figure 5 computes the set of possible distances between two authors and the Erdős number of authors. Because the COAUTHOR relation is symmetric, the graph induced by it contains cycles if it contains more than two authors. Then, the rule in line 4 produces AUTHORDISTANCE tuples with ever increasing distance, leading to non-termination.

```
1 AUTHORDISTANCE(name1, name2, 1) :- COAUTHOR(name1, name2), name1 != name2.
2 AUTHORDISTANCE(name1, name2, d) :- AUTHORDISTANCE(name1, name3, d1),
3                                     AUTHORDISTANCE(name3, name2, d2),
4                                     d = d1 + d2.
5 ERDŐSNUMBER(name, d) :- AUTHORDISTANCE(name, "Erdős", d).
```

Figure 5: Program which computes the distance between two authors

The program in Figure 5 has the desired semantics if only the tuples of AUTHORDISTANCE with smallest distance are added to the IDB. This requires that we introduce a

partial ordering of the AUTHORDISTANCE relation,

$$\text{AUTHORDISTANCE}(m, n, d_1) \sqsubseteq \text{AUTHORDISTANCE}(m, n, d_2) \text{ iff } d_1 \leq d_2$$

and only the least tuple with respect to the $\sqsubseteq$ partial order is preserved in the IDB. The introduction of partial ordering between tuples is called *subsumption* [Kö+93].

An alternative approach to subsumption is to extend Datalog with lattice support, thus allowing the introduction of a lattice structure over the tuples in the AUTHORDISTANCE relation. The Flix language [MYL16] implements this approach and it demonstrated capabilities of encoding complex inter-procedural program analysis frameworks such as IFDS [RHS95] and IDE [SRH96].

The Datalog dialect we use in this work does not support lattices, but we consider them an extension of Datalog orthogonal to our approach, which could be used to enhance the expressiveness of our static code checker frameworks.

2.2 Program Analysis Systems Using Datalog

The Datalog language has been employed in describing program analyses both at high-level, on the AST, and at low-level, on the intermediate representation. In the following paragraphs we present a selection of systems that employ Datalog for program analysis in similar ways with our work (CodeQuest, QL), demonstrate the scalability of Datalog analyses (Doop) or explore language extensions and alternative evaluation modes (IncA).

CodeQuest CodeQuest [HVM06] is one of the first systems that employed Datalog for source code queries, arguing that the language provides the right balance between expressivity and an efficient evaluation model. In particular, the authors of CodeQuest identify Datalog's ability to express recursive queries as crucial for exploring the graph structures arising in source code queries, such as type hierarchies and call graphs. Instead of defining its own representation of the analyzed program, the system exposes internal Eclipse [AIS] data structures as Datalog relations.

QL QL [Avg+16] is a declarative object-oriented programming language aimed at implementing static analyses in the CodeQL program analysis system. In contrast to CodeQuest, QL provides a high-level programming language, which includes classes, subtyping and virtual dispatch. The AST of the analyzed program is directly exposed as built-in types (*entity types*) in the QL language.

The CodeQL system provides a mechanism for populating the relations that represent the program (i.e., the entity types) together with a rich library of predicates and static checks for multiple programming languages (C, C++, C#, Java, etc.).

Doop Doop [BS09] is a pointer analysis framework for Java programs. The framework implements a suite of tightly-composed analyses,[5] with configurable object and

---

[5]For a discussion on analysis composition, see Section 3.1.

call-site sensitivities. By specifying the analyses in Datalog, Doop demonstrates significant performance improvements over previous work. The analyzed program is represented as EDB facts, corresponding to the intermediate representation for Java bytecode produced by the Soot program analysis framework [VR+10]. A system implementing the ideas of Doop, but for LLVM-based languages, is CClyzer [BS16]. CClyzer represents as EDB facts only the subset of LLVM IR instructions which are relevant to pointer analyses.

IncA IncA [Sza+18] is a Datalog system with support for lattices and an incremental evaluation model. Although aimed at program analysis, the increments have fact granularity, thus the incremental scheme is independent of the domain. Without accounting for the time to produce the EDB facts, the system is capable of incrementally running inter-procedural lattice-based analyses such as constant propagation and interval analysis in time scales that do not prohibit the integration of such analyses inside an IDE workflow [SEB21].

2.3 The MetaDL Datalog Dialect

The Datalog dialect we are using has evolved through its use in METADL, JAVADL and CLOG. In this section we present its common core, which we refer to as the METADL Datalog dialect. In Figure 6 we present its grammar, which mostly follows established Datalog conventions.

Program		$::=$	$\overline{K}$
Clause	K	$::=$	$R \mid F$
Fact	F	$::=$	$\overline{H}.$
Rule	R	$::=$	$\overline{H}\,{:}{-}\,\overline{B}.$
Head literal	H	$::=$	$P\left(\overline{t}\right)$
Body literal	B	$::=$	$H \mid !H \mid v = e \mid e < e \mid e == e \mid etc.$
Term	t	$::=$	$e \mid \_ \mid k \mid {'}P$
Expression	e	$::=$	$v \mid f\left(\overline{e}\right) \mid e + e \mid etc.$
Variable	v	$\in$	*Variables*
Function	f	$\in$	*Functions*
Predicate symbol	P	$\in$	*Predicates*
Constant	k	$\in$	$\mathbb{Z} \cup Strings \cup \{\texttt{undef}\}$

Figure 6: The syntax of the Datalog language dialect

Names The namespace of predicates and functions is global, while each rule introduces its local namespace for variables. Predicate names start with an uppercase character, while the names of variables and functions start with a lowercase character.

Types The METADL dialect is a statically typed language, without explicit type declarations. We rely instead on monomorphic type inference. A METADL term can have one of four types: INTEGER, STRING, ASTNODE and PREDREF. ASTN-ODE is the type of AST nodes, while PREDREF is the type of predicate references, $'P$. The type of a METADL predicate is a product type $\tau_1 \times \ldots \times \tau_i$, where $\tau_i \in \{$INTEGER, STRING, ASTNODE, PREDREF$\}$. For convenience, the infinite predicates $==$, $! =$ and $=$ are polymorphic, with type $\tau \times \tau$ where τ ranges over all the term types.

Infinite Predicates and Constructors Similar to other Datalog dialects, we introduce a set of *infinite* predicates, which do not restrict the range of variables appearing in them. We use these predicates for comparisons ($e == e$, $e < e$ etc.) and for constructing values ($v = e$).

Our Datalog dialect makes a syntactic distinction between the predicate for equality testing $e_1 == e_2$ and the predicate $v = e$ for assigning a value to a variable. The predicate for equality testing requires that variables occurring in e_1 and e_2 are range-restricted by other literals in the clause and does not itself restrict the range of the variables. The predicate for assignment requires that only the variables occurring in the expression e in its right-hand side are range-restricted.

Operators and Built-in Functions The language provides the usual arithmetic operators and a set of built-in functions, such as `cat`, for concatenating two strings. Particular implementations, such as CLOG, extend this set of built-in functions, with functions aimed at accessing the source location of an AST node, such as `src_file` : ASTNODE $\rightarrow$ STRING, or for querying properties of the control-flow graph, `cfg_entry` : ASTNODE $\rightarrow$ ASTNODE.

I/O Instead of relying on special syntax for input and output, we assign two predicates a special meaning. The predicate

$$\text{EDB}(predicate : \text{PREDREF}, file : \text{STRING}, format : \text{STRING})$$

enumerates the EDB predicates, together with the *file* from which to load the facts and the *format* of that file. The supported formats are `"csv"` and `"sqlite"`, the latter being intended for internal use. Symmetrically, the predicate

$$\text{OUTPUT}(predicate : \text{PREDREF}, file : \text{STRING}, format : \text{STRING})$$

enumerates the output predicates and their destination file and format.

Legacy Syntax Through its evolution, the METADL dialect used a different concrete syntax for some of its terms, compared with the one described in Figure 6. In Figure 7, we enumerate the relevant differences.

	Current Syntax		Legacy Syntax
Negated body literal	$!H$	$\equiv$	$\mathbf{NOT}(H)$
Equality literal	$e_1 == e_2$	$\equiv$	$\text{EQ}(e_1, e_2)$
Comparison literal	$e_1 < e_2$	$\equiv$	$\text{LT}(e_1, e_2)$
Assignment literal	$v = e$	$\equiv$	$\text{BIND}(v, e)$

Figure 7: Equivalent syntax for the METADL dialect

3 Static Code Checkers

Static code checkers are tools that automatically search for *defects* in a program, without executing the program.

The defects reported by static code checkers include:

- execution of code with undefined behavior, such as reads of uninitialized variables or out-of-bounds array accesses;

- execution of code that leads to program termination: null pointer dereferences;

- misuse of APIs: memory leaks, double free of a pointer;

- partial definition of functions: missing cases in a `switch`;

- illegal data flow: data leaks;

- use of code constructs that hurt readability;

- infringement of project-specific coding guidelines.

Depending on their sophistication, static checkers may report defects to different degrees. Some limit themselves at performing a syntactic check, while others rely on sub-analyses such as data flow analysis to detect defects. For example, there are multiple ways to prevent a double call to `free` error in C code:

1. enforce a syntactic rule: every call to `free` is immediately followed by a statement which sets the freed pointer variable to `NULL`.

2. check that the freed pointer variable is not an argument to free on any subsequent statements;

3. check that `free` is not called on any pointer pointing to the same memory as the freed pointer;

All three approaches incur tradeoffs. The first one misses the cases when the freed memory is referred by multiple pointers and requires source code modifications. However, it requires only syntactical information and thus static checking can be fast. The second approach relies on analyses that a compiler performs, such as name analysis

and CFG construction, thus it can have a low runtime cost and detect some instances of the defect, without requiring any changes to the source code. The last approach is the most demanding. Since pointers can themselves be stored on the heap, it requires heap modeling and pointer analysis, analyses which are known to be expensive. Like the second approach, it does not require any source code modification.

As illustrated in the previous example, choosing a static checking approach is a tradeoff between the accuracy of the result and the available time and compute resources. The less resources a checker uses, the more likely it is to be widely used in practice. Resource-intensive checkers are reserved to use-cases where the cost of a defect manifesting itself after deployment exceeds the cost of running the analysis.

3.1   Structure of Static Code Checkers

From an architectural perspective, static code checkers are composed from three main components: a fact extraction component, an analysis component and a detector component. The fact extraction component (or the *fact extractor*), parses the analyzed program, builds an internal representation (an AST or a three address code) and then translates this to the domain of the analysis, filtering out the irrelevant parts.

The analysis component implements the various static analyses relevant for a subset of detectors implemented by the static code checker.

Finally, the detector component contains the set of detectors implemented by checker. These detectors act as client analyses for the analysis component. The detector component is responsible for filtering and interpreting the results of the analyses and producing the reports.

Depending on the detectors supported by the static code checker, the analysis component has varying complexity. To classify the structure of the analysis component, we adopt the categories *tight* and *loose* composition, introduced by Bronevetsky et al. [Bro+13], which we extend with another category, of *detector-specific* analyses.

For some checkers, the analysis is intertwined with the detector (Figure 8) and the properties computed by the analysis are *detector-specific*. These checkers do not build an AST of the analyzed program, but instead recover local syntax, name and type information by directly processing the stream of scanner tokens or bytecode instructions, as in the case of CppCheck [Mar] and SpotBugs [Spo] respectively.

Many static code checkers reuse infrastructure from compilers, thus their analyses mimic the pipelined architecture of compilers. The analyses are thus *loosely composed*, and they are run either independently, or in a sequence, where analyses early in the sequence inform later analyses (Figure 9).

For improved precision, static code checkers may rely on complex analysis setups, where multiple analyses mutually share information to refine their results, thus being *tightly composed* (Figure 10). An example is the combination of call graph construction, pointer analysis and data-flow analysis for object-oriented languages. To achieve this tight composition, the analyses are integrated into a blackboard architecture, which may be explicit, such in the case of the OPAL system [Hel+20], or implicit, such in
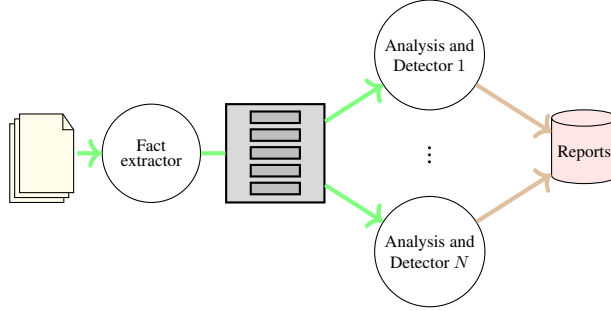
Figure 8: Static code checker with detector-specific analyses
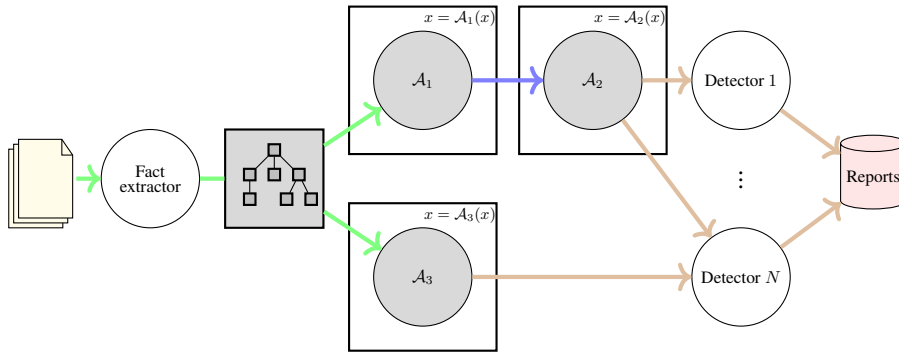
Figure 9: Static code checker with analyses in loose composition

Datalog-based systems (Doop [BS09] for Java pointer analysis, MadMax [Gre+20] for Ethereum smart contracts). Thus we distinguish three classes of analysis components: detector-specific, loosely-composed and tightly-composed.

3.2 Detector Classes

A desirable property of a detector is to have a small number of false-positives. In practice, this imposes a correlation between the classes of defects it can precisely detect and the complexity of the analyses employed by the detector and available in the static code checker framework. From this angle, we distinguish four categories of detectors: style, semantic, local and global.

Style Detectors *Style detectors* are limited to mostly syntactic checks over the analyzed program. Detectors in this category are aimed at enforcing good coding styles, rather than finding defects that can manifest in some execution of the program. Detectors for insufficient use of parentheses are an example of style detectors. Since style

$$
\begin{aligned}
x &= \mathcal{A}_1(x,y,z)\\
y &= \mathcal{A}_2(x,y,z)\\
z &= \mathcal{A}_3(x,y,z)
\end{aligned}
$$
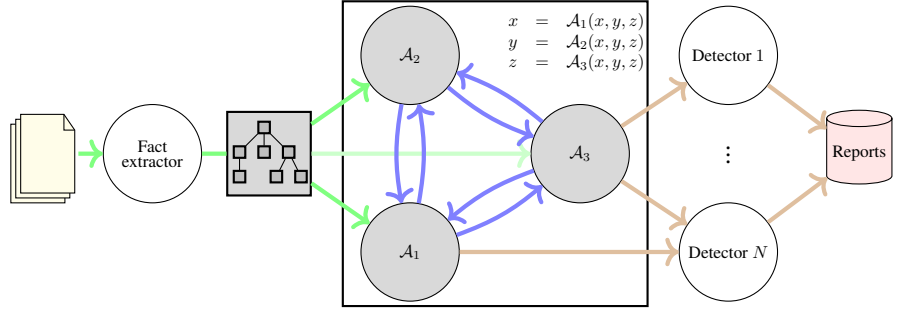
Figure 10: Static code checker with analyses in tight composition

detectors do not need any sophisticated analyses, all static code checkers discussed in this section implement some detectors from this category.

Semantic Detectors   In addition to structural information about the program, *semantic detectors* rely on name and type information. As the style detectors, they are employed at enforcing good coding practice (e.g. a detector for name shadowing), but some are capable of detecting actual defects (a public method exposing a private field).

Local Detectors   *Local detectors* report more complex defects than the ones in the style category, but the reasoning for such reports is confined to a single procedure, thus they rely only on intra-procedural analyses. Systems providing this sort of detectors build at least one representation of the object program, an AST or other intermediate representation such as three-address code.

Beside name and type analysis, local detectors rely on local pointer and data-flow analyses to derive their results. An example of a code checker which supports this class of detectors is Error Prone [Aft+12]. Another system implementing this style of detectors is SpotBugs [HP04] (formerly known as FindBugs), which implements its checks by recovering syntactic and semantic information from Java bytecode instructions.

Global Detectors   Global detectors aim at finding defects that are not localized to a single function, thus they require a set of inter-procedural analyses. Depending on the system, these analyses might be loosely or tightly composed.

3.3   Selected Static Code Checkers

To assess the expressiveness and the performance of our static code checkers, we compare the detector definition styles, the reports and the running times with a selection of relevant static code checkers. We describe these code checkers in the following paragraphs.

SpotBugs SpotBugs [Spo] (previously FindBugs [HP04]) is a static code checker for Java. In contrast to the other checkers discussed in this section, SpotBugs ingests the analyzed program as Java bytecode, and not as source. This makes the definition of checks particularly cumbersome, since the checks must be defined in terms of a stack-based intermediate representation.

The program representation shared by all checks is the Java bytecode. The majority of checks implemented by SpotBugs fall in the category of style checks. SpotBugs also contains a small number of semantic and local detectors (mostly related to multi-threading). The local detectors rely on CFG construction and an intra-procedural data-flow analysis. SpotBugs provides a plug-in mechanism, allowing for extension with custom checks.

Error Prone Another static code checker targeting Java is Error Prone [Aft+12]. Error Prone is not a stand-alone tool, but it is intended to be used as a plugin to the Java compiler, thus having access to the compiler AST, which it uses as the common internal representation.

As for SpotBugs, the majority of Error Prone checks are style and semantic checks. Only the checks related to nullness propagation are local checks, relying on intra-procedural dataflow information provided by the Checker Framework [Che].

One particular feature of Error Prone is that the a check can also define fixes, which can be automatically applied. Besides a plugin mechanism which can be used for adding custom checks, Error Prone also integrates with Refaster [Was13], which allows for definition of patterns and fixes in compilable Java code.

Clang Static Analyzer Clang Static Analyzer [Cla] is a static code checker for C, C++ and Objective-C programs, built on top of Clang compiler infrastructure.

The common representation used by the checks is the Clang AST, as well as a control-flow graph. Clang Static Analyzer includes an inter-procedural data-flow framework, in the style of IFDS [RHS95]. In addition, it uses constraint solving to rule out infeasible CFG paths. This enables the Clang Static Analyzer to implement global checks. However, the analyses are not tightly-composed, but rather compute their results independently. The detectors of Clang Static Analyzer are accessible through the *Clang-Tidy* [Tea] tool, which provides its own style and semantic checks.

CodeQL CodeQL is a language independent program analysis framework. It consists of an object-orient logic programming language, QL [DM+07], fact extractors for multiple languages, which build a relational representation of the analyzed program, and a runtime for evaluating QL programs. CodeQL provides a library of static checks, aimed at multiple programming languages, including C, C++ and Java. The core language of QL is Datalog, thus the CodeQL system is capable of supporting tightly-composed analyses.

3.4   A Mapping of Static Code Checkers

In Figure 11, we present a mapping of the discussed static code checkers, on two co-
ordinates: the detector category they support and the style of composition of their
analyses. For comparison, we also include our code checker frameworks, JAVADL and
CLOG.
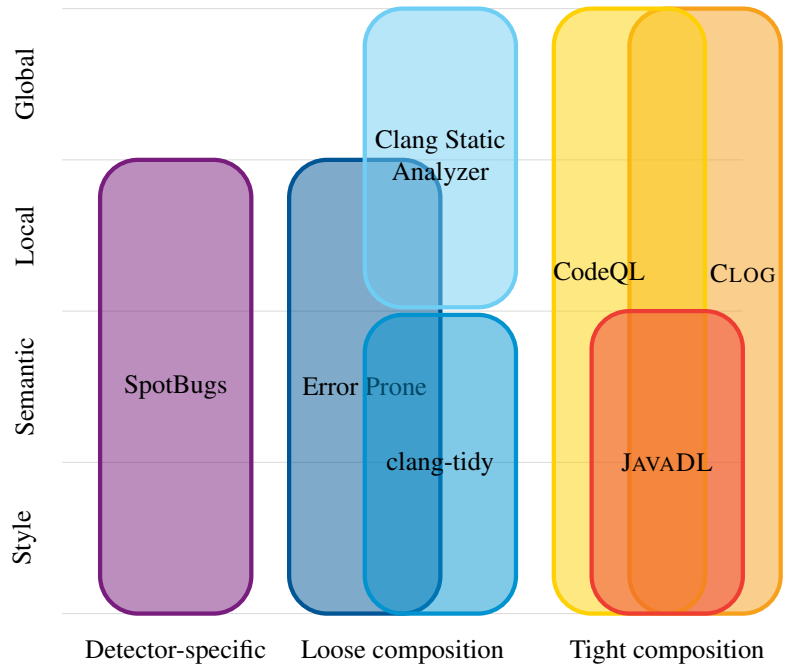
Figure 11: A mapping of static code checkers

4   Syntax Trees

Context-free grammars and representations of programs as abstract syntax trees play
an important role in this thesis. Thus, we briefly illustrate the concepts by adapting
the classical *repmin* example [Bir84] and solve the problem of finding the node of
minimum value from a binary tree. We describe a solution to this problem using the
grammar description dialects used by the JastAdd metacompiler [HM03]. JastAdd
uses a dialect of extended Backus-Naur notation (EBNF) for the specification of the
grammars.

We start by describing the representation of the binary tree. We call this representa-
tion the *abstract* grammar, since it accounts for the logical structure of the tree, rather

for its (textual) materialization. JastAdd uses an object-oriented grammar, which enforces a subtyping relation between its terminals and non-terminals. Thus, on line 1 in Figure 12, we define the `Tree` type, to represent the binary trees. In lines 2 and 3, we declare two of its subtypes: `Fork` to represent a node with two `Tree` children, and `Leaf` to represent the leaves. Terminals and non-terminals on the right-hand side of the production rule have names, for example `Right`, which result in accessor methods, `Fork::getRight()`.

```
1 abstract Tree;
2 Fork : Tree ::= Right:Tree Left:Tree;
3 Leaf : Tree ::= <Value:Integer>;
```

Figure 12: The abstract grammar of the binary tree language

Once the abstract grammar of the binary tree is in place, we define the *concrete* grammar in Figure 13, using the JastAddParser [Jas] EBNF dialect. We also refer to this grammar as the *parsing* grammar to reflect its purpose of transforming a sequence of tokens to an abstract syntax tree (AST). Besides defining the production for each non-terminal and its associated semantic actions, the grammar also defines the type of each non-terminal, which constrains the type produced by the semantic action. In contrast to the abstract grammar, the parsing grammar contains tokens that help avoiding ambiguities, but do not have any semantic value.

```
1 Tree node = leaf
2            | fork
3            ;
4 Leaf leaf = INT {:  return new Leaf(INT) :};
5            ;
6 Fork fork = ( node.left node.right ) {:return new Fork(left, right);:}
7            ;
```

Figure 13: The concrete grammar of the binary tree language. Code between `{:   :}` represents semantic actions.

In Figure 14, we illustrate the concrete syntax tree (or parse tree) for the expression `((4 2) (3 1))`. In general, parsers do not build this tree explicitly, but directly apply the semantic actions and produce the abstract syntax tree. For parsing syntactic patterns, our systems explicitly build the concrete syntax tree, transform it and only then apply the semantic actions.
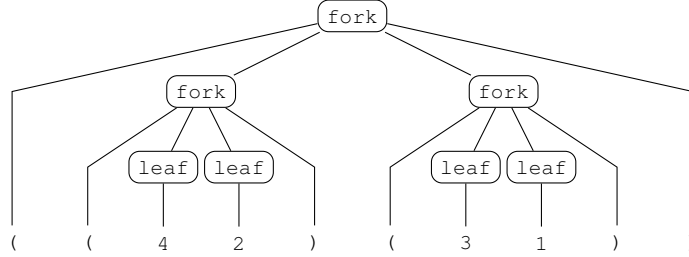
Figure 14: Parsing tree for `((4 2) (3 1))`

To find the node with the minimum value, we introduce equations for the attribute minNode. In Figure 15, line 1, we define the minNode attribute for `Tree` AST nodes. This a synthesized attribute, since it depends only on attributes of the current node and its children, and it is also a reference attribute, since its values are themselves AST nodes. For each of the concrete subtypes of `Tree`, we provide equations to compute the minimum node. Thus, `Leaf` nodes are minimum nodes (line 2), while for `Fork` nodes, we choose the minimum between the left and right subtrees (line 3). We observe that in attribute grammar systems we describe the minNode attribute by using equations and not directly by values, which is a distinguishing feature between attribute grammars and code property graphs.

```
1 syn Tree Tree.minNode();
2 eq Leaf.minNode() = this;
3 eq Fork.minNode() = getLeft().minNode().getValue() <
4                     getRight().minNode().getValue() ?
5                     getLeft().minNode() : getRight().minNode();
```

Figure 15: Equations describing the node with minimum value

In Figure 16, we illustrate the AST for the tree `((4 2) (3 1))` together with the values of the $minNode$ attribute. The values of this attribute are references to nodes in the same AST.
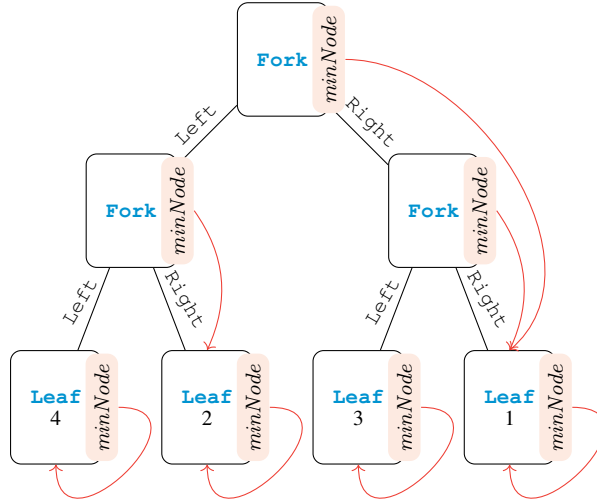
Figure 16: Abstract syntax tree for `((4 2) (3 1))`. Red arrows represent synthesized reference attributes.

5 Syntactic Pattern Matching

In the wide context of program understanding, pattern matching is the process of searching for a given program fragment, called *pattern*, in the source of a program. Some parts of the pattern may be left unspecified, and thus substituted by *wildcards*. Wildcard matches may be subsequently referred to by using *metavariables*.

The search may be lexical, where the matching is performed token by token, without accounting for the grammar of the programming language. For example, function calls in a C-like language can be found using the `grep` utility with the regular expression `"[ _a-zA-Z][ _0-9a-zA-Z]*([ ^)]*)"`.

The above example illustrates the main limitations of the lexical approach: it confounds distinct syntactic categories — the example pattern matches both function calls and function declarations — and it is brittle in regard to white space. Moreover, the pattern does not distinguish between function calls and function-like macros.

These limitations hint to the alternative approach to pattern matching — *syntactic matching* — which accounts for the context-free grammar of the programming language, thus restricting the matches to subtrees obeying this grammar. The syntactic patterns are also written in a superset of this grammar, extended with wildcards. Thus, in an early syntactic pattern matching system, SCRUPLE [Pau92], the pattern `$f_call(#*)` matches all function calls. Since programming languages are typically described using both a concrete (parsing) grammar and an abstract grammar, syntactic

pattern matchers can be built for any of these.

Pattern matching systems deal with at least two languages, so, to distinguish them, we adopt existing terms from the literature [GAM96]. Thus, the *object language* is the language of the program in which the pattern is searched for, while the *pattern language* is the language used to describe the pattern. Some pattern matching systems allow the matched program fragments to be further processed. The *action language* describes this processing.

5.1 Syntactic Patterns in METADL, JAVADL and CLOG

While the pattern grammar varies with the programming language targeted by the static code checker, the syntactic patterns used in this work share common features. In our systems, syntactic patterns always occur in pattern literals, as part of a Datalog program. The pattern literals have the general form

$$[@s]\,[r]\,\langle:\texttt{C}:\rangle$$

where the variables between [ ] are optional. If present, the variable r has type AST-NODE and binds to the root of the subtree matched by the pattern $\langle:\texttt{C}:\rangle$. The s variable, if present, restricts the match to strict sub-trees of the ASTNODE s. If left free, this variable would produce a large number of matches, binding every node from the root of the subtree matched by $\langle:\texttt{C}:\rangle$ to the root of the AST. We did not find any use of this behavior, which is also detrimental to performance, thus the checker compiler require that other literals in the rule bind this variable.

A syntactic pattern $\langle:\texttt{C}:\rangle$ is a code fragment that corresponds to a syntactic category from the object language. To refer to parts of the matched pattern, we introduce metavariables, a set of variables distinct from the tokens of the object language. Depending on the systems, we denote metavariables by $n in CLOG and METADL, or #m in JAVADL. For example, the pattern $\langle:\texttt{while ($cond) $body}:\rangle$, matches all the `while` loops in a program and, for each match, binds `$cond` to the subtree representing the loop condition and `$body` to the subtree representing the loop body.

For matching parts of a list that we do not further refer to, we introduce gaps, denoted by `...` in METADL and `..` in CLOG and JAVADL. Thus, the pattern $\langle:\texttt{while ($cond) {..}}:\rangle$ matches all the `while` statements that have a compound statement as their body. In Figure 17 we illustrate the matches produced by the pattern $\langle:\texttt{while ($cond) {.. $stmt ..}}:\rangle$, where the metavariable `$cond` ranges over all conditions of a `while` statements and, for any binding of `$cond`, the metavariable `$stmt` ranges over all statements in the compound statement corresponding to the loop body.

Semantically, pattern literals behave like predicates ranging over the entire object program. Each match corresponds to a binding of all the metavariables occurring in the syntactic pattern and of the optional root and subterm variables.
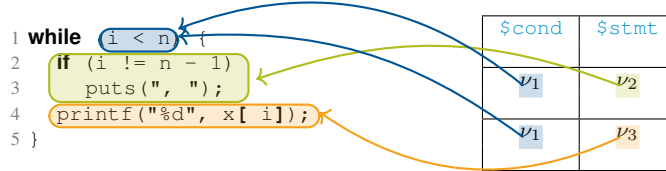
Figure 17: Two matches of the pattern ⟨:while ($cond) {.. $stmt ..}:⟩. ν_i are opaque values of type ASTNODE.

5.2 Related Systems

The idea of using concrete syntax for pattern matching is not new, hence we proceed by describing a selection of systems that use syntactic patterns and discuss the similarities and differences relative to our work.

Early Systems

An early program understanding system using syntactic pattern matching on the concrete syntax is SCRUPLE [Pau92], with C as the object language. The pattern grammar is an extended version of the C grammar, with different wildcards corresponding to the different syntactic categories: statements, types, expressions, function declarations, etc. Unlike the pattern language of CLOG, the SCRUPLE pattern language also contains wildcards to express arbitrary levels of nesting. In CLOG, the pattern language can express only a fixed nesting depth, while arbitrary nesting can be expressed by combining multiple patterns. SCRUPLE performs matching by transforming the pattern to non-deterministic automata and simulating them.

Another early system is TAWK [GAM96], which, unlike SCRUPLE, performs pattern matching on the abstract syntax tree, but it uses the same style of automaton for pattern matching. In addition to the pattern language, TAWK uses C as the action language, which enables the user to express a wide range of analyses.

Stratego

Stratego/XT [Vis04] is a framework for development of program transformation systems based on the term rewriting formalism. While it is not aimed at the development of static code checkers, the framework supports the specification of transformation rules using the concrete syntax of the object language [Vis02].

Combining the syntax of the object language and the syntax of the rewrite specification language requires that the parser of the rewrite specification is able to parse both languages. The parser for the combined language must accept general context-free languages, since the non-trivial subclasses of context-free languages are not closed

under composition. Stratego addresses this by using *scannerless generalized LR parsing*, SGLR [Vis97b]. While the approach used in Stratego is general and robust to changes to any of the combined languages, in CLOG and JAVADL we use a more pragmatic approach: we use an LR parser for parsing the checker specification, while we defer the parsing of the syntactic patterns to a general context-free parser (an Earley parser [Sco08]). We use special tokens (⟨: :⟩) to delimit the two languages.

Stratego/XT uses the syntax definition formalism (SDF) [Vis97a] to define the formal syntax of the language. SDF was successfully used to define disambiguation rules for the combined grammars of Java 5 and AspectJ [BTV06]. In this case, disambiguation is performed by the parser. In our tools, ambiguity arises when extending the grammar of the object languages (C and Java), with metavariables and gaps. However, in our case it is not possible to infer the intent of the user at parse-time, thus we defer disambiguation to the checker developer. We report that a syntactic pattern is ambiguous, and let the developer use predicates to disambiguate between the possible parse trees of a pattern.

Coccinelle

Coccinelle [Pad+08] is a program transformation tool targeting the C and Rust languages. It has a rich pattern language, SmPL [PLM07], for describing *semantic patches*: patterns over the source code and the transformations to apply when these patterns match. A particular strength of Coccinelle is that it performs pattern matching over paths in the CFG rather than on the AST.

Metavariables The Coccinelle SmPL uses metavariables to refer to matched program terms. The metavariables are explicitly declared, together with the syntactic category they can match, for example statements, expressions, identifiers. A special category of metavariables, which does not exist in our systems, are position metavariables that bind to a position in the matched source code. Instead, CLOG relies on the `src_file`, `src_line_start` and `src_col_start` built-in functions to retrieve source information for any AST node bound to a metavariable.

To illustrate, in Figure 18 we present a Coccinelle check[6] which aims at preventing the use of the address of a variable as a branch condition, since such a condition is always true. This check detects the `if` statements in lines 4, 5 and 7 from Figure 19 and produces a report for the user. The first block of the check, in lines 1-5 introduces the metavariables and their syntactic categories, used in the pattern in lines 7 and 8. Finally, it uses inline Python code to output the warnings, between lines 10 and 15.

---

[6]Adapted from `http://coccinellery.org`

```
1 @r@
2 expression x;
3 statement S1,S2;
4 position p;
5 @@
6
7 if@p (&x)
8  S1 else S2
9
10 @script:python@
11 p << r.p;
12 @@
13
14 msg = "ERROR: test of a variable/field address"
15 coccilib.report.print_report(p[0],msg)
```

Figure 18: Coccinelle check for test of variable/field address

```
1 int foo(void) {
2   int x = 5;
3   struct { int f; } s;
4   if (&x) { /* not OK */ }
5   if (&s.f) { /* not OK */ }
6   if (*&x) { /* OK */ } else { /* OK */ }
7   if (&x) { /* not OK */ } else { /* not OK */ }
8 }
```

Figure 19: Unintentional nullness tests of a variable's address

In Figure 20, we present an equivalent CLOG check. We define the IFADDR predicate in lines 1 and 2, with two syntactic patterns, one matching `if` statements with an `else` branch and one, without. The IFADDR relation contains all the AST nodes that represent the expresion whose address is used as condition. In lines 4 and 5, we use the builtin function `src_file`, `src_line_start` and `src_col_start` to collect the source locations in relation IFADDRREPORT — corresponding to the use of position metavariables in Coccinelle. Finally, line 7 marks IFADDRREPORT as an output relation.

```
1 IFADDR($e) :- ⟨:if (&$e) $t else $f:⟩.
2 IFADDR($e) :- ⟨:if (&$e) $t:⟩.
3
4 IFADDRREPORT(f, l, c) :- IFADDR($e), f = src_file($e),
5                          l = src_line_start($e), c = src_col_start($e).
6
7 OUTPUT('IFADDRREPORT, "IfAddrReport.csv", "csv").
```

Figure 20: CLOG check for test of variable/field address

An essential difference between metavariables in Coccinelle and those in CLOG, is that, in Coccinelle, metavariables bind program terms, while in CLOG they bind AST nodes. In CLOG, a metavariable occurring twice in the same pattern results in a matching failure, since all AST nodes have unique identities, even if the subtrees they bind are structurally identical. This makes it cumbersome to test two expressions for structural equality in CLOG, but it is necessary for the correctness of semantic predicates, such as the typing relation, where two structurally identical expressions may have different types in different scopes.

Isomorphisms   The C language allows for multiple idiomatic ways of expressing the same intent, which may differ depending on the adopted coding style. Coccinelle provides a mechanism to declare such isomorphisms, allowing the analysis writer to use only one of the idioms in the patterns. CLOG lacks an explicit mechanism of expressing pattern isomorphisms, but it relies instead on the user to introduce a predicate to refer to all equivalent terms, as on lines 1 and 2 in Figure 20.

The Coccinelle check in Figure 18 relies on an isomorphism defined in the standard library to match both variants of the `if` statement and report on lines 4 and 5 in Figure 19. In Figure 21 we reproduce the relevant isomorphism.

```
1 Statement
2 @ drop_else @
3 expression E;
4 statement S1;
5 pure statement S2;
6 @@
7 if (E) S1 else S2 => if (E) S1
```

Figure 21: Isomorphism definition in Coccinelle

Sequences   Coccinelle patterns match on CFG paths, which makes sequences essential to specifying such patterns. CLOG patterns can also use sequences, but they only match lists at the same level of nesting, for example arguments of a function or a sequence of statements in a block.

Consider the code samples in Figure 22, which contain dereferences of a pointer previously tested for nullness. Figure 23 contains a Coccinelle check[7] that detects the dereference of `q` in line 19, on a path on which it is known to be null, due to the test in line 14. The same check does not report on the dereference of `q` at line 9, because it is dominated by the assignment in line 7.

---

[7]Adapted from `http://coccinellery.org`

```
1 struct S { int f;} s;
2 int g;
3
4 int good(struct S *q) {        13 int bad(struct S *q) {
5   if (q == NULL) {             14   if (q == NULL) {
6     do {                       15     while (g) {
7       q = &s;                  16       q = &s;
8     } while (g);               17     }
9     q->f = 0;                  18     q->f = 0;
10  }                            19   }
11  return 0;                    20   return 0;
12 }                             21 }
```

Figure 22: Dereference of null pointer

The Coccinelle check in Figure 23 introduces a pattern that matches dereferences of
a pointer previously tested for being null. Between lines 9 and 11, the check defines a
sequence, which matches any path on the CFG on which the variable tested for nullness
is not assigned (line 9) before being dereferenced (line 10). The `...` symbol matches
any sequence of statements, but it can be qualified by a `when` clause which excludes a
pattern from the path (line 9).

```
1 @@
2 expression E, E1;
3 identifier f;
4 statement S;
5 @@
6
7 * if (E == NULL)
8 {
9   ...when != E = E1
10 * E->f
11   ...
12 }
13 else S
```

Figure 23: Coccinelle check for null dereferences

In Figure 24, we present an equivalent check, written in CLOG. In lines 4–7, it con-
tains patterns to match the `if` statements of interest. Since CLOG's matching mecha-
nism acts after preprocessing, in lines 1 and 2 we also provide a definition of NULL.
Between lines 10 and 13, the check explicitly declares where the paths of interest begin
— at the entry in the *then* block of the `if` statement — and end — at the CFG suc-
cessor of the if statement (line 10) or at the first assignment to the pointer used in the
condition (line 15). In line 15 we introduce a rule which propagates nullness along the
CFG path. Finally, line 20 contains a rule which checks for dereference statements on
such paths.

```
1 NULL(e) :- e ⟨:0:⟩.
2 NULL(e) :- e ⟨:($t) $f:⟩, NULL($f).
3 // null test if statements
4 IFNULLSTMT(loc, $p, $t) :- loc ⟨:if ($p == $null) $t else $f:⟩,
5                                NULL($null).
6 IFNULLSTMT(loc, $p, $t) :- loc ⟨:if ($p == $null) $t:⟩,
7                                NULL($null).
8 // Beginning and end of the CFG paths inside the if
9 NULLAT(p, b),
10 NULLPATHEND(d, e) :- IFNULLSTMT(loc, p, t),
11                          d = decl(p), d != undef,
12                          b = cfg_entry(t),
13                          CFG_SUCC(loc, e).
14 // End Null paths at assignments
15 NULLPATHEND(p, s) :- s ⟨:$p = $_:⟩, p = decl($p).
16 // Propagate nullness until the end of the path
17 NULLAT(p, s) :- NULLAT(p, t), CFG_SUCC(t, s),
18                    !NULLPATHEND(p, s).
19 // Did the path reach a dereference
20 NULLDEREF(d) :- d ⟨:$p -> $f:⟩,
21                    p = decl($p),
22                    NULLAT(p, d).
```

Figure 24: CLOG check for null dereferences

Comparing the check written in Coccinelle with the one in CLOG, it is apparent that the former expresses the intent of the check, while in the latter the intent is dissimulated between rules about fact propagation along CFG edges. While for the check in the example this is a disadvantage, it also gives the checker writer explicit control, for example by being able to easily specify a flow-insensitive check.

SOUL

The SOUL tool suite for querying programs in symbiosis with Eclipse [DR+11] combines syntactic patterns over the Java language and logic programming in Prolog. This is similar to our approach in JAVADL.

In Figure 25, we illustrate a SOUL pattern[8] intended to find methods that expose the value of private fields. For each match, the pattern binds the metavariables (starting with ?) to the matched AST nodes. Thus, the pattern matches the return statements (lines 4 and 7) the source code in Figure 26.

---

[8]Example from [DR+11]

```
1 if jtClassDeclaration(?classDeclaration) {
2   class ?className {
3     private ?fieldDeclarationType ?fieldName;
4     ?modifierList ?returnType ?methodName(?parameterList) {
5       return ?fieldName;
6     }
7   }
8 }
```

Figure 25: SOUL pattern to find methods that expose the value of a private field

```
1 class Test {
2     private Object field;
3     public Object getField1() {
4         return field;
5     }
6     public Object getField2() {
7         return this.field;
8     }
9 }
```

Figure 26: Java class exposing the value of a private field through getter methods

SOUL supports domain-specific unification, which enables a pattern to match terms that deviate syntactically, but which are semantically equivalent. To achieve this, SOUL makes use of facts computed by an alias analysis to establish that the terms **this**.field and field should unify.

SOUL supports multiple matching strategies for pattern matching. The most stringent strategy requires that the pattern precisely matches the AST — this is also the strategy implemented by JAVADL. A lenient matching strategy requires only that control-flow expressed by the pattern exists in the matched methods, which is similar to the matching strategy adopted by Coccinelle.

In Figure 27 we present an equivalent JAVADL code for detecting return statements that expose private fields. This highlights some differences between the syntactic patterns of SOUL and those of JAVADL. First, in JAVADL, possibly empty lists must be denoted by gaps (..). Otherwise, if two terms appear at consecutive positions in the pattern, they must appear at consecutive position in the matched code. Thus, in this example, the gaps in lines 2, 4 and 8 are essential at specifying that the matched method and the field declaration may appear anywhere in the class. Second, in JAVADL metavariables bind to AST nodes and no unification is performed. Hence, in line 10, we explicitly use the DECL predicate to ensure that #field is bound to the declaration of #ret. Third, the JAVADL pattern matching is exact and JAVADL does not employ alias analysis to unify semantically equivalent expressions, which means that the relation EXPOSEPREDICATE will not contain the returned expression in line 7 in Figure 26, but only the one in line 4. Such alias analysis facts could be easily exposed by a semantic predicate similar to DECL, provided they are made available by

the underlying Java compiler, which is not the case for the current implementation of JAVADL.

```
1  EXPOSEPRIVATE(#ret)  :-  ⟨: class #c {
2                                 ..
3                                 ..  private ..  #fieldType #field;
4                                 ..
5                                 ..  #t #method(..)  {
6                                   return #ret;
7                                 }
8                                 ..
9                              } :⟩,
10                             DECL(#ret, #field).
```

Figure 27: JAVADL rule to find return statements exposing the value of a private field

CONTRIBUTIONS

The goal of this thesis is to enable the construction of fully declarative static code checkers. We incrementally explore the design space and first build a system to analyze Datalog programs in Datalog — METADL. With METADL, we devise methods for representing programs as Datalog relations and to perform syntactic pattern matching in Datalog.

We extend this approach to the Java language and tackle the challenges arising from dealing with a language that is more complex than Datalog and with programs that are much larger than Datalog programs. In the process, we devise ways for automatically generating pattern grammars and dealing with ambiguity arising in syntactic patterns. We integrate with a compiler built using reference attribute grammars, ExtendJ [EH07], which we use as a source of semantic relations that are not computable in Datalog. Since running time is a crucial aspect for static code checkers, we devise an incremental evaluation model for our analyses.

We further explore the design space by targeting another programming language, C, and integrating with a traditional pass-based compiler. Here, we adopt an alternative approach to pattern matching by delegating it to the compiler and computing the pattern matches on-demand.

Along the way, we study qualitative properties of our systems by implementing static checks specific to each of the analyzed languages, as well as quantitative properties, by evaluating the precision and running times of static code checkers implemented using our frameworks.

We materialized the techniques we describe in this work in three static code checker frameworks aimed at different programming languages: METADL for Datalog, JAVADL for Java and CLOG for C. We have used these frameworks to build detectors aimed at their respective languages and demonstrated their effectiveness, both in result quality and in running time.

To enumerate, this thesis brings the following contributions:

1. A relational representation of analyzed programs in Datalog (Paper I);

2. A relational representation of syntactic patterns in Datalog (Paper I);

3. A method for automatically synthesizing pattern grammars from concrete grammars (Paper II);

4. An approach for dealing with ambiguity arising from the use of syntactic patterns (Paper II);

5. An integration between Datalog, syntactic patterns and a RAG-based compiler (Paper II);

6. An alternative integration with a traditional, pass-based, compiler (Paper III);

7. An incremental evaluation scheme for declarative static code checkers (Paper II);

8. Quality and performance evaluation of the code checker frameworks on realistic workloads (Papers II and III).

1 Relational Representation of Programs and Syntactic Patterns

The natural way to build a fully declarative system which combines syntactic pattern matching and Datalog is to devise a method for performing syntactic pattern matching directly in Datalog. To achieve this, we must represent in Datalog the syntactic patterns and the analyzed program.

1.1 Program Representation

To be able to analyze a program in Datalog, we need to represent the program relationally, as EDB relations. In both METADL (Paper I) and JAVADL (Paper II), we use a parser specified using the JastAddParser specification (cf. Section 4) to build an AST of the analyzed program.

For METADL, the parsing grammar is the METADL parsing grammar itself, while for JAVADL we use the pre-existing grammar from ExtendJ [EH07]. In both cases, the parsers build the ASTs of the analyzed program using nodes defined by the abstract grammar using the JastAdd format (cf. Section 4). Collectively, the relations that encode the information about the analyzed program form the *relational representation* of the program. Besides encoding the AST structure, the relational representation includes relations for encoding the source location of the AST nodes. In addition, the JAVADL relational representation encodes semantic information, such as the *type* relation and the *declaration* relation. The latter relation contains pairs of named nodes and their respective definitions.

The METADL Relational Representation   The relational representation used in METADL has the following schema:

- for non-terminals:

$$SyntacticCategory(n : \text{INTEGER}, i : \text{INTEGER}, c : \text{INTEGER})$$

where n represents an unique identifier of the node, i the index of its child and c the unique identifier of its child node at index i.

- for terminals:

$$SyntacticCategory(n : \text{INTEGER}, \ t : \text{STRING})$$

where n is the unique identifier of the node and t the token string.

- for source location:

$$\text{SRC}(n : \text{INTEGER}, \ l : \text{INTEGER})$$

where n is the node identifier and l the start line of the AST node.

This relational representation is close to the one used by .QL [DM+07], with two major differences. First, in METADL, the tuples representing AST nodes encode a reference from a node to its children, in contrast with .QL, which encodes a reference from a node to its parent. We opted for this encoding because it makes the encoding of queries for incomplete AST fragments slightly more compact. Second, the source location of AST nodes is encoded in a separate relation. We chose this encoding, because the source location is rarely accessed by the analyses and it is only necessary when presenting the results of the checker.

The JAVADL Relational Representation While METADL deals with a small number of syntactic categories, this does not hold for the Java grammar. Moreover, different versions of the Java language add new semantic categories, which goes against our previous approach of encoding them in the database schema as predicate names. Thus, in JAVADL we encode the AST structure in a uniform way, in the AST relation:

$$\text{AST}(k : \text{STRING}, \ n : \text{ASTNODE}, \ i : \text{INTEGER}, \ c : \text{ASTNODE}, \ t : \text{STRING})$$

where k represents the syntactic category of the node, n the node, i the index of child c. If the node is a name or a literal, then t represents its token. Since Java programs contain multiple files and many of their AST nodes span multiple lines, we extended the SRC relation accordingly

$$\text{SRC}(n : \text{ASTNODE}, \quad l_s : \text{INTEGER}, \quad c_s : \text{INTEGER}, \\ l_e : \text{INTEGER}, \quad c_e : \text{INTEGER}, \quad f : \text{STRING})$$

where n represents an AST node, l_s and c_s are its start line and column, l_e and c_e are its end line and column and f is the source file.

In addition to the encoding of the AST structure, the relational representation used JAVADL also contains semantic information. This is computed by the ExtendJ compiler and encoded into the

$$\text{ATTR}(n : \text{ASTNODE}, a : \text{STRING}, m : \text{ASTNODE})$$

relation. Here, n represents a node, a indicates what semantic property is encoded and m represents the value of that property a for node n. The possible values for a are `type` or `decl`, so the only semantic relations encoded in the relational representation are type and name information. There is a good reason for including these two relations in the relational representation of the program: in Java, name and type analysis are mutually dependent and type analysis is undecidable [Gri17] so there is no hope for computing them in Datalog.

1.2 Syntactic Patterns

Detectors implemented by static code checkers rely on identifying specific constructs in the source code and then testing their properties.The systems we introduce in this thesis use syntactic patterns to identify constructs of interest in the analyzed program. In our static code checker systems, we write syntactic patterns using a concrete grammar that is a superset of the grammar of the analyzed language (Datalog, C or Java), but the pattern matching is performed on the abstract grammar. The patterns generally match over the entire program, but they may be explicitly restricted, such that the match is performed on a given node or on a subtree of the AST rooted at a given node.

In Figure 1 we present a detector for expressions that use both `&&` and `||` boolean operators without explicit parentheses. This is a detector commonly implemented by static code checkers, such as Error Prone for Java or clang-tidy for C and C++. In this example, the syntactic pattern ⟨:$l || $r1 && $r2:⟩ matches all the problematic expressions, where the `&&` expression is the right operand of the `||` operator. The *metavariables* $r1 and $r2 bind the operands of the `&&` expression and can be referred in other atoms in the same Datalog rule. In this case, we use the built-in SRC predicate to extract the source range of these operands and to add the suggested locations of the parentheses to the relation MISSINGPARENS. The second rule (on line 4) collects all the locations in the program where the `&&` operator occurs in the left operand of the boolean `||` operator.

```
1 MISSINGPARENS(ls, cs, le, ce, f) :- ⟨:$l || $r1 && $r2:⟩,
2                                      SRC($r1, ls, cs, _, _, f),
3                                      SRC($r2, _, _, le, ce, _).
4 MISSINGPARENS(ls, cs, le, ce, f) :- ⟨:$l1 && $l2 || $r:⟩,
5                                      SRC($l1, ls, cs, _, _, f),
6                                      SRC($l2, _, _, le, ce, _).
```

Figure 1: Missing parentheses detector

In Figure 2 we present a detector, which looks for `printf` calls throughout the program — maybe forgotten after a debugging session — and collects their locations for reporting. Because the actual arguments of the `printf` function are irrelevant to the analysis, we use the *gap* — denoted by `..` — to match and ignore a possibly empty sublist of terms. Along with *metavariables*, *gaps* extend the analyzed language grammar to form the *pattern* grammar of CLOG and JAVADL.

```
1 PRINTF(f, l, c) :- p ⟨:fprintf(stdout, ..):⟩, SRC(p, l, c, _, _, f).
2 PRINTF(f, l, c) :- p ⟨:printf(..):⟩, SRC(p, l, c, _, _, f).
```

Figure 2: Searching for `printf` calls

Grammar of Syntactic Patterns   Formally, the pattern grammar extends the grammar of the analyzed languages as follows. For each syntactic category, represented in the grammar definition by N, we introduce two new non-terminals, N_p and N_l, defined as:

$$
\begin{array}{lll}
N_p & ::= & N \quad | \quad \mathit{MetaVar} \\
N_l & ::= & N_p \quad | \quad \mathit{Gap}
\end{array}
$$

We replace all the right-hand-side occurrences of N by N_p if the occurrence is not in a list. If the occurrence is in a list, then we use N_l, to allow for gaps. We apply this transformation both to the parsing grammar and to the abstract grammar. While the transformation is conceptually straight-forward, the abstract grammar is typed, in the following sense: there is a one-to-one mapping from non-terminals to types which are in a subtyping relationship. Moreover, the non-terminals occurring on the right-hand-side are also typed. For example, the Java abstract grammar we use in JAVADL defines the logical operators as follows:

Binary	◁	Expr	::=	left:Expr	right:Expr
LogicalExpr	◁	Binary			
AndLogicalExpr	◁	LogicalExpr			
OrLogicalExpr	◁	LogicalExpr			

where $N_1 ◁ N_2$ denotes that N_1 is a direct subtype of N_2, corresponding to the subclassing mechanism in Java. The parsing grammar contains the corresponding parsing rules and their semantic actions:

```
Expr cor   ::= cand.e₁                    | e₁
            |  cor.e₁   "||" cand.e₂ | new OrLogicalExpr(e₁, e₂)
Expr cand ::= ...                          |
```

Figure 3: Fragment from the Java concrete grammar used by JAVADL

Thus, the type of the semantic action is constrained by the declared type of the non-terminal, Expr, so when transforming the *cand* rule, we need to introduce a $cand_p$ non-terminal, that has type Expr.

$$
\begin{array}{llll}
\text{Expr } cor_p & ::= & cor.e_1 & e_1 \\
 & | & t & \texttt{new MetaVar}_{Expr}(t) \\
\text{Expr } cand_p & ::= & cand.e_1 & e_1 \\
 & | & t & \texttt{new MetaVar}_{Expr}(t) \\
\text{Expr } cor & ::= & cand_p.e_1 & e_1 \\
 & | & cor_p.e_1 \texttt{ "||" } cand_p.e_2 & \texttt{new OrLogicalExpr}(e_1,\ e_2)
\end{array}
$$

Figure 4: Fragment from the concrete pattern grammar of JAVADL

To match the constraints introduced by having a typed abstract grammar, when deriving the abstract grammar for patterns, we need to conservatively introduce multiple *MetaVar* syntactic categories, one corresponding to each non-terminal. In the particular case of $Expr$, we add the following rules to the definition of the abstract pattern grammar:

$$
\begin{array}{ll}
MetaVar_{Expr} \vartriangleleft \text{Expr} ::= t{:}\text{Terminal} \\
Gap_{Expr} \qquad \vartriangleleft \text{Expr}
\end{array}
$$

To avoid the tedium of manually applying the transformations described above, we have built tooling on top of the JastAdd meta-compiler and the JastAddParser to automatically extend the language grammars to pattern grammars. This has enabled us to integrate the transformation in the build process of JAVADL and CLOG, which avoids the need to manually maintain pattern grammars for the Java and C languages.

Ambiguity of Syntactic Patterns   With the introduction of metavariables, the pattern grammars for C and Java become ambiguous. To deal with ambiguity, we implement an Earley parsing algorithm, described by Scott [Sco08], which accepts any context-free grammar and produces a compact representation of the parse result, a *shared packed parse forest* (SPPF), which shares common subtrees between parse results.

We identify two sources of ambiguity in the pattern grammar: constructs in the original grammar that are ambiguous after the introduction of metavariables and trivial production rules in the original grammar.

An example of constructs in the original grammar that become ambiguous after extending the grammar with metavariables are constructor and method definitions — the pattern ⟨:$m $name(..)   {..}:⟩ matches both. For such cases, we opt for emitting a warning and letting the analysis writer disambiguate by using other literals in the same Datalog clause. In this particular case, the conjunction
⟨:$m $name(..)   {..}:⟩, TYPE($m, _)
matches method definitions, while
⟨:$m $name(..)   {..}:⟩, !TYPE($m, _)
matches constructor definitions.

(a) SPPF   (b) Parse tree
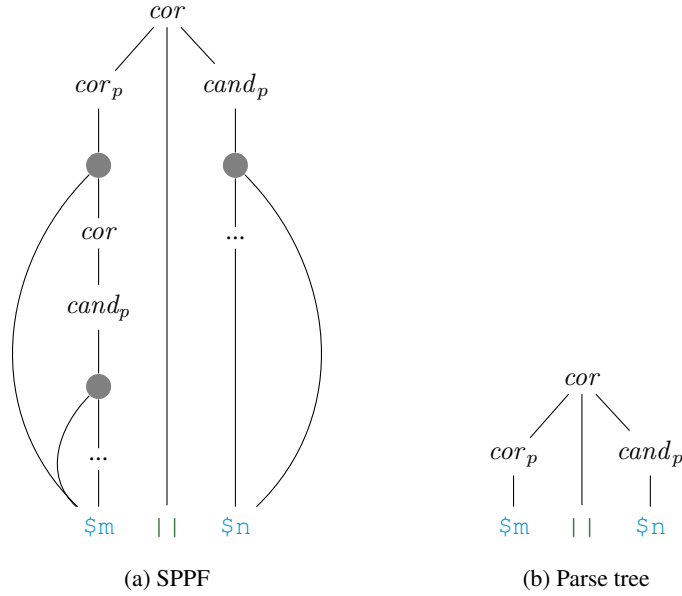
Figure 5: Parsing the pattern ⟨: $m || $n :⟩

The other source of ambiguity is the presence of trivial production rules in the original grammar. Such a trivial production rule is the first rule in Figure 3, which produces multiple parses for the pattern ⟨: $m || $n :⟩, shown in Figure 5a as an SPPF. The circular nodes (●) mark an alternative between two subparses. However, to construct an AST, the parser must apply the semantic actions defined in the original grammar, written in Java and thus opaque to any meta-analysis. Moreover, the abstract grammar of the original language does not define any alternative nodes. Hence, to produce an AST, the parser needs a tree and not an SPPF. The naive option to enumerate all the trees from the SPPF leads to a number of trees exponential in the number of alternative nodes which is prohibitive and we observe that the only relevant parse is the tree shown in Figure 5b. We exploit this observation and add a post-processing step, which compresses all the paths that end in a metavariable and contain only trivial productions, with pass-through semantic actions. Only after this step we proceed with enumerating the parse trees and applying the semantic actions to build the pattern ASTs. We depict the AST for the pattern discussed earlier in Figure 6.

Patterns as Datalog Rules To enable syntactic pattern matching using Datalog, we translate patterns to a set of Datalog clauses as follows. The translation scheme assigns a fresh predicate name to each syntactic pattern. For each of the ASTs produced by parsing the syntactic pattern, it assigns each AST node a fresh variable. The metavariable nodes keep their declared variables. A traversal of the AST produces
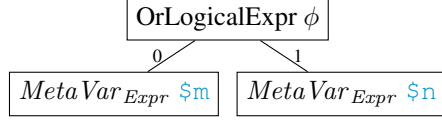
Figure 6: AST for the pattern $\langle\!: \texttt{\$m} \ \texttt{||} \ \texttt{\$n} :\!\rangle$

atoms following the schema of the AST predicate, discussed in section 1.1. Concretely, the rule generated for the pattern AST in Figure 6 is:

```
P_fresh($m, $n) :- AST("OrLogicalExpr", φ, 0, $m, _),
                   AST("OrLogicalExpr", φ, 1, $n, _).
```

Automatic Synthesis of Pattern Grammars   The mechanism of extending a language grammar to a pattern grammar is general. It is fully automated and we have applied it to Datalog, Java and C grammars. The overhead of adapting our approach of combining syntactic pattern matching and Datalog to a new language is small: it only requires a description of the abstract grammar and of the concrete grammar in the JastAdd format. In practice, this means that our framework can be repurposed to generate at least style checkers for other languages. Indeed, achieving more than style checking requires integration with other compilers, as presented in Paper II and Paper III.

2   Integrations

Static code checkers are practical tools, generally aimed at programming languages with a wide adoption. Thus, to validate our approach at building static code checkers we targeted the Java and C languages. By the choice of compilers for these languages we also explored the interactions of a Datalog system with a reference attribute grammar system, ExtendJ [EH07], and with a pass-based compiler, Clang.

2.1   JAVADL: Static Code Checking for Java

To evaluate our approach on realistic workloads, we targeted the Java language and built the JAVADL system, which integrates our Datalog system with fact extraction from the ExtendJ compiler [EH07].

From the ExtendJ compiler we use the parser, the type analysis and the name analysis. We also import the abstract and concrete grammar definitions to derive the pattern grammars in JAVADL.

Type and name analyses are crucial components to any analysis beyond simple syntactic checks. However, in the presence of generics, these analyses are undecidable for Java [Gri17]. On the other hand, the analyzed program is equivalent to an EDB input to the Datalog program, and the Datalog language has a PTIME data complexity, so it is not possible to implement name and type analyses in Datalog. To circumvent

(a) Synthesis of a JAVADL checker

(b) Runtime view of JAVADL
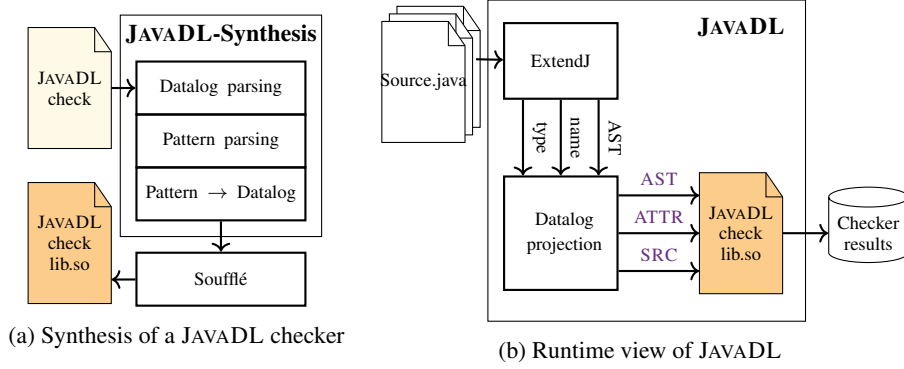
Figure 7: Compile- and run-time overview of JAVADL

this shortcoming, we import the results of these analyses from the ExtendJ compiler as facts, using the schema described in section 1.1.

We have chosen to integrate with ExtendJ because it comes with explicit EBNF descriptions of the concrete parsing grammar and of the abstract grammar, thus enabling our framework to automatically derive the pattern grammars. Other Java compilers, such as javac [Cor] or the Java Parser [Jav] lack these explicit definitions of the concrete and abstract grammars.

Because the relational representation of the program (i.e. the AST and the ATTR relations) for a realistic Java program contain tuples in the order of millions, we integrated a state-of-the-art Datalog engine, Soufflé [Sch+16], to synthesize an executable version of the Datalog program. At checker run-time we use this synthesized executable instead of interpreting the Datalog rules using the internal Datalog engine of JAVADL.

In Figure 7a we depict the process of synthesizing a checker library. The JAVADL compiler parses the checker definition, parses the patterns and represents them as plain Datalog clauses, then translates the internal Datalog dialect to the Souffleé dialect. Finally, it invokes the Soufflé compiler to synthesize executable code. At runtime (Figure 7b), JAVADL proceeds by invoking the ExtendJ compiler on the analyzed files, to build the AST. In the *Datalog projection* phase it populates the relational representation of the AST, then it evaluates the attributes that define type and name information on the nodes for which these attributes are relevant, and then finally it executes the Soufflé synthesized code, which produces the results. This approach enabled us to implement common static detectors that produce comparable results and run in comparable time with other widely deployed static code checkers, as we discuss in section 4.

2.2 CLOG: Static Code Checking for C

To further explore the capabilities of our approach, we have built CLOG, a static code checker system aimed at the C language.

In contrast to Java, the C language does not support modular compilation, but it relies on preprocessor mechanisms to emulate it. The use of include files has one undesired consequence: the AST of a C compilation unit is much larger than the one for a comparable compilation unit for Java, the reason being that it contains AST nodes for all the included files, transitively. This introduces wasteful duplication in the relational representation which is hard to de-duplicate: preprocessor directives may change the configuration of the included files, from compilation unit to compilation unit, thus the AST fragments corresponding to includes cannot be shared between compilation units.

Since importing the AST from a compiler and performing pattern matching in Datalog is not feasible, in CLOG we explore a complementary approach to the one in JAVADL: we delegate pattern matching to the Clang compiler infrastructure and we run only the rest of the analysis in Datalog. This approach comes with its challenges: an abstract grammar built for syntactic pattern matching must be close to the parsing grammar but the abstract grammar used by a compiler such as Clang is tuned towards compiler analyses. We bridge this mismatch in representation by introducing a translation phase between the internal abstract pattern grammar used by CLOG and the Clang abstract grammar used by its AST pattern matching mechanism. Despite the challenges, the choice of integrating with Clang also brings benefits, such as the access to CFG information which facilitates the implementation of data-flow analyses in CLOG The blend between syntactic patterns, logic programming and semantic information (CFG, name and type analyses) enabled us to build checkers able to detect common weaknesses found in C programs.

3 Alternative Evaluation Modes

The declarative specification of our static code checkers allows us to experiment with different evaluation modes, with the aim of reducing running time. Besides the default exhaustive evaluation mode, we introduce two alternative evaluation strategies: incremental and on-demand.

3.1 Incremental Evaluation

Static code checkers are usually integrated in the continuous integration infrastructure of the project or run as pre-commit hooks, so it is important that their results are quickly available. In both of these use-cases, it is only a limited part of the analyzed project that changes between consecutive runs of the code checker, so there is an opportunity to reuse the checker results for parts of the project that are not modified.

We use this observation to demonstrate one strength of our declarative approach to static code checking: the underlying evaluation model of the code checker is not fixed.

We exploit this by introducing an incremental evaluation scheme for JAVADL.

At the core of the incremental scheme lies the observation that some predicates in a static code checker depend only on information from a single compilation unit. This is the case of all predicates that represent syntactic patterns, but also of other predicates representing local analyses. This means that the set of tuples in this local predicates can be partitioned such that each partition is in a one-to-one correspondence with a compilation unit and it needs to be recomputed only when its respective compilation unit changes. To identify such predicates, we devise a meta-analysis over JAVADL programs which we describe in Paper II. This analysis allows the JAVADL compiler to split the original JAVADL program into a *local* part, which is evaluated only on the set of modified compilation units and a *global* part, which combines information from multiple compilation units and is evaluated on every run.

A complication to this incremental scheme is introduced by the use of semantic predicates to represent the typing relation (TYPE) and the name use-declaration relation (DECL), corresponding to the `type` and `decl` attributes in ExtendJ. While these predicates can be cached for each compilation unit, their computation depends on multiple compilation units, thus their provenance needs to be tracked and cached alongside them. To compute provenance, we use the attribute evaluation tracing mechanism of ExtendJ. This enables JAVADL to compute, for each compilation unit, the set of files, that, when changed, require a re-run of the local phase on that respective compilation unit.

3.2 On-Demand Evaluation

The scheme for relational representation of patterns introduced in JAVADL (cf. Section 1.2) has a major performance drawback: it computes the pattern matches over the whole program, independent on their actual use in Datalog clauses. While this drawback is alleviated by the incremental evaluation scheme, because all predicates corresponding to patterns are local and are cached between runs, in CLOG we compute matches on-demand, for cases where the variable representing the root of the pattern is bound by other predicates in the rule.

In the example below, the root variable $r of the pattern $r ⟨:$r1 && $r2:⟩ is bound by the previous pattern in the clause, ⟨:$l || $r:⟩. Thus, a local pattern match at node $r is sufficient.

```
MISSINGPARENS(ls, cs, le, ce, f) :- ⟨:$l || $r:⟩, $r ⟨:$r1 && $r2:⟩,
                                    SRC($r, ls, cs, le, ce, f).
```

In addition to performing on-demand pattern matching, CLOG computes the control-flow graph on-demand, only for functions for which a CFG edge is traversed by the analysis. The adoption of the on-demand model for evaluating pattern matches and semantic relations enables CLOG to utilize only the internal Datalog engine, with a performance comparable to Clang Static Analyzer.

4   Evaluation

Since this work is aimed at building static code checkers, in our evaluation we demonstrate that the tools we have built using declarative techniques are able to perform on par with established static code checkers. Through our experiments, we aim to answer the following over-arching question: *can we express, using our tools, static code checkers that are comparable in precision and running time with the widely adopted static code checkers in use today?*

We answer this question in three stages, for JAVADL as well as for CLOG. First, we select a set of common static bug checks, that we implement using our tools. Second, we report on the precision and recall of these checks on realistic and synthetic workloads, by comparing with results from other tools or from a ground truth, where this is available. Third, we compare the runtime performance of the implemented checks, and discuss alternative evaluation strategies.

4.1   Expressiveness

To show that the language provided by our tools is expressive enough for detectors commonly found in static code checkers, we implemented detectors also present in other tools. Because exhaustively implementing all detectors found in a static checker tool is beyond the scope of this thesis, we systematically selected several of these detectors. In our evaluation, we also demonstrate that beyond being able to encode detectors present in other tools, JAVADL and CLOG provide a *concise* notation for expressing such detectors.

To avoid bias in our selection, we have relied on other sources for selecting the most relevant detectors for the target language. In the case of JAVADL, we relied on a study [HP18] on the report quality of static checkers for Java, which listed the first five warnings reported by a set of widely used static code checkers. For CLOG, we relied on the MITRE annual report listing the top software weaknesses [Mit].

4.2   Analysis Quality

The expressiveness question is not fully answered unless the detectors implemented by JAVADL and CLOG are shown to be useful. We demonstrated this by comparing the results produced by the checks implemented using our tools with results produced by establishes static code checkers: SpotBugs [HP04], Error Prone [Aft+12] and Clang Static Analyzer [Cla].

To select the projects on which to evaluate the tools without bias, we have again resorted to other studies in literature. For JAVADL, we have used the Defects4J [JJE14] suite, extended by Habib and Pradel in the same study we used for selecting the relevant checks [HP18]. For evaluating CLOG, we followed previous work in using a synthetic suite, Juliet [AS17], as well as reusing real-world programs from a program fuzzing benchmark, Magma [HHP22], which contains ground truth information. To enable a

quantitative comparison, we used the familiar metrics of

$$precision = \frac{|W \cap W_T|}{|W|}$$

and

$$recall = \frac{|W \cap W_T|}{|W_T|}$$

where W is the set of reports produced by our tool, while W_T is the set of reports produced by the tool that we compare against. Where ground truth information is available, W_T represents the set of the reports in the ground truth.

4.3 Performance

It is essential that static code checkers produce their results in a timely fashion, since these tools are often integrated in the continuous integration system or even in the build system of a project.

In this thesis, we have evaluated the runtime performance of JAVADL and CLOG and compared to Error Prone, SpotBugs, and Clang Static Analyzer respectively. Because JAVADL supports two evaluation modes — exhaustive and incremental — we have built a test harness that runs JAVADL incrementally on a sequence of 500 consecutive commits from each project. For the incremental running mode of JAVADL we provide a detailed analysis of the running time, as well as a discussion on how the project structure (interdependencies between compilation units) influences the running times.

In our performance evaluation, we demonstrate that JAVADL has a performance comparable to the baseline tools we are comparing against. For a majority of the projects it even exceeds the performance of the baseline tool, in at least one of the evaluation modes. For CLOG, the performance picture is not as clear. On synthetic benchmarks, we demonstrate running times about 2-3 times slower that the baseline, while on realistic loads the running times are one order of magnitude faster than the baseline. We believe that the latter result is due to our analyses being less precise than the ones implemented by the baseline.

5 Summary

We developed the contributions of this thesis in multiple publications and we employed them in multiple systems, thus in Table 1 we present a mapping of these contributions to publications and show which of the code checker systems uses the respective contribution.

Contribution	Paper I	Paper II	Paper III	METADL	JAVADL	CLOG
A relational representation of analyzed programs in Datalog	■			▨		
A relational representation of syntactic patterns in Datalog	■			▨		
A method for automatically synthesizing pattern grammars from concrete grammars		■				▨
An approach for dealing with ambiguity arising from the use of syntactic patterns		■				▨
An integration between Datalog, syntactic patterns and a RAG-based compiler		■				▨
An alternative integration with a traditional, pass-based, compiler			■			▨
An incremental evaluation scheme for declarative static code checkers		■			▨	
Quality and performance evaluation of the code checker frameworks on realistic workloads		■	■		▨	

Table 1: Summary of contributions by publication and tool

CONCLUSIONS

The use of logic programming and syntactic patterns for static code checking has a long history, as we have discussed in Chapter II. However, this thesis is the first to demonstrate that combining syntactic pattern matching and logic programming is a feasible method for building fully declarative static code checkers that are *competitive* in running time and analysis quality with state-of-the-practice static code checkers.

A concern about declarative programming is that the resulting programs are slow. On the contrary, we demonstrate the benefits of the declarative approach by introducing an incremental evaluation scheme, which does not require any modification to the analysis specification. Incrementalization of existing static code checkers would be a practically impossible task, due to their imperative style. In our evaluation, we discuss the performance characteristics of our incremental scheme and we show that it is capable of reducing running times.

A major contribution is the underlying framework we developed for the code checkers described in this thesis. It includes tools for automatic generation of pattern grammars, a parser implementation for general context-free grammars, components for representing programs as Datalog facts and patterns as Datalog rules. Besides semantic information, the system is language-agnostic and can be adapted to other languages, given a formal description of their grammar. We proved the versatility of this framework by integrating with two different styles of compilers, one attribute grammar-based (ExtendJ) and one pass-based (Clang), to provide support for two of the most used programming languages, Java and C.

1 Limitations

Evaluation We evaluated the systems presented in this thesis on a small set of software projects, while the static code checkers we compare with are widely used. This means that these checkers may have been tuned differently in regards to precision and recall. The other static checkers we use in the evaluation of our tools contain hundreds of detectors. Since building a set of detectors for JAVADL or CLOG that is comparable

in size requires a significant effort, we have restricted our comparisons to a small selection of detectors. To avoid bias in our selection, we relied on third party studies and, to ensure a fair comparison, we enabled only the detectors that had equivalents in our systems. Nevertheless, it is unclear what is the performance impact of analysis sharing in other static checkers and what opportunities for analysis sharing arise when implementing more detectors in our static checkers. Thus, extrapolating the performance results of our static checkers to a comprehensive set of detectors is not straight-forward.

Completeness   In the development of our analyses we have observed that the use of syntactic patterns speeds up the initial phase of the development of a bug detector: describing the program fragments that are relevant to analyses, independent of the abstract grammar of the language. While the initial phase is fast, knowing when all the cases with relevant semantics have been covered by patterns is hard. We illustrate this issue in the C language, where the expressions `base + offset`, `base[offset]` and even `offset[base]` are semantically equivalent, but syntactically distinct. This limitation may be mitigated by the use of isomorphism declarations, in the style of Coccinelle.

2   Future Work

Besides addressing the limitations enumerated in the previous paragraphs, we see two possible directions of continuing the work presented in this thesis.

The first direction is to experimentally identify limitations in the expressivity of the JAVADL and CLOG languages. A question to be explored from the user's perspective is whether the current combination of syntactic patterns and Datalog rules is accessible for users without background in logic programming, or whether an alternative surface syntax is desirable.

A possible use of the artifacts associated with this dissertation is to systematically explore the feature-space for declarative code checkers and identify which language features are required for implementing a broad set of detectors and what modularity features are necessary for analysis sharing between these detectors. This exploration is achievable by systematically encoding detectors from other tools in our analysis specification languages.

The second direction is to explore the impact of alternative design decisions on the current systems, such as pattern matching on the CFG rather than on the AST, in the style of Coccinelle, or unification between name declarations and name uses, as in the SOUL system. Besides understanding if it is feasible to implement these alternative designs in our current Datalog-based frameworks, assessing their impact on expressivity and performance are also valid research questions.

On a longer perspective, perhaps the most lucrative use of the declarative analysis specification languages described in this dissertation is to act as a common representation between the user and the static code checker, a representation that is understand-

able and explainable for both, but which is not necessarily produced directly by the user. In light of this, we expect that automatic derivation of syntactic patterns from code examples is the next step in overhauling the interface between the user and the static code checker. In the same vein, we think it is feasible to exploit common structures in many of the analyses — such as information propagation along graph edges — and infer the analysis rules from bug examples.

REFERENCES

[AIS]   Eclipse Foundation AISBL. *Eclipse IDE*. URL:
            https://eclipseide.org Accessed 2025–02–22.

[AHV95]   Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of
            Databases*. Addison-Wesley, 1995.

[Aft+12]   Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and
            Sundaresan Krishnan. "Building Useful Program Analysis Tools Using
            an Extensible Java Compiler". In: *2012 IEEE 12th International Working
            Conference on Source Code Analysis and Manipulation*. 2012.

[Aft+12]   Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and
            Sundaresan Krishnan. "Building useful program analysis tools using an
            extensible Java compiler". In: *2012 IEEE 12th International Working
            Conference on Source Code Analysis and Manipulation*. IEEE. 2012.

[Are+15a]   Molham Aref et al. "Design and Implementation of the LogicBlox
            System". In: *Proceedings of the 2015 ACM SIGMOD International
            Conference on Management of Data*. SIGMOD '15. ACM, 2015.

[Are+15b]   Molham Aref et al. "Design and Implementation of the LogicBlox
            System". In: SIGMOD '15. Association for Computing Machinery,
            2015.

[AS17]   NSA Center for Assured Software. *Juliet C/C++ 1.3 Test Suite*. 2017.
            URL: https://samate.nist.gov/SARD/test-suites/112
            Accessed 2025–01–27.

[Avg+16]   Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and
            Max Schäfer. "QL: Object-oriented Queries on Relational Data". In:
            *30th European Conference on Object-Oriented Programming (ECOOP
            2016)*. Vol. 56. Leibniz International Proceedings in Informatics
            (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[BS16]   George Balatsouras and Yannis Smaragdakis. "Structure-sensitive
            points-to analysis for C and C++". In: *International Static Analysis
            Symposium*. Springer. 2016.

[BF07]   William C. Benton and Charles N. Fischer. "Interactive, scalable,
            declarative program analysis: from prototype to implementation". In:
            *Proceedings of the 9th ACM SIGPLAN International Conference on
            Principles and Practice of Declarative Programming*. PPDP '07.
            Association for Computing Machinery, 2007.

[Bir84]   R. S. Bird. "Using circular programs to eliminate multiple traversals of
            data". en. In: *Acta Informatica* 21.3 (1984).

[BS09]   Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative
            specification of sophisticated points-to analyses". In: *Proceedings of
            OOPSLA '09*. ACM, 2009.

[BTV06]   Martin Bravenboer, Éric Tanter, and Eelco Visser. "Declarative, formal,
            and extensible syntax definition for AspectJ". In: *ACM SIGPLAN
            Notices* 41.10 (2006).

[Bro+13]   G Bronevetsky, M Burke, S Aananthakrishnan, J Zhao, and V Sarkar.
            Compositional dataflow via abstract transition systems. Tech. rep.
            Lawrence Livermore National Lab.(LLNL), Livermore, CA (United
            States), 2013.

[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*.
            2015.

[Mit]   *CWE Top 25 Most Dangerous Software Weaknesses*. URL:
            https://cwe.mitre.org/top25/ Accessed 2025–01–27.

[CGT89]   Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What you always
            wanted to know about Datalog (and never dared to ask)". In: *IEEE
            transactions on knowledge and data engineering* 1.1 (1989).

[CB16]   Maria Christakis and Christian Bird. "What developers want and need
            from program analysis: an empirical study". en. In: *Proceedings of the
            31st IEEE/ACM International Conference on Automated Software
            Engineering*. ACM, 2016.

[Cla]   *Clang Static Analyzer*. URL:
            https://clang-analyzer.llvm.org Accessed 2025–01–27.

[Cor]   Oracle Corporation. *OpenJDK*. URL: https://openjdk.org
            Accessed 2025–01–17.

[Cou97]   Patrick Cousot. "Types as abstract interpretations". In: *Proceedings of the
            24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming
            Languages*. POPL '97. Association for Computing Machinery, 1997.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Association for Computing Machinery, 1977.

[DM+07] Oege De Moor et al. ".QL: Object-oriented queries made easy". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2007.

[DR+11] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. "The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Association for Computing Machinery, 2011.

[DBR] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*. SOAP 2019.

[DR] Alexandru Dura and Christoph Reichenbach. "Clog: A Declarative Language for C Static Code Checkers". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024.

[DR24] Alexandru Dura and Christoph Reichenbach. *Clog: A Declarative Language for C Static Code Checkers*. 2024. URL: https://doi.org/10.5281/zenodo.10525151.

[DRS21a] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection". In: *Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA* (2021).

[DRS21b] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. *JavaDL: Automatically Incrementalizing Java Bug Pattern Detection*. 2021. URL: https://doi.org/10.5281/zenodo.5090141.

[EH07] Torbjörn Ekman and Görel Hedin. "The JastAdd Extensible Java Compiler". In: *SIGPLAN Not.* 42.10 (2007).

[Eng+20] Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. "How software engineering research aligns with design science: a review". In: *Empirical Software Engineering* 25 (2020).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987).

[Gre+20]   Neville Grech et al. "MadMax: analyzing the out-of-gas world of smart contracts". en. In: *Communications of the ACM* 63.10 (2020).

[Gri17]   Radu Grigore. "Java Generics are Turing Complete". In: *ACM SIGPLAN Notices* 52.1 (2017).

[GAM96]   W.G. Griswold, D.C. Atkinson, and C. McCurdy. "Fast, flexible syntactic pattern matching and processing". In: *4th Workshop on Program Comprehension WPC '96*. 1996.

[HP18]   Andrew Habib and Michael Pradel. "How Many of All Bugs Do We Find? A Study of Static Bug Detectors". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018.

[HVM06]   Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: Scalable Source Code Queries with Datalog". In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP'06. Springer-Verlag, 2006.

[HHP22]   Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *SIGMETRICS Perform. Eval. Rev.* 49.1 (2022).

[Hed00]   Görel Hedin. "Reference attributed grammars". In: *Informatica (Slovenia)* 24.3 (2000).

[HM03]   Görel Hedin and Eva Magnusson. "JastAdd: An Aspect-oriented Compiler Construction System". In: *Sci. Comput. Program.* 47.1 (2003).

[Hel+20]   Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. "Modular Collaborative Program Analysis in OPAL". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Association for Computing Machinery, 2020.

[HP04]   David Hovemeyer and William Pugh. "Finding bugs is easy". In: *Acm sigplan notices* 39.12 (2004).

[Jas]   *JastAddParser*. URL: https://jastadd.cs.lth.se/web/tool-support/jastaddparser.php Accessed 2025–03–17.

[Jav]   *JavaParser*. URL: https://javaparser.org Accessed 2025–01–17.

[Joe]   *Joern - The Bug Hunter's Workbench*. URL: https://joern.io Accessed 2025–03–06.

[Joh+13]   Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and
            Robert Bowdidge. "Why don't software developers use static analysis
            tools to find bugs?" In: *Proceedings of the 2013 International
            Conference on Software Engineering*. ICSE '13. IEEE Press, 2013.

[Jon90]   Larry G. Jones. "Efficient evaluation of circular attribute grammars". In:
            ACM Trans. Program. Lang. Syst. 12.3 (1990).

[JJE14]   René Just, Darioush Jalali, and Michael D Ernst. "Defects4J: A database
            of existing faults to enable controlled testing studies for Java programs".
            In: *Proceedings of the 2014 International Symposium on Software
            Testing and Analysis*. 2014.

[KU76]   John B. Kam and Jeffrey D. Ullman. "Global Data Flow Analysis and
            Iterative Algorithms". In: *J. ACM* 23.1 (1976).

[Knu68]   Donald E. Knuth. "Semantics of context-free languages". In: *Theory of
            Computing Systems* 2.2 (1968).

[KV15]   Shriram Krishnamurthi and Jan Vitek. "The real software crisis:
            repeatability as a core value". In: *Commun. ACM* 58.3 (2015).

[Kö+93]   Gerhard Köstler, Werner Kießling, Helmut Thöne, and Ulrich Güntzer.
            "The differential fixpoint operator with subsumption". en. In: *Deductive
            and Object-Oriented Databases*. Springer, 1993.

[MYL16]   Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. "From Datalog to
            Flix: a Declarative Language for Fixed Points on Lattices". In: *ACM
            SIGPLAN Notices* 51.6 (2016).

[MH07]   Eva Magnusson and Görel Hedin. "Circular reference attributed
            grammars — their evaluation and applications". In: *Science of Computer
            Programming* 68.1 (2007).

[Mar]   Daniel Marjamäki. *CppCheck*. URL:
            https://cppcheck.sourceforge.io Accessed 2025–01–09.

[NCP90]   Jay F. Jr. Nunamaker, Minder Chen, and Titus D.M. Purdin. "Systems
            Development in Information Systems Research". In: *Journal of
            Management Information Systems* 7.3 (1990).

[PLM07]   Yoann Padioleau, Julia L. Lawall, and Gilles Muller. "SmPL: A
            Domain-Specific Language for Specifying Collateral Evolutions in
            Linux Device Drivers". en. In: *Electronic Notes in Theoretical Computer
            Science* 166 (2007).

[Pad+08]   Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller.
            "Documenting and automating collateral evolutions in linux device
            drivers". In: *SIGOPS Oper. Syst. Rev.* 42.4 (2008).

[Pau92]   Santanu Paul. "SCRUPLE: a reengineer's tool for source code search". In: *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research - Volume 1*. CASCON '92. IBM Press, 1992.

[Rag+18]   Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. "User-guided program reasoning using Bayesian inference". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.

[Rep95]   Thomas W Reps. "Demand interprocedural program analysis using logic databases". In: *Applications of Logic Databases*. Springer, 1995.

[RHS95]   Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995.

[Ric53]   Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953).

[Rio+24]   Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin, and Christoph Reichenbach. "Efficient demand evaluation of fixed-point attributes using static analysis". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*. 2024.

[Rio+21]   Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. "A Precise Framework for Source-Level Control-Flow Analysis". In: *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*. IEEE Computer Society, 2021.

[Sad+18]   Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from building static analysis tools at Google". In: *Commun. ACM* 61.4 (2018).

[SRH96]   Mooly Sagiv, Thomas Reps, and Susan Horwitz. "Precise interprocedural dataflow analysis with applications to constant propagation". In: *Theoretical Computer Science* 167.1 (1996).

[Sch+16]   Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. "On Fast Large-scale Program Analysis in Datalog". In: *Proceedings of the 25th Int. Conf. on Compiler Construction*. CC 2016. ACM, 2016.

[Sco08]   Elizabeth Scott. "SPPF-style parsing from Earley recognisers". In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008).

[SKV13]   Anthony M Sloane, Lennart CL Kats, and Eelco Visser. "A pure embedding of attribute grammars". In: *Science of Computer Programming* 78.10 (2013).

[Spo]   *SpotBugs*. URL: `https://spotbugs.github.io` Accessed 2025–02–25.

[SW12]   Terrance Swift and David S Warren. "XSB: Extending Prolog with tabled logic programming". In: *Theory and Practice of Logic Programming* 12.1-2 (2012).

[Sza+18] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. "Incrementalizing Lattice-Based Program Analyses in Datalog". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018).

[SEB21]   Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. "Incremental whole-program analysis in Datalog with lattices". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Association for Computing Machinery, 2021.

[Tea]   The Clang Team. *Clang-Tidy*. URL: `https://clang.llvm.org/extra/clang-tidy/index.html` Accessed 2025–03–04.

[Che]   *The Checker Framework*. URL: `https://checkerframework.org/` Accessed 2025–02–25.

[Tur36]   A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: 42 (1936). This is the paper that introduced what is now called the *Universal Turing Machine*.

[VR+10]   Raja Vallée-Rai et al. "Soot: A Java Bytecode Optimization Framework". In: *CASCON First Decade High Impact Papers*. CASCON '10. IBM Corp., 2010.

[VW+10]   Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. "Silver: an Extensible Attribute Grammar System". In: *Science of Computer Programming* 75.1–2 (2010).

[Vis04]   E. Visser. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Lecture Notes in Computer Science* 3016 (2004).

[Vis97a] Eelco Visser. *A family of syntax definition formalisms*. Tech. rep. 1997.

[Vis97b] Eelco Visser. *Scannerless generalized-LR parsing*. Tech. rep. 1997.

[Vis02]   Eelco Visser. "Meta-Programming with Concrete Object Syntax". In: *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. GPCE '02. Springer-Verlag, 2002.

[Was13]   Louis Wasserman. "Scalable, example-based refactorings with refaster".
            In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring
            Tools*. WRT '13. Association for Computing Machinery, 2013.

[Yam+14]   Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck.
            "Modeling and Discovering Vulnerabilities with Code Property Graphs".
            In: *2014 IEEE Symposium on Security and Privacy*. 2014.

[ZSS20]   David Zhao, Pavle Subotić, and Bernhard Scholz. "Debugging
            Large-scale Datalog: A Scalable Provenance Evaluation Strategy". In:
            ACM Transactions on Programming Languages and Systems (TOPLAS)
            42.2 (2020).

INCLUDED PAPERS

METADL: ANALYSING DATALOG IN DATALOG

Abstract

Datalog has emerged as a powerful tool for expressing static program analyses. Program analysis researchers have built nontrivial code bases in Datalog, but tool support for working with Datalog itself has been lacking. In this paper, we introduce METADL, a language extension to Datalog that enables source-level Datalog program analysis within Datalog. We describe several program analyses implemented in METADL and report on initial experiences. Our findings show that the language is effective for real-life Datalog analysis and can simplify working with Datalog source code.

1 Introduction

Declarative programming is a powerful approach for program analysis [HM03], and Datalog is playing a key role in this development [HVM06], especially for points-to analysis [BS09].

Datalog offers concise notation and eliminates the need for manual management of *worklists* [VR+10], a common feature in imperative program analyses. In imperative implementations, when multiple analyses are mutually supportive [Lun+09], each component analysis must interact with other analyses' worklists, which breaks modularity and complicates development and experimentation. Datalog's *semi-naïve evaluation strategy* [Ull89] automates worklist management, freeing developers to focus on

analysis logic.

Today, researchers have built Datalog-based analyses with thousands [Sch+16] or even tens of thousands [BS09] of rules. Understanding and working with such code bases can be challenging, as Datalog has no notion of data hiding or modularity: all information is global by design.

To help developers manage Datalog code bases, we are developing METADL, a program analysis tool for Datalog in Datalog. METADL programs can read other Datalog programs-under-analysis into queryable Datalog relations and use syntactic pattern matching to access these relations concisely. For example, we might compute a relation ARITY(pn, a), relating predicates *pn* to their parameter count (*arity*) a with rules like the following (see Section 3 for the full example):

$$\text{ARITY}(pn, \$i) \text{ :- } [\ldots, \$p(\ldots, \$i : \$x), \ldots \text{ :- } \ldots \; .], \text{ID}(\$p, pn).$$

This rule matches *metavariables* ($\$p$, $\$i$, $\$x$) to code from the program under analysis, for every instance of the syntactic pattern enclosed in square brackets. Here, $\$p$ matches occurrences of Datalog head literals (predicates with arguments) in a program, $\$x$ matches the right-most argument in each matching literal, and $\$i$ is the numeric index of $\$x$ in the argument list to $\$p$. The *gaps* ($\ldots$) describe sequences whose content we ignore. Outside the pattern, the literal ID($\$p, pn$) extracts the name of the predicate bound to $\$p$ into *pn*. Similar queries make it easy to e.g., find all locations in which a predicate occurs on the left-hand side of a rule, identify all predicates that don't contribute to interesting output, or identify inefficient code or refactoring opportunities.

2   Background

METADL is an extension of Datalog, a declarative language that computes *relations*, effectively database tables without duplicate rows (i.e., with set semantics), from other relations. Each relation is bound to a *predicate symbol* that represents the relation in the program, and in the following we will use the two terms interchangeably.

As an example, consider Figure 1, which shows a program that computes the table of all subtypes given input data that describes a program in a Java 1.4-style language. Line 1 loads a relation from the file `direct-superclass.csv` into a relation with predicate symbol SUPERCLASS, and line 2 does the same for IMPLEMENTSINTERFACE.

Line 4 then specifies that *any pair* $\langle \text{t}, p \rangle$ *that is in the relation* SUPERCLASS *must also be in the relation* SUPERTY. Such rules (or *horn clauses*) take the general form

$$P_1(\overline{x_1}) \text{ :- } P_2(\overline{x_2}), \ldots, P_k(\overline{x_k}).$$

where the P_i are predicate symbols and the $\overline{x_j}$ are sequences of variables and constants. Semantically, such rules are right-to-left implications: for all substitutions ρ from variables in $\overline{x_2} \ldots \overline{x_k}$ to constants, if we can show that the *body literals* $P_2(\rho(\overline{x_2}))$, $\ldots$, $P_k(\rho(\overline{x_k}))$ are true, then the *head literal* $P_1(\rho(\overline{x_1}))$ must also be true. To show that a

```
1 EDB("direct-superclass.csv", 'SUPERCLASS).
2 EDB("direct-interface.csv", 'IMPLEMENTSINTERFACE).
3
4 SUPERTY(t, p) :- SUPERCLASS(t, p).
5 SUPERTY(t, p) :- IMPLEMENTSINTERFACE(t, p).
6 SUPERTY(t, a) :- SUPERTY(t, super), SUPERTY(super, a).
7
8 OUTPUT('SUPERTY).
```

Figure 1: Datalog application code example: compute the transitive supertype relation
for a Java-like language.

literal is true, we either look it up in tables loaded from disk (as in lines 1 and 2) or
(recursively) derive it from any of the rules in the program. When $k = 1$, $P_1(\overline{x_1})$ is
always true. In this case, we can omit the ':-' symbol, as in lines 1 and 2.

Lines 4 and 5 therefore copy all tuples $\langle t, p \rangle$ from SUPERCLASS and IMPLE-
MENTSINTERFACE into SUPERTY, computing the union of these two relations. Line 6
then computes the transitive closure of the SUPERTY relation.

Finally, line 8 specifies that the computed SUPERTY relation should be written to
disk once computed.

Like most Datalog systems, METADL adds features for arithmetic, string opera-
tions, multiple head literals, and negation. Negation introduces a semantic compli-
cation, as it allows us to write self-contradictory rules such as $A(x)$:- **NOT**$(A(x))$.
Contradictions can also arise through longer chains of reasoning. We follow existing
tools (and mathematical tradition) in requiring negation to not be recursive, i.e., when-
ever predicate P depends on a negated predicate Q, we must be able to fully compute
Q before computing P. This process is called *stratification* (Section 4.3).

3 METADL

METADL adds a number of language features to simplify program analyses of Dat-
alog programs. Consider the program in Figure 2, which checks that all predicates
in the input program have the same arity and reports all disagreements in the relation
ARITYERROR.

Line 1 illustrates the IMPORT *pseudopredicate*, which loads an external Datalog
program into a single predicate (PROGRAM). Pseudopredicates, which also include
EDB and OUTPUT from Figure 1, follow the syntax of Datalog predicates but have
special semantics (Section 3.2). As a result of Line 1, the predicate PROGRAM repre-
sents a complex relation that encodes the structure of the input program (Section 3.3).

While it is possible to analyse Datalog programs by directly accessing this *repre-
sentative relation*, we provide a more concise syntax for program access in the form
of syntactic patterns. From the programmers' perspective, a pattern describes the syn-
tax that they want to match; our system rewrites this pattern into relational Datalog
constraints.

```
1 IMPORT("input-program.dl", 'PROGRAM).
2
3 analyze('PROGRAM) {
4   ARITY(p_name, a, loc) :-
5     [ ...:- ..., $p(...,$i:$x), .... ],
6     BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
7   ARITY(p_name, a, loc) :-
8     [ ...:- ..., NOT($p(...,$i:$x)), .... ],
9     BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
10  ARITY(p_name, a, loc) :-
11    [ ..., $p(...,$i:$x), ...:- .... ],
12    BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
13  ARITY(p_name, a, loc) :-
14    [ ..., $p(...,$i:$x), .... ],
15    BIND(a, $i+1), ID($p, p_name), SRC($p, loc).
16 }
17 ARITYERROR(p, loc_i) :- ARITY(p, i, loc_i),
18    ARITY(p, j, loc_j), NEQ(i, j).
19 OUTPUT('ARITYERROR).
```

Figure 2: Program to check that the arities of predicates used within a program are consistent.

For example, Line 5 uses the pattern

$$[...:- ..., \texttt{\$p(...,\$i:\$x)}, ...]$$

This pattern will match any positive (non-negated) Datalog literal that occurs on the right-hand side of a Datalog rule. Names preceded by dollar signs are metavariables. If we apply this rule to the code in Figure 1, $p will match the four literals occurring on the right hand side of rules, $x will match their right-most argument, and $i will match the index of the right-most argument in the parameter list (always 1 in that example, since our offsets start at 0 and all relations in Figure 1 are binary). For instance, in Line 6 of Figure 1, the above pattern would match twice, once for each SUPERTY literal on the right hand side, and $x would match the variables super (for SUPERTY(t, super)) and a (for SUPERTY(super, a)).

Returning to the analysis in Figure 2, we now increment $i by one and assign the result to a (BIND(a, $i+1)). Finally, we read the name of the predicate $p into p_name. (using ID($p, p_name)) and its source location into $loc (using SRC($p, loc)).

We provide three more rules that extract arity from the remaining sources of arity information: negated literals (line 7), head literals (line 10; such literals cannot be negated) and head literals in rules that omit the :- symbol (line 14).

We give a full overview of our pattern rules in Section 3.4.

METADL programs can process multiple input files at once (e.g., for code differencing or dependency tracking). We provide **analyze**('P) { ... } blocks (lines 3–16), where P is a representative relation. Within the curly braces of such a block, the

scope of all patterns and related pseudopredicates (e.g., ID) is set to exactly the program(s) represented by P.

3.1 Types

METADL has a simple static type system with type inference (Section 4.4). Each predicate P must have a fixed arity, and each argument must be of exactly one type. We support three types: *Int* for integers, *String* for strings, and *PredRef* for *predicate references* of the form ' P (where P is a predicate).

We use such predicate references for input and output, but they can also communicate information between programs under analysis and METADL analyses (Section 4.2).

We represent the AST nodes of programs under analysis as integers (Section 3.3). Metavariables in patterns thus always bind to an integer, and developers can use pseudopredicates to extract information from these node IDs.

3.2 Pseudopredicates

METADL provides several pseudopredicates. EDB, IMPORT, and OUTPUT, which interface with the harddisk, require constant parameters. Several other pseudopredicates (EQ, NEQ, GT, ...) test for (in)equalities. The special pseudopredicate BIND allows evaluating expressions. For example, BIND(x, y + 2 * z) will compute $y + (2 \cdot z)$ and bind the result to x. The first argument to BIND must be a variable.

The remaining pseudopredicates function like regular Datalog predicates, but only within **analyze** blocks:

- ID(n, name) relates AST nodes n to their names, for nodes with names (i.e., predicates and variables).

- SRC(n, loc) relates AST nodes n and their source locations loc.

- STR(v, s) and INT(v, i) relate arguments to their constant string or integer values.

- REF(v, n) extracts the predicate symbols from predicate references.

- EXPR(expr, i, subexpr) relates expressions expr, which can occur in the BIND pseudopredicate, and their subexpressions subexpr at (zero-based) index i. For instance, the expression 'x + 7' will have subexpression 'x' at index 0 and '7' at index 1.

3.3 Relational Representation of Datalog Programs

Figure 3 captures the in-memory representation of the AST produced by parsing the following Datalog rule:

SUPERTY(t, ancestor) :- SUPERTY(t, super), SUPERTY(super, ancestor).

When importing an AST, we assign every node in the AST a unique identifier and relate these node IDs in predicates whose names reflect the METADL AST (e.g. RULE, LIST, ATOM, VARIABLE, etc.). The relations for terminal nodes (e.g., VARIABLE) relate node IDs to node contents (e.g., the variable name), while relations for nonterminal nodes capture the AST structure. Our encoding scheme for AST nodes uses triples $\langle p, i, c \rangle$, where p is the parent node ID, i is the (zero-based) child index, and c is the child node. With this scheme, the AST in Figure 3 is represented by the tables in Figure 4.
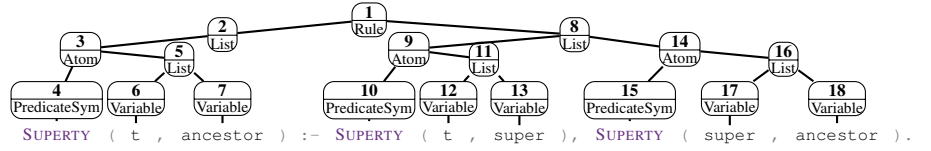
Figure 3: In-memory AST representation of a Datalog rule.

RULE		
1	0	2
1	1	8

ATOM		
3	0	4
3	1	5
9	0	10
9	1	11
14	0	15
14	1	16

LIST		
2	0	3
5	0	6
5	1	7
8	0	9
8	0	14
11	0	12
11	1	13
16	0	17
16	1	18

PREDICATESYM	
10	"Superty"
15	"Superty"
4	"Superty"

VARIABLE	
12	"t"
13	"super"
17	"super"
18	"ancestor"
6	"t"
7	"ancestor"

Figure 4: Datalog table representation of the AST in Figure 3.

This representation uses a nontrivial number of relations. This in turn is at odds with our desire to provide a simple interface to the IMPORT pseudopredicate, so we currently compress all such relations into a single relation and decompress that relation again for processing **analyze** blocks.

3.4 Pattern Matching for Analysing Datalog

We currently support two forms of patterns: one for matching rules with the ':-' symbol, and one for rules without. The language accepted inside the patterns is Datalog extended with metavariables, index metavariables, and gaps.

Metavariables bind to predicate symbols, variables, constants (including predicate references), and expressions, and always start with the symbol '$'. Metavariables are the sole mechanism for directly connecting information from a pattern to literals outside of the pattern, where they behave identically to regular variables.

When a metavariable $x is part of a sequence of literals or parameters, it has an associated index that is accessible via an index metavariable, prefixed by ':' (e.g., $i:$x).

Metavariables allow limited variability in our patterns; for instance, the partial pattern $p($v, $w) will match any positive literal with two arguments of any kind. However, metavariables by themselves are insufficient to match literals with an unknown number of arguments , or rules with an unknown number of literals. Therefore, we also allow *gaps*.

In their simplest form (e.g., $p(...)), gaps specify that we permit any number of elements in an argument list or a list of literals. When gaps are adjacent to metavariables or literal Datalog code, they also relax positioning constraints.

If an element is adjacent to a gap on only one side, then the element's position is fixed relative to its neighbouring element. For example, `[ $p($v, ..., $w).]` matches a single literal and binds $v to the first parameter and $w to the last parameter. If the matched literal is unary, then $v = $w.

If an element has gaps both to its left and to its right, its position is unconstrained in the list that it is a part of. This is a conscious design decision to allow patterns such as `[ ..., P(...), ..., Q(...), .... ]` to match predicates P and Q in any order (even if Q appears before P). If the order is significant, programmers can use index metavariables (e.g., in `[ ...,$i:P(...),..., $j:Q(...),....])` to enforce the order by requiring the inequality LT($i, $j).

Relational Representation of Patterns

We implement pattern matching by rewriting patterns into conjunctions of Datalog literals. The translation scheme is analogous to generating representative relations of imported programs (Section 3.3), with the main differences being that we (a) preserve metavariables throughout the translation and (b) introduce fresh variables for parts of the patterns that we do not wish to match (as part of gaps).

For example, recall this pattern from Figure 2: `[ ...:- ..., $p(...,$i:$x), .... ].` We translate this pattern to the following conjunction:

```
1 RULE(v0, 0, v1), LIST(v1, 0, v4),
2 RULE(v0, 1, v2), LIST(v2, vj, v6), ATOM(v6, 0, $p),
3 ATOM(v6, 1, v3), LIST(v3, $i, $x),
4 BIND(v8, $i + 1 ), NOT(LISTPROJ(v3, v8))
5 # Helper predicate:
6 LISTPROJ(n, i) :- LIST(n, i, ignore).
```

All the variables (excluding metavariables) in lines 1–4 are fresh. Line 1 binds v0 to a rule that has at least one predicate in its head. Line 2 asserts that the same rule v0 has a child at index vj, and that the child must be an atom v6 with predicate symbol $p. Here, vj is an implicit index variable. Line 3 binds $x to a term at position $i in

the list of terms in atom v6. Line 4 ensures that that term has no right sibling (at offset $i + 1$), as our example pattern requires x to be in a rightmost position. LISTPROJ is a helper predicate (line 6).

4 Applications

To examine the utility of our approach, we have built several program analyses in METADL, of which we report on five: arity checking (Figure 2), Cartesian product checking, deprecation checking, stratification, and type inference.

4.1 Checking for Cartesian Products

State-of-the-art Datalog engines like Soufflé [Sch+16] use an eager evaluation strategy. This means that rules such as

```
P(x, y) :- Q(x), R(y).
```

can be wasteful: given j elements in Q and k elements in R, we must compute a table of $j \times k$ elements. If we instead eliminate the above rule and replace $P(x, y)$ on the right-hand side of all remaining rules by $Q(x), R(y)$, we can avoid this cost.

We have written a static checker that detects such projections, reporting any that are consistent across all left-hand side occurrences of a given predicate symbol. Our checker reports both light warnings (one projection) and serious warnings (two or more projections), using a total of 22 rules and 5 syntactic patterns. To illustrate, its final rule is:

```
1 CartesianProjectionWarning(p_name, $i, $j) :-
2     VarProjectedN(p_name, $i, q_name),
3     VarProjectedN(p_name, $j, q2_name),
4     NOT(ProjectionIndicesSharedN(p_name, $i, $j)).
```

with VARPROJECTED determining that in all rules with the predicate named p_name on their left hand side, index $i (resp. $j) a projection from one fixed index of right-hand-side relation q_name (resp. q2_name), further implying that the right-hand-side relation is not filtered in any way. The last line ensures that $i and $j are not only distinct but also always come from different predicates on the right-hand side; this check is slightly more fine-grained than ensuring that q_name and q_name2 are distinct and will also capture e.g. A(x, y) :- B(x), B(y).

We have tested this checker on a self-contained miniature version of DOOP (170 rules) but found no interesting issues. We expect that such static checkers will be most useful during development of new Datalog code.

4.2 Deprecation Checking

Predicate references allow us to implement a light-weight Java-style deprecation checker:

```
1 Depr(p) :- [ Deprecated($p). ], REF($p, p).
2 Warn(p, l) :- [ ...:- ..., $p(...), .... ],
3                 ID($p, p), Depr(p), SRC($p, l).
```

If DEPRECATED(′P). occurs in a program, positive references to P in the body will trigger a warning.

4.3 Stratification

Stratification is part of semi-naïve Datalog evaluation. The purpose of stratification is to (i) ensure that no relation P depends on the negation of P, or the negation of a predicate that depends on P, and (ii) construct an evaluation order over all predicates that will produce the correct result.

Stratification computes a list of *strata*, where each stratum is a set of predicate symbols that depend only on the same stratum and on all previous strata. A stratum contains at least one predicate but may contain more, if the predicates have mutual dependencies. Figure 5 gives a stratification algorithm for standard Datalog, in METADL.

In this figure, we first compute direct dependencies (both positive and negated) between predicates, then the transitive closure of these dependencies, DEP. We then compute which predicates must be evaluated in the same stratum due to circular dependencies, SAMESTRATUM, and the set of predicates that need to be evaluated in the parent stratum of the stratum represented by a predicate, PARENTSTRATUM. Finally, we check that no stratum has a negated dependency on itself and report violations in ERROR.

```
1  analyze('Program) {
2    DIRECTDEP(p_name, q_name) :-
3      [ ...,$p(...),... :- ...,$q(...), ....],
4      ID($p, p_name), ID($q, q_name).
5
6    DIRECTDEPNEG(p_name, q_name),
7    DIRECTDEP(p_name, q_name) :-
8      [ ...,$p(...),... :- ...,NOT($q(...)), ....],
9      ID($p, p_name), ID($q, q_name).
10 }
11 DEP(p_name, q_name) :- DIRECTDEP(p_name, q_name).
12 DEP(p_name, q_name) :- DEP(p_name, rn),
13   DIRECTDEP(rn, q_name).
14
15 SAMESTRATUM(p_name, q_name) :- DEP(p_name, q_name),
16   DEP(q_name, p_name).
17 PARENTSTRATUM(p_name, q_name) :-
18   DIRECTDEP(p_name, q_name),
19   NOT(SAMESTRATUM(p_name, q_name)).
20
21 ERROR(p_name, q_name) :- DIRECTDEPNEG(p_name, q_name),
22   SAMESTRATUM(p_name, q_name).
```

Figure 5: Stratification of Datalog in METADL.

4.4   Type Inference

A METADL predicate is well-typed iff each of its arguments is used consistently with exactly one of the three METADL types (*Int*, *String*, *PredRef*). A METADL program is well-typed iff all predicates that occur in it are well-typed.

METADL does not have special syntax for type declarations, but developers can set types via rules that never trigger:
```
P(0, "", 'P) :- NEQ(0, 0).
```

The above ensures that P is of type $\langle Int, String, PredRef \rangle$.

In Figure 6 we present part of an implementation of type inference. PREDTYPE(p, i, τ) defines a relation between each predicate symbol p, argument index i, and type τ that may occur at this argument index. Lines 2–4 show how we extract type information from string constants; the process is analogous for other constants and other locations in which literals occur. Lines 7–11 show how we propagate type information across body literals; the process for head literals is analogous. Finally, predicate INCOMPLETETYPE(p, i) (lines 19–20) checks if predicate p at parameter index i lacks type information, and predicate INCONSISTENTTYPE (lines 21–22) checks if it has contradictory type information.

Type inference in rules containing the BIND and EQ pseudopredicates and arithmetic expressions can be described similarly, using the EXPR pseudopredicate.

```
1  analyze('Program) {
2    # Infer types from ground terms in facts (strings)
3    PREDTYPE(p_n, $i, "String") :-
4      [...,$p(...,$i:$v,...),... .], ID($p, p_n), STR($v, x).
5      — analogous rules omitted —
6    # Propagate types through variables
7    PREDTYPE(q_name, $j, t) :-
8      [ ...:- ..., $p(...,$i:$v,...), ...,$q(...,$j:$w, ...), .... ],
9      ID($p, p_name), ID($q, q_name),
10     ID($v, v_name), ID($w, w_name),
11     EQ(v_name, w_name), PREDTYPE(p_name, $i, t).
12     — analogous rules omitted —
13   # Compute the term indices for each predicate
14   TERMINDEX(p_name, $i) :-
15     [ ..., $p(...,$i:$v,...), ....], ID($p, p_name).
16     — analogous rules omitted —
17 }
18 ISTYPED(p_name, i) :- PREDTYPE(p_name, i, x).
19 INCOMPLETETYPE(p_name, i) :- TERMINDEX(p_name, i),
20     NOT(ISTYPED(p_name, i)).
21 INCONSISTENTTYPE(p_name, i) :- PREDTYPE(p_name, i, t1),
22     PREDTYPE(p_name, i, t2), NEQ(t1, t2).
```

Figure 6: Highlights of Datalog type inference in METADL.

5   Implementation

Our implementation of METADL is based on the Jast-Add [HM03] extensible compiler generator. It consists of a 'baseline' Datalog implementation and a separate METADL language extension module that relies on JastAdd's rewriting and non-terminal attribute features to transform `analyze` blocks and patterns to plain Datalog.
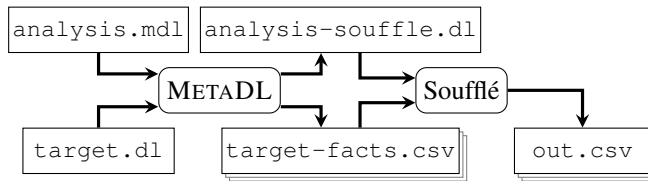
Figure 7: Evaluation strategy for METADL when using Soufflé as external Datalog engine.

Our system has its own Datalog backend, using the *naïve* evaluation strategy [Ull89]. We only use this mechanism for the pseudopredicate IMPORT, then defer to an external Datalog backend (currently the Soufflé system [Sch+16]). We serialise the current rule set and all internal facts (especially our representative relations) into a backend-specific format, run the backend engine (Figure 7), then read back the results.

We have experimented with our analyses on our own code, on a self-contained miniature version of DOOP (437 lines, 170 rules) that we have ported to METADL, and on tests and synthetic benchmarks. For example, our checker from Section 4.1 can analyse the miniature DOOP in around two seconds; growing the target program ten-fold (1700 rules) still allows us to finish in under ten seconds. Despite being in an early stage of development, our tool is practical for analysing medium-sized code bases.

6   Related Work

Program analysis in Datalog has been an area of active research at least since Whaley and Lam [Wha+05], though their system required substantial manual representation tuning. Later systems based on LogicBlox [Are+15; BS09] and Soufflé [Sch+16] scaled more easily. While program analysis in the latter systems has focussed on back-end properties, the CodeQuest system [HVM06] demonstrated the formalism's utility for front-end analyses. Unlike ours, the above systems targeted Java or similar general-purpose languages.

The use of pattern matching has a long tradition in the functional programming community, though we are not aware of support for gaps and indices for program analysis in the same vein as our system. Coccinelle [Law+10] supports ranges (including recursive nesting) for analysing C programs.

Analysing programs of a given language within the same language was a central topic in LISP and is also supported in other languages, primarily with the goal of supporting meta-programming [LJ05; Swa+06]. While our goal is to support similar facilities in METADL in the future, the need for stratification raises hurdles towards offering full metacircularity.

7 Conclusions and Future Work

We have presented METADL, a Datalog extension for loading, analysing, and syntactic pattern-matching over Datalog programs. Our initial results show that the system can concisely express a variety of interesting program analyses and run them in a practically useful time frame. In future work, we plan to extend our pattern matching support to allow metavariables to match more syntactic constructs (including entire rules) and enable Datalog code transformation, using an extended version of our quotation syntax.

Acknowledgements

References

[Are+15]   Molham Aref et al. "Design and Implementation of the LogicBlox System". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015.

[BS09]   Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of OOPSLA '09*. ACM, 2009.

[DBR]   Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*. SOAP 2019.

[HVM06]   Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: Scalable Source Code Queries with Datalog". In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP'06. Springer-Verlag, 2006.

[HM03]   Görel Hedin and Eva Magnusson. "JastAdd: An Aspect-oriented
            Compiler Construction System". In: *Sci. Comput. Program.* 47.1 (2003).

[LJ05]   Ralf Lämmel and Simon Peyton Jones. "Scrap Your Boilerplate with
            Class: Extensible Generic Functions". In: *SIGPLAN Not.* 40.9 (2005).

[Law+10]   Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and
            Gilles Muller. "Finding Error Handling Bugs in OpenSSL Using
            Coccinelle". In: *European Dependable Computing Conference*. 2010.

[Lun+09]   Jonas Lundberg, Tobias Gutzmann, Marcus Edvinsson, and Welf Löwe.
            "Fast and precise points-to analysis". In: *Information and Software
            Technology* 51.10 (2009). Source Code Analysis and Manipulation,
            SCAM 2008.

[Sch+16]   Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann.
            "On Fast Large-scale Program Analysis in Datalog". In: *Proceedings of
            the 25th Int. Conf. on Compiler Construction*. CC 2016. ACM, 2016.

[Swa+06]   Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. "A Monadic
            Approach for Avoiding Code Duplication when Staging Memoized
            Functions". In: *Proceedings of the 2006 ACM SIGPLAN Symposium on
            Partial Evaluation and Semantics-based Program Manipulation*. PEPM
            '06. ACM, 2006.

[Ull89]   Jeffrey D. Ullman. "Bottom-up beats top-down for datalog". In: *Proc. of
            the 8th ACM SIGACT-SIGMOD-SIGART symposium on Principles of
            database systems*. ACM Press, 1989.

[VR+10]   Raja Vallée-Rai et al. "Soot: A Java Bytecode Optimization Framework".
            In: *CASCON First Decade High Impact Papers*. CASCON '10. IBM
            Corp., 2010.

[Wha+05]   John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam.
            "Using Datalog and Binary Decision Diagrams for Program Analysis".
            In: *Proc. of the 3rd Asian Symp. on Prog. Lang. and Systems*. Vol. 3780.
            Lecture Notes in Computer Science. Springer-Verlag, 2005.

JavaDL: Automatically Incrementalizing Java Bug Pattern Detection

Abstract

Static checker frameworks support software developers by automatically discovering bugs that fit general-purpose bug patterns. These frameworks ship with hundreds of detectors for such patterns and allow developers to add custom detectors for their own projects. However, existing frameworks generally encode detectors in imperative specifications, with extensive details of not only *what* to detect but also *how*. These details complicate detector maintenance and evolution, and also interfere with the framework's ability to change how detection is done, for instance, to make the detectors incremental.

In this paper, we present JavaDL, a Datalog-based declarative specification language for bug pattern detection in Java code. JavaDL seamlessly supports both exhaustive and incremental evaluation from the same detector specification. This specification allows developers to describe local detector components via *syntactic pattern matching*, and nonlocal (e.g., interprocedural) reasoning via *Datalog-style logical rules*. We compare our approach against the well-established SpotBugs and Error Prone tools by re-implementing several of their detectors in JavaDL. We find that our implementations are substantially smaller and similarly effective at detecting bugs on the Defects4J benchmark suite, and run with competitive runtime performance. In our experiments, neither incremental nor exhaustive analysis can consistently outperform

the other, which highlights the value of our ability to transparently switch execution modes. We argue that our approach showcases the potential of *clear-box static checker frameworks* that constrain the bug detector specification language to enable the framework to adapt and enhance the detectors.

1   Introduction

Static bug checkers have become an essential tool for many developers. For example, Vassallo et al. [Vas+20] show that many Open Source developers run tools like FindBugs [Aye+08] (now SpotBugs), Checkstyle [Bur+21], PMD [Cop05], SonarQube [Bra+14], and Error Prone [Aft+12] on a daily basis for quality assurance. Moreover, they found that 66% of the Open Source projects that they surveyed mandate that contributors must use static checkers in some form. Checker frameworks typically offer a selection of bug pattern detectors, and then provide extension mechanisms through which developers can add more checks. Examples of such specialized checks include project-specific language idioms and coding standards as well as bug patterns connected to a specific API. Developers who build custom detectors mostly write plugins in an imperative general-purpose language to examine some program representation that is specific to the checker framework, e.g., an abstract syntax tree or a byte code representation.

However, given the properties of imperative specifications, these checker implementations are *black-box detectors* in that the checker framework knows little about the internal wiring of these detectors beyond how to run them. Thus, the framework cannot examine the internals of each detector to e.g. analyze the version of the input language that the checker can handle or the types of bugs that might report. It must instead rely on static and manually curated meta-information or on dynamic (and incomplete) sampling. We argue that black-box bug checker architectures limit innovation at the architectural level, and propose *clear-box bug checker frameworks* as an alternative. Clear-box architectures enable the checker framework to *automatically* enhance detectors, e.g. by deriving confidence scores [Rag+18] to rank bug reports, tracing the detector's chain of reasoning to produce explanations [ZSS20], or incrementalizing computations to speed up execution [Sza+18]. By contrast, a black-box checker framework must ask each detector developer to manually integrate such features, requiring $O(n)$ effort over the number of detectors as opposed to $O(1)$ for clear-box frameworks. Notably, no established black-box checker framework (to the best of our knowledge) supports any combination of the above features.

In this paper, we explore the value of clear-box bug detection in JAVADL, a novel bug checker framework for Java that can process a wide range of user-specified bug detectors and execute them both incrementally and exhaustively. Incremental evaluation can often outperform exhaustive evaluation by re-using intermediate results, but for larger changes, the cost of retaining intermediate results may be greater than the benefits from re-use. JAVADL follows Raghothaman et al. [Rag+18] and Szabó et al.

[Sza+18] in building on Datalog [CGT89], a declarative logic programming language that has become a popular choice for implementing analyses over powerset lattices, including complex points-to analyses [BS16]. Datalog-based analyses have two parts: (1) *fact extraction*, which today is generally a black-box analysis that maps the input program to a set of *input facts* (e.g., Def-Use edges [Heo+19]), and (2) *fixpoint computation*, which runs the actual Datalog program on the input facts. JAVADL minimizes its dependency on black-box fact extractors by providing syntactic pattern matching on the input abstract syntax tree (AST) for the full Java 8 syntax. This pattern matching support eliminates much of the *interoperability* concern that Madsen, Yee, and Lhoták [MYL16] observed for Datalog in program analysis, i.e., the need for "*tedious mapping*" and manual (de)serialization code in fact extraction.

For convenience, JAVADL additionally provides detectors with a small but extensible set of facts (name and type analysis information) from the ExtendJ Java compiler [EH07].

We take JAVADL as one point in the design space for clear-box static checkers and explore how it compares to contemporary static checker tools with regard to expressiveness and performance. Our experiments show that JAVADL offers performance comparable to that of existing checkers and enables the concise implementation of different types of bug detectors, in the common, broad sense of the term [Rei21], i.e. including both syntactic "bad smell" detectors and semantic, inter-procedural detectors that reason across compilation units. Our results also show that JAVADL can effectively incrementalize these checkers, but that incremental analysis may not always outperform exhaustive analysis. Our contributions in this paper are the following:

- the first (to the best of our knowledge) static checker framework with support for syntactic pattern matching over the full Java grammar (up to Java 8),

- a Datalog language extension for syntactic pattern matching that extends over our earlier work [DBR] by handling both syntactic ambiguity and semantic information,

- an incremental evaluator for inter-procedural Datalog-based program analysis,

- a prototype implementation of our approach, the JAVADL system,[1]

- a comparison of performance and quality between JAVADL, SpotBugs, and Error Prone, with a validated [DRS21b] and an improved artifact [DRS21c] for reproducibility.

2   The JAVADL language

As an introduction, consider the Covariant equals() bug pattern, shared by the FindBugs and Error Prone checkers. In Java, all objects inherit the method

---

[1] https://github.com/lu-cs-sde/metadl

`equals(Object)` for equality checking. If class $C \neq$ `Object` wants to provide custom equality tests, it must *override* `Object.equals(Object)` with its own C.`equals(Object)` method. Note that Java requires that C.`equals` must accept *any* `Object` parameter. However, programmers may miss this requirement and instead define a method that only accepts a subtype of `Object`, e.g. C.`equals(`C`)`. Since Java also supports *overloading*, this method may work as expected in some contexts but cause subtle bugs elsewhere.

Figure 1 shows a complete JAVADL specification that includes a detector for this pattern. For now, we skip over the input and output handling and focus on the detector itself, in lines 3–15.

Here, line 3 declares that the checker records a WARNING at source location $\langle$`ls, cs, le, ce, f`$\rangle$ (in order: start line & column, end line & column, source file) whenever class `c` defines an `equals(c)` method (COVARIANTEQUALS, lines 5–10) but no `equals(Object)` method (OBJECTEQUALS, lines 11–15).

The code in these lines is a standard Datalog Horn clause,

$$P_0(\overline{x_0}) :\text{-} P_1(\overline{x_1}), \ldots, P_n(\overline{x_n}).$$

where P_i are predicate symbols and $\overline{x_i}$ are sequences of variables and constants. The semantics of these clauses are that for any assignment ρ that maps all variables in $\overline{x_0}$, $\ldots, \overline{x_n}$ to constants, whenever for all $i \in \overline{1..n}$ the relations P_i contain the tuple $\rho(\overline{x_i})$, the tuple $\rho(\overline{x_0})$ must also be in the relation P_0. We call $P_0(\overline{x_0})$ the *head literal* and the other $P_i(\overline{x_i})$ the *body literals* and follow most Datalog dialects in allowing body literals to be negated (with the obvious semantics). When $n \neq 0$, the clause is also called a *rule*, and if $n = 0$, it is called a *fact* and usually omits the left arrow ':-'.

Lines 5–9 show how JAVADL extends clauses with pattern matching to define CO-VARIANTEQUALS:

`⟨: class #_ { .. public boolean equals(#t #_) { .. } .. } :⟩`

This pattern captures all class declarations `c` that declare an `equals` method with a single parameter. The pattern matching braces $\langle: \cdots :\rangle$ contain Java source code, extended with *syntactic metavariables* such as `#t`, which here binds to the type name of the formal parameter to `equals`, and *wildcards* (`#_`) that we here use to ignore the name of that same parameter. The *gaps* (`..`) in the pattern ignore *sequences* of program elements: here, they allow any number of declarations to precede or follow the declaration of `equals`, and any sequence of statements inside the `equals` method body.

This pattern binds `#t` and `c` to AST nodes in the Java input program. To check if `c` defines a method `equals(c #_)`, we must also ensure that `c` and `#t` describe the same type. JAVADL represents each type by the type's declaration AST node, so `c` already *is* a type, but `#t` is only a name and could resolve to different types in different contexts. We resolve its type with the *semantic predicate* DECL and ensure that that type is `c` by requiring that DECL(`#t, c`) must hold.

OBJECTEQUALS in lines 11–15 analogously captures all classes `c` that override `Object.equals(Object)`.

```
1 EDB('ANALYZEDFILES, "AnalyzedFiles.csv", "csv").
2 java ('ANALYZEDFILES) {
3   WARNING(ls, cs, le, ce, f) :- COVARIANTEQUALS(c), NOT(OBJECTEQUALS(c)),
4                                    SRC(c, ls, cs, le, ce, f).
5   COVARIANTEQUALS(c) :- c ⟨: class #_ {
6                                  ..
7                                  public boolean equals(#t #_) { ..  }
8                                  ..
9                              } :⟩,
10                         DECL(#t, c).
11  OBJECTEQUALS(c) :- c ⟨: class #_ {
12                             ..
13                             public boolean equals(Object #_) { ..  }
14                             ..
15                         } :⟩.
16 }
17 OUTPUT('WARNING, "Warnings.csv", "csv").
```

Figure 1: Covariant equals check implemented in JAVADL

We have now seen the essence of how JAVADL combines Datalog, syntactic patterns, and semantic predicates like DECL and SRC to analyze source code. JAVADL provides some additional features for handling input and output, which we demonstrate in the remaining parts of Figure 1.

Returning to the top of our example, line 1 specifies that the relation ANALYZED-FILES contains all tuples from an external data source (a .csv file). Line 2 then starts a java block over ANALYZEDFILES. This block sets the scope for pattern matching and semantic predicates in lines 3–15: our bug detector will analyze all the source files listed in the file AnalyzedFiles.csv. Finally, line 17 declares that all tuples in the relation WARNING must be written to the file Warnings.csv.

2.1 Non-Local and Semantic Analyses

From the perspective of a typical imperative checker framework, the properties we describe in Figure 1 amount to boolean checks on Java classes. To see how JAVADL expresses more complex semantic properties, consider the Missing **switch** Default bug pattern. Figure 2a shows a **switch** statement over int values that only captures three cases; this code may indicate that the programmer forgot to handle other possible cases. Both Error Prone and SpotBugs flag such code, and JAVADL can capture this case with syntactic pattern matching and a small amount of logical reasoning.

However, relying purely on syntactic reasoning leads to a false positive for Figure 2b. This **switch** over an enumeration is exhaustive and therefore requires no **default**. SpotBugs and Error Prone will not flag the **switch** in Figure 2b, but *will* flag the nonexhaustive **switch** in Figure 2c.

To distinguish these three cases, we need to know that variable state is an **enum** value and what values it can take. That means that we need state's static type, and find that type's possible **enum** values (e.g., FAIL and OK), which can be *non-local*

```
1 switch(getInt())) {
2     case 0: return "zero";
3     case 1: return "one";
4     case 2: return "two";
5 }                              (a)
```

```
1 enum State { OK, FAIL }
2 switch(state) {
3     case FAIL: return false;
4     case OK: return true;
5 }                              (b)
```

```
1 enum State { OK, FAIL, HALT }
2 switch(state) {
3     case FAIL:
  return false;
4     case OK: return true;
5 }                              (c)
```

Figure 2: Examples for the Missing **switch** Default bug pattern. Code box (a) lacks a default case. Box (b) does not, since it covers all possible **enum** cases explicitly. Box (c) uses the same **switch** block as box (b) but extends the **enum** with (HALT), meaning that the **switch** again lacks one case.

information if **enum** State resides in a different compilation unit. (More precise analyses are also conceivable, cf. Section 5.2.)

Figure 3 shows how we can specify this bug pattern in JAVADL. Predicate SWITCH (line 1) matches all **switch** statements, while SWITCHHASDEFAULT(s) (line 2) matches only those who contain **default** clauses. Next, CASEONENUM(s, τ, m) (line 3) captures all **case** branches for **enum** member m, plus the surrounding **switch** statement s and **enum** type τ. Here, DECL$(\#\text{c}, m)$ maps **enum** values from Figure 2 to their declarations, while TYPE$(\#\text{v}, \tau)$ relates the expression $\#\text{v}$ that we are discriminating over to its type τ, which is an **enum** type iff $\tau\langle:\textbf{enum}\;\#\_\;\{\;..\;\}:\rangle$ holds.

CASEONENUM(s, τ, m) thus uses both semantic and non-local information: DECL and TYPE provide the results of ExtendJ's name and type analysis, respectively. TYPE(e, τ) relates expressions e and their Java types τ, and we follow ExtendJ in identifying τ with class, interface, or enum definitions for all types for which we can find one in a source or .class/.jar file. In the latter case, we use ExtendJ's partial decompilation facilities to map the high-level bytecode class structure (excluding method bodies) into an AST representation. For example, when applied to Figure 2b, CASEONENUM would contain $\langle s, \text{State}, \text{OK}\rangle$ and $\langle s, \text{State}, \text{FAIL}\rangle$, where s is the **switch** block in line 2, and State, OK, and FAIL are the declaration sites of **enum** type State and its values OK and FAIL, respectively. These semantics are independent of whether State is defined in the same source file as the **switch** statement, in a different source file, or in Java bytecode.

Continuing our example, predicate ENUMMEMBER$(\tau, \#\text{m})$ in line 6 relates **enum** types τ and their member values $\#\text{m}$, while SWITCHWITHOUTENUMMEMBER(s, τ) checks whether switch s over **enum** type τ is missing at least one enum value (m, in line 7), SWITCHONALLENUMMEMBERS(s) checks that SWITCHWITHOUTENUMMEMBER does not hold for s, and

```
1 SWITCH(s) :-               s ⟨:switch (#_) { .. }:⟩.
2 SWITCHHASDEFAULT(s) :-  s ⟨:switch (#_) { ..  default:  ..  }:⟩.
3 CASEONENUM(s, τ, m) :- s ⟨:switch (#v) { ..  case #c:  ..  }:⟩,
4                           DECL(#c, m), TYPE(#v, τ),
5                           τ ⟨:enum #_ { .. }:⟩.
6 ENUMMEMBER(τ, #m) :- τ ⟨:enum #_ { .., #m , ..  ; .. }:⟩, ID(#m, _).
   // ID filters out declarations #m that are not identifiers, thus not enum values
7 SWITCHWITHOUTENUMMEMBER(s, τ) :-
8    CASEONENUM(s, τ, _), ENUMMEMBER(τ, m), NOT(CASEONENUM(s, τ, m)).
9 SWITCHONALLENUMMEMBERS(s) :-
10   CASEONENUM(s, τ, _), NOT(SWITCHWITHOUTENUMMEMBER(s, τ)).
11 SWITCHWITHOUTDEFAULT(s) :-
12   SWITCH(s), NOT(SWITCHHASDEFAULT(s)), NOT(SWITCHONALLENUMMEMBERS(s)).
```

Figure 3: JAVADL specification for the SwitchNoDefault bug pattern, excluding input and output handling.

SWITCHWITHOUTDEFAULT(s) finally collects all **switch** statements that neither have a default nor cover all **enum** members.

Non-local bug patterns like SWITCHWITHOUTDEFAULT are the interesting cases for incremental analysis: if the code changes in either the **switch** statement or in the **enum** type, we must re-evaluate all potentially affected reports, ideally without re-evaluating the unaffected ones (Section 4).

2.2 Language Definition

Figure 4 summarizes the JAVADL syntax. Our language is based on Datalog with negation, where literals in rule bodies may be negated **NOT**(P($\ldots$)), following the standard ("stratification") requirement that they must not transitively depend on their own negations (e.g., P(x) :- **NOT**(P(x)).).   JAVADL extends Datalog with java blocks, syntactic patterns, and predicate references to support syntactic pattern matching.

Syntactic patterns allow us to match terms in a given input program. JAVADL surrounds syntactic patterns J by quotes $⟨:J:⟩$ that we write <: J :> in the source code. Here, J is any Java code fragment that can be rooted at a single AST node; these can be statements, classes, types, import statements or any other syntactic production from the Java grammar. For instance, the pattern $⟨:$#e.toString():⟩$ will match any calls to the method toString() in the input program and bind the expression on which the call is made to the pattern metavariable #e. Pattern metavariables like #e describe AST nodes, and we can use them outside of syntactic patterns as normal Datalog variables. In some cases we are interested in reasoning about the root of a syntactic pattern explicitly. JAVADL's *rooted matching* syntax expresses e.g. the set of occurrences of the literal 0 in the input program as $z⟨:0:⟩$; as usual in logic programming, we can use this pattern both to test whether z is an AST node for a 0 and to find all z with that property. Using pattern metavariables for rooted matching allows us to recurse, e.g. to conservatively underapproximate the set of all expressions that evaluate

Program		$::= \overline{D_i}$
Declaration	$D ::=$	$C \mid \texttt{java}(r)\,\{\overline{C_i}\}$
Clause	$C ::=$	$\overline{H_i}\,\texttt{:-}\,\overline{B_j}.$
Head literal	$H ::=$	$P\,(\overline{t_i})$
Body literal	$B ::=$	$H \mid \textbf{NOT}(H) \mid \langle\!:J:\!\rangle \mid v\langle\!:J:\!\rangle$
Term	$t ::=$	$v \mid \_ \mid k \mid r \mid e$
Predicate reference	$r ::=$	$\texttt{'}\,P$
Expression	$e ::=$	$v \mid \texttt{to\_number}(e) \mid e\texttt{+}e \mid e\texttt{*}e \mid ...$
Variable	$v ::=$	$v_b \mid v_m$
Predicate symbol	$P \in$	$PredSym$
Constant	$k \in$	$\mathbb{Z} \cup String$
Pattern language	$J \in$	$JavaPatternLanguage$
Basic variable	$v_b \in$	Var
Pattern metavar.	$v_m \in$	$\#v_b$

Figure 4: Syntax for the JAVADL language.

to 0:
```
1    ZERO(#z) :- #z⟨:0:⟩.
2    ZERO(#z) :- #z⟨:0 + #z2:⟩, ZERO(#z2). // And so on.
```
We can thus derive complex semantic properties directly from syntactic patterns.

When matching patterns, we often want to leave parts inside the pattern unspecified; we can do so by using gaps (..). For example, in Figure 1 we used gaps to ignore declarations inside class bodies, since these were irrelevant to our analysis. We also allow gaps within sequences of elements. For instance, the pattern ⟨:new #T[] { #first , .. }:⟩, will match array initializers with at least one (but possibly more) elements, and bind the first element to #first.

Meanwhile, *predicate references* quote a predicate, e.g. ' SRCS, and turns it into a logical object, akin to a pointer to a C variable or a `java.lang.Class` object. We use this mechanism to describe properties *about* predicates, and especially to expand predicates that describe a set of source files into ASTs through a `java` block:

$$\texttt{java}(\texttt{'}\,\textsc{Srcs})\ \{\ ...body\ ...\}$$

All syntactic patterns in the *body* of this block will reason about precisely the ASTs of the programs whose sources are in SRCS. This design gives us a scope for syntactic patterns without having to make AST roots explicit everywhere; we can think of `java` blocks as "broadcasting" implicit parameters to all rules in the body. This design follows our earlier MetaDL `analyze` blocks [DBR], adding rooted patterns and support for syntactic ambiguity (Section 3.3). While `java` blocks are not essential to our approach and could be implicit, we include them to enable future use cases e.g. for pre-analyses that grow the list of source files to handle dynamic class loading.

2.3 Pseudopredicates

JAVADL incorporates a number of language constructs with the syntactic form of predicates but special semantics. We categorize these *pseudopredicates* into three groups: *I/O pseudopredicates* for accessing external data, *infinite pseudopredicates* for arithmetic and comparison, and *semantic predicates* that expose precomputed semantic information. We have already introduced our two I/O pseudopredicates, which may only occur as facts (i.e., not in rules):

- EDB($'$P, `"TabulatedP.csv"`, `"csv"`) imports an external database (SQLite3 or CSV) into a relation (here: P). Like facts from `java` blocks, these imported relations provide *ground facts*, rather than derived facts; the 'E' in EDB ('extensional') alludes to this property.

- OUTPUT($'$R, `"TabulatedR.db"`, `"sqlite"`) exports the contents of the R predicate to the database file `"TabuletedR.db"`.

Our infinite pseudopredicates are the only predicates that allow (nested) arithmetic expressions inside their literals. We only allow these pseudopredicates in rule bodies:

- EQ(e_0, e_1) holds iff the value of e_o is equal to the value of e_1.

- LT(e_0, e_1) holds iff the value of e_0 is less than the value of e_1.

- BIND(v, e) holds iff the variable v equals the value of expression e. Unlike EQ, BIND can also bind variables.

Finally, semantic predicates provide additional information about programs under analysis. They are only available inside `java` blocks:

- ID(n, m) relates the node n to its name m, if the node represents a named AST fragment (e.g., a variable, class or method).

- SRC(n, l_s, c_s, l_e, c_e, f) relates the AST node n to its source code location, where the pairs $\langle l_s, c_s \rangle$ and $\langle l_e, c_e \rangle$ represent its start and end positions in the source file f.

- PARENT(n, c) relates the AST node c to its parent n.

- INT(n, s) relates the integer literal n to its value s, represented as a string. We provide similar predicates for the numeric literals (FLOAT, LONG, etc.).

- STR(n, s) relates the string literal n to its string value s.

- MOD(n, m) relates a declaration n to its modifiers (**public**, **static** etc.).

- DECL(n, n_d) relates the name n to its declaration n_d.

- TYPE(n, n_t) relates an AST node that represents a name or expression n to the AST node n_t that represents the node's type. Due to limitations in our frontend, there are corner cases in Java 8 where this relation may be inaccurate.

We did not find it necessary to add a feature for referencing specific Java types. As we show later in Figure 11 for `java.lang.String`, the existing JAVADL features suffice for this task.

The TYPE and DECL relations allow JAVADL code to refer to code from external `jar` files, though pattern matching support on such compiled code excludes statements and expressions.

Types   JAVADL is a statically typed language, without explicit type declarations. Instead, it relies on monomorphic type inference. Currently, JAVADL supports four types for variables: *Int*, *String*, *ASTNode* for AST nodes, and *PredRef* for predicate references.

3 Implementation

JAVADL can run bug checkers either exhaustively (on the entire program) or incrementally (only on the parts of a program that have changed). We first describe our implementation for exhaustive evaluation, and discuss the refinements for incremental evaluation in Section 4.

3.1 Architecture

Figure 5 and Figure 6 show an overview of our prototype implementation, with an initial *synthesis* step illustrated in Figure 5 and the steps of the *exhaustive evaluation* in Figure 6.

In Datalog synthesis, our prototype compiles JAVADL into efficient native checker code with the help of the high-performance Datalog compiler Soufflé [Sch+16]. In this step, our compiler first parses the Datalog fragment of the JAVADL program using an LALR parser, but leaves syntactic patterns ($\langle : \ldots : \rangle$) unprocessed. It calls a separate parser to parse these (often highly ambiguous) patterns (Section 3.3) and then translates them into Datalog queries (Section 3.3). The compiler then emits Datalog in Soufflé's Datalog dialect (adding explicit type annotations and input/output specifications), and calls Soufflé to generate the native checker library. This library implements the Datalog portion of the bug checker's exhaustive variant and is independent of the program under analysis, so we only need to re-synthesize it when we add, remove, or alter bug detectors.

In exhaustive evaluation, the driver first determines the set of Java files for each `java` block to analyze[2], then passes these files to the Java compiler ExtendJ [EH07].

---

[2]If a `P` in a `java` (`'P`) block is not loaded directly, we additionally use partial interpretation (not explored in this work).

The driver then translates relevant parts of ExtendJ's Java AST representation into in-memory tables for our native checker binary ("Datalog projection"), by flattening the AST structure (Section 3.2) and tabulating semantic information (e.g., the TYPE and DECL) predicates. Finally, the driver passes these tables to the native library, which completes Datalog evaluation and reports all detected bugs.
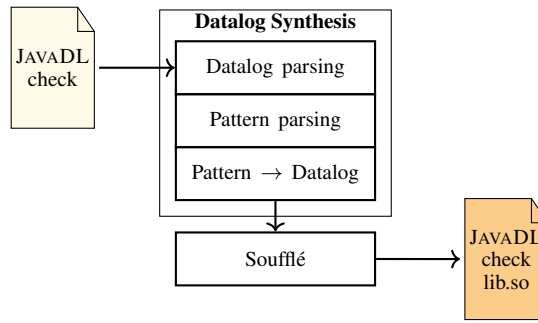
Figure 5: Compilation of a JAVADL checker specification to a Datalog library
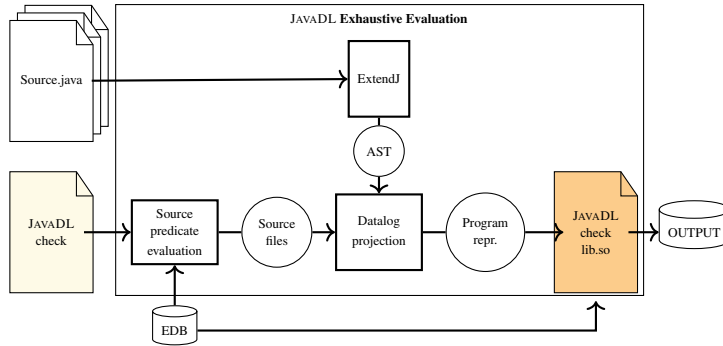
Figure 6: Exhaustive evaluation of a compiled JAVADL checker on a Java program

3.2 Program Representation

Our syntactic pattern matcher operates entirely on logical relations. Thus, we compile syntactic Java patterns into special pattern-matching Horn clauses (Section 3.3) during Datalog synthesis, and translate Java source code into logical facts. Since ExtendJ's AST faithfully preserves the concrete syntax, we do not distinguish between ASTs and parse trees.

Our approach extends the relational representation of ASTs from our earlier work [DBR] to compactly represent ExtendJ's Java AST, together with terminal symbols, source location information, and semantic attributes (TYPE, DECL etc.).

We map AST nodes to 64-bit integers, and represent the connections between them with the following three relations:

1. $\mathrm{AST}(k : \mathit{String}, n : \mathit{ASTNode}, i : \mathit{Int}, c : \mathit{ASTNode}, t : \mathit{String})$ represents a node n, of syntactic production k. If n has no children, we store the node information as a single fact $\mathrm{AST}(k, n, \text{-}1, \text{-}1, t)$. If k represents a terminal name or literal, then t is additionally the corresponding lexeme. Otherwise, if n is a nonterminal, c represents the ith child node.

2. $\mathrm{ATTR}(n : \mathit{ASTNode}, a : \mathit{String}, m : \mathit{ASTNode})$ holds if node n's attribute a has value m.

3. $\mathrm{SRC}(n : \mathit{ASTNode}, l_s : \mathit{Int}, c_s : \mathit{Int}, l_e : \mathit{Int}, c_e : \mathit{Int}, f : \mathit{String})$ relates node n and its source location, represented as starting and ending line and column information, and file name.

3.3 Syntactic Pattern Matching

Our JAVADL compiler translates syntactic patterns into logical literals. This delegates the heavy lifting of syntactic pattern matching to the Datalog engine and allows us to freely intersperse pattern matching and other forms of program analysis.

The Java Pattern Grammar

We construct our pattern grammar by transforming the ExtendJ Java grammar. For each syntactic category, represented by a nonterminal N, we introduce two fresh nonterminals N_p, which adds metavariables ($\#v$), and N_l, which also adds gaps (..):

$$
\begin{aligned}
N_p &\quad ::= \quad N \mid \mathit{MetaVarID} \\
N_l &\quad ::= \quad N_p \mid \texttt{".."}
\end{aligned}
$$

We then update all other rules in the BNF grammar to replace right-hand-side occurrences of N by N_p, except for sequences of N (e.g., parameter lists), which we replace by N_l. This process is fully automatic to support future changes to ExtendJ.

Our pattern grammar accepts any Java program but introduces ambiguities. For example, the pattern $\langle :\#\texttt{t}\ \#\texttt{n}()\ \{\ ..\ \}:\rangle$ can be parsed both as a constructor definition or as a method definition. The pattern compiler statically detects such ambiguities and emits a warning, but defaults to allowing any of the possible parses. The bug detector writer can then choose to disambiguate by using other predicates in the same clause as the pattern. In the example above, the rule could e.g. test if $\#\texttt{t}$ is a modifier or a type with the literal $\mathrm{MOD}(\#\texttt{t},\ \_)$. If the bug detector writer instead chooses to retain this ambiguity, $\#\texttt{t}$ will match both constructor and method definitions. These ambiguities only affect syntactic patterns; when we later parse Java input programs, we rely on ExtendJ and its unambiguous Java grammar.

Relational Representation of Patterns

We transform syntactic patterns to Datalog clauses, following our earlier work [DBR] and De Roover et al. [DR+07], except for encoding the semantic category of the matched node in tuples rather than in predicates.

To illustrate, let us first consider how ExtendJ would parse the expression `1 + 1`. Figure 7a shows the corresponding part of the ExtendJ Java grammar: ExtendJ will represent the syntactic category `add_expr` by an AST node **AddExpr** with two **IntegerLiteral**(1) child nodes.

We parse the pattern e⟨: #x + 1 :⟩ similarly, except with the grammar in Figure 7b, which adds rules for gaps and metavariables.

We represent gaps and metavariables by custom AST nodes, and assign each node a *node identifier* (Figure 8). For metavariables and the roots of rooted patterns, we set these identifiers to be the corresponding (meta)variables. For all other nodes, we introduce a fresh variable (e.g. ϕ_1, ϕ_2).

For each such pattern p, we then create a fresh Datalog predicate PAT_p, with its arity equal to the number of free variables in the pattern, including the optional root. We then replace all occurrences of p by PAT_p to desugar JAVADL to plain Datalog.

```
1 Expr add_expr ::= add_expr.e1 PLUS mul_expression.e2
2    {: return new AddExpr(e1, e2); :}
3 Expr mul_expression ::= ...
4       | literal
5 IntegerLiteral literal ::= LITERAL.l
6    {: return new IntegerLiteral(l); :}
```

(a) Pattern Rule

```
1 Expr add_expr ::= add_expr.e1 PLUS mul_expr.e2
2       {: return new AddExpr(e0, e2); :}
3   | METAVARID.id
4       {: return new ExprMetaVar(id); :}
5   | ...
6 Expr mul_expr ::= ...
7   | literal
8   | METAVARID.id
9       {: return new ExprMetaVar(id); :}
10 IntegerLiteral literal ::= LITERAL.l
11      {: return new IntegerLiteral(l); :}
12   | METAVARID.id
13      {: return new IntegerLiteralMetaVar(id); :}
```

(b) Extended Grammar Rule

Automatic Pattern Grammar Extension

Figure 7: Pattern grammar generation and pattern parsing.

In Figure 8, the **MetaVarExpr** node corresponds to the syntactic category **Expr** in the abstract grammar. Since we tag our AST nodes with their syntactic production (e.g., **IntegerLiteral**) rather than with nonterminal syntactic categories, we encode their subtyping relationship in an internal predicate SUPERTYPE(*String*, *String*). We generate this predicate statically, based on the analysis of the type hierarchy in the abstract grammar, and use it to set a bound on the syntactic categories that a metavariable can match. This transforms the pattern $e\langle: \#x + 1 :\rangle$ into the Datalog rule in Figure 9. We use the same approach to translate gaps and the constraints that they introduce as in MetaDL [DBR].
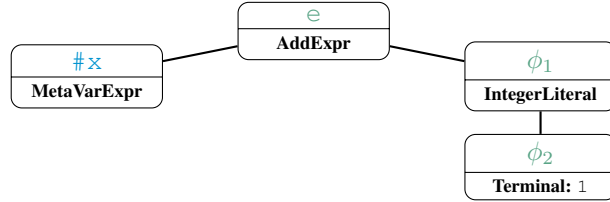
Figure 8: The abstract syntax tree for the pattern $e\langle: \#x + 1 :\rangle$

```
1 PAT(e, #x) :- AST("AddExpr", e, 0, #x, _),
2                AST("AddExpr", e, 1, φ₁, _),
3                SUPERTYPE(κ₁, "Expr"),
4                AST(κ₁, φ₁, 0, φ₂, _),
5                AST("Terminal", φ₂, _, "1").
```

Figure 9: Expanded pattern

Parsing Ambiguous Patterns

Our automatically generated pattern grammar (Figure 7b) is highly ambiguous, and unsuitable for most parsing algorithms. We therefore adapt Scott [Sco08]'s shared packed parse forest (SPPF) variant of the Earley parser. This algorithm compresses ambiguity effectively, but solves only part of the ambiguity in our case.

To illustrate, consider the pattern $\langle: \#e_1 + \#e_2 + \#e_3 :\rangle$. Parsing this pattern yields the SPPF in Figure 10a, which contains two SPPF *alternative* nodes. These are not parse tree nodes but instead mark a choice in the tree: either child of an alternative node will yield a correct parse tree.

While a purely syntactic system could match over these alternatives directly, JAVADL must map syntactic categories to AST node names (Section 3.3). However, ExtendJ's parser uses a black-box specification of the mapping between concrete and abstract syntax that only provides us with opaque 'parser actions' ({: ... :} in Figure 7) that we can call, but not inspect automatically.

To avoid having to produce all four ASTs for our example, or 2^n in general, we simplify the parse forests by shortening the path from each metavariable or gap terminal to

the root. Concretely, we find all derivation chains of the form $N \to M \to MetaVarID$ and replace them with $N \to MetaVarID$. We do this transformation recursively, starting from the leaves towards the root, while also merging alternatives if they contain the same derivation chain. We perform a similar transformation for gaps. For the example in Figure 10a, we apply the transformation twice for $N =$ **MulExpr** and $M =$ **IntegerLiteral**. The intuition behind this transformation is that instead of enumerating all the possible syntactic categories a metavariable could have, we provide an upper bound for them.

Going back to the example in Figure 10a, the metavariable `#e3` could match the syntactic categories **IntegerLiteral** or **Expr**. After the transformation (Figure 10b), `#e3` matches all the nodes having the syntactic category **Expr**, which is more general and includes **IntegerLiteral**.
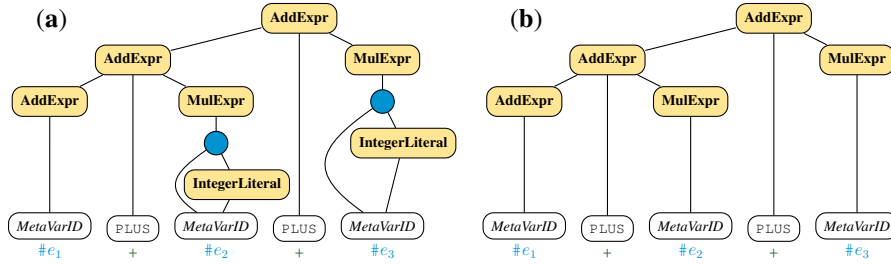
Figure 10: The original (a) and simplified (b) SPPFs for the expression `#e1` + `#e2` + `#e3`. ● denotes alternatives.

In effect, the transformation merges multiple derivation chains that end up in a *MetaVarID* (or a gap) and yields an SPPF with significantly fewer edges. In practice, this enables us to quickly enumerate the parse trees. For the addition pattern discussed above, this approach produces a single parse tree (Figure 10b).

3.4   Semantic Attribute Extraction

ExtendJ is a full-fledged Java compiler, and therefore performs a variety of program analyses. Some of these are useful for writing bug detectors, so we export them for easy access in JAVADL code.

ExtendJ is implemented in the *Reference Attribute Grammar* system JastAdd [HM03], meaning that its program analyses are computed as attributes over AST nodes, and that these attributes may be references to other AST nodes. For example, ExtendJ computes the declaration site of a variable or a method as a reference to the AST node that contains the node's declaration, and the type of a variable as a reference to the AST node of the type's class definition. Some of these AST nodes are not AST

nodes in the classical sense. ExtendJ computes *synthetic AST nodes* (higher-order attributes [VSK89]) on-demand to represent e.g. primitive types and classes loaded from external jar files.

Since ExtendJ runs on the Java Virtual Machine and Soufflé is a native executable, implementing on-the-fly attribute evaluation through callbacks to Java would require considerable engineering effort. We instead opted to tabulate program representation relations directly into Soufflé's data structures, using Java Native Interface calls. While tabulating the program's original AST requires only a simple tree traversal, the evaluation of attributes may produce new synthetic AST nodes that we must traverse, add to the program representation relation, and recursively re-examine to extract their attributes. We implement a fixed point algorithm here for exhaustive attribute evaluation.

JAVADL currently supports the evaluation of two attributes that may produce fresh AST nodes:

- DECL, mapping a node to the AST node declaring it, and

- TYPE, mapping an AST node to the node declaring its type

In general, the attribute extraction mechanism is easy to extend. It takes no more than 10 lines of code to expose a new attribute as a Datalog relation. This property of our implementation facilitates easy sharing of analyses between ExtendJ and JAVADL.

4   Incremental Evaluation

When developers integrate static checkers into their workflow, these checkers often become part of continuous integration or code review [Sad+15; Cal+15]. In those scenarios, the checker runs frequently, and each run sees largely the same source code, meaning that exhaustive analysis can be wasteful.

Consider the analysis in Figure 11. This check, inspired by Spot-Bugs' DM_STRING_CTOR, looks for explicit String object constructions, **new** String(#v), where #v is already a String. Since Java Strings are immutable, this construction is inefficient. On line 1, the predicate NEWSTRING identifies source locations $\langle f, l, c \rangle$ that explicitly call a String constructor with a single argument of type τ. On line 3, STRINGCLASS finds the unique java.lang.String class. Finally, on line 5 BADNEWSTRING filters NEWSTRING to only report string constructions from existing string objects.

```
1 NewString(τ, f, l, c) :- n ⟨:new String(#v):⟩,
2                          TYPE(#v, τ), SRC(n, l, c, _, _, f).
3 StringClass(s) :- s ⟨:.. class String { .. }:⟩,
4                  SRC(s, _,_,_,_, "java/lang/String.class").
5 BadNewString(f, l, c) :- NewString(τ, f, l, c), StringClass(τ).
```

Figure 11: Check for wasteful String construction, split into three separate predicates to simplify our exposition.

We see that different parts of one bug detector may need to be updated at different times: we only need to recompute STRINGCLASS if the standard library changes, but must update NEWSTRING on changed parts of the source code. Finally, we must update BADNEWSTRING if either of the other relations changes.

4.1   Incremental Architecture

Changes to source code can lead to the retraction of a previously true fact, e.g., when a developer changes the superclass of some class `C` from `A` to something else. Retractions can have consequences: if `C` inherited `m` from `A`, it will no longer do so. However, the consequences of retractions may not be obvious: `C` might also have obtained `m` from a second source, e.g., a Java `default` interface method.

The literature has proposed a number of different techniques for incremental updates of Datalog-style analyses [GMS93; Sza+18]. However, we found these techniques unsuitable for JAVADL: first, they assume that all ground facts are permanently stored in a database. In JAVADL, the majority of ground facts are AST nodes that we derive from source files (i.e., these facts are actually derived and "ground" only from the perspective of Datalog), and serializing these facts to disk would incur nontrivial storage cost. Second, we assign identities to these nodes based on their order in the parsed file, which means that a small change at the beginning of a file would trigger individual retractions for almost all AST nodes in that same file.

While we expect to be able to avoid the second problem through a suitably tuned AST differencing algorithm [NRL18] analogously to CFG-based differencing [AB14], this would not address the first problem. We aimed to keep the AST in-memory as much as possible, so we use a conservative provenance tracking scheme that partitions facts by compilation unit, tracks which compilation unit partition contributed to which other partition, and propagates retractions along partition dependencies.

This partitioning scheme reflects both the level of granularity of the ASTs that we obtain from ExtendJ and JAVADL's use as a static checker: such tools typically run e.g. in Continuous Integration environments, where the degree of incrementality that they encounter is that of one or more revision control commits.

Our incremental evaluator first computes a set of *stale* files from a set of modified, added, and deleted source files, to which it transitively adds all other source files that transitively depend on stale files. It then discards data from these stale files and re-analyzes them as needed. This adds several components over the exhaustive analysis (Section 3), which we summarize in Figure 12:

1. *Predicate Separation* (Section 4.2) sorts JAVADL predicates into *local* predicates (e.g., NEWSTRING and STRINGCLASS in Figure 11), which we can run on individual source files, and *global* predicates which can depend on multiple source files (e.g., BADNEWSTRING in Figure 11).

2. *Attribute provenance tracking* (Section 4.3) tracks source file dependencies.

3. The *Intermediate Result DataBase (IRDB)* (Figure 12) caches local predicates, tagged with the source file from which we have derived them, as well as attribute provenance.

Moreover, we split evaluation into two phases:

1. Local predicate evaluation, whose results (P_L) we cache in the IRDB, and which we only re-run on stale files, and

2. Global predicate evaluation, which we always run once at the end, which depends on all local predicates and obtains them either from the cache (P_C) or from in-memory data from the current run (P_L) (Section 4.2).
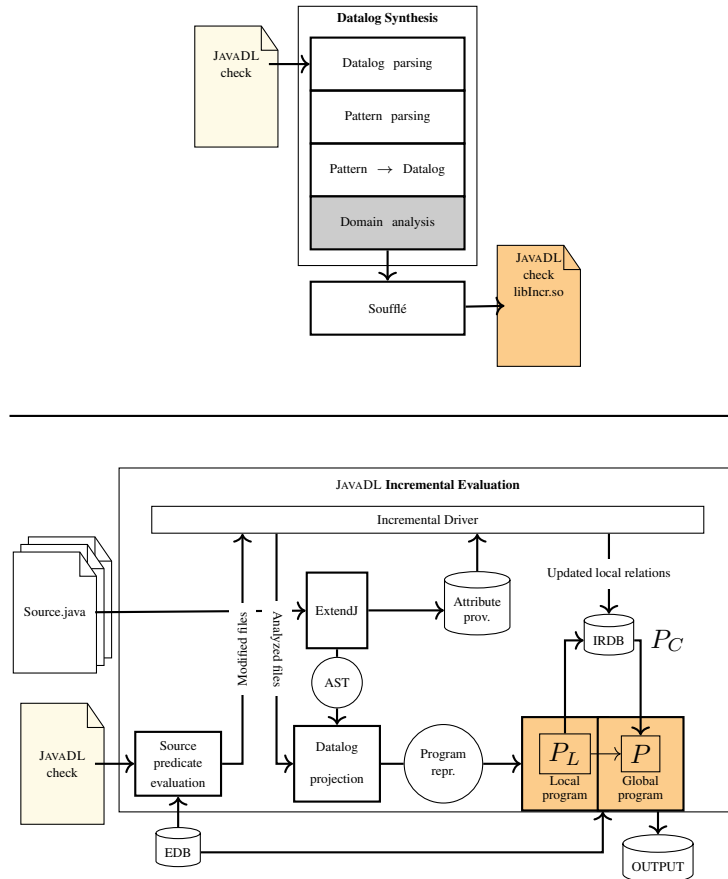
Figure 12: Incremental evaluation of a JAVADL analysis on a Java program.

4.2 Separating Local and Global Rules

Consider again Figure 11: as we argued above, predicate NEWSTRING and the rule
defining it refer to information from a single source file. However, that claim is not
entirely obvious. While the syntactic pattern only connects nodes from the same com-
pilation unit (n and #v), the pair $\langle \#v, \tau \rangle$ from TYPE may refer to any type τ in any
compilation unit. However, we can still evaluate this rule locally: if we replace TYPE
with TYPE_f, a subset of TYPE that only knows the type information of all AST
nodes in file f, and set f to be the file that contains n, then $\langle \#v, \tau \rangle \in \text{TYPE}_f \iff$
$\langle \#v, \tau \rangle \in \text{TYPE}$. We can apply the exact same reasoning to SRC in the rule for
NEWSTRING, and to the body of the STRINGCLASS rule.

However, BADNEWSTRING may combine tuples from distinct compilation units:
$\langle \tau, f, l, c \rangle$ from any compilation unit that called the `String` constructor, and $\langle \tau \rangle$
from `java/lang/String.class`. Hence, we must compute BADNEWSTRING
during global evaluation.

We formalize these intuitions in the following, but first note that the separation be-
tween local and global predicates is only relevant for predicates with variables of type
ASTNode or predicates that depend on such rules. In practice, we have encountered
user-defined predicates without *ASTNode* variables only as (short) block- or passlist-
s/allowlists, so we do not consider them further here.

For our formalization, we introduce three concepts:

1. *Domain separation signatures* (DSS) (Section 4.2), which capture which of the
   (*ASTNode*) variables of a predicate or a rule must always be in the same compi-
   lation unit. For example, the DSS for the rule body of NEWSTRING partitions
   the variables $\{n, \#v, \tau\}$ must separate $\{n, \#v\}$ from and $\{\tau\}$, since τ may come
   from a different compilation unit.

2. The *local variable set* (Section 4.2), which is the set of variables in a rule or
   parameters in a predicate that represent the compilation unit for which we can
   evaluate the predicate or rule locally. For example, we can evaluate the rule for
   NEWSTRING in the compilation unit represented by $\{n, \#v\}$, but not the one
   represented by $\{\tau\}$.

3. *AST predicates*, which are predicates that (transitively) depend on AST facts.

Domain Separation Signatures

Domain Separation Signatures (DSS) describe, for both rules and predicates, which
(*ASTNode*) parameters to the head literal must come from the same compilation unit.
We denote DSS as partitions over the variables in each rule body:

$$R_1 = \text{n}\langle\text{:new String(\#v) :}\rangle \qquad\qquad \text{DSS}(R_1) = \{\{n, \#v\}\}$$
$$R_2 = \text{n}\langle\text{:new String(\#v) :}\rangle, \text{TYPE}(\#v, \tau) \quad \text{DSS}(R_2) = \{\{n, \#v\}, \{\tau\}\}$$

and analogously for predicates, where we partition by parameter index (Figure 13).

```
P(n, #v) :- n⟨:new String(#v) :⟩                          DSS(P) = {{1, 2}}
TYPE(n, τ) :- . . . .                                   DSS(TYPE) = {{1}, {2}}
SRC(n, _, _, _, _) :- . . . .                            DSS(SRC) = {{1}}
NEWSTRING(τ, _, _, _) :- P(n, #v), TYPE(#v, τ),    DSS(NewString) = {{1}}
                  SRC(n, _, _, _, _).
STRINGCLASS(s) :- s⟨:.. class String {..} ..:⟩   DSS(StringClass) = {{1}}
```

Figure 13: Domain Separation Signatures for predicates. 1 refers to the first parameter, 2 to the second etc.

Whenever the DSS separates two variables, we may need to defer their binding to global predicate evaluation (with some exceptions, cf. Section 4.2). We thus want a DSS that is as *coarse* as possible:

Definition 1. *Let N be a finite set and σ_1, σ_2 two partitions of N. We say that the partition σ_1 is coarser than σ_2 and we write $\sigma_2 \sqsubseteq \sigma_1$ iff $\forall x. \forall y. x \sigma_2 y \Rightarrow x \sigma_1 y$.*

At the same time, we require that in each partition of a DSS, all variables in that partition must always bind to AST nodes in same compilation unit. To find the coarsest possible DSS that satisfies this requirement, we construct a DSS lattice:

Definition 2. *We define the join ($\sqcup$) and meet ($\sqcap$) for partitions over N as follows, and note the resultant top (coarsest and trivial) as well as bottom (most refined) partitions:*

$$x(\sigma_1 \sqcup \sigma_2)y \quad \Leftrightarrow \quad \exists z. x\sigma_1 z \wedge z\sigma_2 y \qquad \top = \{N\}$$
$$x(\sigma_1 \sqcap \sigma_2)y \quad \Leftrightarrow \quad x\sigma_1 y \wedge x\sigma_2 y \qquad \bot = \{\{n\} | n \in N\}$$

Definition 3. *Let P be a predicate of type $(\tau_1, ..., \tau_n)$ and $N = \{i | \tau_i = ASTNode\}$. The domain separation signature of predicate P is the coarsest partition σ of N such that if $i\sigma j$ then for all tuples $\langle v_1, \ldots, v_n \rangle \in P$, v_i and v_j must be bound to AST nodes from the same compilation unit.*

For a predicate P we denote by $\sigma_P^{\overline{x}}$ the equivalence relation induced by $\mathsf{DSS}(P)$ on all $\overline{x} = x_1, \ldots, x_k$. Let $\sigma_P = \mathsf{DSS}(P)$. Then $i\sigma_P j$ implies that any values bound to x_i and x_j must always come from the same compilation unit, so we define $x_i \sigma_P^{\overline{x}} x_j \iff i\sigma_P j$

We now conjunctively combine induced constraints for Datalog rule bodies of the form

$$R = P_1(\overline{x_1}), \ldots, P_n(\overline{x_n}), \mathbf{NOT}(P_{n+1}(\overline{x_{n+1}})), \ldots, \mathbf{NOT}(P_{n+m}(\overline{x_{n+m}}))$$

Definition 4. *The domain separation signature of a rule body R is the partition σ_R of the set V of variables of type ASTNode in the rule such that*

$$\sigma_R = \bigsqcup_{k=1,n} \sigma_{P_k}^{\overline{x_k}}$$

Inferring Domain Separation Signatures

Since predicate definitions may have recursive dependencies, the above definitions are not yet sufficient for inference. We begin inference by observing that the DSS for syntactic patterns is always trivial (all variables are in the same compilation unit), and similarly for semantic predicates, e.g. $\mathsf{DSS}(\mathrm{ID}) = \{\{1\}\}$. The exceptions are $\mathsf{DSS}(\mathrm{TYPE}) = \mathsf{DSS}(\mathrm{DECL}) = \{\{1\}, \{2\}\}$.

As a converse to our earlier definition, the equivalence relation σ_R on the variables in a rule induces an equivalence relation on the head literal, $\mathrm{P}(\overline{x})$, where we define $i\sigma_{\mathrm{P}[R]}^{\overline{x}}j \iff x_i\sigma_R x_j$ with x_i and x_j in $\overline{x}$. We compute the signatures of the predicates as a fixpoint, starting with $(\sigma_\mathrm{P})_0 = \top$:

$$(\sigma_P)_{k+1} = (\sigma_P)_k \sqcap \bigsqcap_{\mathrm{P}(\overline{x}):-\mathrm{R}} \sigma_{\mathrm{P}[R]}^{\overline{x}}$$

The number of iterations is bounded by the height of our (finite) lattice.

Definition 5. *We say that a DSS σ_P is sound iff after* exhaustive *evaluation of the* JAVADL *program that contains* P, *whenever* $\langle n_1, \ldots, n_k \rangle \in \mathrm{P}$ *and* $i\sigma_\mathrm{P}j$ *with* $i, j \in \{1, \ldots, k\}$, *we always have that* n_i *and* n_j *are from the same compilation unit. We extend this definition to* σ_R.

Theorem 4.1. *For any* JAVADL *program, DSS inference is sound for all* P.

Proof. Sketch: Assume that $i\ \sigma_\mathrm{P}\ j$, but exhaustive evaluation derives $\overline{n} = \langle n_1, \ldots, n_k \rangle \in \mathrm{P}$ s.th. n_i and n_j are AST nodes from different compilation units. Considering the Datalog horn clauses as sequents, we show by coinduction that any proof for $\mathrm{P}(\overline{n})$ must be infinite, which is a contradiction.

First note that P cannot be a built-in predicate, whose DSS are trivially sound. By fixpoint construction of σ_P, we must then have for all rule bodies R with $\mathrm{P}(\overline{x})$:- R that $x_i\ \sigma_R\ x_j$. Let

$$R = \mathrm{P}_1(\overline{y_1}), \ldots, \mathrm{P}_n(\overline{y_{n+1}}), \textbf{NOT}(\mathrm{P}_{n+1}(\overline{y_{n+1}})), \ldots, \textbf{NOT}(\mathrm{P}_{n+m}(\overline{y_{n+m}}))$$

We can ignore the negated literals, as they do not contribute to DSS and can at most remove tuples. Observe that $y\ \sigma_\mathrm{P}^{\overline{x}}\ z$ only holds if y and z are in $\overline{x}$, so $x_i\ \sigma_R\ x_j$ iff there exists a positive literal $\mathrm{P}_k(\overline{y_k})$ with $x_i\ \sigma_{\mathrm{P}_k}^{\overline{y_k}}\ x_j$. Since $x_i \neq x_j$, that literal cannot be built-in: if $\mathrm{P}_k = \mathrm{PARENT}$ (or similar local AST traversal), n_i and n_j are in the same compilation unit (contradiction), while for TYPE and DECL the DSS does not allow $x_i\ \sigma_{\mathrm{P}_k}^{\overline{y_k}}\ x_j$. By elimination, P_k must be a user-defined predicate. Let $\overline{y_k} = y_0, \ldots, y_m$ s.th. $x_i = y_a$ and $x_j = y_b$. Then P_k is unsound, because there exists $\overline{v} = \langle v_1, \ldots, v_m \rangle \in \mathrm{P}_k$ with $v_a = n_i$, $v_b = n_j$. Hence v_a and v_b are from different compilation units, but $\mathsf{DSS}\mathrm{P}_k)$ claims $a\ \sigma_{\mathrm{P}_k}\ b$. By coinduction hypothesis, the derivation of $\mathrm{P}_k(\overline{v})$ is infinite. $\square$

Local Variable Set

While the DSS tells us which variables are always in the same compilation unit, that information is not sufficient for telling whether we can evaluate a rule locally. For instance, consider the following rules, both of which have the DSS $\{\{1\}, \{2\}\}$:

$$
\begin{aligned}
Q(\#n, \tau) \quad &:\text{-} \quad \text{TYPE}(\#n, \tau), \langle :.. \quad \text{class C \{ ..} \quad \#n \ .. \quad \} : \rangle. \\
U(n, \tau) \quad &:\text{-} \quad \text{TYPE}(n, \tau), \tau \langle :.. \quad \text{class C \{ ..} \quad \} : \rangle.
\end{aligned}
$$

Q finds all fields and methods $\#n$ and their types τ in any class named C, while U finds all expressions and declarations n whose type is any class named C. Q we can compute locally, but not U: during incremental evaluation, TYPE only knows the data that we extract from ExtendJ for the compilation unit under analysis, i.e., the types of local nodes, but no external references to local nodes.

Definition 6. *For a predicate* P, *we define the local variable set as* $\mathsf{LV}(P) \in \mathsf{DSS}(P)$ *such that we can evaluate* P *locally iff all variables in* $\mathsf{LV}(P)$ *are bound to local AST nodes.*

As we saw with predicates Q and U, we can evaluate TYPE locally only if parameter 1 is bound to a local AST node. Thus, the local variable set of TYPE is $\{1\}$, while for P from Figure 13, the local variable set consists of all its variables, $\{1, 2\}$.

Analogously to domain separation signatures, each predicate induces a local variable set. So if $i \in \mathsf{LV}(P_k)$ and $L = P_k(\ldots, x_i, \ldots)$ is a literal, then $x_i \in \mathsf{LV}(L)$ (analogously for negated literals).

Definition 7. *For a rule* $R = L_0 :\text{-} L_1, \ldots, L_k$, *we define the local variable set* $\mathsf{LV}(R)$ *as the element of* $\mathsf{DSS}(R)$ *that contains at least one local variable from all literals that have a local set, i.e.,*

$$
\mathsf{LV}(R) = S \in \mathsf{DSS}(R) \text{ s.t. for all } L_i \in \{L_1, \ldots, L_k\}, \mathsf{LV}(L_i) = \emptyset \text{ or } \mathsf{LV}(L_i) \cap S \neq \emptyset.
$$

If such an element exists, R *is a* local rule, *otherwise we set* $\mathsf{LV}(R) = \emptyset$.

Corollary 4.1.1. *If* $\mathsf{LV}(R) = S$ *is nonempty, then by construction, all literals in* R *depend only on AST nodes from the same compilation unit.*

Returning to our example in Figure 11 and labeling the rule bodies on lines 1–3 as R_1, R_2, R_3, respectively, we have the local variable sets: $\mathsf{LV}(R_1) = \{n, \#v\}$, $\mathsf{LV}(\text{NEWSTRING}) = \emptyset$, $\mathsf{LV}(R_2) = \{s\}$, $\mathsf{LV}(\text{STRINGCLASS}) = \{s\}$ and $\mathsf{LV}(R_3) = \emptyset$. R_1 and R_2 are therefore local rules.

Incrementalizing by Rule Localization

If all rules R for a predicate P are local, we consider P a *local predicate*. All other predicates are *global predicates*. For each local P, we introduce two helper predicates P_L and P_C. P_L contains the tuples of P that we compute in the current evaluation pass,

Local

NEWSTRINGL$(\tau, f, t, c, \mathbf{u})$:-
    $\langle$:new String(#v):$\rangle$,
    TYPE(#v, τ), SRC($n, l, c, \_, \_, f$),
    BIND($\mathbf{u}$, cuidOf(n)).
OUTPUT($'$NEWSTRINGL).
EDB($'$NEWSTRINGC).
NEWSTRING(τ, f, l, c) :- NEWSTRINGL$(\tau, f, l, c, \_)$.
NEWSTRING(τ, f, l, c) :- NEWSTRINGC$(\tau, f, l, c, \_)$.

STRINGCLASSL$(s, \mathbf{u})$:-
    $s\langle$:.. class String {..}:$\rangle$,
    SRC($s, \_, \_, \_, \_, "$.../String.class$"$),
    BIND($\mathbf{u}$, cuidOf(s)).
OUTPUT($'$STRINGCLASSL).
EDB($'$STRINGCLASSC).
STRINGCLASS(s) :- STRINGCLASSL$(s, \_)$.
STRINGCLASS(s) :- STRINGCLASSC$(s, \_)$.

BADNEWSTRING(f, l, c) :- NEWSTRING(τ, f, l, c), STRINGCLASS(τ). **Global**

Figure 14: Global and local rules in the incrementalized version of Figure 11. The shaded areas fuse the updated and cached results and thereby recover the same results as if we had run an exhaustive analysis.

while P_C contains the tuples from cached compilation units. We augment P_C and P_L with an additional marker to store its source compilation unit.

JAVADL transforms our running example (Figure 11) into the program in Figure 14. Here, $\mathbf{u}$ is the compilation unit marker, which we extract via the function cuidOf(n). Since we encode all *ASTNode* objects as unique 64 bit integers that incorporate their compilation unit identifier, cuidOf(n) has practically no overhead. The shaded areas highlight how we fuse P_L and P_C.

4.3 Attribute Provenance

JAVADL rules can reference compilation units in two ways: through pattern-matching, and through semantic predicates (DECL, TYPE). We tabulate the latter directly from ExtendJ attributes. However, individual tuples in these relations can have nontrivial provenance. Consider Figure 15. When we analyze class C, DECL for the call `foo()` yields the definition of `foo()` in class A. However, the provenance of this information also includes `B.java`, since ExtendJ must check that class B does not override that method, i.e., changes to B can affect the DECL information in C.

To compute attribute provenance, we trace attribute evaluation with a tracing mechanism provided by JastAdd [SH11], the attribute grammar framework that underlies ExtendJ. We have extended this tracing to report source file accesses in ExtendJ, and to cache summaries for attributes that JastAdd itself caches. This technique allows us to support provenance tracking for arbitrary attributes, including future extensions. Our implementation does not yet track negative dependencies, which can arise with wildcard import statements, e.g. `import java.util.*;`. Adding support for these would require some engineering effort but no changes to our conceptual framework; in our evaluation, we systematically verified that they had no impact on the bug reports.

```
1 class A {
2   void foo() { }
3 }        A.java
```

```
1 class B extends A {
2   // void foo() {
   }
3 }        B.java
```

```
1 class C extends B {
2   void bar() {
   foo(); }
3 }        C.java
```

Figure 15: Nontrivial provenance: The `foo()` call in class `C` references `A`, but also depends on `B`.

5   Evaluation

To understand the value of JAVADL as a proof-of-concept clear-box bug checker framework, we selected an experimental setup (Section 5.1) to explore the following questions:

- **RQ1**: *To what degree can* JAVADL *be used to specify common static checks?* (Section 5.2)

    - **RQ1.1**: *How expressive is* JAVADL *compared to other tools?*

    - **RQ1.2**: *How precise is* JAVADL *compared to other tools?*

- **RQ2**: *How does the execution time for* JAVADL *compare to the state of the art?* (Section 5.3)

    - **RQ2.1**: *How does the execution time of* JAVADL *compare to other tools?*

    - **RQ2.2**: *How effective is the automatic file-level incrementalization that* JAVADL *enables?*

5.1   Experimental Setup

Selection of Tools and Bug Patterns   Modern bug checkers come with hundreds of bug patterns. To select representative patterns, we turned to a recent study by Habib and Pradel [HP18], who compare the SpotBugs, Error Prone, and Infer checkers on an extended version of the Defects4J [JJE14] data set[3]. The study identifies the five most commonly triggered detectors ("top 5 warnings") for each of the three selected detectors. We selected these warnings for our analysis and compared our approach against SpotBugs' and Error Prone's detector implementations. We excluded both Infer and its Top-5 detectors from our comparison, since (a) Infer's unique separation-logic based approach [Rey02; O'H19] distinguishes it from the checkers that we listed in the introduction to such a degree that it is unclear that a comparison would generalize, and (b) its Top-5 detectors depend on flow-sensitive analyses, for which JAVADL does not currently provide special support (Section 5.2), so that they would have required substantial additional effort to add.

---

[3]`https://github.com/rjust/defects4j/pull/112`

Table 1 summarizes these detectors and their implementations. The bug patterns marked with '⋆' are the Top-5 for SpotBugs (SB-Top-5) or Error Prone (EP-Top-5). SB-Top-5 and EP-Top-5 overlap in one detector (Boxed Primitive Constructor), for a total of nine bug detectors. For completeness, the table also lists the Covariant `equals()` detector that we discuss in Section 2.

Selection of Java Benchmarks For our evaluation setup we adapted the framework developed by Habib and Pradel [HP18] to JAVADL and updated the versions of the Defects4J data set (v2.0[4]) as well as of the checkers (SpotBugs v4.0.3, Error Prone v2.4). We used all the projects from Defects4J, except `Gson`, for which we were unable to retrieve the project properties, and `Collections` that we were not able to compile with ExtendJ. For each Defects4J project, we chose the version tagged as `D4J_project_id_FIXED_VERSION`, where *id* is the highest bug ID for *project* in the Defects4J database. We implemented two JAVADL programs, one for SB-Top-5 and one for EP-Top-5. Throughout the evaluation, we compare these two JAVADL programs directly against these recent, unaltered versions of SpotBugs and Error Prone.

Gathering Data on Analyzer Precision To assess precision, we compare the reports of our JAVADL detectors on Defects4J against the reports produced by SpotBugs and Error Prone. We followed Defects4J's configuration in excluding the projects' unit tests from this analysis. To compare JAVADL, we utilize a notion of *relative recall and precision*:

$$recall_T = \frac{|W_{\text{JAVADL}} \cap W_T|}{|W_T|} \qquad precision_T = \frac{|W_{\text{JAVADL}} \cap W_T|}{|W_{\text{JAVADL}}|}$$

where W_T is the set of warnings reported by T. These notions of precision and recall are *relative to a checker T*, so they do not represent ground truth, but rather degree of agreement with T.

5.2 RQ1: Expressiveness and Precision

RQ1.1: Expressiveness

We were able to express all detectors that we implemented in JAVADL purely with Datalog-style logical rules, syntactic pattern matching, and our small set of carefully selected semantic relations. We encountered no situation that we thought would be simpler to implement imperatively, though we are of course biased.

Table 1 shows the sizes of the bug pattern implementations in Error Prone, Spot-Bugs, and JAVADL, in lines of code. Some implementations additionally support suggested fixes, the implementation of several bug checkers may be entangled in the same implementation, and different tools may provide different library functionality (not counted here), so this comparison is not precise. With that in mind, the ratio (tool LOC

---

[4]Commit `eaebff11c`

	Baseline Static Checker Framework				JavaDL		
	★ **Bug Pattern**	**ID**	**LOC**	**Notes**	**LOC**	**Rules**	**Sem.Pred.**
Error Prone	★ Boxed Primitive Constructor	`BoxedPrimitiveConstructor`	229	115 + 114	9	3	DECL
	★ Missing `@Override`	`MissingOverride`	84	82 + 2	48	30	DECL
	★ Useless Type Parameter	`TypeParameterUnusedInFormals`	108		27	18	DECL
	★ Complex Operator Precedence	`OperatorPrecedence,` `UnnecessaryParentheses`	138	99 + 39	37	37	
	★ == on References	`ReferenceEquality`	97	**dataflow**	88	48	DECL,TYPE
	Covariant `equals()`	`NonOverridingEquals`	132	116 + 16	15	9	DECL
SpotBugs	★ Boxed Primitive Constructor	`DM_NUMBER_CTOR,` `DM_STRING_CTOR`	1415	share 52	9	3	DECL,TYPE
	★ Expose Internal Representation	`EI_EXPOSE_REP,` `MS_EXPOSE_REP,` `EI_EXPOSE_REP2,` `EI_EXPOSE_STATIC_REP2`	138		29	20	DECL
	★ Naming Convention Violation	`NM_METHOD_NAMING_CONVENTION,` `NM_FIELD_NAMING_CONVENTION,` `NM_CLASS_NAMING_CONVENTION`	499	share 12	17	10	
	★ Missing **switch** Default	`SF_SWITCH_NO_DEFAULT`	289	share 4	21	8	DECL,TYPE
	★ Field Never Written To	`UWF_UNWRITTEN_FIELD,` `UWF_UNWRITTEN_PUBLIC_OR` `_PROTECTED_FIELD`	1032	share 14 **dataflow**	51	47	DECL
	Covariant `equals()`	`EQ_ABSTRACT_SELF`	541	share 18	15	9	DECL

Table 1: Overview of selected bug patterns and their size in Error Prone, SpotBugs, and JAVADL. Patterns marked with ★ are among the Top-5 (Section 5.1). In column **Notes**, $a + b$ means that b of the lines implemented auto-fixups. *share n* means that the detector(s) were sharing their implementation with n additional detectors. Except for DM_NUMBER_CTOR and DM_STRING_CTOR, which were in two separate files (sharing 1 and 51, respectively), all detectors listed above (for all tools) were in a single source file each. **dataflow** means that the detector additionally used the Error Prone or SpotBugs data flow analysis engine (not counted for LOC). Column **Sem. Pred.** lists semantic predicates that the JavaDL implementations depended on (Section 5.2).

/ JAVADL LOC) yields a range of 1.1–12.8 (mean: 4.9) for Error Prone (when excluding LOCs for fixes) and 4.8–157.2 (mean: 38.4) for SpotBugs. If we normalize for implementation sharing across SpotBugs detectors, i.e., assume that any set of n shared SpotBugs detectors with ℓ LOC could be refactored to precisely $\frac{\ell}{n}$ LOC each, the ratio becomes 1.3–4.8 (mean 2.8), though we expect this to be an under-approximation. Either interpretation indicates that JAVADL requires fewer LOC than the other tools. Detector == on References stands out as being almost the same size for Error Prone and JAVADL. However, Error Prone's implementation utilizes a flow analysis framework whose size we did not include in the measurements. Our implementation of this detector otherwise follows Error Prone's heuristics, including checking whether the reference type defines or inherits a custom `equals()` method (28 LOC in JAVADL).

Based on manual inspection of the SpotBugs and Error Prone detectors, we believe that the main reasons for why JAVADL specifications are more concise are that (a) syntactic patterns can simplify many nested **if** statements in imperative code, and (b) looping and filtering is implicit in Datalog.

Limitations in Syntactic Patterns   When encoding bug patterns in JAVADL, we sometimes found that relying directly on syntactic patterns led to code duplication.   For example, matching declarations #d with the enclosing class ⟨:**class** #c {.. #d ..}:⟩, enum ⟨:**enum** #c {.. #d ..}:⟩, or interface ⟨:**interface** #i {.. #d ..}:⟩ requires three separate rules that we cannot combine into one pattern ⟨:#td #c {.. #d ..}:⟩, since the keywords **class**, **interface**, and **enum** do not themselves form a syntactic category for #td in the Java grammar.

Semantic Reasoning   The above limitation materializes especially when we analyze semantic properties that do not cleanly map to a single syntactic construct. For example, the patterns ⟨:**this.**f:⟩ and ⟨:f:⟩ are syntactically distinct, but often semantically identical. In those cases, syntactic patterns quickly become ineffective, and we found that we instead relied on Datalog relations that captured semantic properties.

As the right-most column in Table 1 shows, all but two of our detectors rely on the static name analysis information that we extract from ExtendJ, and three also rely on type analysis information, while only two detectors did not include semantic information from ExtendJ.

While we can export additional information from ExtendJ, detectors can also introduce their own helper predicates and, in principle, share them. After implementing the bug patterns, we surveyed our implementation and identified several helper predicates that we consider potentially reusable (possibly with minor refactoring), such as the subtyping relation, convenience extractors for type and membership information, method signature equality checks, and relations to mark defining and using occurrences of fields. While these observations show the importance of semantic information, all of our detectors used multiple syntactic patterns, and the Complex Operator Precedence detector demonstrated that semantic information is not always sufficient for

real-life bug detectors, as it relied heavily on syntactic information that is no longer visible e.g. in Java bytecode.

Flow-Sensitive Analysis   Some program analyses, such as null-pointer derefer-ence analysis or taint analysis [Arz+14], rely on knowing the program execution order, typically modeled as a control-flow graph (CFG). We have not added CFG informa-tion to JAVADL due to the nontrivial engineering effort for building correct and precise CFGs for a mature language like Java, but see no conceptual obstacle to building a JAVADL CFG predicate library. Alternatively, we can import the CFGs that Riouak et al. [Rio+21]'s recent IntraJ system superimposes over ExtendJ AST for Java 7.[5]

To better understand the importance of flow-sensitive analyses, we approximated their prevalence among the SpotBugs and Error Prone detectors by instrumenting each framework's CFG construction code and running individual detectors to analyze the SpotBugs core classes (967 files, 108kLOC) to see which detectors would trigger CFG construction. For SpotBugs, which groups related detectors into 167 *visitors*, we found that 35 of the 167 visitors (21%) triggered CFG construction, and 3 of Error Prone's 500 checkers (0.6%). These numbers are likely under-approximations but indicate that flow-sensitive analysis is important, but only for a minority of today's bug detectors.

We observed that two of the detectors that we had previously implemented were based on detectors that triggered CFG construction (Table 1, column **Notes**).

RQ1.2: Precision

Table 2 reports on checker precision and recall, comparing the Top-5 SpotBugs and Error Prone detectors against our JAVADL re-implementations. Several of the patterns contain subcategories (Table 1), and we report on the subcategories in cases where our implementation computed them separately. Whenever JAVADL reported bugs that Error Prone or SpotBugs did not report, we manually investigated 10 randomly sam-pled cases (or all, if less than 10), and proceeded analogously when the baseline tools reported a bug that JAVADL did not report.

For the EP-Top-5, most checkers have a high degree of agreement. Lowest (with 77% relative recall) is `UnnecessaryParentheses`. According to our sampling, the mismatch is due to the JAVADL detector not reporting parentheses surrounding unary operators, and Error Prone not reporting on parentheses surrounding case labels and return values.

For Missing `@Override`, JAVADL fails to produce warnings when the overrid-den method is a library method, since ExtendJ fails to evaluate attributes on some AST nodes created from bytecode, which excludes those nodes from the TYPE relation. The recall for `OperatorPrecedence` is due to differences in reporting: for in-stance, for `a && b || c && d`, Error Prone produces two warnings, one per `&&`, while JAVADL produces a single report for the entire expression, whose root is the `||`.

---

[5]This framework was not yet available at the time of our experiments, while ExtendJ's earlier data flow framework [Söd+13] was unmaintained and incompatible with recent versions of ExtendJ.

Turning to the bottom (SB-Top-5) half of the table, JAVADL is able to detect all `DM_STRING_CTOR` and `NM_METHOD_NAMING_CONVENTION` warnings reported by SpotBugs. The lower recall rates on Missing **switch** Default due to over-reporting in SpotBugs, caused by its lack of access to AST data (SpotBugs operates on Java bytecode [Spo]). For `UWF_UNWRITTEN_FIELD`, JAVADL does report SpotBugs' single warning, but at the field's definition, rather than the access location. The other two warnings are unwritten fields that SpotBugs fails to detect. Similarly, we found that JAVADL's warnings for `UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD` were correct.

The low precision of Expose Internal Representation is due to JAVADL assuming that all reference types are mutable, while SpotBugs uses domain knowledge to filter out known immutable classes like `String`.

For `DM_NUMBER_CTOR`, we observed that SpotBugs and JAVADL match on 20 additional reports, with SpotBugs reporting the bug one line off from its source location. The remaining 4 reports that SpotBugs produces are constructor calls where the argument is a cast to `String`. JAVADL produces 12 additional correct warnings.

Overall, none of the differences indicate systematic limitations and could be addressed by evolving the JAVADL bug patterns.

Static Checker Framework (Baseline Tool)		Number of Results			Precision	
Name	**Bug Pattern**	JAVADL	**Tool**	**Common**	**Precision**	**Recall**
EP-Top-5	⋆ Boxed Primitive Constructor	429	425	423	98.60	99.53
	⋆ Missing `@Override`	4533	5341	4393	96.91	82.25
	`OperatorPrecedence`	84	100	82	97.62	82.00
	⋆ == on References	1176	1235	1169	99.40	94.66
	⋆ Useless Type Parameter	99	95	95	95.96	100.00
	`UnnecessaryParentheses`	257	174	134	52.14	77.01
SB-Top-5	`DM_NUMBER_CTOR`	190	182	158	83.16	86.81
	`DM_STRING_CTOR`	5	5	5	100.00	100.00
	⋆ Expose Internal Representation	3546	161	108	3.05	67.08
	`NM_CLASS_NAMING_CONVENTION`	0	0	0	N/A	N/A
	`NM_FIELD_NAMING_CONVENTION`	9	0	0	0.00	N/A
	`NM_METHOD_NAMING_CONVENTION`	1120	38	38	3.39	100.00
	⋆ Missing **switch** Default	223	87	81	36.32	93.10
	`UWF_UNWRITTEN_FIELD`	3	1	0	0.00	0.00
	`UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD`	3	0	0	0.00	N/A

Table 2: Number of results reported by ErrorProne, SpotBugs, and JAVADL for the selected bug patterns on the Defects4J repositories. "Precision" and "recall" are relative to the baseline tool (cf. Section 5.1). We report on bug subcategories for detectors that implement the subcategories separately.

Project	Touched files/commit				Added lines/commit				Deleted lines/commit				Lines of code		Files	
	Mean	M.	90%	Max.	Mean	M.	90%	Max.	Mean	M.	90%	Max.	Start	End	Start	End
Cli	5.30	2	8	149	211.09	16	216	17844	176.54	4	79	19167	0	7070	0	52
Codec	3.15	1	5	170	57.86	6	111	1835	34.13	3	57	1430	13492	19568	85	122
Compress	3.39	1	4	180	70.52	10	171	1969	35.23	3	52	1965	31478	41838	251	341
Csv	1.71	1	3	20	28.97	5	61	1139	20.35	2	29	1152	3597	6117	25	31
J...Core	3.15	2	6	66	93.39	42	235	2839	33.52	9	75	1513	22477	43803	146	256
J...Databind	3.92	2	7	95	71.44	39	169	967	35.33	9	95	800	102361	114590	816	925
Jsoup	2.91	2	6	28	52.44	16	104	5287	20.40	3	32	2168	8854	20403	58	117
Math	7.16	2	12	210	211.95	30	501	28873	143.60	10	173	28291	6122	22749	74	281
Mockito	5.80	3	13	201	130.39	18	123	38603	33.42	8	69	1412	11014	38868	204	532
Time	8.79	2	10	628	583.62	29	472	127673	164.89	3.5	92	43798	45238	176827	240	656

Table 3: Distribution of commit sizes (touched files, added lines, removed lines) and the sizes of the projects at the start and at the end of the commit range. "M." stands for median and "90%" stands for the 90th percentile.

5.3   RQ2: Execution Time Performance

Gathering Data on Run-Time Performance   To assess the efficiency of JAVADL for bug detection in a practical usage scenario, we iterated through a series of consecutive Git commits of Defects4J projects and measured the time that it took to run JAVADL, Error Prone, and SpotBugs on each commit. For these measurements, we included each projects' unit tests, as we would in practice.

To ensure a fair comparison, we configured Error Prone and SpotBugs to only run their respective Top-5 detectors, and ran JAVADL separately for our implementations of SB-Top-5 and EP-Top-5. Unlike Error Prone and SpotBugs, JAVADL supports incremental evaluation, so we ran JAVADL with four configurations in total: {SB-Top-5, EP-Top-5} × {exhaustive, incremental}, with incremental runs re-using the IRDB from the previous incremental run (except for the first run).

For incremental analysis, we ran these experiments on 500 commits of the Defects4J benchmarks[6], and for exhaustive analysis we ran them on 50 equidistant commits from this range, including the first and the last commits. We selected these commits to be the closest predecessors of the commit we used for precision analysis, counting only commits that modified source files.

Table 3 shows the distributions of the number of touched files (modified, added or removed) from the selected sequence, and project sizes in the last four columns. We argue that these numbers show that the projects are realistic software projects at different stages in their life time.

For our performance analysis, we excluded several benchmarks: Chart, since it did not use Git (which our scripts required), and Closure, JxPath and Lang, because some of the commits among the 500 that we examined crashed ExtendJ and corrupted our IRDB. While recovering the IRDB from a backup would be trivial in a production system, we argue that including such runs would impair the representativeness of

---

[6]Except for Cli where the Git history contains only 360 commits.
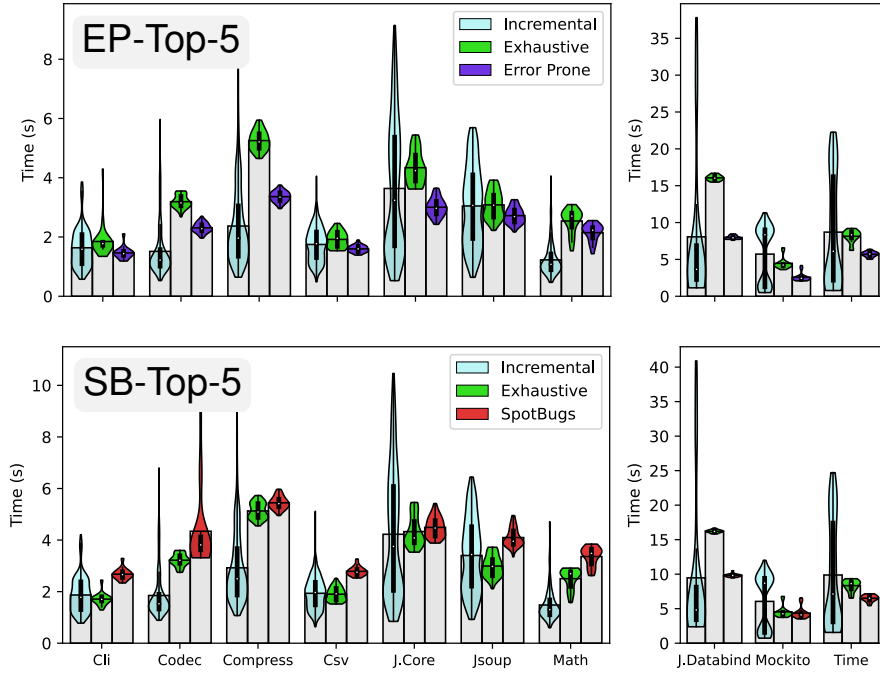
Figure 16: Running time distribution on 500 consecutive commits for EP-Top-5 (above) and SB-Top-5 (below). The shaded boxes represent the mean.

the incremental measurements. For the remaining projects we made a best effort to process them with all tools but found that SpotBugs and Error Prone's dependence on `javac` prevented many comparisons. We adjusted builds by tweaking compiler settings, adding libraries, or excluding problematic files (without examining the impact on our later measurements). Despite our efforts, some projects had a substantial number of failing builds (Cli: 18, Time: 25, Math: 29, Databind: 35, JacksonXML: 42) for the baseline checkers. We excluded all measurements for the affected revisions from our analysis and removed JacksonXML entirely. For incremental runs, we also the removed measurements for the 5 following/preceding revisions.

We ran our experiments on an Intel(R) Core(TM) i7-11700K CPU system, running at a fixed 3.6 GHz with 128 GiB RAM on Ubuntu 18.04.5 with Linux 5.13.7-051307-generic and OpenJDK 11.0.11+9-Ubuntu-0ubuntu2.18.04 inside a Docker container.

RQ2.1: Performance Comparison

Figure 16 shows the distribution of the measured running times for the incremental and exhaustive JAVADL analyses compared to the baseline detectors, i.e., Error Prone (above) and SpotBugs (below).

We observe that overall JAVADL performs competitively to the existing checkers, irrespective of code base size. Our incremental checker often but not universally outperforms the exhaustive checker. The similarity between the performance for JAVADL on the SB-Top-5 and EP-Top-5 suggests that JAVADL's performance depends more on the project structure than on the selection of checkers.

We further observe in the EP-Top-5 diagram that the Math, JacksonDatabind and Time projects have peaks in running times that are more than four times higher than the average. While similar in relative magnitude, the peaks have different causes. For Math, the peak is caused by a merge commit from a release branch (`6ef3b2932f`). For JacksonDatabind, the largest running time due to a merge commit (`33840a208b`) that modifies a significant number of files, though we also observed a set of high measurements caused by commits that modify files that are referenced by many other files (e.g., `DeserializationContext.java` in commit `5b8f0d9923`). The maximum running time for the Time project is caused by a major move/rename commit `53feb3fa56`, affecting 176 out of the total of 656 files in the project. Our brief analysis of the running time peaks also showed that for merge commits and major move/rename commits, the running time is higher than the time needed for the initial run of the analysis. A continuous integration system could use this observation to attempt to predict when it is more efficient to delete the IRDB and re-start from scratch, or to temporarily switch to exhaustive analysis.

Overall, our performance measurements suggest that our prototype framework offers run-time performance comparable to that of state-of-the-practice systems, with neither incremental nor exhaustive mode consistently outperforming the other.

RQ2.2: Incremental Performance

To explain the difference in incremental execution time between the projects, we turn to Figure 17, which shows the average time spent by JAVADL in each phase for the EP-Top-5 (a) and SB-Top-5 (b) detectors. While the running time of the analysis program is overall shorter for the incremental runs (including IRDB read and write), it is also clear that the incremental runs suffer from the overhead of the provenance tracking mechanisms that increase the time spent generating the program representation (*Datalog projection*). The Datalog projection phase grows with number of analyzed files, which includes the number of modified files as well as the files that need to be revisited because they contain an attribute whose value depends on modified files. On slower hard disks, we have also observed a second overhead, for deleting stale IRDB facts (not visible here).

We observed that it is mainly not commits that modify many files that contribute most to high running times, but commits that modify files on which other files' at-
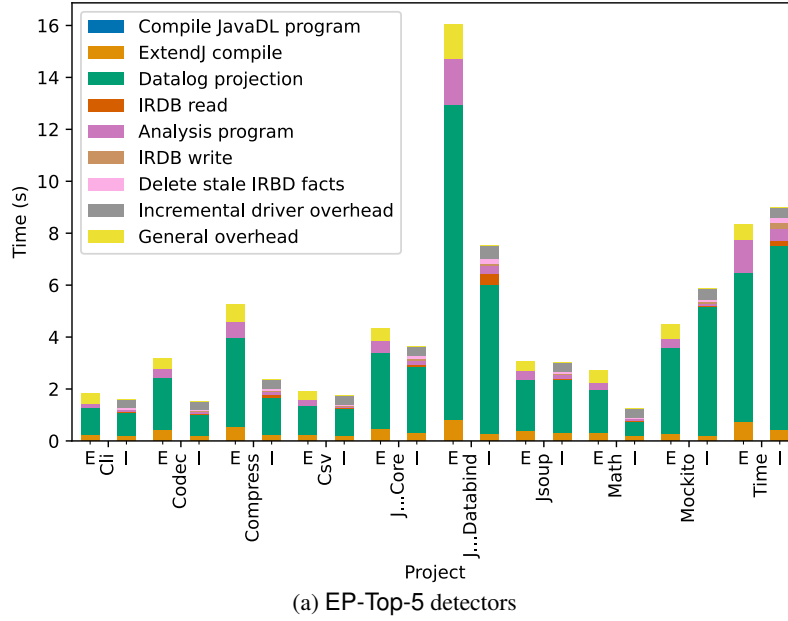
(a) EP-Top-5 detectors
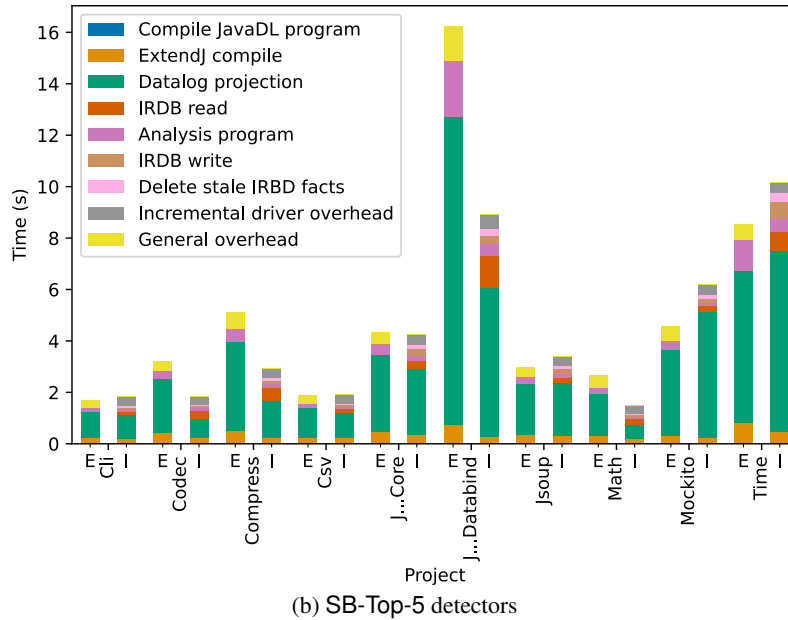
(b) SB-Top-5 detectors

Figure 17: Average running time for exhaustive (E) and incremental (I) runs, split by evaluation phases
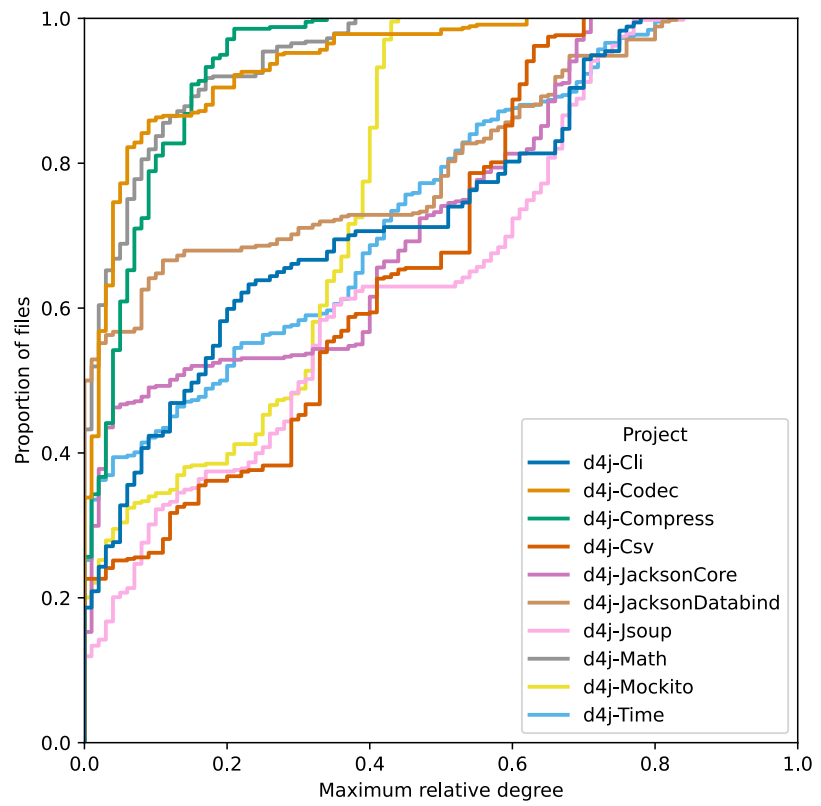
Figure 18: Cumulative distribution of the maximum relative degree in the analyzed commits

tributes depend. We characterize this property with the notion of the *relative degree* of a file F, which is the fraction of the files in the project that contain an attribute whose value was computed using F and whose analyses we must thus re-run when F changes.

Figure 18 shows the cumulative distribution function computed over all the commits in each project. We can observe that for the Codec, Compress and Math projects, 80% of the commits have a degree of at most 0.1 and a further 10% a maximum degree of 0.2. They are followed by JacksonDatabind, which also has about 70% of the commits with a maximum degree of 0.2. These are the same projects in which our incremental analysis is most effective at outperforming the exhaustive analysis. Meanwhile, in Mockito 30% of the commits have a degree of less than 0.1 and 30% of the commits have a maximum degree of 0.4, which is also the maximum for this project, in line with the pronounced bimodal distribution of the running times for the project. These observations indicate that the commit history is correlated with the running times of the incremental analysis, and thus can help inform which evaluation mode to choose for JAVADL for a given commit in a given project.

Analyzing the average running times spent in each phase for SB-Top-5 (Figure 17(b)), we observe that the amount of time needed to read, write and delete facts in the IRBD is more than twice that of the EP-Top-5 detectors. This indicates that the number of facts that are produced in the local evaluation pass but used in the global one is significantly higher in SB-Top-5.

The above observation reflects that we have not yet automated all locality optimizations that we are aware of, and manually optimized only EP-Top-5 for locality. Our main manual optimization was to split rules for locality. For instance, we could compress our earlier example (Figure 11) into:

```
1 BADNEWSTRING(f, l, c) :- n ⟨:new String(#v):⟩, TYPE(#v, τ),
2                          SRC(n, l, c, _, _, f),
3                          τ ⟨:.. class String { .. }:⟩,
4                          SRC(s, _, _, _, _, "java/lang/String.class").
```

However, this rule is global and uses local predicates, so the predicates' tuples would be stored in the IRDB without further optimization. SRC in particular contains one tuple per AST node; materializing it on disk would be expensive. In Figure 11, we avoided this problem by extracting only the source locations of interest into the NEW-STRING predicate, analogously to the Magic Sets transformation [Ban+85]. We expect to automate this process in future work.

5.4 Discussion and Limitations

One limitation of our language that we have so far omitted is that, unlike e.g. Prolog, our Datalog dialect is not Turing-complete but effectively restricted to **PTIME** [Imm99]. While we did not find this restriction limiting in our experiments, it does bar JAVADL from expressing **EXPTIME**-complete analyses (e.g., bounded symbolic execution) or semi-decidable analyses (e.g., Java type analysis [Gri17]). Thus, JAVADL offers less expressivity than afforded by e.g. general-purpose IDE/IFDS-based incrementalization frameworks [AB14]; JAVADL is subject to the usual limitations of Datalog in this

space. We defer to Madsen, Yee, and Lhoták [MYL16] for a detailed discussion.

We emphasize that this restriction is a trade-off: the syntactic simplicity of JAVADL ensures that JAVADL detectors are *clear-box detectors* and can be analyzed and transformed to integrate confidence score computation [Rag+18], to explain their chains of reasoning [ZSS20] or for incrementalization (as shown in this paper), or for integrations of these techniques, e.g., to use knowledge of recent changes to adjust confidence scores.

This restricted expressive power comes with more easily checkable correctness guarantees. For comparison, existing free-form (or black-box) IDE/IFDS-based approaches place complex *semantic* constraints of distributivity on transfer/flow functions [RHS95] to ensure consistent results. Analyses that fail to meet these constraints may work correctly on some inputs but fail in corner cases or when run in different evaluation modes. By contrast, JAVADL projects will (in principle — i.e., not accounting for implementation bugs) evaluate correctly whenever they type-check.

Summary   Overall, our results show that our JAVADL implementation enables both exhaustive and incremental evaluation, with performance comparable to state-of-the-practice tools, and that the JAVADL language can concisely express a variety of both syntactic and semantic checks, both intra- and inter-procedural. While we have only analyzed the concepts that underlie JAVADL on one language (the subset of Java 8 supported by ExtendJ) and on a limited set of bug detectors, we argue that our results yield data points to support the case for clear-box bug detectors as devices for enabling architectural advances in bug checking.

6   Related Work

JAVADL combines four strands of work: domain-specific languages for program analysis, Datalog dialects, pattern matching languages, and systems with alternative but equivalent execution models.

To the best of our knowledge, the first system to demonstrate the connection between DSLs for program analysis, logic programming, and pattern matching was the SOUL system by De Roover et al. [DR+11]. While SOUL does not aim to support multiple execution models or report performance on par with contemporary bug checkers, other aspects of its design are closely related to ours: SOUL provides syntactic patterns in a similar style as JAVADL [DR+07; De 09], albeit restricted to five syntactic categories. It performs logical reasoning through a SmallTalk-based Prolog implementation (rather than Datalog), connects to the Eclipse JDT framework (rather than to ExtendJ), and provides data flow analysis support via Soot [VR+10]. SOUL's use of Prolog results in a top-down evaluation strategy, which permits incremental evaluation and simplifies integration with traditional imperative analysis code, at the cost of performance [Ull89], where Datalog has shown its strength in recent years [Sch+16; BS09]. The SOUL pattern language also omits some language features such as gener-

ics, but captures the majority of the Java syntax supported by Eclipse in 2011, whereas our language is directly based on the grammar and AST of a Java compiler through an automatic transformation (with minimal manual intervention) and thus faithfully captures the entire Java syntax. Unlike SOUL, our work did not yet explore data flow analysis (as we discuss in Section 5.2).

Beyond SOUL, Visser [Vis02] argues in favor of syntactic pattern matching, with Fischer and Visser [FV04] opting for concrete syntax over AST matching by arguing that "the simple abstract syntax approach quickly becomes cumbersome as the schemas become larger". Their AutoBayes system combines syntactic pattern matching with Prolog, compared to which DeepWeaver [Fal+07] adds program rewriting for implementing aspect weaving. Another related system with syntactic pattern matching is our earlier MetaDL [DBR], which analyzes a Datalog dialect, in which it needs to match only one syntactic category. MetaDL uses a different AST encoding that maps each syntactic rule or AST node to a separate predicate (Section 3.3), which we found to be less efficient for Java than our approach here. Like SOUL, MetaDL has not seen a systematic evaluation for bug patterns, unlike JAVADL. A final closely related system is Cohen et al's JTL [CGM06], which provides a Datalog-style language together with a Java-like query syntax for querying Java class files for structural and dataflow information. Similar to JAVADL and SOUL, JTL exposes a number of built-in pre-computed predicates to aid program analysis, though it does not expose the AST and thus limits the amount of reasoning that it allows.

There are many other systems that support syntactic pattern matching. Visser [Vis02] uses GLR parsing for processing syntactic patterns, instead of Earley parsing as in our work. GLR parsing may improve the performance of our JAVADL specification frontend. To address possible bugs or inefficiencies from concrete syntax patterns, Kats, Kalleberg, and Visser [KKV11] offer an interactive tool that suggests AST categories for concrete syntax tree patterns. Similarly, but on the concrete syntax level, Huang, Zook, and Smaragdakis [HZS08] automatically infer the syntactic type of syntactic patterns. We believe that this technique can enable optimizations and improve static checking in our syntactic patterns.

Kats, Bravenboer, and Visser [KBV08] extend syntactic parsing in an extensible compiler that allows language extensions to normalize to mixtures of source code and bytecode, similarly to the Attribute Grammar system Silver [VW+10]. This style of normalization allows specifying analyses and transformations at the concrete syntax tree level while matching at a more abstract level, which (in effect) automatically generalizes analyses and transformations. The trade-off is that the effect of a match or transformation becomes less obvious, and that it may become impossible to match syntactic peculiarities that are not visible at the AST level. For example, JAVADL can distinguish between `(x)` and `((x))`, which is necessary for our Unnecessary-Parentheses bug pattern, but a system that uses matching after normalization may lack suitable information (e.g., in SpotBugs).

While not Datalog, two other closely related domain-specific languages are the program query language PQL [MLL05] and the earlier commercial .QL [DM+07] system

(now CodeQL). Neither system provides support for syntactic matching in the style of JAVADL or SOUL. PQL provides a SQL-like interface that allows not only static analysis, but also dynamic analysis and instrumentation, though it does not provide fine-grained AST / parse tree matching or support for analyzing arithmetic or primitive values. PQL is thus more useful for analyzing dynamic protocols, e.g., to check if a test during execution releases an external resource (e.g., a file handle) exactly once, while JAVADL is more useful for general-purpose static checking. Dynamic information can also be critical for analyzing reflection [LTX19]. For static tools like JAVADL, the lack of dynamic knowledge could be mitigated through ground facts from external tools [Bod+11]. .QL, meanwhile, combines ideas from attribute grammars and SQL, giving it powerful static analysis capabilities, but without JAVADL-style syntactic matching capabilities.

While Datalog and its dialects have aided program analysis for over a decade, most existing tools like DOOP [BS09] rely on a separate fact extraction mechanism. DOOP's Java fact extractors translate either a Soot- or WALA-specific IR [VR+10; FD12] to a common representation, encoded as a set of predicates. This predicate-based Datalog IR encodes enough information for DOOP's points-to and call graph analyses, but (1) lacks source locations (necessary for precise error reports), (2) represents the program under analysis linearly, without any notion of nesting of type definitions, blocks or expressions (necessary for syntactic checks such as Complex Operator Precedence), and (3) hardcodes the set of predicates that it exposes to Datalog and requires detail knowledge of DOOP internals and Soot or WALA IRs to evolve or maintain this fact extraction code. By contrast, JAVADL provides the analysis code with a full representation of the analyzed program, which enables a broad range of static checks.

The literature has proposed several extensions to Datalog such as general-purpose lattices in Flix [MYL16] and IncA [Sza+18]. These valuable extensions are orthogonal to ours. While JAVADL automates fact extraction by directly exposing the program AST, Basten and Klint [BK08] propose a language-parametric scheme for using AST annotations to determine which facts to extract. We hypothesize that JAVADL could automate these AST annotations through bug pattern meta-analysis, to reduce the number of facts that we project from ExtendJ to Soufflé.

Earlier Prolog-based approaches to program analysis include Janzen and De Volder's JQuery system [JDV03], based on Prolog extended with aspect pointcut-style predicates and intended for Java code browsing, and a SOUL-like system Eichberg et al. [Eic+07], without syntactic matching but with Eclipse IDE integration and Prolog-based incrementalization. Due to the different (top-down vs. bottom-up) evaluation modes, their techniques are not applicable to JAVADL. By using Prolog, both systems are in principle Turing complete, which may enable them to express more powerful analyses but loses the termination guarantee and bottom-up evaluation strategy (more efficient for whole-program analysis) that Datalog provides. Overall, the perhaps earliest discussion of logic programming for program analysis is due to Reps [Rep95], who focused on incrementalizing analysis through the Magic Sets transformation [Ban+85].

Attribute Grammars [Knu68] are another declarative programming paradigm for program analysis. Tools such as AbleC [Kam+17] for C and ExtendJ (formerly JastaddJ) for Java [EH07; ÖH13], which underlies JAVADL, provide declarative features for computing attributes of AST nodes by synthesizing information from other AST nodes (including other attributes). Computations can be functional (in AbleC, based on Silver [VW+10]) or imperative (in ExtendJ, based on JastAdd [HM03]). Unlike Datalog, attribute grammar systems provide special support for reasoning over tree structures. Silver provides support for syntactic pattern matching, but (to the best of our knowledge) no support for computations over general-purpose relations, while JastAdd does not provide syntactic pattern matching support, but can support general-purpose relations [Mey+18] (a feature not yet explored for purposes of program analysis). Attribute grammars have been used for bug finding [Söd+13], and some analyses may be easier to encode in attribute grammars. If so, JAVADL can import them through ExtendJ attributes. Attribute Grammars support multiple alternative execution models, like Datalog. *Reference attribute* grammars, supported in JastAdd, are evaluated on-demand [HM03] and can be evaluated both concurrently [ÖH17] and incrementally [SH12].

Arzt and Bodden [AB14] have demonstrated incremental program analysis for (distributive) IFDS- and IDE-based analyses in their REVISER system, assuming the presence of a program differencing algorithm, and CHEETAH by Do et al. [Do+17] demonstrates related techniques to prioritizing local bug detection during live editing. Their approaches to granularity are more fine-grained than ours: REVISER operates on CFG node differences, and CHEETAH's notions of locality cover eight different layers, of which our file level is only one. Neither system supports syntactic matching. They focus exclusively on semantic properties, and both assume existing facilities to map the object language to graphs for flow analysis, whereas JAVADL only requires an AST.

Lhoták and Hendren [LH04]'s Jedd system takes a near-converse approach to ours: they extend Java with special support for operations over relations and SAT solving, and use it in the context of Soot [VR+10] for imperative program analyses. More recently, Opal [Hel+20] demonstrated complex and high-performance program analyses by combining imperative implementations of program analyses in a Datalog-like blackboard architecture. While the analyses themselves are not clear-box specifications, Opal obtains some ability to alternate execution modes from hand-written meta-information and manual incrementalization. Other program analysis systems such as Soot [VR+10], WALA [FD12], or Spoon [Paw+16] provide program analysis facilities as regular Java libraries.

7 Conclusion

We have introduced JAVADL, the first static checker framework (to the best of our knowledge) that can run any **PTIME**-computable static bug detector on Java from a single specification both exhaustively and incrementally, while automatically rewriting

the specification to optimize it for incremental evaluation. Our tool combines syntactic patterns for local reasoning with declarative, Datalog-style nonlocal reasoning. We have demonstrated that the efficiency of our prototype implementation, based on an existing high-performance Datalog engine, is competitive to state-of-the-practice systems, that our specification language can concisely express typical bug detectors, and that its incremental and exhaustive evaluation are both able to outshine each other in different usage scenarios. We argue that our results demonstrate the value and viability of *clear-box bug checker frameworks*, which constrain the bug detector specification language in order to obtain the ability to analyze and transform detector specifications for different usage modes.

Acknowledgements

References

[Aft+12]   Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. "Building Useful Program Analysis Tools Using an Extensible Java Compiler". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012.

[AB14]   Steven Arzt and Eric Bodden. "Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Association for Computing Machinery, 2014.

[Arz+14]   Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *Acm Sigplan Notices* 49.6 (2014).

[Aye+08]   Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and Johan Penix. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008).

[BS16]   George Balatsouras and Yannis Smaragdakis. "Structure-sensitive points-to analysis for C and C++". In: *International Static Analysis Symposium*. Springer. 2016.

[Ban+85]   Francois Bancilhon, David Maier, Yehoshua Sagiv, and
            Jeffrey D Ullman. "Magic sets and other strange ways to implement
            logic programs". In: *Proceedings of the fifth ACM SIGACT-SIGMOD
            symposium on Principles of database systems*. 1985.

[BK08]   Bas Basten and Paul Klint. "DeFacto: Language-Parametric Fact
            Extraction from Source Code". In: *Revised Selected Papers of the First
            International Conference on Software Language Engineering*. Vol. 5452.
            Lecture Notes in Computer Science. Springer International Publishing,
            2008.

[Bod+11]   Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and
            Mira Mezini. "Taming reflection: Aiding static analysis in the presence
            of reflection and custom class loaders". In: *2011 33rd International
            Conference on Software Engineering (ICSE)*. IEEE. 2011.

[Bra+14]   Simon Brandhof et al. *SonarQube*. 2014. URL:
            `https://github.com/SonarSource/sonarqube`.

[BS09]   Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative
            specification of sophisticated points-to analyses". In: *Proceedings of
            OOPSLA '09*. ACM, 2009.

[Bur+21]   Oliver Burn et al. *Checkstyle 9.0*. 2021. URL:
            `https://checkstyle.sourceforge.io/`.

[Cal+15]   Cristiano Calcagno et al. "Moving fast with software verification". In:
            NASA Formal Methods Symposium. Springer. 2015.

[CGT89]   Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What you always
            wanted to know about Datalog (and never dared to ask)". In: *IEEE
            transactions on knowledge and data engineering* 1.1 (1989).

[CGM06]   Tal Cohen, Joseph Gil, and Itay Maman. "JTL: the Java tools language".
            In: *ACM SIGPLAN Notices* 41.10 (2006).

[Cop05]   Tom Copeland. *PMD applied*. Vol. 10. Centennial Books Arexandria, Va,
            USA, 2005.

[DM+07]   Oege De Moor et al. ".QL: Object-oriented queries made easy". In:
            *International Summer School on Generative and Transformational
            Techniques in Software Engineering*. Springer. 2007.

[De 09]   Coen De Roover. "A Logic Meta Programming Foundation for
            Example-Driven Pattern Detection in Object-Oriented Programs".
            PhD thesis. Vrije Universiteit Brussel, 2009.

[DR+07]   Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and
            Laurence Duchien. "Behavioral similarity matching using concrete
            source code templates in logic queries". In: *Proceedings of the 2007
            ACM SIGPLAN symposium on Partial evaluation and semantics-based
            program manipulation*. 2007.

[DR+11]   Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers.
          "The SOUL Tool Suite for Querying Programs in Symbiosis with
          Eclipse". In: *Proceedings of the 9th International Conference on
          Principles and Practice of Programming in Java*. PPPJ '11. Association
          for Computing Machinery, 2011.

[Do+17]   Lisa Nguyen Quang Do et al. "Just-in-Time Static Analysis". In:
          *Proceedings of the 26th ACM SIGSOFT International Symposium on
          Software Testing and Analysis*. ISSTA 2017. Association for Computing
          Machinery, 2017.

[DBR]   Alexandru Dura, Hampus Balldin, and Christoph Reichenbach.
          "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th
          ACM SIGPLAN International Workshop on the State of the Art in
          Program Analysis*. SOAP 2019.

[DRS21a] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg.
          "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection".
          In: *Proceedings of the ACM on Programming Languages, Volume 5,
          Issue OOPSLA* (2021).

[DRS21b] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg.
          JavaDL: Automatically Incrementalizing Java Bug Pattern Detection.
          2021. URL: https://doi.org/10.5281/zenodo.5090141.

[DRS21c] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg.
          "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection".
          In: (2021). (artifact, updated after OOPSLA AEC evaluation).

[Eic+07] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and
          Klaus Ostermann. "Automatic incrementalization of prolog based static
          analyses". In: *International Symposium on Practical Aspects of
          Declarative Languages*. Springer. 2007.

[EH07]   Torbjörn Ekman and Görel Hedin. "The JastAdd Extensible Java
          Compiler". In: *SIGPLAN Not.* 42.10 (2007).

[Fal+07] Henry Falconer et al. "A Declarative Framework for Analysis and
          Optimization". In: *Compiler Construction*. Springer Berlin Heidelberg,
          2007.

[FD12]   Stephen Fink and Julian Dolby. *WALA–The TJ Watson Libraries for
          Analysis*. 2012.

[FV04]   Bernd Fischer and Eelco Visser. "Retrofitting the AutoBayes Program
          Synthesis System with Concrete Syntax". In: *Domain-Specific Program
          Generation: International Seminar, Dagstuhl Castle, Germany, March
          23-28, 2003. Revised Papers*. Springer Berlin Heidelberg, 2004.

[Gri17]   Radu Grigore. "Java Generics are Turing Complete". In: *ACM SIGPLAN
          Notices* 52.1 (2017).

[GMS93]   Ashish Gupta, Inderpal Singh Mumick, and
            Venkatramanan Siva Subrahmanian. "Maintaining views incrementally".
            In: *ACM SIGMOD Record* 22.2 (1993).

[HP18]   Andrew Habib and Michael Pradel. "How Many of All Bugs Do We
            Find? A Study of Static Bug Detectors". In: *Proceedings of the 33rd
            ACM/IEEE International Conference on Automated Software
            Engineering*. 2018.

[HM03]   Görel Hedin and Eva Magnusson. "JastAdd: An Aspect-oriented
            Compiler Construction System". In: *Sci. Comput. Program.* 47.1 (2003).

[Hel+20]   Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and
            Mira Mezini. "Modular Collaborative Program Analysis in OPAL". In:
            *Proceedings of the 28th ACM Joint Meeting on European Software
            Engineering Conference and Symposium on the Foundations of Software
            Engineering*. ESEC/FSE 2020. Association for Computing Machinery,
            2020.

[Heo+19]   Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik.
            "Continuously reasoning about programs using differential Bayesian
            inference". In: *Proceedings of the 40th ACM SIGPLAN Conference on
            Programming Language Design and Implementation, PLDI 2019,
            Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019.

[HZS08]   Shan Shan Huang, David Zook, and Yannis Smaragdakis.
            "Domain-Specific Languages and Program Generation with
            Meta-AspectJ". In: *ACM Trans. Softw. Eng. Methodol.* 18.2 (2008).

[Imm99]   N. Immerman. *Descriptive Complexity*. Graduate texts in computer
            science. Springer New York, 1999.

[JDV03]   Doug Janzen and Kris De Volder. "Navigating and querying code
            without getting lost". In: *Proceedings of the 2nd international conference
            on Aspect-oriented software development*. 2003.

[JJE14]   René Just, Darioush Jalali, and Michael D Ernst. "Defects4J: A database
            of existing faults to enable controlled testing studies for Java programs".
            In: *Proceedings of the 2014 International Symposium on Software
            Testing and Analysis*. 2014.

[Kam+17]   Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk.
            "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).

[KKV11]   Lennart C. L. Kats, Karl T. Kalleberg, and Eelco Visser. "Interactive
            Disambiguation of Meta Programs with Concrete Object Syntax". In:
            Software Language Engineering. Springer Berlin Heidelberg, 2011.

[KBV08]   Lennart C.L. Kats, Martin Bravenboer, and Eelco Visser. "Mixing Source and Bytecode: A Case for Compilation by Normalization". In: *SIGPLAN Not.* 43.10 (2008).

[Knu68]   Donald E. Knuth. "Semantics of context-free languages". In: *Theory of Computing Systems* 2.2 (1968).

[LH04]   Ondřej Lhoták and Laurie Hendren. "Jedd: a BDD-based relational extension of Java". In: *ACM SIGPLAN Notices* 39.6 (2004).

[LTX19]   Yue Li, Tian Tan, and Jingling Xue. "Understanding and Analyzing Java Reflection". In: *ACM Trans. Softw. Eng. Methodol.* 28.2 (2019).

[MYL16]   Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. "From Datalog to Flix: a Declarative Language for Fixed Points on Lattices". In: *ACM SIGPLAN Notices* 51.6 (2016).

[MLL05]   Michael Martin, Benjamin Livshits, and Monica S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. Association for Computing Machinery, 2005.

[Mey+18] Johannes Mey et al. "Continuous model validation using reference attribute grammars". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 2018.

[NRL18]   Krishna Narasimhan, Christoph Reichenbach, and Julia Lawall. "Cleaning up copy–paste clones with interactive merging". In: *Automated Software Engineering* 25.3 (2018).

[O'H19]   Peter O'Hearn. "Separation logic". In: *Communications of the ACM* 62.2 (2019).

[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*. 2013.

[ÖH17]   Jesper Öqvist and Görel Hedin. "Concurrent circular reference attribute grammars". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 2017.

[Paw+16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "Spoon: A library for implementing analyses and transformations of java source code". In: *Software: Practice and Experience* 46.9 (2016).

[Rag+18]   Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and
            Mayur Naik. "User-guided program reasoning using Bayesian
            inference". In: *Proceedings of the 39th ACM SIGPLAN Conference on
            Programming Language Design and Implementation, PLDI 2018,
            Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.

[Rei21]   Christoph Reichenbach. "Software Ticks Need No Specifications".
            English. In: *Proceedings of the 43rd International Conference on
            Software Engineering: New Ideas and Emerging Results Track*. 43rd
            International Conference on Software Engineering: Software
            Engineering in Practice, ICSE 2021 ; Conference date: 23-05-2021
            Through 29-05-2021. IEEE - Institute of Electrical and Electronics
            Engineers Inc., 2021.

[Rep95]   Thomas W Reps. "Demand interprocedural program analysis using logic
            databases". In: *Applications of Logic Databases*. Springer, 1995.

[RHS95]   Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise
            interprocedural dataflow analysis via graph reachability". In:
            *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on
            Principles of programming languages*. 1995.

[Rey02]   J.C. Reynolds. "Separation logic: a logic for shared mutable data
            structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in
            Computer Science*. 2002.

[Rio+21]   Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. "A
            Precise Framework for Source-Level Control-Flow Analysis". In: *21st
            IEEE International Working Conference on Source Code Analysis and
            Manipulation, SCAM 2021*. IEEE Computer Society, 2021.

[Sad+15]   Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and
            Collin Winter. "Tricorder: Building a Program Analysis Ecosystem". In:
            International Conference on Software Engineering (ICSE). 2015.

[Sch+16]   Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann.
            "On Fast Large-scale Program Analysis in Datalog". In: *Proceedings of
            the 25th Int. Conf. on Compiler Construction*. CC 2016. ACM, 2016.

[Sco08]   Elizabeth Scott. "SPPF-style parsing from Earley recognisers". In:
            Electronic Notes in Theoretical Computer Science 203.2 (2008).

[Söd+13]   Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson.
            "Extensible intraprocedural flow analysis at the abstract syntax tree
            level". In: *Science of Computer Programming* 78.10 (2013).

[SH11]   Emma Söderberg and Görel Hedin. "Automated Selective Caching for
            Reference Attribute Grammars". In: *Software Language Engineering*.
            Springer Berlin Heidelberg, 2011.

[SH12]   Emma Söderberg and Görel Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Tech. rep. 98. LU-CS-TR:2012-249, ISSN 1404-1200. Lund University, 2012.

[Spo]   *SpotBugs 4.4.1 Bug Descriptions*. 2021. URL: https://spotbugs. readthedocs.io/en/stable/bugDescriptions.html.

[Sza+18]   Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. "Incrementalizing Lattice-Based Program Analyses in Datalog". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018).

[Ull89]   J. D. Ullman. "Bottom-up Beats Top-down for Datalog". In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '89. Association for Computing Machinery, 1989.

[VR+10]   Raja Vallée-Rai et al. "Soot: A Java Bytecode Optimization Framework". In: *CASCON First Decade High Impact Papers*. CASCON '10. IBM Corp., 2010.

[VW+10]   Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. "Silver: an Extensible Attribute Grammar System". In: *Science of Computer Programming* 75.1–2 (2010).

[Vas+20]   Carmine Vassallo et al. "How developers engage with static analysis tools in different contexts". In: *Empirical Software Engineering* 25 (2 2020).

[Vis02]   Eelco Visser. "Meta-Programming with Concrete Object Syntax". In: *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. GPCE '02. Springer-Verlag, 2002.

[VSK89]   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)*. 1989.

[ZSS20]   David Zhao, Pavle Subotić, and Bernhard Scholz. "Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42.2 (2020).

CLOG: A DECLARATIVE LANGUAGE FOR C STATIC CODE CHECKERS

Abstract

We present CLOG, a declarative language for describing static code checkers for C. Unlike other extensible state-of-the-art checker frameworks, CLOG enables powerful interprocedural checkers without exposing the underlying program representation: CLOG checkers consist of Datalog-style recursive rules that access the program under analysis via syntactic pattern matching and control flow edges only. We have implemented CLOG on top of Clang, using a custom Datalog evaluation strategy that piggy-backs on Clang's AST matching facilities while working around Clang's limitations to achieve our design goal of representation independence.

Our experiments demonstrate that CLOG can concisely express a wide variety of checkers for different security vulnerabilities, with performance that is similar to Clang's own analyses and highly competitive on real-world programs.

1 Introduction

While the C programming language enforces certain correctness properties that all C programs must satisfy, C programmers have been utilizing supplementary static check-

ers to enforce additional constraints for most of the language's existence — the release of C in 1973 [Rit93] was followed by the release of `lint` only 4 years later [Joh77].

Since then, static program analysis has made substantial advances. Modern software engineers can draw from frameworks that offer interprocedural data flow analyses (e.g., *Phasar* [SHB19]), complex call graph and points-to analyses with intricate inter-dependencies (e.g., *cclyzer* [BS16]) and efficient memory models via separation logic (*Infer* [O'H19]). In practical software development, developers may also adopt checker frameworks that trade precision or soundness for speed, for easier integration into the build workflow, such as the *Clang Static Analyzer* [1], *clang-tidy*[2] and *CppCheck*.[3]

However, most of these tools (*Phasar*, *Infer*, *Clang Static Analyzer*, *clang-tidy*, *CppCheck*) are not easily extensible: they only supply a fixed set of built-in *general-purpose* checkers. Thus, they are not designed to tackle the increasing reliance of modern software on external libraries, to support custom checks for internal APIs, or to incorporate custom, project-specific tweaks to existing analyses. Adding a new analysis or customizing an existing one depends on the internal representation of the program used by the tool (usually an abstract syntax tree (AST) or a 3-address intermediate representation (IR)), and these internal representations often evolve as the underlying tool itself evolves. For example, Clang offers a command-line interface for AST pattern matching (*clang-query*) that can find program locations corresponding to a user-defined pattern, but users must express these patterns in Clang's idiosyncratic pattern language.

Several recent tools, *Coccinelle* [LM18], *CodeQL* [Avg+16], and *cclyzer*, therefore explicitly offer domain-specific languages that enable software engineers to supply their own bug patterns or to tweak existing ones. We observe that these tools split bug detection into two *phases*:

1. *First phase*: moving from the concrete domain (AST or IR) to an abstract domain

2. *Second phase*: combine information in the abstract domain to derive the conclusions, computing fixpoints as needed

Both *cclyzer* and *CodeQL* provide a declarative, Datalog-style, analysis description that facilitates the development of custom analyses, but only in the second phase of the analyzer. The analyses are still dependent on the internal representation of the analyzed program, be it an AST or an IR. *Coccinelle*'s code pattern language, meanwhile, allows developers to specify *syntactic patterns* that resemble C code, with various pattern extensions e.g. for intraprocedural control flow dependencies between patterns. More complex connections between patterns (e.g., those that require fixpoints) require custom Python or OCaml code.

In this paper, we introduce CLOG, a declarative language that combines Datalog-style reasoning with *Coccinelle*-style syntactic pattern matching over the C language.

---

[1] https://clang-analyzer.llvm.org/
[2] https://clang.llvm.org/extra/clang-tidy/
[3] https://cppcheck.sourceforge.io/

For the first analysis phase, syntactic patterns allow us to describe C code patterns, without exposing the internal representation, unlike *cclyzer* and *CodeQL*. For the second analysis phase, Datalog-style reasoning allow us to freely combine syntactic patterns across arbitrary flow and dependency edges, avoiding the need to escape to scripting languages, as in *Coccinelle*, and enabling integration with the existing body of work on points-to analysis and context-sensitivity in Datalog.

To illustrate our approach, consider a common warning between static code checkers: the use of `goto` statement (Figure 1). Although there is variation in what the tools consider an admissible use of `goto`s, both *clang-tidy* and *CodeQL* agree on discouraging back-jumps (Figure 1a).

```
1 void f(void) {
2 loop:
3   if (1)
4     goto loop;         (a)
5 }
```

```
1 int g(int cond) {
2   int exit_code = 0;
3   if (cond) {
4     exit_code = 1;
5     goto error_exit;
6   }
7   return 0;
8 error_exit:
9   cleanup();
10   return exit_code;   (b)
11 }
```

```
1 int h(int cond) {
2   if (cond)
3     return 0;
4   else
5     goto error_exit;
6   error_exit:
7   // cleanup();
8   return 1;            (c)
9 }
```
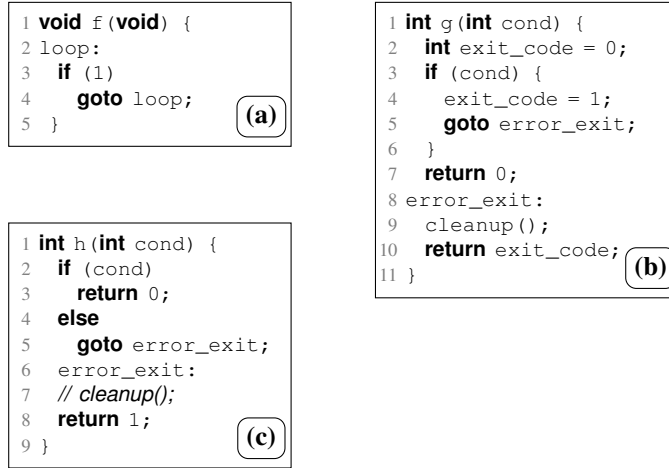
Figure 1: Examples for the `goto` check. Case (a) is a back-jump. Case (b) is a forward jump to the single label in the function. Case (c) is a forward jump to a label followed by a return.

In Figure 2 we show a possible approach for implementing a back-jump checker in *CodeQL*. The checker collects all `goto` statements in a relation `GotoStmt` and all label statements in `LabelStmt` (line 1), from these relations selects the `goto` and label statements that refer to the same label (line 3), compares their source locations (line 4), and, if successful, generates a report (line 5). While this checker provides a concise description of the check, it still relies on the GOTOSTMT and LABELSTMT relations, which are defined externally. The checker writer must be aware about how the `goto` and label statements are represented in the abstract syntax tree (AST) and what their attributes are (e.g., `.getTarget()`).

In contrast to *CodeQL*, CLOG aims to hide implementation details internal to the analysis system such as the AST nodes and their attributes and introduces *syntactic patterns* to match arbitrary terms of the analyzed programs, including, but not limited to, terms representing single AST nodes.

```
1 from GotoStmt goto, LabelStmt label
2 where
3   goto.getTarget() = label and
4   label.getLocation().getStartLine() < goto.getLocation().getStartLine()
5 select goto, "Goto jumps to a label that appears before the goto."
```

Figure 2: Implementation of a `goto` check in CodeQL

```
1 WARNBACKWARDGOTO(g, l) :-
2   g ⟨:goto $label; :⟩,
3   l ⟨:$label :  $s:⟩,
4   src_line_start(g) > src_line_start(l).
```

Figure 3: Implementation of a `goto` check in CLOG

Figure 3 depicts the CLOG implementation of the same `goto` check, where we use syntactic patterns instead of built-in relations. Following Datalog notation, `:-` represents logical right-to-left implication ($\Leftarrow$) and commas represents conjunction. The syntactic pattern in line 2 matches all `goto` statements in the program, binds them to variable `g` and their labels to `$label`. Analogously, the pattern in line 3 matches all label statements `l`, with the same label, `$label`. To distinguish between program names and Datalog variables, we prefix variables used inside patterns with the `$` sign, thus `$label` can bind to any label in the program and `$s` to any statement. Since these are variables of the analysis program, we call them *metavariables*. On line 4, the program compares source line numbers, and if, this succeeds, it adds the tuple (`g`, `l`) to the WARNBACKWARDGOTO relation.

Syntactic patterns are not limited to single statements, but we can freely compose them as long as the result is a statement, expression, declaration or definition. For example, a program that detects labels before a `return` statement is:

```
1 LABELEDRETURN(l) :- l ⟨:$label :  return $r; :⟩.
2 LABELEDRETURN(l) :- l ⟨:$label :  return; :⟩.
```

Such a program may warn about missing cleanup code or hint that a return can be used directly instead of a `goto` (Figure 1c).

To enable our framework to handle industrial-quality code, including large code bases and C language extensions, we have implemented its frontend on top of Clang. This allows us to combine state-of-the-art Datalog evaluation techniques with a custom *pattern embedding* strategy that offloads parts of syntactic pattern matching and semantic analysis to Clang's own pattern matching and analysis facilities, on demand.

To summarize, our contributions are:

1. CLOG, a declarative language that combines syntactic patterns and Datalog-style reasoning for program analysis of C programs;

2. A prototype implementation of CLOG[4];

---

[4] https://github.com/lu-cs-sde/clog

3. An execution strategy that allows us to automatically offload parts of most CLOG analyses to Clang;

4. A comparison of speed and quality between 5 analyses implemented in CLOG and the Clang Static Analyzer, with a validated artifact [DR24].

2   The CLOG Language

We introduce the CLOG language through two examples that illustrate both the language and the workflow that we use to develop static checkers in CLOG. We first show how we can construct recursive inclusion-based analyses (Section 2.1) to catch misuses of a typical internal API, and then show how CLOG's built-in knowledge about control flow (Section 2.2) can help identify violations of an API protocol for an external library. Sections 2.3-2.6 then give a full overview over the language.

2.1   Recursive Patterns: Arena Allocators

To illustrate how CLOG can help developers find bugs related to internal APIs, we look at arena allocators [Cha+17]. These custom allocators can speed up memory allocation and (bulk) deallocation. Since arenas can be used to store temporary data structures that have a shorter lifetime than the program, pointers allocated in them may coexist along pointers allocated using `malloc`. However, calling `free` on an arena-allocated pointer is undefined behavior, so it is important that the two kinds of pointers are not confused in the program. In Figure 4 we provide three examples of such API misuse.

```
 1 void entry0(arena *ma) {
 2   int *p = aalloc(ma, sizeof(int));
 3   int *q = malloc(sizeof(int));
 4   // do work
 5   free(p);
 6   free(q);
 7 }
 8 void cleanup1(int *x, int *y) {
 9   free(x);
10   free(y);
11 }
12 void entry1(arena *ma) {
13   int *p = aalloc(ma, sizeof(int));
14   int *q = malloc(sizeof(int));
15   // do work
16   cleanup1(p, q);
17 }

18 int * alloc21(arena *ma) {
19   int *p = aalloc(ma, sizeof(int));
20   return p;
21 }
22 int * alloc22(arena *ma) {
23   int *p = malloc(sizeof(int));
24   return p;
25 }
26 void entry2(arena *ma) {
27   int *p = alloc21(ma);
28   int *q = alloc22(ma);
29   // do work
30   free(p);
31   free(q);
32 }
```

Figure 4: Wrong uses of an arena allocator API

```
1 ExprPointsToArena(e)  :- e ⟨:aalloc(..):⟩.
2 ExprPointsToArena(e)  :- e ⟨:($t) $f:⟩, ExprPointsToArena($f).
3 VarPointsToArena($p)  :- ⟨:$t *$p = $e:⟩, ExprPointsToArena($e).
4 VarPointsToArena(p)   :- ⟨:$p = $e:⟩, p = decl($p), p != undef,
5     ExprPointsToArena($e).
6 ExprPointsToArena($p) :- ⟨:$p:⟩, p = decl($p), p != undef,
7     VarPointsToArena(p).
8 FreeOfArenaPtr($p)    :- ⟨:free($p):⟩, ExprPointsToArena($p).
```

Figure 5: Intra-procedural check for `free`-ing arena-allocated memory.

In Figure 5, we give an analysis that can detect the first of these cases. Like all CLOG programs, the analysis consists of a set of extended Horn clauses that conceptually take the form $P_0(\overline{x_0}) :- P_1(\overline{x_1}), \ldots, P_n(\overline{x_n})$ where the P_i are the names of user-defined or built-in relations (or, equivalently, *predicates*). These rules follow the usual Datalog semantics: Let $\mathcal{X}$ be the set of all variables that occur in $\overline{x_0}, \ldots, \overline{x_n}$. Whenever we have a function ρ that maps each $x \in \mathcal{X}$ to a constant such that for all $i \in 1, \ldots n$, the relation P_i contains the tuple $\rho(\overline{x_i})$, then the relation P_0 also contains the tuple $\rho(\overline{x_0})$.

In CLOG, $P_i(\overline{x_i})$ may also represent a syntactic pattern (for $i \neq 0$). For example, in Figure 5, line 1 states that if e is a specific element of the program under analysis that has the syntactic form ⟨:aalloc(..):⟩ (where '..' is a wildcard that matches any sequence of elements), then we must conclude that ExprPointsToArena(e) is true. Syntactic patterns like e ⟨:aalloc(..):⟩ behave like predicates quantified over the entire program under analysis, that is: a syntactic pattern matches if there exists a substitution of the pattern's metavariables with terms from the analyzed program such that the resulting term is present in the analyzed program.

We follow most Datalog dialects in further extending the language with comparison predicates, represented inline by the operators (e.g. >, ==, etc.), pure functions (e.g. `src_line_start` which maps a program term to its source location) and stratified negation (using the '!' operator).

To catch the first misuse case (line 5), we describe an intra-procedural, flow-insensitive analysis in Figure 5. In line 2, we also add the cast expressions to the same relation. Line 3 introduces a new relation, VarPointsToArena, which contains all the pointer variables pointing to arena-allocated memory. First, we add all variables that are initialized by an expression that points to an arena. In line 4, we handle variable assignments, which are matched by the ⟨:$p = $e:⟩ pattern, where $p is bound to an identifier and we use the built-in function `decl` to look-up the corresponding declaration, p. If the look-up is successful, and the right-hand side of the assignment points to an arena, then we deduce that the variable p also points to an arena. Line 6 contains the dual of the previous rule, stating that if variable points to an arena, then a reference to that variable, ⟨:$p:⟩, is an expression pointing to an arena. We observe that there is a circular dependency between the VarPointsToArena and ExprPointsToArena relations - this is handled by the fix-point semantics of Datalog. Finally, in line 8, we

collect all the problematic calls to `free` on an arena-allocated pointer in the relation FREEOFARENAPTR.

As hinted earlier, this checker does not catch cases where the allocation and the call to `free` are happening in different functions. To catch the error on line 9, we need to track the flow of values through function calls. We achieve this by appending two rules to the program.

```
 9 CALL(call, $actual, $formal) :- call ⟨: $c(.., $actual, ..)  :⟩,
10    $callee = decl($c),
11    ⟨:$rt $callee(.., $pt $formal, ..)  { .. }:⟩,
12    index($formal) == index($actual).
13 VARPOINTSTOARENA(f) :- CALL(_, e, f), EXPRPOINTSTOARENA(e).
```

The first rule (line 9) defines the CALL predicate which contains the `formal` and `actual` argument pairs for each `call` expression. We use the `index` function to retrieve the index of a term in a list, in this case the index of the actual and formal argument. The second rule (line 13) states that if an actual argument is an expression pointing to an arena, then the formal argument points to an arena.

To catch the error on line 30, we also have to track values through function returns. Another two rules materialize this:

```
14 RETURN(call, $val) :- call ⟨:$c(..):⟩,
15    $callee = decl($c),
16    @$callee ⟨:return $val;:⟩.
17 EXPRPOINTSTOARENA(call) :- RETURN(call, e), EXPRPOINTSTOARENA(e).
```

The rule on line 14 defines the RETURN predicate which maps all call expressions to the expressions returned by the `$callee`. On line 16 we illustrate the use of restricted syntactic patterns, where the pattern @`$callee` ⟨:`return` `$val`; :⟩ matches only the `return` statements from `$callee`.

We observe that we can develop the checker incrementally, by appending new rules, to transform it into an inter-procedural analysis. Although the checker is far from being sound or complete, it already covers interesting cases and it is a starting point for incremental development, for example by adding call-site sensitivity. Adding various flavors of context-sensitivity to Datalog analyses has already been demonstrated by tools like *Doop* [BS09], thus we do not explore this direction as part of this work.

2.2   Control Flow: API Protocol for MPI

A common restriction is that the API functions need to be called in a certain order. For example, the OpenMPI library expects that each call to the non-blocking send function `MPI_Isend` is followed by a call to `MPI_Wait` (or similar). In Figure 6 we show 4 examples of API misuse, where the calls to `MPI_Isend` are not paired with calls to `MPI_Wait` with the same request handle, `req`.

```
1 void good(void) {
2   MPI_Request req;
3   MPI_Isend(ARGS, &req);
4   // do work
5   MPI_Wait(&req, 0);
6 }
7
8 void bad0(void) {
9   MPI_Request req;
10  MPI_Isend(ARGS, &req);
11  // do work
12 }
13
14 void bad1(void) {
15  MPI_Request req;
16  MPI_Isend(ARGS, &req);
17  // do work
18  MPI_Wait(&req, 0);
19  MPI_Isend(ARGS, &req);
20  // do work
21 }
```

```
22 void bad2(void) {
23  MPI_Request req;
24  MPI_Isend(ARGS, &req);
25  // do work
26  MPI_Isend(ARGS, &req);
27  // do work
28  MPI_Wait(&req, 0);
29 }
30
31 void bad3(void) {
32  MPI_Request req;
33  MPI_Isend(ARGS, &req);
34  if (cond) {
35    // do work
36    MPI_Wait(&req, 0);
37  }
38 }
```

Figure 6: Examples of non-blocking OpenMPI calls.

```
1 ISEND(c, req) :- c⟨:MPI_Isend($a0, $a1, $a2, $a3, $a4, $a5, &$req):⟩,
2   req = decl($req), req != undef.
3 WAIT(c, req) :- c⟨:MPI_Wait(&$req, $a1):⟩, req = decl($req),
4   req != undef.
5 ISENDCHAIN(c, c) :- ISEND(c, _).
6 ISENDCHAIN(c, t) :- ISENDCHAIN(c, s), ISEND(c, r), !WAIT(s, r),
7   CFG_SUCC(s, t).
8
9 WARNING(s) :- ISEND(s, r), ISENDCHAIN(s, s1), s != s1, ISEND(s1, r).
10 WARNING(s) :- ISENDCHAIN(s, c), f = enclosing_function(s),
11  CFG_EXIT(f, exit), exit == c, ISEND(s, r), !WAIT(c, r).
```

Figure 7: CLOG checker for mismatched OpenMPI calls.

In Figure 7 we implement a checker to detect these cases of API misuse. First, the checker identifies the involved API calls using syntactic patterns (lines 1 and 3) and defines two predicates, ISEND and WAIT which enumerate these call expressions (c) together with the variable that stores the request handle (req). In the next two rules (lines 5-7), we inductively define the ISENDCHAIN relation, which maps an Isend call to its control-flow successors that are not Iwait instructions with the same request handle. For defining the ISENDCHAIN relation we rely on the CFG_SUCC built-in predicate, that maps s to all its control-flow successors t. On line 9, we emit a warning whenever there exists a control-flow path on which two Isend instructions with the

same handle occur without a `Wait` in between (Figure 6, `bad2`). On line 10 we emit a warning if, starting from an `ISend` call, we can build a path that reaches a function exit, but does not contain a matching `Wait` call. To identify the function exits we rely on the `enclosing_function` built-in function, which maps a term to its enclosing function definition, and on the CFG_EXIT predicate which maps a function definition (`f`) to all its exits (`exit`).

2.3 Language Overview

As we have seen earlier, the CLOG language is an extension of standard Datalog. We extend the syntax for body literals B to allow for pattern literals, T, and the syntax for terms, where we allow for function application ($f(\overline{e})$) and for a special constant, `undef`. In Figure 8 we give the full syntax of the language.

Program		::= $\overline{K}$
Clause	K	::= $\overline{H} \,\text{:-}\, \overline{B}$
Head literal	H	::= $P\left(\overline{t}\right)$
Body literal	B	::= $H \mid !H \mid T \mid v = e \mid v > e \mid \dots$
Pattern literal	T	::= $[@t]\,[t]\,\langle : C : \rangle$
Term	t	::= $e \mid \_ \mid k \mid 'P$
Expression	e	::= $v \mid f\left(\overline{e}\right) \mid e + e \mid \dots$
Variable	v	::= $v_b \mid v_m$
Function	f	$\in$ *Functions*
Predicate symbol	P	$\in$ *Predicates*
Constant	k	$\in \mathbb{Z} \cup$ *String* $\cup \{\text{undef}\}$
Pattern	C	$\in$ *CPatternLanguage*
Ordinary variable	v_b	$\in$ *Var*
Metavariable	v_m	::= $\$v_b$

Figure 8: The syntax of the CLOG language

The CLOG language lacks explicit type declarations, but its predicates are statically typed. We rely instead on monomorphic type inference to deduce the type of predicates and variables. The possible types for variables are: *Integer*, *String*, *ASTNode* and *PredRef*. The *ASTNode* type represents program terms, thus all metavariables have this type. The *PredRef* is the type of predicate references, $'P$.

2.4 Pattern Literals

Pattern literals are predicates over the abstract syntax tree of the analyzed program. A pattern literal `@s` `r` $\langle : C : \rangle$ consists of a syntactic pattern, $\langle : C : \rangle$, and two optional nodes: the root, `r`, and the subterm restriction `@s`. The syntactic pattern matches terms in the analyzed program. When a match occurs, the metavariables in the pattern are bound and so is the optional root variable, `r`, which binds the whole matched term.

The optional subterm restriction, @s, restricts the matching to a strict subterm of the term s. For example,

⟨:while ($cond) $body:⟩, @$body ret ⟨:return $e;:⟩

matches all return statements occurring in the body of a `while` loop, and binds them to variable `ret` and their respective return expression to `$e`. The pattern, ⟨:C:⟩, can have any of the following syntactic categories: *expression*, *statement*, *declaration* or *function definition*. Terms in the pattern are either concrete, following the C grammar, or they are left abstract and replaced with a metavariable. Metavariables can be used in place of concrete terms from the following syntactic categories: *identifier*, *init-declarator*, *parameter-declaration*, *expression*, *statement*. In places where a list of these is required, but the list elements are not relevant, a *gap* (`..`) can be used. For example, ⟨:printf(.., $e, ..):⟩ matches any function call to `printf`, and binds its arguments, in turn, to `$e`. The occurrence of the call expression `printf("Hello %s!", name)` results in two pattern matches, one when `$e` binds the string literal `"Hello %s!"` and another, when it binds the identifier `name`.

2.5   Built-in Predicates

Our implementation assigns special semantics to a set of predicates. These can be grouped into three categories: control-flow predicates, I/O and infinite predicates.

Control-flow predicates expose the intra-procedural control-flow graph to the CLOG program. While traversing the control-flow graph is achievable by using only syntactic patterns, this increases the verbosity of the code, therefore we expose the following predicates:

- CFG_SUCC (n, n_s) maps the term n to its successors in the control-flow graph, n_s. Since the C language leaves unspecified the order of evaluation in some cases (e.g. subexpressions, function arguments), the CFG_SUCC relation represents only one of the possible orderings. The variable n must be bound by other literals in the same clause.

- CFG_EXIT (f, n_e) maps the function definition f to all its exits, n_e. The variable f must be bound by other literals in the same clause.

The I/O predicates enable the CLOG program to read or write relations to a tabular format (CSV or SQLite3) and they are most frequently used to read analysis parameters or to output analysis results. As the I/O predicates, the infinite predicates are identical to the ones used by *JavaDL*, and therefore we refer the reader to [DRS21]. As syntactic sugar, CLOG provides infix notations for comparison (==, <=, etc.) and variable binding (=).

2.6   Built-in Functions

CLOG provides a set of predefined functions. These functions are free of side-effects and their use is allowed inside the operands of the comparison predicates or as the

right operand of the = predicate. CLOG requires that the arguments to the functions are either other expressions (i.e. arithmetic or function application) or that they are bound variables. The purpose of these functions is to expose properties of the analyzed program that are not expressible through syntactic patterns.

Type and Name Analysis Functions

While type and name analysis for C can be expressed as a Datalog program, doing so bloats the analysis program and hinders readability. Therefore, CLOG exposes these semantic properties through predefined functions[5].

- `type(e)` - the type of the expression e.

- `decl(n)` - the declaration of the the identifier n.

Because not all C constructs have a type and, for incomplete C programs, not all identifiers have a declaration, the `type` and `name` functions are partial. In such cases, they evaluate to a special value, `undef`.

Program Structure Functions

CLOG provides convenience functions that enable the traversal of the program structure:

- `parent(n)` retrieves the parent term of n, if it exists.

- `enclosing_function(n)` retrieves the enclosing function of term n, if it exists.

Names

The CLOG programs are general over the set of names, and thus identifiers can be replaced with metavariables inside syntactic patterns. However, metavariables can bind terms that may or may not have a name, for example the pattern ⟨:`$l` + `$r`:⟩ matches the expression `a` + `1` and only the metavariable `$l` binds an identifier. To retrieve the variable name, we introduce a function `name(n)` that maps the term n to its name if it is an identifier or a named declaration, or to the empty string otherwise.

Control-Flow Functions

In addition to the control-flow predicates, CLOG defines the `cfg_entry(f)` function that maps a term f to the entry node of its control-flow graph. Terms that have a CFG are function definitions, statements and expressions.

---

[5]In the CLOG implementation, these functions are prefixed by `c_`, e.g. `c_decl`, `c_name`, etc.

Source Location Functions

To support report generation, CLOG defines a set of functions that retrieve the source location of a program term n: `src_line_start(`n`)`, `src_col_start(`n`)`, `src_file(`n`)`, etc. Since reports sometimes occur in macros, reporting the original location can be confusing, so we provide another set of functions that retrieve the expansion location: `src_exp_file(`n`)`, `src_exp_line_start(`n`)`, etc.

3 Implementation

3.1 Overview

We built the CLOG prototype using two major components:

1. A Datalog implementation that we extended with support for syntactic patterns, functions and built-in predicates.

2. A Clang library, (*Clang-Clog*), which provides support for pattern matching, CFG predicates and built-in functions.

We chose to use Clang as the parser for the analyzed program over using our own C parser because we wanted CLOG to be able to analyze real-world C programs that may use language extensions beyond the standard C grammar used by our parser. Moreover, by choosing a mature compiler such as Clang, we also gain access to other standard compiler analyses, such as name and type analysis and CFG construction.

In its implementation, CLOG reuses infrastructure from previous systems that combine syntactic pattern matching and Datalog: *MetaDL* [DBR] and *JavaDL* [DRS21]. Both these tools implement syntactic pattern matching in Datalog and translate the AST of the entire analyzed program to Datalog relations. We have experimented with the same approach in an early CLOG prototype, but it proved impractical, because for C programs the AST contains nodes for all included files (transitively) and serializing the AST into Datalog relations dominated the running time of CLOG. Instead, we opted for using Clang's own AST matching infrastructure, provided by the *LibASTMatchers*[6] library. This way, CLOG could perform pattern matching directly on the Clang AST and only serialize to relations the AST fragments that match.

The *LibASTMatchers* provides a domain-specific language (DSL) for building up a tree of matchers from a predefined set of base matchers. For example, an exact matcher for the `int x;` variable declaration is

```
varDecl(hasName("x"), hasType(qualType(isInteger())),
     unless(hasInitializer(anything()))).bind("$m")
```

where `varDecl` matches a `VarDecl` AST node, `hasName` and `hasType` are predicates on its name and type; `unless` is a matcher that succeeds when its inner matcher

---
[6] `https://clang.llvm.org/docs/LibASTMatchers.html`

fails (in this case, `hasInitializer`). The `bind` attribute specifies a name for the matching AST nodes.

To use the *LibASTMatchers* library, CLOG translates the syntactic patterns to AST matchers. However, the CLOG pattern grammar and the Clang abstract grammar have been designed for different purposes. On one hand, the goal for the CLOG pattern grammar is to be close to the C grammar even after adding metavariables. On the other hand, the Clang grammar is optimized for compiler analyses.

To a C programmer, the pattern ⟨:`$t` `$f`, `*$g`:⟩ matches two variable declarations, a variable `$f` with type `$t` and another variable, `$g`, with type pointer-to-`$t`, without any initializer. In terms of the C grammar, this means that the type of `$g` is split between two non-terminals: the *type specifier* `$t` and the *declarator*, `*$g`. However, the Clang AST represents this using two disjoint entities: a `VarDecl` which has a `QualType` as its type attribute. This is reflected in the LibASTMatchers DSL as the matcher:

```
varDecl(hasType(qualType(pointsTo(qualType().bind("$t")))),
        unless(hasInitializer(anything()))).bind("$g")
```

In the AST matcher, as in the Clang abstract grammar, the `qualType` matcher is an attribute of `varDecl`, while in the C grammar these nodes reside in different subtrees of the *declaration* non-terminal. This means that building the Clang AST matcher in the semantic actions is not feasible, thus CLOG uses three intermediate translation steps. First, it parses the syntactic patterns to an *internal* AST, which contains the same non-terminals as the C11 grammar extended with metavariables. Secondly, it translates the *internal* AST to a *pseudo-Clang* AST, which contains the same AST nodes as the Clang AST, but it also allows concrete nodes from abstract grammatical categories such as *expression* (`clang::Expr`), *statement* (`clang::Stmt`) *declaration* (`clang::Decl`) or *qualified type* (`clang::QualType`), which are marked as being a metavariable or a gap. Thirdly, CLOG traverses the *pseudo-Clang* AST and generates the AST matchers.

CLOG Extensions to Clang AST Matching

We have extended the set of AST matchers to cover all *pseudo-Clang* nodes generated by CLOG.

In CLOG, the gap construct (`..`) in syntactic patterns allows for parts of a list to remain unspecified, while still preserving the matching order of the list members that are specified. To support the semantics of gaps, we implemented a family of matchers that match a particular children list of a node against a list of sub-matchers. The sub-matchers must match the list elements in order, but the matched elements are not necessarily consecutive. For the pattern ⟨:`$f($a, $b)`:⟩, CLOG generates the matcher

```
callExpr(callee(expr().bind("$f")), argumentCountIs(2),
            hasArgument(0, expr().bind("$a")),
            hasArgument(1, expr().bind("$b")))
```

which matches function call expressions with precisely two arguments, while for

⟨:$f(..,$a,.., $b,..) :⟩, it generates:
```
callExpr(callee(expr().bind("$f")),
     argumentDistinct(expr().bind("$a"), expr().bind("$b")))
```
The `argumentDistinct` matcher ensures that its inner matchers match distinct arguments of the call expression.

Limitations

A fundamental limitation of CLOG is that it matches the syntactic patterns against Clang ASTs, which means the metavariables always bind Clang AST nodes. Thus, we configured the pattern grammar generation to allow metavariables only for the terminals and non-terminals that have a corresponding Clang AST node. For example, CLOG rejects patterns that have metavariables in place of qualifiers, such as ⟨:$q int *$f:⟩ because Clang represents qualifier as fields of the `clang::QualType` node. However, the pattern ⟨:const int *$f:⟩ is valid, because CLOG can check that the AST node bound by $f is a pointer to a `const`-qualified integer.

While the use of Clang AST can enable future extensions of CLOG to analyze C++ programs, this comes with challenges. The most significant challenge is that the language must introduce a distinction between terms existing in the source and terms arising from template instantiation, `auto` type deduction and default methods.

3.2   CLOG at Runtime

Figure 9 provides a runtime view of the CLOG implementation. The evaluation of a CLOG program proceeds with the parsing of the analysis code. The parsing of syntactic patterns is deferred to the *Pattern parser*. Then the semantic analysis phase ensures that the program is well formed. This is followed by a plan generation phase, where CLOG generates an evaluation plan according to the semi-naive evaluation strategy [Ull89]. In the evaluation phase, CLOG executes the evaluation plan to produce the results.

Clang-CLOG Interface

CLOG interfaces with Clang through a library built using the Clang *LibTooling* support library[7]. This *Clang-*CLOG library handles the parsing of the analyzed sources and builds the ASTs for all sources. In addition to building the ASTs, it supports pattern matching through Clang's *LibASTMatchers* library, the evaluation of built-in functions and CFG queries.

Pattern Parser

The CLOG parser defers the parsing of patterns to an Earley parser [Sco08], which is capable of handling general context-free grammars and ambiguity. Support for ambiguity is necessary, since the patterns lack context to disambiguate cases such as t * a;
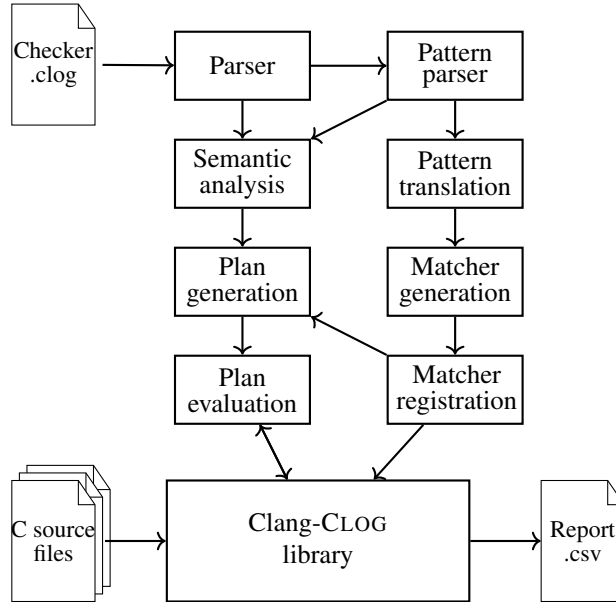
---

[7]https://clang.llvm.org/docs/LibTooling.html

Figure 9: Phases of a CLOG checker evaluation

or `f(x);`[8]. The pattern grammar follows the C grammar given in the Annex A of the C11 language standard [Iso]. This grammar is automatically extended when CLOG is built, to accept metavariables and gaps for a configurable set of non-terminals. The result of the pattern parsing phase is an AST with nodes from the *internal grammar*.

Pattern Translation

During the pattern translation phase, CLOG translates the pattern ASTs from the *internal* grammar to a *pseudo-Clang* grammar.

Matcher Generation

In this phase, CLOG traverses the *pseudo-Clang* AST and generates matchers in the LibASTMatchers DSL.

The translation from the pattern ASTs to the AST matchers is not always one-to-one. Due to ambiguities in the C language, CLOG generates two AST matchers for the pattern ⟨:`$t *$p`:⟩, one for the pointer declaration and another for the multiplication. This is reflected in the evaluation plan as a disjunction of two literals.

---

[8]A variable `x` of typedefed type `f` or a function call?

Matcher Registration

In this phase, CLOG registers the matchers with the Clang-CLOG library. The Clang-CLOG library parses the matcher DSL, builds the AST matchers and returns a unique identifier for each matcher, that the Datalog engine uses during the plan evaluation phase for retrieving the results of the match. CLOG registers a matcher a single time to avoid the cost of parsing the matcher DSL each time it evaluates a pattern literal.

The Clang-CLOG library supports three kinds of pattern matchers:

- *node matcher* - attempts to match one given node;

- *subtree matcher* - finds all matches in all descendants of one given node;

- *global matcher* - finds all matches across all ASTs.

All pattern literals without a subtree restriction or a root node, of the shape $\langle:C:\rangle$, correspond to *global matchers*. For the ones that have a root literal, but not a subtree restriction, `r` $\langle:C:\rangle$, CLOG generates a *node matcher* if the root variable `r` is bound by other literals in the clause. Otherwise, it generates a global matcher. Patterns with a subtree restriction, `@s r` $\langle:C:\rangle$, result in *subtree matchers*.

Plan Generation and Evaluation

In this phase, CLOG translates the Datalog rules to an evaluation plan. The operations of this plan are similar to the Relational Algebra Machine used by the *Soufflé* Datalog engine [Sch+16].

In contrast to the approach in [DRS21], where the entire AST of the analyzed program is materialized as Datalog relations, in CLOG we have implemented an *on-demand* approach, where we only materialize the AST nodes that are matched by the syntactic patterns and the ones that are exposed through the CFG predicates and built-in functions.

In addition, for pattern literals, we optimize for cases where the root variable of the pattern is bound by a literal occurring earlier in the clause. In this case, CLOG runs a *node matcher* on the root node, instead of a *global matcher* on the entire AST. In the current implementation, the plan generator preserves the order of literals in the clause, with the exception of the infinite and binding predicates, which it may reorder.

For the CFG_SUCC and CFG_EXIT predicates, the Clang-CLOG library lazily computes the CFGs only for functions for which these predicates are queried. In effect, this means that the CFG is computed only for functions relevant to the analysis.

The evaluation of the analysis proceeds with running the global matchers. This is handled by the Clang-CLOG library using Clang's AST matching API, which runs all the global matchers in a single traversal over the whole AST. This stage is followed by the execution of the Datalog plan. In the final stage, CLOG writes the output relations to disk.

4 Evaluation

In this section we ask the following questions:

1. ***RQ1*** How capable is CLOG to express code checkers with good precision and recall rates?

2. ***RQ2*** How does the execution time of CLOG compare to other tools?

To answer these question we implemented several checkers in CLOG and ran them on synthetic benchmarks and on real programs. We compared our results against the Clang Static Analyzer (CSA). We chose CSA because both itself and CLOG have access to the same underlying AST and we wanted to assess the analysis expressivity and speed, without comparing the underlying AST representations. To compare with CSA[9], we ran the relevant checkers using the *clang-tidy* frontend. We used an AMD EPYC 7713P 64-Core Processor with 504 GB RAM for our evaluation.

4.1 Synthetic Benchmarks

In line with earlier work [WS14; Che16], we adopted the Juliet 1.3[10] test suite in our evaluation. Juliet is a collection of synthetic tests aimed at assessing static analysis tools.

Experimental Setup

To evaluate the expressivity of CLOG we attempted to implement checkers for the first 15 weaknesses listed in the *2023 CWE Top 25 Most Dangerous Software Weaknesses* published by MITRE [11] and run these checkers on Juliet. For the cases where the Juliet suite did not contain tests for a specific weakness, we exploited the hierarchical organization of the CWE database and we used the tests for the weakness' direct descendants. We present our mapping from listed CWEs to Juliet test sets in Table 1.

For each Juliet test set corresponding to a CWE we implemented a checker in CLOG. We explicitly list the enabled checkers in Table 2. In our analysis development, we followed an iterative process in which we aimed for increasing the checker's recall, without having a precision that is lower than the precision of the Clang checker.

Discussion

In Table 2 we present a comparison regarding precision, recall and running times of the CSA and CLOG checkers.

---

[9]Built from `https://github.com/llvm/llvm-project`@98acd74683
[10]`https://samate.nist.gov/SARD/test-suites/112`
[11]`https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html`

#	CWE	Juliet tests
1	CWE-787 Out-of-bounds Write	CWE121_Stack_Based_Buffer_Overflow CWE122_Heap_Based_Buffer_Overflow CWE123_Write_What_Where_Condition CWE124_Buffer_Underwrite
4	CWE-416 Use After Free	CWE416_Use_After_Free
5	CWE-78 Improper Neutralization of Special Elements Used in an OS Command	CWE78_OS_Command_Injection
6	CWE-20 Improper Input Validation	CWE134_Uncontrolled_Format_String CWE114_Process_Control CWE190_Integer_Overflow CWE785_Path_Manipulation_Function ...
7	CWE-125 Out-of-bounds Read	CWE126_Buffer_Overread CWE127_Buffer_Underread
8	CWE-22 Improper Limitation of a Pathname to a Restricted Directory	CWE23_Relative_Path_Traversal CWE36_Absolute_Path_Traversal
12	CWE-476 Null Pointer Dereference	CWE476_NULL_Pointer_Dereference
14	CWE-190 Integer Overflow or Wraparound	CWE190_Integer_Overflow

Table 1: A mapping from common weaknesses (CWEs) to Juliet test sets. We omit CWEs that are not relevant to C programs.

CWE	CSA checkers	GT	True positive		False positive		Precision		Recall		Time (s)	
			CSA	CLOG	CSA	CLOG	CSA	CLOG	CSA	CLOG	CSA	CLOG
CWE-78	alpha.security.taint.*	1520	0	1008	0	0		100.00	0	66.31	35.82	119.39
CWE-121	*ArrayBoundsConfig*	4036	2132	240	6810	20	23.84	92.30	52.82	5.94	80.14	171.09
CWE-122	*ArrayBoundsConfig*	2606	1174	240	3542	20	24.89	92.30	45.04	9.20	50.84	128.46
CWE-124	*ArrayBoundsConfig*	1288	720	300	3593	0	16.69	100.00	55.90	23.29	27.22	73.01
CWE-126	*ArrayBoundsConfig*	972	449	160	3180	0	12.37	100.00	46.19	16.46	20.11	46.40
CWE-127	*ArrayBoundsConfig*	1288	720	300	3593	0	16.69	100.00	55.90	23.29	27.12	73.15
CWE-134	alpha.security.taint.*	1900	570	1278	780	0	42.22	100.00	30.00	67.26	48.98	312.61
CWE-416	unix.Malloc	138	36	108	0	0	100.00	100.00	26.08	78.26	2.27	2.65
CWE-476	core.NullDereference	270	174	150	16	0	91.57	100.00	64.44	55.55	4.73	6.27

Table 2: CSA and CLOG results on Juliet test sets. *ArrayBoundsConfig* represents the checkers: alpha.security.ArrayBound, alpha.security.ArrayBoundV2, alpha.security.taint.*, alpha.unix.cstring.OutOfBounds, security.insecureAPI.*. GT is the number of ground-truth reports.

To detect **CWE-78 OS Command Injection** we implemented an inter-procedural data-flow analysis. CSA did not produce any warnings, even though we configured the taint analysis to use the same propagation rules as CLOG. We adapted the same checker for **CWE-134 Uncontrolled Format String**, with a different set of taint sources and sinks.

To achieve high recall on the test sets **CWE-121 Stack Based Buffer Overflow**, **CWE-122 Heap Based Buffer Overflow**, **CWE-124 Buffer Underwrite**, **CWE-126 Buffer Overread**, **CWE-127 Buffer Underread** the checkers need to perform constant propagation or numerical domain analysis. Both these analyses are difficult to encode in Datalog without general lattice support, which is a limitation of our implementation.

These checkers also brought to light a mismatch between the pattern grammar and the Clang AST grammar. The pattern for matching array declarations ⟨:$t $a[$n]:⟩ allows an expression for the array length, $n. However, the Clang AST contains two nodes for arrays: one for fixed-length and another for variable-length arrays. The fixed-length array node does not have a child node for encoding the length, but it encodes it explicitly, as an integer field. In effect, this means that there is no AST node that the variable $n can bind, and this is not compatible with our assumption, that each metavariable binds an AST node. Fortunately, this is not a fundamental limitation of our translation scheme, since it can be circumvented by adding a custom AST matcher which instantiates a constant integer AST node when matching the size of an array.

In the **CWE-416 Use after Free** checker we implemented an intra-procedural data-flow analysis. An intra-procedural analysis proved sufficient, because freed pointers are reported when passed as arguments or return values.

To detect **CWE-476 Null Pointer Dereference** we implemented an inter-procedural data-flow analysis. To reduce false positives, we added rules that exclude paths dominated by a null test of a variable. One of these rules is

```
NotNullPath(s, d) :- ⟨:if ($v) $t else $f:⟩,
    d = decl($v), d != undef, s = cfg_entry($t).
```

where we define a predicate, NotNullPath, to mark that a variable d is not null on paths starting from s.

To achieve good precision and recall on the **CWE-190 Integer Overflow** test set, the analysis needs to perform numerical domain analysis, a CLOG limitation we have also seen earlier, on the tests sets for **CWE-121,122,124,126,127**. Detecting cases of **CWE-123 Write What Where Condition** requires heap modeling, which out of the scope of this work. CSA also fails to report any warning on this test set. The test sets **CWE-114**, **CWE-785** use the Windows API, while **CWE-23** and **CWE-36** contain only C++ sources. We did not implement checkers for these test sets.

In Table 3 we list the sizes of CLOG programs, counted as number of rules. We were able to implement all checkers concisely, with no more than 35 rules and most of them with 20-30 rules. We also note that, for all checkers, the number of syntactic patterns is close to the number of rules, which shows that they are a well-utilized language feature.

Looking back at **RQ1**, we conclude that CLOG is expressive enough to encode high precision checkers for typical data-flow analyses, even inter-procedural. However, we encountered difficulties in achieving good recall on the checkers for buffer accesses.

CWE	Predicates	Pattern literals	Rules
CWE-78	19	18	24
CWE-121	8	23	25
CWE-122	8	23	25
CWE-124	7	18	19
CWE-126	8	22	24
CWE-127	9	17	20
CWE-134	20	18	25
CWE-416	13	23	28
CWE-476	14	31	35

Table 3: Predicate, rule and pattern literal counts for CLOG programs

This class of vulnerabilities highlighted a limitation of CLOG: it can't express analyses that use lattices other than the power set. Fortunately this is a known limitation of Datalog and is addressed by approaches orthogonal to ours [Sza+18; MYL16].

In reply to **RQ2**, in Table 2, we observe that the running times of CLOG are about 2-3 times slower than CSA, but this is not surprising if we consider that we implemented CLOG partly as a Java application, partly as a native library and the two do not share a heap, so all the results of pattern matching must be copied from a native heap to the Java heap. In spite of this, the fact that CLOG is fully declarative enables other optimization approaches, such as parallelizing the evaluation engine or incrementalization, in the style of *IncA* [Sza+18] or *JavaDL* [DRS21].

4.2   Realistic Workloads

Inspired by a study on the effectiveness of static C code analyzers [LBP22], we have reused programs from the Magma v1.1 fuzzing benchmark [HHP22] as targets for our testing. Magma contains a set of programs with known vulnerabilities and their respective fixes. Conveniently, these fixes can be enabled or disabled through a preprocessor symbol. To avoid assumptions about how the location of a fix corresponds to the location of a report, we adopt a *differential approach*: we compare the reports between the fixed and the faulty versions of a program.

From the checkers we have implemented for the Juliet benchmark, we selected the ones with good recall and full precision that also have corresponding vulnerabilities in the Magma programs: **CWE-416** and **CWE-476**.

From the Magma programs, we selected *openssl*, *sqlite*, *libxml2* and *libpng*. We were not able to properly extract the compilation commands for *libtiff* and *php* and we discarded *poppler* for being mostly a C++ project.

In Table 4 we present the results of running CLOG and CSA analyses on the selected Magma programs. For **CWE-476**, both checkers discovered a real issue in *sqlite*, while for **openssl** only CSA succeeded. For the *sqlite* issue, CSA generates one report, while CLOG generates four. The difference is that CSA reports only the first dereference of a null pointer, while CLOG reports all. In the **openssl** case, our implementation of the null

CWE	Program	CSA		CLOG		Time (s)	
		Vuln.	Fixed	Vuln.	Fixed	CSA	CLOG
CWE-476	libpng	0	0	0	0	29.87	3.73
	openssl	25	24	217	217	452.78	191.14
	sqlite3	18	17	387	383	1109.19	155.86
	libxml2	33	33	425	425	349.69	23.47
CWE-416	libpng	0	0	0	0	31.80	1.31
	openssl	0	0	0	0	492.01	61.55
	sqlite3	0	0	0	0	1183.19	4.59
	libxml2	0	0	0	0	372.68	3.40

Table 4: CSA and CLOG report numbers and running times on Magma test programs. *Vuln.* columns refer to the vulnerable version.

pointer dereference checker does not handle uninitialized variables, while CSA does. This is a limitation of our analysis set, but not of CLOG itself, since an uninitialized variable checker is another instance of data-flow analysis, a class of analyses we have showed that CLOG can express. For **CWE-416**, both CSA and CLOG fail to find any cases of use-after-free vulnerabilities, even though one vulnerability is present in each of the *libxml2*, *sqlite3* and *openssl* programs. Contrary to results we have seen on the Juliet test suite, the running times for CLOG are significantly better than for CSA.

5   Related Work

Declarative approaches to the static analysis of program semantics have a rich history based on a variety of techniques, including attribute grammars [Knu68], pattern-matching on algebraic data types [App98], term rewriting [KBV08], logical queries [MLL05], flow-sensitive types [FTA02], and combinations such as Cobalt's use of integrated modal logic and term rewriting [LMC05].

Most approaches operate on some representation of the AST and/or CFG, often involving intermediate code; custom extensions must then follow these (generally tool-specific) abstractions [Avg+16; SHB19; NRL17]. This is a well-understood challenge in the field of API protocol checking [BBA09; FTA02], which attempts to identify API-specific bugs. Some tools try to address this challenge by encoding analysis rules in an internal DSLs (e.g., embedding them inside API code via Java annotations [BBA09]). However, these approaches are limited by the host language's annotation facilities.

External DSLs based on syntactic pattern matching therefore offer an appealing alternative. For C, *Coccinelle* [LM18] offers facilities for code matching and transformation, based on syntactic patterns. In contrast to CLOG, the *Coccinelle* DSL that describes the syntactic patterns requires the user to explicitly declare metavariables and their syntactic categories. *Coccinelle* provides a scripting interface (Python and OCaml) that the users can use to combine pattern matching with their custom analysis and build bug finding tools [Law+09]. The scripting interface serves approximately the

same purpose as the Datalog language in CLOG.

Other tools that combine syntactic pattern matching with logical queries include *SOUL* [DR+11], which combines AST pattern matching with Prolog-style logic programs to analyze Java programs, though *SOUL* restricts patterns to five predefined syntactic categories, and *JavaDL* [DRS21], which is closest to our work in spirit in that it offers Datalog-style rules and syntactic pattern matching, on Java programs. CLOG reuses parts of the *JavaDL* infrastructure, specifically the grammar transformation mechanism and *JavaDL*'s Datalog implementation. Unlike CLOG, *JavaDL* offers no built-in predicates for CFG traversal, which limits its ability to express flow-sensitive analyses. Internally, *JavaDL* performs pattern matching by encoding ASTs as tables of Datalog facts and relying on an external Datalog engine, while CLOG translates syntactic patterns to queries for the Clang AST matching mechanism and materializes only the results of these queries as Datalog tuples.

CLOG is not the first Datalog-style tool for C: *cclyzer* [BS16] implements declarative points-to analyses for C and C++ in Datalog. Compared to CLOG, which works on the AST, the *cclyzer* analyses are implemented in terms of LLVM IR instructions. Like *JavaDL*, *cclyzer* relies on the high-performance Datalog engine *Soufflé* [Sch+16]. Despite its wide adoption for program analysis tasks [BS16; BS09], *Soufflé* itself does not extend the Datalog language with features particularly geared towards program analysis.

6   Conclusions

We have shown that the CLOG language can express powerful custom checkers for C code without exposing program representation internals. While our experiences suggest that implementing a language like CLOG on top of Clang may require nontrivial internal plumbing to align program structure and control flow, we find that the cost for this abstraction is modest: CLOG-based checkers may even run faster than Clang's own checkers, despite delivering competitive results. Like most Datalog-based approaches, CLOG is limited to boolean (product) lattices but scales to a wide variety of practical analyses, including interprocedural and data-flow analyses. Overall, we argue that CLOG's combination of language simplicity, expressivity, and performance make it uniquely suited for building custom C code checkers.

Acknowledgements

References

[App98]   Andrew W. Appel. *Modern Compiler Implementation: In ML*. 1st. Cambridge University Press, 1998.

[Avg+16]   Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[BS16]   George Balatsouras and Yannis Smaragdakis. "Structure-sensitive points-to analysis for C and C++". In: *International Static Analysis Symposium*. Springer. 2016.

[BBA09]   Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. "Practical API protocol checking with access permissions". In: *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer. 2009.

[BS09]   Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of OOPSLA '09*. ACM, 2009.

[Cha+17]   Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. "Instruction punning: Lightweight instrumentation for x86-64". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.

[Che16]   Xi Cheng. "RABIEF: range analysis based integer error fixing". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016.

[DR+11]   Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. "The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Association for Computing Machinery, 2011.

[DBR]   Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*. SOAP 2019.

[DR]   Alexandru Dura and Christoph Reichenbach. "Clog: A Declarative Language for C Static Code Checkers". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024.

[DR24]   Alexandru Dura and Christoph Reichenbach. *Clog: A Declarative Language for C Static Code Checkers*. 2024. URL: https://doi.org/10.5281/zenodo.10525151.

[DRS21]   Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection". In: *Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA* (2021).

[FTA02]   Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. "Flow-sensitive type qualifiers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002.

[HHP22]   Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *SIGMETRICS Perform. Eval. Rev.* 49.1 (2022).

[Iso]   *ISO/IEC 9899:2011 – Information technology – Programming languages - C*. Standard. International Organization for Standardization.

[Joh77]   Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[KBV08]   Lennart C.L. Kats, Martin Bravenboer, and Eelco Visser. "Mixing Source and Bytecode: A Case for Compilation by Normalization". In: *SIGPLAN Not.* 43.10 (2008).

[Knu68]   Donald E Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968).

[Law+09]   Julia L Lawall et al. "WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009.

[LM18]   Julia Lawall and Gilles Muller. "Coccinelle: 10 years of automated evolution in the Linux kernel". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018.

[LMC05]   Sorin Lerner, Todd Millstein, and Craig Chambers. "Cobalt: A Language for Writing Provably-Sound Compiler Optimizations". In: *Electronic Notes in Theoretical Computer Science* 132.1 (2005). Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004).

[LBP22]   Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Association for Computing Machinery, 2022.

[MYL16]   Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. "From Datalog to Flix: a Declarative Language for Fixed Points on Lattices". In: *ACM SIGPLAN Notices* 51.6 (2016).

[MLL05]   Michael Martin, Benjamin Livshits, and Monica S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. Association for Computing Machinery, 2005.

[NRL17]   Krishna Narasimhan, Christoph Reichenbach, and Julia Lawall. "Interactive Data Representation Migration: Exploiting Program Dependence to Aid Program Transformation". In: *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM 2017. ACM, 2017.

[O'H19]   Peter O'Hearn. "Separation logic". In: *Communications of the ACM* 62.2 (2019).

[Rit93]   Dennis M. Ritchie. "The Development of the C Language". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Association for Computing Machinery, 1993.

[Sch+16]   Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. "On Fast Large-scale Program Analysis in Datalog". In: *Proceedings of the 25th Int. Conf. on Compiler Construction*. CC 2016. ACM, 2016.

[SHB19]   Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2019.

[Sco08]   Elizabeth Scott. "SPPF-style parsing from Earley recognisers". In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008).

[Sza+18]   Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. "Incrementalizing Lattice-Based Program Analyses in Datalog". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018).

[Ull89]   J. D. Ullman. "Bottom-up Beats Top-down for Datalog". In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '89. Association for Computing Machinery, 1989.

[WS14]   Andreas Wagner and Johannes Sametinger. "Using the juliet test suite to compare static security scanners". In: *2014 11th International Conference on Security and Cryptography (SECRYPT)*. IEEE. 2014.

POPULAR SCIENCE SUMMARY

Static Program Analysis   Software is gaining ground in our world and is running in almost all objects that are connected to a power source. For objects that have a power source and are not running a computer program, there exists a variant of them that does. Likewise, processes central to our society — transportation systems, health records, logistics — are software controlled. Thus, to have reliable everyday objects and a functional society we need reliable software.

One way of achieving reliable software is by studying the behaviour of the computer programs and trying to avoid the undesired behaviours. The branch of computer science concerned with studying the behaviour of computer programs is called *program analysis*. The first approach to program analysis, is dynamic: we run the program on various inputs and observe their behaviour. While this approach can unveil some undesired behaviours, it will not exclude all possible undesired behaviours.

The second approach to program analysis considers the program as a description of program's behaviour and tries to restrict the description in order to restrict the behaviour. Since in this approach we do not run the program, we call it *static* program analysis.

Static program analyses are already included in compilers, the tools that translate programs from a human-readable format to machine code. But compilers can only include those analyses that do not limit the generality of the programming language — for example they can ensure that we are not mixing values of different types. Static analyses that are specific to an application domain, those that are considered too restrictive for the whole domain of the programming language, or those which take too much time to run are delegated to specialized tools called *static code checkers*.

Static Code Checkers   Static code checkers contain a set of detectors which aim to identify common bug patterns in computer programs. Since the detectors that ship with the static code checkers are aimed at general bug patterns, these checkers also provide an extension mechanism to enable the definition of project-specific detectors or for the customization of existing ones.

In this thesis we identify limitations of the current static code checkers, that hinder their adaptation to specific project needs:

Checker-specific representations In their definition, the detectors depend on a specific internal program representation and auxiliary data structures. These data structures are checker-specific.

Fixed runtime behaviour The code checkers use a fixed evaluation model, which assumes that for each run, the whole program needs to be re-checked. In practice, this is rarely the case, since typical change sets are limited to a small number of files from a project.

Declarative Code Checkers To address the limitations of current code checkers, in this thesis we explore a novel method for building static code checkers. Instead of specifying *how* the detectors are built in terms of handling the program representation and the auxiliary data structures, we introduce a family of declarative specification languages, that move the focus to specifying *what* are the detectors looking for. To achieve this, we combine two existing declarative techniques: syntactic pattern matching and logic programming.

We use syntactic patterns of the same shape as the language we are checking to break the dependency between the static checker and the checker-specific representations of the program. This eliminates the need to understand the internal representation of the program, and should allow for ad-hoc definition of static code checkers, by software engineers that are familiar with the project they want to check, but not necessarily with the internals of a static code checker.

To combine information between multiple syntactic pattern matches and enable non-local reasoning over a program, our declarative specification allows for the definition of judgements of the shape *if-condition-then-consequence*, in the style of the Datalog logic programming language.

The use of syntactic patterns and Datalog-style rules facilitates the complete decoupling of the detector specification from its implementation details. And once the specification is decoupled from the implementation, we are free to vary the runtime mode of the checker to address the evolving needs of a software project. Thus, we introduce an incremental evaluation mode that recomputes only the reports which depend on modified source files. Switching between incremental and exhaustive schemes does not require changes to the detector specification. This enables the runtime mode to match the project's needs: use the exhaustive mode when changes are pervasive, such in the early stages of a project's lifetime and the incremental mode for localized changes, suitable for mature projects.

The detector specification language is small: the syntactic patterns are essentially fragments of the language targeted by the code checker and the logical rules are a subset of first order logic. This allows for a concise specification of detectors. Besides being concise, we show that the detectors implemented using our declarative language

are also expressive. The quality of the reports produced by the detectors using our specification language is comparable with established static code checkers.

Since code checkers are part of the development loop or of continuous integration, it is important that computing the results of a detector does not take unnecessarily long. For all its elegance, a declarative static code checker is useless if it takes too much time to deliver the results. Our checkers are not an exception: we empirically demonstrate that their runtime performance is comparable with widely used static checkers.

Hence, this thesis provides evidence that static code checkers that are concise and expressive in their definition, accurate in their results and fast in their running time are indeed feasible.