Incrementalizing Static Analyses in Datalog

Dissertation submitted for the award of the title
"Doctor of Natural Sciences"
to the Faculty of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University Mainz
in Mainz

Tamás Szabó

Master of Science in Computer Science,
Budapest University of Technology and Economics, Hungary,
born in Debrecen, Hungary.

Mainz, June 30, 2020

Referees:

[redacted]	[redacted]
	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

Date of oral examination: January 11, 2021.
An electronic version of this dissertation is available at `https://openscience.ub.uni-mainz.de`.

D77

Contents

Summary vii

Zusammenfassung ix

Acknowledgments xi

1 Introduction 1
- 1.1 Trade-offs in the Design of Static Analyses . . . 3
- 1.2 Techniques for Improving the Performance of Static Analyses . . . 6
- 1.3 Challenges in Incrementalizing Static Analyses . . . 7
- 1.4 Incremental Analysis Framework . . . 9
- 1.5 Contributions . . . 10
- 1.6 Outline . . . 13

2 Static Analysis with Datalog 15
- 2.1 Introduction to Datalog . . . 16
- 2.2 Datalog by Example . . . 16
- 2.3 Recursion and Fixpoint Computation . . . 19
- 2.4 Incrementalizing Standard Datalog . . . 21
- 2.5 Chapter Summary . . . 26

3 The IncA Incremental Analysis Framework 27
- 3.1 Introduction . . . 28
- 3.2 Incremental Program Analysis with IncA . . . 29
- 3.3 Syntax and Compilation of IncA Datalog . . . 34
  - 3.3.1 Syntax of IncA Datalog . . . 34
  - 3.3.2 Compilation to Graph Patterns . . . 35
- 3.4 Compiler Optimizations for IncA . . . 36
- 3.5 Technical Realization and IDE Integration . . . 39
  - 3.5.1 Architecture Overview . . . 39
  - 3.5.2 Implementation for MPS . . . 40
  - 3.5.3 Applicability in other IDEs . . . 41
- 3.6 Case Studies . . . 41
  - 3.6.1 Control Flow Analysis . . . 41
  - 3.6.2 Points-to Analysis . . . 42
  - 3.6.3 Well-formedness Checks for mbeddr C . . . 42
  - 3.6.4 FindBugs for Java . . . 43
- 3.7 Performance Evaluation . . . 43
  - 3.7.1 Evaluation Setup . . . 43
  - 3.7.2 Evaluating Run Time (Q 3.1) and Optimization Impact (Q 3.3) . . . 45
  - 3.7.3 Evaluating Memory (Q 3.2) and Optimization Impact (Q 3.3) . . . 46

3.8 Chapter Summary . . . 47

4 Incrementalizing Lattice-Based Program Analyses 49

4.1 Introduction . . . 50
4.2 Challenges of Incrementalizing Lattice-based Program Analysis . . . 51
4.3 Incremental Lattice-Based Program Analysis with IncA . . . 54
4.3.1 Incremental Execution of Non-Recursive Analyses . . . 54
4.3.2 Incremental Execution of Recursive Analyses . . . 55
4.4 Incremental Recursive Aggregation with DRed_L . . . 57
4.4.1 Assumptions of DRed_L on the Input Datalog Rules. . . . 57
4.4.2 Support Data Structures . . . 58
4.4.3 DRed_L Algorithm . . . 59
4.4.4 Incremental Aggregator Function . . . 63
4.5 Formal Semantics of DRed_L and Correctness Proof . . . 63
4.5.1 Semantics of Recursive Aggregation . . . 63
4.5.2 Correctness of the Algorithm - Proof Sketch . . . 64
4.6 Integrating Lattices into the IncA Framework . . . 66
4.6.1 Extensions to IncA Datalog. . . . 66
4.6.2 Runtime System . . . 67
4.6.3 Compiler . . . 68
4.6.4 IDE Integration . . . 69
4.7 Case Studies . . . 69
4.7.1 Jimple as Subject Language . . . 69
4.7.2 Strong-update Points-to Analysis . . . 70
4.7.3 String Analyses . . . 71
4.8 Performance Evaluation . . . 72
4.8.1 Evaluation Setup . . . 73
4.8.2 Evaluating Run Time (Q 4.1) . . . 73
4.8.3 Evaluating Memory (Q 4.2). . . . 74
4.8.4 Discussion . . . 75
4.9 Chapter Summary . . . 75

5 Incrementalizing Lattice-based Inter-procedural Analyses 77

5.1 Introduction . . . 78
5.2 Prior Work and Problem Statement . . . 80
5.3 Incremental Lattice-Based Program Analysis with LADDDER . . . 85
5.3.1 Initial Analysis with LADDDER . . . 86
5.3.2 Incremental Analysis with LADDDER . . . 88
5.3.3 Monotonicity, Assumptions, and Guarantees of LADDDER . . . 89
5.4 Incremental Aggregation in LADDDER. . . . 91
5.5 Formal Semantics of LADDDER and Correctness Proof . . . 93
5.5.1 Concepts. . . . 93
5.5.2 Refined Assumptions of LADDDER on the Input Datalog Rules . . . 95
5.5.3 Semantics . . . 95
5.5.4 Correctness Properties . . . 96
5.5.5 Proofs of Correctness Properties . . . 96

5.5.6 Additional Notes 98
5.6 Performance Evaluation 98
5.6.1 Evaluation Setup 98
5.6.2 Evaluating Incrementalizability (Q 5.1) 99
5.6.3 Evaluating Run Time (Q 5.2) 100
5.6.4 Evaluating Memory (Q 5.3). 101
5.6.5 Evaluating Optimization Impact (Q 5.4) 101
5.6.6 Discussion on the Set-based Analysis 102
5.7 Chapter Summary 103

6 A DSL for Incremental Program Analysis 105
6.1 Introduction 106
6.2 Background: Overload Resolution in Featherweight Java. 107
6.3 Overload Resolution with IncA Datalog 108
6.4 Syntax and Compilation of IncA$_{fun}$. 112
6.4.1 Syntax of IncA$_{fun}$. 113
6.4.2 Compilation of IncA$_{fun}$. 113
6.5 Case Studies with IncA$_{fun}$ 119
6.5.1 Overload Resolution for Featherweight Java with IncA$_{fun}$ 119
6.5.2 Rust Program Analyses with IncA$_{fun}$ 121
6.6 Chapter Summary 123

7 Textual Front End for Incremental Program Analysis 125
7.1 Introduction 126
7.2 Requirements for Textual Front Ends and Prior Work 127
7.3 AST Differencing with *hdiffs* 132
7.4 Integrating *hdiffs* with IncA 138
7.4.1 Subject Program as Virtual EDB Relations 139
7.4.2 Requirements for Correct and Efficient EDB Updates 139
7.4.3 Updating EDB Relations 141
7.4.4 Language-independent Implementation 146
7.5 Evaluation 146
7.5.1 Evaluating Ease of Integration (Q 7.1) 147
7.5.2 Evaluating Run Time (Q 7.2) 148
7.5.3 Discussion 149
7.6 Future Work 149
7.7 Chapter Summary 151

8 Related Work 153
8.1 Incremental Static Analysis 155
8.2 Techniques for Speeding up Static Analyses 160
8.3 Datalog and Its Applications 164
8.4 DSLs for Program Analysis. 169
8.5 Tree Differencing Techniques 172
8.6 Chapter Summary 174

9 Conclusions and Future Work **177**
9.1 Revisiting the Thesis of the Dissertation 177
9.2 Suggestions for Future Work 179

Bibliography **183**

Curriculum Vitæ **199**

Summary

Static analyses are tools that reason about the behavior of computer programs without actually executing them. They play an important role in many areas of software development because they can catch potential runtime errors or reason about the security or performance of subject programs already at development time. For example, Integrated Development Environments (IDEs) use a variety of static analyses to provide continuous feedback to developers, as they edit their programs. IDEs run type checkers to verify that subject programs are free of type inconsistencies, or they run data-flow analyses to find security vulnerabilities. In turn, developers can fix or improve their programs before those go into production, thereby saving significant costs on the harmful effects of failures.

Designing static analyses that provide continuous feedback in IDEs is challenging because analyses must balance a vexing trade-off between *precision* and *performance*. On the one hand, an analysis must precisely capture the program behavior by considering all possible concrete executions. On the other hand, an analysis running in an IDE must update its results after a program change in a fraction of a second, otherwise it interrupts the development flow, which is counterproductive. However, these two requirements are in conflict with each other, as more precision comes with more computational cost. Given that we cannot loosen the timing constraint in an IDE, the challenge is how to speed up static analyses, so that we sacrifice on precision as little as possible.

We study how to use *incrementality* to speed up static analyses. Instead of repeatedly reanalyzing the *entire* subject program from scratch, an incremental analysis reuses previous results and updates them based on the *changed* code parts. An incremental analysis can achieve significant performance improvements over its non-incremental counterpart *if* the computational cost of updating analysis results is proportional to the size of the change and not the size of the entire program. However, precise static analyses typically use features that significantly increase computational costs by requiring the re-analysis of *more* than just the changed code parts. Moreover, incrementalization requires sophisticated algorithms and data structures, so specialized solutions often do not pay off in terms of development effort. We claim that it is possible to automatically incrementalize static analyses, and incrementality can significantly improve the performance of static analyses.

We design an *incremental static analysis framework* called IncA. IncA offers a Datalog-based language for the specification of static analyses. The declarative nature of Datalog allows analysis developers to focus on the analyses themselves by leaving the incrementalization to a Datalog solver. We design Datalog solver algorithms to automatically incrementalize analyses and deliver the kind of performance that analyses running in IDEs need. We prove our solver algorithms correct, making sure that analyses incrementalized by IncA compute the exact same results as their non-incremental counterparts. The architecture of IncA is generic, as it is independent of any particular subject language, program, or analysis, enabling the integration of IncA into different IDEs.

We evaluate IncA by incrementalizing existing static analyses for a number of subject languages. We implement well-formedness checks for DSLs, FindBugs rules, data-flow analyses, inter-procedural points-to analysis for Java, plus a type checker for Featherweight Java and Rust. We find that IncA consistently delivers good performance across all of these analyses on real-world programs. Based on these results, we conclude that IncA enables the use of realistic static analyses for continuous feedback in IDEs.

Zusammenfassung

Statische Analysen sind Werkzeuge zur Softwareentwicklung, die Informationen zum Laufzeitverhalten von Programmen liefern, ohne sie auszuführen. Sie spielen eine wichtige Rolle in vielen Bereichen der Softwareentwicklung. Zum Beispiel können statische Analysen frühzeitig mögliche Laufzeitfehler erkennen, oder über die Sicherheit und Performanz von Programmen Auskunft geben. Des Weiteren sind sie Bestandteil von integrierten Entwicklungsumgebungen (IDEs) um Softwareentwicklern kontinuierlich Rückmeldung bei der Programmentwicklung zu geben. Zum Beispiel verwenden IDEs Type Checker um sicherzustellen, dass Programme keine Typinkonsistenzen enthalten, oder sie führen Datenflussanalysen aus um Sicherheitslücken zu finden. Dies erlaubt Entwicklern ihre Programme zu verbessern bevor sie produktiv eingesetzt werden, um hohe Kosten von Fehlern im Produktiveinsatz zu vermeiden.

Die Entwicklung von statischen Analysen, die in IDEs kontinuierliches Feedback liefern, ist eine besondere Herausforderung, da sie einen Kompromiss zwischen *Präzision* und *Performanz* finden müssen. Auf der einen Seite müssen Analysen präzise das Programmverhalten modellieren, indem sie alle möglichen Ausführungspfade berücksichtigen. Auf der anderen Seite müssen Analysen in IDEs ihr Ergebnis in Sekundenbruchteilen aktualisieren, um nicht den Arbeitsfluss von Softwareentwicklern zu unterbrechen. Jedoch ist es schwierig beiden Anforderungen gleichzeitig zu erfüllen: Mit einer höheren Präzision erhöhen sich häufig auch die Laufzeitkosten der Analyse. Da wir allerdings in IDEs auf eine niedrig Laufzeit angewiesen sind, ist die Herausforderung herauszufinden, wie man statische Analysen beschleunigen kann ohne zu viel Präzision aufzugeben.

In dieser Arbeit erforschen wir wie man statische Analysen mit Hilfe von *Inkrementalität* beschleunigen kann. Anstatt das *gesamte* Programm bei jeder Programmänderung von neuem zu analysieren, verwendet eine inkrementelle Analyse das vorherige Analyseergebnis wieder und berechnet das neue Ergebnis basierend auf den *Programmänderungen*. Das verbessert signifikant die Laufzeit einer inkrementellen Analyse im Vergleich zu einer nicht-inkrementellen Analyse, *wenn* die Laufzeitkosten der Aktualisierung proportional zur Größe der Änderung und nicht zur Größe des Programms sind. Allerdings müssen präzise inkrementelle Analysen bei einer Änderung häufig *mehr* reanalysieren als nur den Code der Änderung selbst, was ihre Laufzeit verlangsamen kann. Dazu kommt, dass inkrementelle Analysen ausgeklügelte Algorithmen und Datenstrukturen benötigen, sodass sich eine händische Inkrementalisierung wegen dem hohen Entwicklungsaufwand nicht lohnt. Wir behaupten, dass es möglich ist statische Analysen automatisch zu inkrementalisieren und dass Inkrementalisierung die Performanz von statischen Analysen verbessert.

In diesem Zuge haben wir ein *inkrementelles statisches Analyse Framework* namens IncA entwickelt. IncA verwendet für die Spezifikation von Analysen einen Datalog Dialekt. Datalog ist eine deklarative Programmiersprache, die es Analyseentwicklern erlaubt sich auf die Implementierung der Analyse zu konzentrieren, während die Inkrementalisierung von einem Datalog Solver übernommen wird. Das IncA Backend unterstützt mehrere Algorithmen zum Lösen von Datalogregeln, deren Korrektheit wir formal bewiesen haben. Das bedeutet, dass die inkrementalisierte Analyse dasselbe Ergebnis liefert, wie die nicht inkrementelle Variante. Des weiteren ist IncA nicht an spezifische Analysen und Programmiersprachen gekoppelt, was es ermöglicht IncA in verschiedene IDEs zu integrieren.

Wir evaluieren IncA indem wir existierende statische Analysen für verschieden Programmiersprachen inkrementalisiert haben. Insbesondere haben wir eine Wohlgeformtheitsüberprüfung für DSLs, FindBugs Regeln, Datenflussanalysen, eine Interprozedurale Zeigeranalyse für Java, sowie einen Typechecker für Featherweight Java und Rust in IncA implementiert. In all unseren Experimenten haben die inkrementalisierten Analysen für realistische Programme eine gute Performanz gezeigt. Daraus schlussfolgern wir, dass sich IncA eignet statische Analysen zu implementieren, die schnell genug sind um Entwicklern in einer IDE kontinuierliches Feedback zu geben.

Acknowledgments

This dissertation is the result of a long and often difficult journey with many ups and down along the way. It took me almost 6 years to complete my Ph.D. studies, and I was a student at three universities in two countries. I had the pleasure to surround myself with a number of wonderful people who made the struggles easier and who were there to share the happiness and successes with me. I dedicate this chapter to them.

First of all, I would like to thank [redacted] for the years we spent together. When we started working, I quickly realized that your critical attitude required a completely different mindset also from me. This was and still is a good thing, but I was just not yet used to it at that time, so it also required some adjustments on my side. Throughout the 6 years, we essentially worked remotely together, and that also came with its own set of challenges, until we both learned how we can be productive together. I think you imparted your critical attitude to me well, and I learned that I should always think before I answer, I should always dare to ask, and that I should always come prepared. You had a good sense of how far you can push me, so that I have both freedom and the necessary amount of guidance at the same time. I still remember the ASE paper writing experience where you were demanding last minute changes an hour before the deadline, but you were there and ready to discuss if needed until the last minute. You also know how to have fun, which is also very important; I will always remember our trip together in Boston, you dancing with us until the morning during my wedding, or the great night outs in Delft. No doubt, you had a significant impact on my professional and personal growth, and I thank you for that.

Next in line is [redacted]. You have been my secondary supervisor all along, and I thank you for that. I always enjoyed working with you. You bring a lot of passion into everything you do. You have this down-to-earth and very practical attitude, which I really like. I have learned a lot from you about (technical) writing and presenting. It was sometimes annoying that you could always find more things to improve during revisions, but you always presented your feedback in a very structured manner. I will always remember the word *scientification*: Let them just do their thing with all the Greek symbols. Thank you for all the years we worked together, and I hope that we will have the chance to work together again in the future.

I would also like to thank [redacted] for the collaborations. I think I was very lucky that I got to work together with you after my M.Sc. studies again. Your technical depth and meticulousness still amaze me, and I thoroughly enjoyed all the detailed discussions we conducted throughout the years. You never got bored of my questions, and you always found time to explain all the Datalog details to me.

My Ph.D. journey spanned three countries, as I followed [redacted] along. Coupled with the remote nature of my work, this proved to be the best thing that could happen to me. Every time I visited the respective research groups at the universities, the trips always ended up being memorable. Somehow the trips were always a good mixture of intense research meetings, sight seeings, social outings, and exploration of local cuisine.

First, I would like to thank [redacted] for allowing me to join the Software Technology Group at TU Darmstadt. Even though, I spent only a year in Darmstadt, that time was still enough for me to realize that I made a good choice when I decided to pursue a Ph.D. I got acquainted with a number of people in the group, and I had my first presentations, reading group participation, and research discussions. I would like to thank [redacted], [redacted], [redacted], and [redacted] for the enlightening discussions we had.

Second, I moved on to TU Delft with [redacted]. I would like to thank [redacted] for allowing me to join the Programming Languages Group. I have actually spent most of my time in Delft, as I was a student there for 3 years. I enjoyed every visit to Delft. The whole team was so welcoming every time, and the whole environment fueled my professional growth a lot. I would like to thank all the group members ([redacted], [redacted], [redacted], [redacted], [redacted], [redacted], [redacted], [redacted], [redacted], [redacted], and [redacted]) for the interesting discussions and the fun we had together after work. [redacted], I will never forget the bike trip we had to the sandy beaches of The Hague.

Third and last, I spent two years at JGU Mainz in the Programming Languages group led by [redacted]. At this point, I would like to say a special thank you to [redacted] for all the work and fun together. You were there all along during my Ph.D., as we started roughly at the same time back in Darmstadt. I always admired how meticulous you are and your solid theoretical foundations in all things programming languages. I also would like to thank [redacted] for all the discussions we had and the interesting work we did on IncA. It was funny and interesting to see your professional development as you started your Ph.D., the struggles you went through (just like I did), and your first successes with accepted papers. I hope the foundations laid out with IncA will prove to be useful for your future research.

I would also like to thank my Ph.D. defense committee members for their service and the interesting discussions we conducted during the defense.

Now, on to the industrial side of things. itemis and its key people ([redacted], [redacted], and [redacted]) deserve a huge thank you. You all have managed to build an environment that appreciates and values research and innovation. The 4+1 working scheme has allowed me to spend dedicated work time on doing my Ph.D. I would also like to thank everybody in the Stuttgart office for all the years we spent together. I thank you, [redacted], [redacted], [redacted], [redacted], and [redacted], for all the work together.

I would also like to thank members of IncQuery Labs. Thank you, [redacted] and [redacted], for allowing me to build upon Viatra Query in my research and for the work we did together during the past couple of years.

Finally, I want to finish off in Hungarian to address my family. Először is köszönöm szépen a szüleimnek az éveken át tartó támogatást. Mindig ott voltatok mellettem. Ha kellett, vigasztaltatok amikor nem úgy sikerült valami ahogy szerettem volna, máskor velem együtt örültetek a sikereimnek. Sokszor elmondtátok, hogy sok mindent nem értetek abból amit csinálok, ennek ellenére mindig buzdítottatok arra, hogy a kutatás nekem való lenne.

Legvégül jutok a legfontosabb személyhez ebben az egész történetben. Fanni, köszönöm Neked azt, hogy végig mellettem voltál. Szerintem nem is tudod azt, hogy Neked mennyire nagy szereped van abban, hogy ez az egész dolog sikerrel végződött. Nyugodt körülményeket teremtettél ahhoz, hogy én azt tudjam csinálni amit szeretek, mindig ott voltál amikor valami bántott, és mindig volt időd arra, hogy beszámoljak a legújabb történésekről a programozási nyelvek területén. Dani, te pedig a legjobb baba vagy a világon. Pont tudtad, hogy akkor kell megérkezni, amikor én már a disszertáció írásának nagy részén túl voltam. Neked azt köszönöm hogy a jó kedveddel mindig minket is felvidítottál.

Tamás
Stuttgart, January 2021

1 Introduction

Static analysis is a method for reasoning about the structure or behavior of a subject program without actually executing it. This is in contrast to dynamic verification techniques like software testing or program slicing, which require a subject program to run. Given that real-world software systems often have complex dependencies or require special hardware to run (e.g. GPUs or cloud environment), the ability to reason about subject programs statically has the potential to catch runtime errors early on, already at development time. In turn, this helps to improve the quality of software, e.g. in terms of safety, security, or performance, before that software actually goes into production, saving significant costs on potentially harmful effects of software failures [71].

Static analyses show up in many areas of modern software development. IDEs frequently use static analyses to provide automated feedback to developers about subject programs by running type checkers or data-flow analyses. They do this by considering all possible inputs of a subject program to account for all possible executions. Continuous integration (CI) servers use static analyses to comment on pull requests about violations of coding standards. Using coding standards is typical in software companies because they help enforce a uniform coding style or prevent the use of bad coding practices. Compilers use static analyses to enable optimizations such as constant propagation and other semantics-preserving code transformations.

As a concrete example, consider an IDE that warns programmers about unreachable (or dead) code in C programs. A part of a program is considered unreachable if it can never be reached in any execution irrespective of the input. Given a concrete subject program, an unreachable code analysis reports concrete occurrences of unreachable code parts. A well-known bug that is related to unreachable code is *goto fail* that affected all iPhone devices back in 2014 [1]. The bug was in the implementation of a C function that performed SSL key verification. The relevant excerpt of the function looks as follows:[1]

[1]https://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c.auto.html.

```
 1 static OSStatus SSLVerifySignedServerKeyExchange(...) {
 2   ...
 3   if (...)
 4     goto fail;
 5     goto fail;
 6   if (...)
 7     goto fail;
 8   err = sslRawVerify(ctx,
 9     ctx->peerPubKey,
10     dataToSign,     /* plaintext */
11     dataToSignLen,  /* plaintext length */
12     signature,
13     signatureLen);
14   if(err) {
15     sslErrorLog(...);
16     goto fail;
17   }
18   fail:
19     SSLFreeBuffer(&signedHashes);
20     SSLFreeBuffer(&hashCtx);
21     return err;
22 }
```

By duplicating line 4, the developer accidentally introduced an unconditional `goto` instruction in line 5. That is, whenever the program execution reaches line 5, the execution will jump to label `fail` unconditionally. None of the code in lines 6-17 are reachable, irrespective of the program input. This is a problem because lines 6-17 would do important verification steps, particularly the `sslRawVerify` function, but the function call is part of the unreachable code. Without these verification steps, attackers could access sensitive user data even when the communication was protected by SSL/TLS.

Modern IDEs run unreachable code analysis as developers edit programs, so that developers can fix their code by eliminating the unreachable parts, thereby improving, for example, code quality and compilation performance (due to smaller code size). If we open the above code in the CLion IDE,[2] lines 6-17 are immediately highlighted (similar to the gray color in the above listing) with a message indicating that they are unreachable. This means that we could easily catch the root cause of the goto fail error. Moreover, CLion updates the highlighting as code gets changed. For the sake of this example, assume we replace lines 3-4 from above as follows:

```
1 static OSStatus SSLVerifySignedServerKeyExchange(...) {
2   ...
3   if (false)
4     goto fail;
5   ...
6 }
```

The `goto fail;` in line 4 immediately gets highlighted, indicating that this is also unreachable code now. Assume now that we change the code as follows:

[2]`https://www.jetbrains.com/clion/`

```
 1 static OSStatus SSLVerifySignedServerKeyExchange(...) {
 2   ...
 3   if (check())
 4     goto fail;
 5   ...
 6 }
 7 
 8 static boolean check() {
 9   return false;
10 }
```

Interestingly, there is no highlighting at all, even though it is clear that line 4 is still unreachable, given that `check` simply returns false. *Why is that the case?*

The problem is that this code snippet would now require an *inter-procedural* analysis, which is an analysis that reasons across function calls. In contrast, CLion uses an *intra-procedural* analysis,[3] which is not sufficient to precisely reason about the above code. In turn, CLion cannot prove the above code unreachable, so it chooses to not do any highlighting. The reason for using an intra-procedural analysis is that it is computationally much cheaper than an inter-procedural analysis. An inter-procedural analysis would need to consider the effect of (transitively) called functions. To statically analyze C function calls, the analysis would need to reason about function pointers, as well. There is also a range of other program constructs that the analysis would need to consider to precisely analyze real-world subject programs, like the iOS operating system. Coupled with the sheer size of realistic subject programs, it is a daunting task to efficiently run the analysis in IDEs. However, if an analysis takes a long time to compute new results after a program change, then the development workflow will be interrupted with lengthy pauses.

1.1 Trade-offs in the Design of Static Analyses

The previous example of unreachable code analysis already highlights that an analysis developer must balance several trade-offs when designing a static analysis. First, there is the aspect of *precision*, which entails *reliability* and *accuracy*. Accuracy and reliability are actual quantifiable metrics in the static analysis community, but here we only discuss their intuitive meaning.[4] A fully reliable analysis never misses a real bug, therefore we can always trust the results of such analysis. A fully accurate analysis only reports about bugs that a real program execution would actually exhibit. We will discuss these terms in more details later in this section. Second, there is the aspect of *performance*. A more performant analysis delivers analysis results faster than a less performant one. The left part of Figure 1.1 shows the design trade-offs graphically. We now discuss how the intra-procedural and inter-procedural versions of the unreachable code analysis balance these trade-offs, which we also visualize in the right part of Figure 1.1.

The intra-procedural unreachable code analysis was good at finding unreachable code parts locally inside a function, but it failed when it was required to reason about the

[3] At least, version 2019.3.6, which was used at the time of writing this dissertation.

[4] Note that the actual terms used in the static analysis literature differ from our terminology. The synonym of our notion of reliability is *recall*, while accuracy is the synonym for *precision*. We chose to not use the traditional terms because precision is a heavily overloaded word in the context of this dissertation.

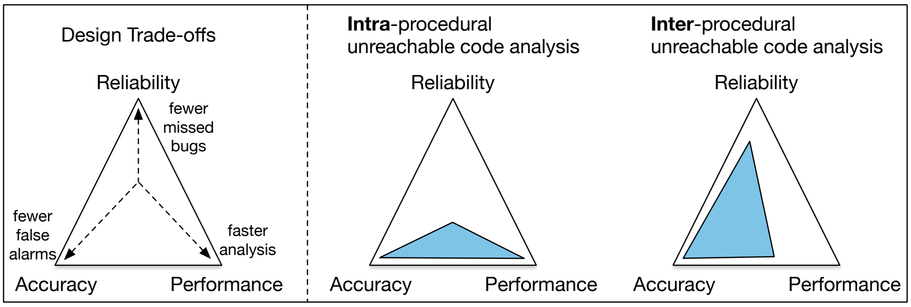


Figure 1.1: Trade-offs in the design of static analyses.

effects of function calls. The analysis has low reliability because it is easy for a potential unreachable code part to go undetected if spotting that would involve inter-procedural reasoning. If the analysis misses a truly unreachable code part, then we say that the analysis has a *false negative* result. A client of the results of the unreachable code analysis is the error reporter in the editor of CLion. This error reporter was intentionally designed in a way that it would only mark a code part unreachable if the underlying analysis can guarantee that the code part is indeed unreachable. Without inter-procedural reasoning, the analysis could not guarantee this for our last example above, so there was no error highlighting at all. This is to avoid potential *false positive* alarms, as those constitute one of the primary reasons why developers disuse static analyses [65]. In turn, the analysis has high accuracy. Finally, the intra-procedural analysis has good *performance*, as CLion could deliver updated analysis results after every keystroke without interrupting the development flow.

Now, let us consider how the design trade-offs would change with the inter-procedural unreachable code analysis. First, it would have improved reliability due to the inter-procedural reasoning, so it could detect more bugs. Second, we could keep accuracy the same as before by not changing the error reporter, so that we would still aim to rule out false alarms as much as possible. Third, performance would get a hit, as functions cannot be analyzed in isolation anymore. We must consider the effect of all transitively reachable functions when statically examining a function call. As this can easy involve a long chain of function calls, re-analysis after program changes may incur significant computational cost, hindering the possibility to provide live feedback to developers.

The perceived reader may wonder about two questions at this point. *Why would a static analysis ever report false (positive or negative) results? Would it be possible to just maximize all design trade-offs?* The answers to these questions lie in a fundamental work from the field of computability theory called Rice's theorem [99]. The theorem states that a fully reliable and fully accurate analysis (regarding a program's output behavior) cannot terminate for all possible subject programs. Therefore, a static analysis must either work in a best-effort manner with reduced reliability, or it must approximate the program behavior with reduced accuracy. We have already seen how the unreachable code analysis in CLion works in a best-effort manner; It tries its best to help developers spot unreachable code parts, as long as the analysis does not require inter-procedural reasoning. The price of this behavior is that unreachable code parts may go undetected. To demonstrate approximations, consider that we equip the unreachable code analysis with the ability to reason about loop conditions, so that it can statically identify unreachable loop bodies. To this end, the analysis must be able to reason about loop conditions, which entails e.g. reasoning about the possible runtime values of number-typed variables. Of course, one option is to accurately track all possible values of such variables, but this can easily become computationally intractable or lead to nontermination. Instead of reasoning over the *concrete* value domain of integers, the analysis could resort to use an *abstract* domain that somehow abstracts over the concrete one, e.g. by only telling the smallest interval that contains all possible values or by only tracking the possible sign of values. It is important to see that by making the abstract domain capture less information about the concrete one, we can make the analysis terminate and even achieve good performance. However, the price of the approximations is that an analysis may not be able to faithfully reason about all possible concrete executions of a program. For example, a sign domain is not sufficient to determine the outcome of a parity check. In turn, the analysis may report false alarms that real executions would not exhibit.

Finding the right balance between reliability, accuracy, and performance typically boils down to a careful analysis of the needs of analysis clients. Let us look at a few example application areas and what analysis clients expect in terms of the trade-offs:

- *Providing feedback in IDEs*: Performance is crucial. Sub-second analysis time is expected, as interrupting the development flow with lengthy pauses is counterproductive and hinders the adoption of static analyses, as also demonstrated by Johnson et al. in their survey with software engineers [65]. While retaining high performance, reliability and accuracy are to be maximized.
- *Optimizations in compilers*: Compilers can settle with longer run times of static analyses because build times on real-world code bases can easily take several minutes, so adding a few extra minutes to run an analysis may not be an issue. However, optimizations in compilers require high reliability, otherwise an optimization may alter the behavior of the subject program.
- *Pull requests*: CI servers must deliver feedback on pull requests in at most a few tens of minutes. Sadowski et al. report that the Tricorder analysis framework must produce feedback as a bot reviewer on a pull request on Google-scale code bases in less than 5-10 minutes, otherwise developers switch context and start working on other issues, severely limiting the perceived usefulness of analysis results [103]. Sadowski et al. also argues that high accuracy is of paramount importance because developers quickly lose trust in static analysis tools if presented with false alarms.
- *Major software releases*: Certain static analyses may be too expensive to compute on every commit or build. For example, Jordan et al. perform security analysis on the entire JDK with the Soufflé analysis framework in around 15 hours, requiring 75 GB of memory [66]. In return, the analysis has both high accuracy and reliability because it reasons inter-procedurally. It may be sufficient to perform such an analysis only for every major release of a software.

Our goal is to speed up static analyses, so that they can serve the basis for continuous feedback in IDEs, while sacrificing as little as possible on reliability or accuracy. This way we can shift the landscape of the above examples by delivering sub-second run times for analyses that were originally not amenable to applications in IDEs.

1.2 Techniques for Improving the Performance of Static Analyses

There is a long line of research investigating how to improve analysis performance with different techniques ranging from specialized algorithms to more generic techniques like parallelism or incrementality. We survey the relevant literature in Chapter 8 in more details, here we highlight a select few of the approaches. This list is already sufficient to make an important observation: There is no single best technique for every application domain, as each one of these techniques come with limitations, as well:

- *Specialized algorithms or data structures* are frequently used to speed up one-off static analyses for a specific subject language. For example, Dietrich et al. designed specialized algorithms for Java to compute the points-to information of variables, and they managed to analyze the entire JDK in a minute [32]. A limitation of this approach is that the high development effort of such analyses must be spent on a case-by-case basis.
- A *compositional analysis* analyzes parts of a subject program in isolation and then composes the analysis result from the smaller pieces. For example, enforcing secure coding standards (e.g. MISRA [56]) falls into this category because individual code parts (e.g. functions) can be analyzed in isolation, as they do not have dependency on any other code part. Type checkers are also often composable because typically functions can be analyzed in isolation by relying on the type signatures of other functions. A limitation of this approach is that many practically-relevant analyses do not compose well due to complex dependencies in subject programs.
- *Parallelization* of analysis subtasks is an obvious idea when running on multi-core hardware. For example, Méndez-Lojo et al. present an approach for parallelizing a points-to analysis, and they achieve a three-fold speedup compared to the sequential version of the same analysis [83]. However, efficient parallelization requires regular computational problems, that is, when the subtasks can be distributed equally to processors *prior* to the computation. In contrast, computational problems are often irregular where the workload distribution can only be determined at runtime. Méndez-Lojo et al. devised specialized solutions to make points-to analysis amenable to parallelization.
- A *demand-driven analysis* only produces results that are asked for and when they are asked for by analysis clients, instead of eagerly performing analysis on the entire subject program. The Boomerang framework exploits this idea, and it can deliver highly precise points-to information for Java on the ballpark of few seconds when asked at a specific call site of a function [118]. Unfortunately, this approach is limited to certain classes of analyses and cannot be applied generally, as we discuss in Section 8.2.
- The idea of *incrementalization* is to efficiently reuse previously computed analysis results. An incremental analysis entails two steps. First, perform a from-scratch analysis on an entire subject program to compute the initial analysis result. Then, when the subject program changes, reuse the previous result and update it based on

the changed code parts. This is in contrast to repeated re-analysis from scratch on the entire subject program, and it can lead to significant speed-ups. For example, Erdweg et al. have shown that incremental type checking can achieve order-of-magnitude speedups compared to non-incremental re-analysis and deliver update times that are fast enough for live feedback in IDEs [36]. However, incremental analyses are often one-off solutions with specialized algorithms, and it is often not obvious how to achieve good incremental performance in general.

We study how to use incrementality to speed up static analyses in this dissertation. We chose this technique because of two primary reasons. First, it has been shown both in the domain of program analysis (see above) and even outside of program analysis that incrementalization can bring asymptotic performance improvements over non-incremental computations; e.g. computational geometry [67], machine learning [122], or big data processing [18]. Second, incrementalization perfectly aligns with analyses running in IDEs, as developers typically modify subject programs with *small* changes compared to the size of the entire subject program. Due to the small changes, we should be able to update analysis results significantly faster than a non-incremental re-analysis. However, incrementalizing static analyses entails significant challenges, as we discuss next.

1.3 Challenges in Incrementalizing Static Analyses

The number one challenge for incrementalization is *correctness*. In response to a program change, a correct incremental analysis yields the exact same result as its non-incremental counterpart. We use Figure 1.2 to illustrate how an incremental analysis works in comparison to a non-incremental one. Assume that we run the unreachable code analysis on some C program $\texttt{Program}_1$. The analysis produces the initial results $\texttt{Result}_1$ in the form of a set consisting of labels of statements that are unreachable. Then, a developer changes $\texttt{Program}_1$ by adding or removing statements from the program to obtain $\texttt{Program}_2$. The delta between the two programs consists of the set of inserted and deleted statements denoted by $\Delta(\texttt{Program}_1)$. A non-incremental unreachable code analysis re-analyzes $\texttt{Program}_2$ from scratch to obtain the new result $\texttt{Result}_2$. In contrast, an incremental unreachable code analysis reuses the old results $\texttt{Result}_1$ and only re-analyzes the changed code parts $\Delta(\texttt{Program}_1)$ to obtain $\Delta(\texttt{Result}_1)$, which also consists of labels of deleted and inserted statements. Correctness requires that $\texttt{Result}_2$ is the same as $\texttt{Result}_1 + \Delta(\texttt{Result}_1)$. Ensuring correct incremental updates requires *dependency tracking*. This means that an incremental analysis must be able to precisely identify which part of an analysis result depends on which part of the subject program, so that is can invalidate affected parts of the old results and then compute additional new results.

Assuming correct incremental updates, we can now turn our attention to a number of other challenges that affect the performance of incremental analyses. An incremental analysis can only yield performance improvements over the non-incremental counterpart *if* the computational effort for computing the output delta is proportional to the size of the *input delta* and not the size of the *entire input*. One may think that there is no problem here at all, as $\Delta(\texttt{Program}_1)$ is guaranteed to be small, given that developers insert and delete statements in succession, and we could just rerun the incremental analysis after

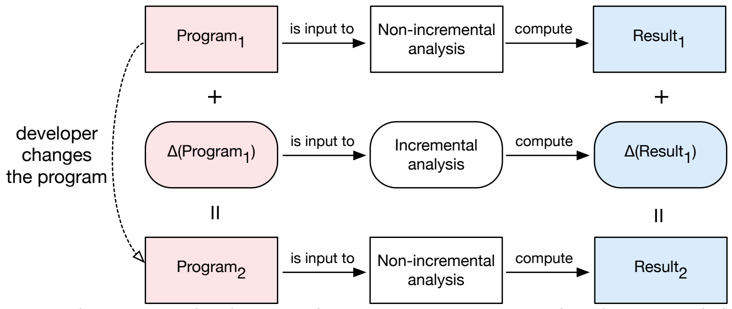


Figure 1.2: Incrementalization visualized. Rectangles concern non-incremental analyses, rounded ones concern incremental analysis. Red color represents input to the analysis, blue color represents output of the analysis.

every program change. However, practically relevant analyses that deliver on reliability and accuracy often exhibit features that can significantly increase the computational cost of updating analysis results even if the program change $\Delta(\texttt{Program}_1)$ is small. We discuss three such features and the challenges they pose for efficient incrementalization:

- *Complex recursive dependencies*: As we wrote above, an incremental static analysis requires dependency tracking to identify which part of the analysis result must be recomputed after a program change. However, precise analyses have many recursive components that need to work together in an intertwined manner, which significantly complicates dependency tracking. For example, reasoning about unreachable code requires the construction of a control flow graph. Given that a C program may use function pointers, we must also perform a points-to analysis, so that we can keep track of the potential targets of a pointer variable, which then allows us to resolve the call targets. Then, the set of targets may change as the call graph is constructed. This shows that points-to analysis and call graph construction need to go hand in hand, reinforcing each other through recursive dependencies. When a subject program changes, we must follow all these recursive dependencies to correctly update the analysis result.
- *Lattices and recursive aggregation*: Static analyses routinely use custom lattices for the representation of the abstract domain [90]. Lattices are partially ordered sets with aggregation operators to compute the least upper bound or greatest lower bound of any two abstract values. Aggregation operators play an important role when analyzing subject programs with cyclic control flow, as it is common to approximate a set of lattice values computed over the different control flow paths with a single aggregate lattice value. Due to cyclic control flow, aggregation also becomes recursive, and the analysis may need to iterate and aggregate repeatedly when analyzing loops until the analysis result stabilizes. This severely complicates efficient reuse of previous results in an incremental setting because a change in the subject program may require a complete unrolling of the previous results, resulting in a complete re-analysis of loops through several iterations again.
- *Inter-procedurality*: Analyzing across function calls opens up a whole new realm of challenges to efficient incrementalization because even a small program change can have a large impact on the overall analysis result. For example, it can easily

happen that a developer mistake in a frequently used library function makes code in a large number of call sites unreachable. This shows that the output delta of an inter-procedural unreachable code analysis may very well be proportional to the input size rather than the size of the input delta. The large impact of the change makes it difficult to deliver on the promise that incrementality comes with significant improvements over a from-scratch analysis.

Although, this list may not be complete, it will become clear when we survey the related work in Chapter 8 that this list is representative in the sense that state-of-the-art static analyses often use some or all of these features to reason about realistic subject programs. In the following, we simply use the word *sophisticated* to refer to analyses that use the above features, even if the word itself has a subjective connotation. This dissertation claims the following thesis:

Incrementalization can significantly improve the performance of sophisticated static analyses. We can achieve this automatically while shielding analysis developers from the technical details of incrementalization.

1.4 Incremental Analysis Framework

We show the thesis to be true by designing and implementing an incremental static analysis framework in this dissertation. We call our framework IncA, which is short for *INC*remental *A*nalysis. IncA automatically incrementalizes static analyses, thereby shielding analysis developers from the technical details of incrementalization. IncA does not make assumptions or restrictions about precision aspects (accuracy or reliability) of analyses, finding the right design trade-off between those is the job of the analysis developer, just like in a non-incremental analysis. The following requirements guide our design and implementation throughout this dissertation:

Correctness (R1): An analysis incrementalized by IncA must compute the same result as its non-incremental counterpart. In other words, incrementally updating an analysis result with IncA in response to a program change must be the same as if we would run the non-incremental analysis from-scratch on the entire updated subject program.

Efficiency (R2): An analysis incrementalized by IncA should be significantly faster than its non-incremental counterpart. We expect to deliver update times on the sub-second ballpark, as we target applications in IDEs that require live feedback.

Expressiveness (R3): IncA must support sophisticated static analyses, that is, it must support recursive analysis specifications, custom lattices with aggregation, and inter-procedural reasoning.

Genericity (R4): IncA must be independent of any particular subject program, subject language, or static analysis.

Declarativity (R5): IncA must automatically incrementalize analyses and hide the technical details of incrementalization from analysis developers.

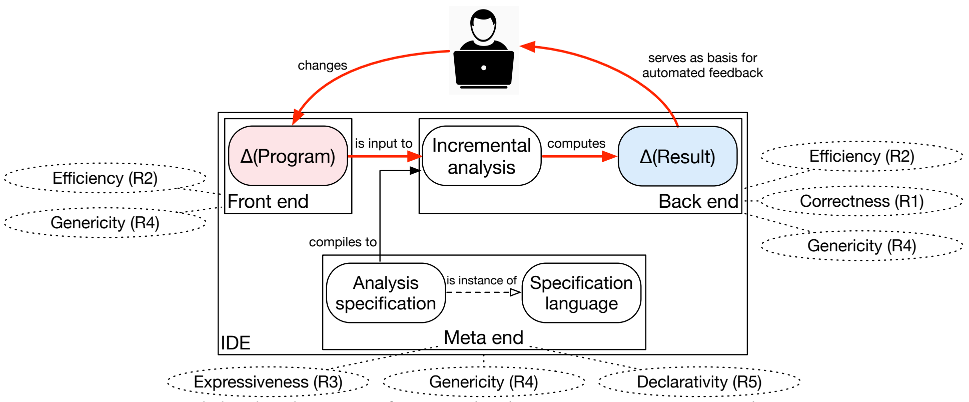


Figure 1.3: High-level architecture of IncA with relevant requirements mapped to components.

Figure 1.3 shows the high-level architecture of IncA. The central element of the architecture is the incremental feedback cycle highlighted in red: (i) A developer changes a subject program, (ii) the delta in the subject program is the input to the incremental analysis, (iii) the incremental analysis computes the delta in the analysis result, and, finally (iv) the result delta is used as a basis for automated feedback in the IDE. It is crucial that this feedback cycle is re-evaluated in sub-second time after program changes, so that we can avoid interrupts in the developer workflow. IncA consists of three main parts, each one being responsible for satisfying one or more requirements of the thesis statement:

Front end This is the component that *software developers* interact with; This is where they develop subject programs. The front end of IncA efficiently computes deltas in the subject program, thereby enabling incremental analyses. The front end is independent of any particular subject program or subject language.

Meta end This is the component that *analysis developers* interact with. To shield analysis developers from the technical details of incrementalization, IncA provides a declarative language for analysis specification. As we will show throughout this dissertation, the specification language is expressive, as it supports a wide range of static analyses, and it is independent of any particular subject language or analysis.

Back end The back end is responsible for the incrementalization of static analyses. This component comes with specialized algorithms and data structures for efficient and correct incremental analyses. The back end is generic because it incrementalizes whatever analysis the meta end can express.

In the following, we briefly review the concrete contributions of this dissertation.

1.5 Contributions

The current design and features of IncA is the result of several iterations and improvements. Concretely, we present five contributions that follow the development history of IncA:

The IncA incremental analysis framework First, we create a baseline version of IncA. This is already a complete incremental analysis framework in the sense that it implements all three "ends". The primary focus here is on supporting analyses with complex recursive dependencies.

In the front end, we use projectional editing, which is a generic editor technology independent of a subject language. Changes in subject programs directly translate to low-level transformations of the underlying abstract syntax tree (AST) in a projectional editor. This aligns perfectly with our incremental approach, as AST deltas can be computed with zero computational overhead in a projectional editor. We use Datalog in the meta end as the specification language. Datalog is a declarative language that allows to encode analyses through relations. It allows analysis developers to focus on what to compute in terms of relations, but incrementalization is left to a Datalog *solver*. We use a state-of-the-art solver algorithm in the back end. This algorithm provides efficient and correct incremental updates for any kind of recursive analyses.

We use IncA to implement practically relevant program analyses: FindBugs for Java to detect common bug patterns, well-formedness checks for domain-specific languages (DSLs), and control flow and points-to analysis for C. We show that IncA is efficient because it delivers sub-second update times for these analyses.

Incrementalizing lattice-based program analyses A limitation of baseline IncA is that analysis developers can only implement analyses that compute sets of tuples in relations as the analysis result. This is equivalent to using the powerset lattice for the abstract domain. While this is sufficient to relate existing entities to each other, for example to construct a control flow graph, it is not sufficient for many other practically relevant analyses that also need to compute and aggregate abstract values as the analysis runs. For example, an interval analysis computes and aggregates interval lattice values at runtime.

We improve the expressive power of IncA by extending its specification language with support for lattices and aggregation. However, we not only need to extend the language, but also the IncA compiler and back end to make them lattice-aware, as well. We show that state-of-the-art solver algorithms are not fit to support lattice-based analyses. To this end, we design a novel algorithm called DRed$_L$ tailored specifically to this problem.

We use IncA equipped with DRed$_L$ to implement a lattice-based points-to analysis for Java and analyses that reason about string values in Java. Our benchmarking on real-world subject programs demonstrate that IncA is efficient, as the analyses deliver millisecond update times.

Incrementalizing lattice-based inter-procedural analyses We identified three main obstacles to efficient incrementalization above: recursive dependencies in analyses, lattices and aggregation, and inter-procedurality. We already discussed the first two, now we account for inter-procedural analyses. We show that the specification language of IncA is already expressive enough to implement inter-procedural lattice-based analyses. The problem is with the back end. As we explained before, inter-procedurality can easily compromise the potential performance gains over a from-scratch recomputation. We find that DRed$_L$ and other state-of-the-art solvers have either poor performance or do not even terminate for realistic implementations of lattice-based inter-procedural analyses.

To efficiently incrementalize lattice-based inter-procedural analyses, we design a novel solver algorithm called Laddder. On the one hand, Laddder uses a non-standard aggregation semantics compared to prior approaches (including DRed_L) which helps to improve the performance of lattice aggregators. On the other hand, Laddder performs extra bookkeeping during fixpoint computation, and this helps to better reuse previous results, which is crucial in an inter-procedural setting.

Inter-procedural analyses need a new approach for benchmarking, as well. Aiming to deliver sub-second raw performance for *any* kind of change for an inter-procedural analysis is hopeless. For example, changing frequently used library functions will always have a large impact on the overall result. It is exactly this impact that our evaluation takes into account. We define the impact of a change as the size of the difference between the old and the new analysis result. In our evaluation, we analyze real-world subject programs with a lattice-based points-to analysis. We empirically show that, given a random series of program changes, low-impact changes dominate, and IncA is efficient in handling those, while, for high impact changes, IncA is at least as fast as a from-scratch re-analysis. Considering all program changes, the average update time is in the sub-second ballpark.

A DSL for incremental static analyses Throughout the implementation of the case studies discussed so far, we used Datalog and lattices in IncA. However, we identified recurring patterns of analysis code and missing language constructs that would make the analysis implementation more readable and less verbose. Based on these observations, we design a declarative DSL for incremental static analyses. Compared to Datalog extended with lattices, our DSL has the same expressive power, it is also generic, but it has several syntactical differences. For example, an analysis encoded in our DSL reasons in a forward or backward style with input and output parameters which is common for data-flow analyses. This is in contrast to computing bare relations with Datalog. Moreover, we add new language constructs in our DSL, such as pattern matching to easily deconstruct ASTs, switch-case statement, if statement, and more. We show how these constructs can be mapped to Datalog.

We demonstrate the use of our DSL through two large-scale case studies. First, we implement overload resolution for Featherweight Java, and we show how does the new DSL help to improve readability and conciseness compared to the functionally equivalent implementation in Datalog. Second, as part of a Master's thesis [21], the new DSL was used to implement a type checker for Rust and other static analyses used in the official Rust compiler. This is the biggest IncA case study so far with 4 KLoC in size for the analysis implementation. We briefly report on this case study, as well.

Textual front end for incremental program analysis Our last contribution is centered around making IncA available for textual front ends, as well. This is important because textual IDEs are the norm rather than projectional editing. However, a major challenge in moving to a textual front end is that deltas in the subject program are not computed for free, which was the case with projectional editing. For textual front ends, we must use a parser to produce the AST, and then we must compute the differences between the old and new ASTs with tree diffing. If this process takes too long, we cannot deliver efficient incremental updates.

We show that with existing state-of-the-art techniques, it is not parsing that is the performance bottleneck, but tree diffing. We find that most tree diffing tools are not satisfactory for our purposes, as they (i) are either slow, (ii) only work with a particular subject language, or (iii) encode AST differences imprecisely. Only recently did Miraldo et al. introduce a generic tree diffing algorithm called *hdiff* that does not have any of these limitations [85]. We adopt *hdiff* and build a textual front end for IncA that is as efficient as the version using projectional editing. We show that our approach can deliver AST diffs in a few tens of milliseconds on average for evolving textual Python code.

We have realized all work described in this dissertation in concrete implementations to guide the design of IncA and to perform benchmarking with real-world static analyses. IncA itself and our benchmarks (analyses and harnesses) are available as open source software at `https://github.com/szabta89/IncA`.

1.6 Outline

The structure of this dissertation directly follows the historical development of IncA. The contributions of this dissertation have either been previously published or are currently under submission by the author in collaboration with others at international conferences and workshops. In the beginning of each chapter, we indicate which paper was the source of the chapter.

- Chapter 2 provides background on Datalog. We review the syntax, semantics, and algorithms for the incrementalization of standard Datalog. This chapter can be skipped if the reader is already familiar with Datalog.
- Chapter 3 presents the design and implementation of the baseline IncA framework.
- Chapter 4 presents our approach for incrementalizing analyses with lattice-based aggregation. We discuss the DRed_L algorithm in detail.
- Chapter 5 adds support for inter-procedural lattice-based analyses. The chapter is centered around our Laddder algorithm.
- Chapter 6 presents the IncA DSL used for incremental static analysis and the case study on implementing overload resolution for Featherweight Java.
- Chapter 7 presents the design and implementation of a textual front end for IncA.
- Chapter 8 puts IncA and our contributions in context with related work.
- Chapter 9 concludes this dissertation and provides an outlook on possible future directions.

2 Static Analysis with Datalog

This chapter shares material with the ASE'16 paper "IncA: A DSL for the Definition of Incremental Program Analyses" [125].

Abstract — Datalog is a logic programming language that is a popular choice when it comes to implementing static analyses. IncA is also a Datalog-based analysis framework. This chapterprovides background material: It reviews the syntax, semantics, and state-of-the-art algorithms for the incrementalization of standard Datalog.

2.1 Introduction to Datalog

Datalog is a logic programming language that originates from the 80s from the database community. Originally, it was used as a specification language for database queries [2]. The language was dormant for a long time, but, in recent years, it started to gain more and more attention in a number of different domains; e.g. distributed computing [27], networking [13], security [66], and program analysis [12, 112]. Datalog allows to formulate queries over relational data in a declarative style. This means that developers focus on *what* to compute in terms of relations and tuples instead of *how* to execute a query. The "how" is left to a Datalog *solver* that computes the result of a Datalog program. State-of-the-art solvers are actually really good at solving the question of "how", as they implement a range of optimizations, like multi-threaded execution, incrementality, or automatic tuning of indexing, that all Datalog programs can benefit from. In the following, we first give an introductory overview on Datalog through an example static analysis. Then, we review recursion and fixpoint computation, both being fundamental to Datalog. Finally, we discuss state-of-the-art algorithms for incrementalizing standard Datalog. For an in-depth overview on Datalog, we refer the reader to *Datalog and Recursive Query Processing* [44], we resort to only discussing details here that are relevant to the contents of this dissertation.

2.2 Datalog by Example

Control flow analysis As an introductory example, we build a simple control flow analysis for C using Datalog. IDEs and compilers use control flow analysis to reason about the execution order of statements in a program and as a building block for further analyses such as uninitialized variables or points-to analysis.

The input of our control flow analysis is a subject program in C, and its output is a control flow graph (CFG) [90]. As an example, consider the C program and its CFG in Figure 2.1. Each node in the CFG represents a statement in the program. A source statement is connected to a target statement in the CFG if there exists an execution trace in which the execution of the source statement immediately precedes the execution of the target statement. Note how control can flow from statement 3 both to 3a and 4 in Figure 2.1. This is because, in the general case, we cannot determine statically what the condition of the `while` statement evaluates to, so, to faithfully represent all possible control flows, we consider both 3a and 4 as potential targets.

```
int temp = readSensor(...);          (1)
int last, err;                       (2)
while (outOfRange(temp)) {           (3)
  last = err;                        (3a)
  temp = readSensor(...);            (3b)
}
log("Last temperature %d", last);   (4)
```

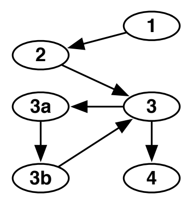


Figure 2.1: A simple C program and its CFG.

```
CFlow(src, trg) :- CSimple(src, trg).
CFlow(src, trg) :- CWhile(src, trg).

CSimple(src, trg) :- PrecedingStatement(src, trg), SimpleStatement(src).

CWhile(src, trg) :- PrecedingStatement(src, trg), WhileStatement(src).
CWhile(src, trg) :- WhileStatement(src), FirstStatement(src, trg).
CWhile(src, trg) :- WhileStatement(trg), LastStatement(trg, src).
```

Figure 2.2: Control flow analysis for a subset of C in Datalog.

The result of a control flow analysis is the CFG, but, in Datalog, we model the CFG with a relation consisting of a set of tuples representing CFG edges. For this example, we restrict ourselves to a subset of C with `while` statements and simple statements that entail a sequential control flow such as assignments. We define the relation `CFlow` for the edges of the CFG, and we compute it as the union of two helper relations:

1. **CSimple** consists of those control flow edges `(src, trg)` where `src` is a simple statement that syntactically precedes `trg` in the source code.
2. **CWhile** consists of control flow edges `(src, trg)` that lead into or out of a `while` statement, which is the case if:

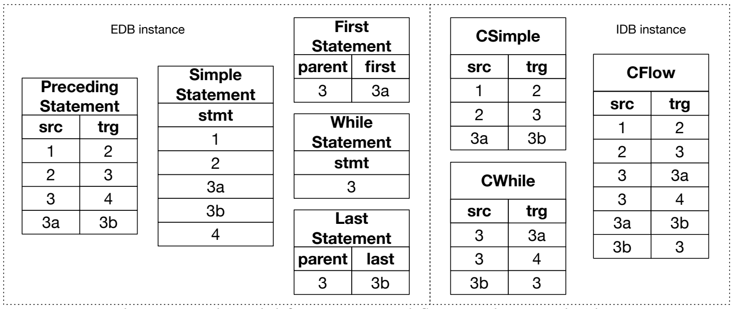
   2.1. `src` syntactically precedes `trg`, and `src` is a `while` statement,
   2.2. `src` is a `while` statement, and `trg` is the first statement in its body,
   2.3. `src` is the last statement in the body of the `while` statement `trg`.

In our example, `CSimple` consists of the tuples `(1, 2)`, `(2, 3)`, and `(3a, 3b)`, and `CWhile` consists of `(3, 3a)`, `(3, 4)`, and `(3b, 3)`. The union of the two relations constitutes `CFlow`. Now, let us implement this analysis with Datalog to compute these tuples.

Analysis implementation in Datalog A Datalog program consists of a set of rules that compute sets of tuples in relations. In standard Datalog [44], a rule r has the form $h :- a_1, \ldots, a_n$ where h is called the rule *head*, and $a_1, \ldots, a_n$ form the rule *body*. h and a_i are called *atoms*. An atom is a relation name (predicate symbol) and a list of *terms* as arguments. A term can either be a constant value or a variable. An atom consisting solely of constant values is called a *ground* atom or *fact*.

Figure 2.2 shows an excerpt of the Datalog rules that compute the tuples of the `CFlow` relation. The `CFlow` rule has two alternative bodies: The first one collects the edges concerning simple control flow predecessors from the `CSimple` rule, while the second one collects the edges concerning `while` statements from `CWhile`. The definition of `CSimple` and `CWhile` directly follows the above reasoning about possible control flows.

We turn our attention now to computing the tuples of `CFlow`. However, first, we introduce necessary terminology. The set of all predicate symbols appearing in a Datalog program is called a *database*. The *extensional database* (EDB) consists of all those predicate symbols that represent the subject program, while the *intensional database* (IDB) is the collection of derived predicate symbols computed by the Datalog program. For example, in Figure 2.2, `CSimple`, `CWhile`, and `CFlow` constitute the IDB, while the other predicate


EDB instance
IDB instance

Preceding Statement

src	trg
1	2
2	3
3	4
3a	3b

Simple Statement

stmt
1
2
3a
3b
4

First Statement

parent	first
3	3a

While Statement

stmt
3

Last Statement

parent	last
3	3b

CSimple

src	trg
1	2
2	3
3a	3b

CWhile

src	trg
3	3a
3	4
3b	3

CFlow

src	trg
1	2
2	3
3	3a
3	4
3a	3b
3b	3

Figure 2.3: The minimal model for our control flow analysis and subject program.

symbols represent the EDB. A *database instance* is a set of facts that constitute the relations represented by the predicate symbols of a database. The *EDB instance* consists of facts that describe the input of the Datalog program. In our example, the constant values in the facts are statements of the subject program. For example, `SimpleStatement` is a unary relation that enumerates statements that are not `while` statements, while `PrecedingStatement` consists of those `(src, trg)` facts where statement `src` syntactically precedes statement `trg`. It is common in Datalog-based analysis frameworks to extract these facts from the data structure (e.g. the AST) representing a subject program.

Model-theoretic semantics There are different ways to define the semantics of a Datalog program. Here, we focus on the *model-theoretic semantics*, which captures the declarative nature of Datalog, that is, it explains what the *IDB instance* consists of, but it does not talk about how to compute that.

A rule is interpreted as a universally quantified implication: The substitutions of the variables in the body imply when the head holds. Given a head atom $R(t_1, ..., t_k)$, the valid substitutions of the terms $t_1, ..., t_k$ yield the tuples of relation R. Multiple Datalog rules can share the same predicate symbol, thereby providing alternative ways to infer tuples in the corresponding relation.

Starting from an (extensional) database instance I, and using a Datalog program P, the job of a Datalog solver is to compute a *minimal model* M of P. A model is a database instance I' that extends I (that is, $I \subseteq I'$), plus it satisfies all Datalog rules when treated as constraints. Given that there may be multiple models satisfying this criteria, a Datalog solver must compute the minimal one, that is, the one that has the least number of tuples out of all models. It has been shown that for any standard Datalog program and *finite* EDB instance, there exists a unique minimal model [132]. Figure 2.3 shows the minimal model computed by our example analysis in Figure 2.2 on the subject program from Figure 2.1. The tuples in the EDB instance are simply extracted from the AST of the subject program. The IDB instance consists of all those CFG edges that satisfy the Datalog rules.

The model-theoretic semantics specifies what to compute as a result of a Datalog program, which aligns perfectly with the declarative nature of the language. However, it does not tell us how to actually obtain such a result. To actually compute the minimal model, we use an alternative semantics called the *fixpoint-theoretic semantics*. Instead of continuing with the control flow analysis in itself, we actually build an analysis on top of the control flow analysis. We do this for two reasons. First, our new example analysis will use recursion, which is a fundamental improvement in the expressivity of Datalog. Note that our control flow analysis is not recursive, as there are no recursive dependencies among its Datalog rules. The fact that the CFG was cyclic for the subject program in Figure 2.1 was because of the loop in the program itself. Second, fixpoint computation is much more interesting for a recursive analysis.

2.3 Recursion and Fixpoint Computation

Uninitialized variables analysis We extend our running example with an uninitialized variables analysis [90], which will be a client of the control flow analysis. The uninitialized variables analysis collects all those variables that *may* not be initialized during the execution of a subject program. Using this information, an error reporter can mark all those reads erroneous where the variable is not guaranteed to be initialized. We use a flow-sensitive analysis, which means that the analysis keeps track of the control flow and computes the set of uninitialized variables per CFG node. Given that subject programs may have cyclic control flow (as is the case in Figure 2.1), we will use a recursive analysis definition.

Consider the subject program in Figure 2.1. Statement 2 declares variables `last` and `err`, but does not initialize them. From here, the uninitialized state spreads across the CFG edges. Even though, `last` gets assigned in statement 3a, due to a developer mistake, it gets assigned with `err` instead of with `temp`. Variable `err` is uninitialized still, so `last` also remains uninitialized. When we reason about the variables at the loop (represented by statement 3), we must consider both branches `2` $\longrightarrow$ `3` and `3b` $\longrightarrow$ `3` that lead to the condition of the loop. Both `last` and `err` are uninitialized on both branches, so the variables are considered uninitialized at statements 3 and 4. Based on this result, the read from `last` at statement 4 is erroneous.

Analysis implementation in Datalog Figure 2.4 shows the implementation of a simple uninitialized variables analysis in Datalog. The two main rules are `UninitializedBefore` and `UninitializedAfter`, and they are mutually recursive. Rule `UninitializedBefore` propagates results from CFG predecessors to CFG successors. A tuple `(stmt, var)` $\in$ `UninitializedBefore` means that `var` is uninitialized before the execution of statement `stmt`. Rule `UninitializedAfter` interprets the different kinds of statements. A tuple `(stmt, var)` $\in$ `UninitializedAfter` means that `var` is uninitialized after the execution of statement `stmt`. This can happen in three ways: (i) `stmt` declares `var`, but does not initialize it (e.g. statement 2 in Figure 2.1), (ii) `stmt` does not declare nor assign to `var`, but `var` was already uninitialized before, and (iii) `stmt` does not declare `var` but assigns a variable reference to it where the referenced `otherVar` is uninitialized. The new syntax ‘‘`not`’’ appearing in line 7 stands for negation, while ‘‘`_`’’ is a wildcard variable that can be substituted with any value. The whole atom `not Assignment(stmt, var, _)` means that `var` does not get assigned in `stmt` because there is *no* value `rhs` that would yield `(stmt, var, rhs)` $\in$ `Assignment`.

Fixpoint-theoretic semantics The *fixpoint-theoretic semantics* specifies *how* to compute the minimal model of a Datalog program given an EDB instance. We apply it to the uninitialized variables analysis to compute its results on the subject program in Figure 2.1.

Given a Datalog program P, the *immediate consequence operator* T_P is a function that

```
UninitializedBefore(stmt, var) :-
  CFlow(src, stmt),
  UninitializedAfter(src, var).

UninitializedAfter(stmt, var) :-
  VariableDeclaration(stmt, var),
  not Assignment(stmt, var, _).
UninitializedAfter(stmt, var) :-
  not VariableDeclaration(stmt, var),
  not Assignment(stmt, var, _),
  UninitializedBefore(stmt, var).
UninitializedAfter(stmt, var) :-
  not VariableDeclaration(stmt, var),
  Assignment(stmt, var, rhs),
  VariableReference(rhs, otherVar),
  UninitializedBefore(stmt, otherVar).
```

Figure 2.4: Initialized variables analysis for a subset of C in Datalog.

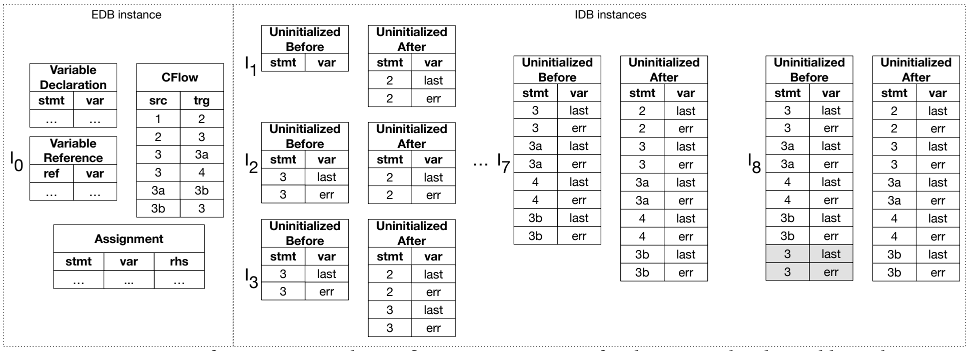
maps a database instance to a database instance. We say that a database instance I' is the immediate consequence of the database instance I if every tuple t in I' is either already in I or t can be derived using one of the rules in P based on tuples that are in I. I' is the result of applying T_P on I. T_P is a monotone operator, that is, if $I_1 \subseteq I_2$ then $T_P(I_1) \subseteq T_P(I_2)$. The process of solving P with T_P based on an EDB instance entails the repeated application of T_P until a fixpoint is reached, that is, until we cannot infer any new tuples. Assuming standard Datalog, it can be shown that (i) this process always terminates, (ii) it always computes the least fixpoint, that is, the smallest among the derivable database instances, and (iii) the least fixpoint equals to the minimal model defined by the model-theoretic semantics [44].

Figure 2.5 shows the fixpoint computation for the uninitialized variables analysis on the subject program in Figure 2.1. I_0 is the EDB instance, which does not only contain facts about the subject program, but also the result of the control flow analysis, as the uninitialized variables analysis makes use of it. The fixpoint computation finishes in eight steps, the final IDB instance being I_8. We omitted $I_4 - I_6$ to improve readability. Note that in I_8, we obtain *redundant derivations* `UninitializedBefore(3, last)` and `UninitializedBefore(3, err)` (marked with gray), as they have already been derived in I_2. There is no need to apply T_P again, as we did not infer *new* tuples in `UninitializedBefore` in I_8.

Negation As we explained before, the T_P operator is monotone, that is, repeated applications of it only ever grows the number of tuples in consecutive database instances. However, negation applied through recursion can easily violate this property, as it may require to retract tuples that have already been inferred in a previous application of T_P. One way to deal with this situation is to restrict the use of negation.

A widely used restriction is to only allow *stratified negation* in Datalog programs [44]. A *strata* is a set of mutually recursive predicate symbols. A Datalog program is stratifiable if there exists an ordering among the strata $S_1, S_2, \ldots, S_n$ of the program such that

- when a rule computing relation $R \in S_i$ uses relation $R' \in S_j$ in its body in a non-negated atom, then $i >= j$,


EDB instance
IDB instances
I0
I1
I2
I3
… I7
I8

I0 — Variable Declaration

stmt	var
…	…

I0 — Variable Reference

ref	var
…	…

I0 — CFlow

src	trg
1	2
2	3
3	3a
3	4
3a	3b
3b	3

I0 — Assignment

stmt	var	rhs
…	…	…

I1 — Uninitialized Before

stmt	var

I1 — Uninitialized After

stmt	var
2	last
2	err

I2 — Uninitialized Before

stmt	var
3	last
3	err

I2 — Uninitialized After

stmt	var
2	last
2	err

I3 — Uninitialized Before

stmt	var
3	last
3	err

I3 — Uninitialized After

stmt	var
2	last
2	err
3	last
3	err

I7 — Uninitialized Before

stmt	var
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err

I7 — Uninitialized After

stmt	var
2	last
2	err
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err

I8 — Uninitialized Before

stmt	var
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err
3	last
3	err

I8 — Uninitialized After

stmt	var
2	last
2	err
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err

Figure 2.5: Series of IDB instances during fixpoint computation for the uninitialized variable analysis.

- when a rule computing relation $R \in S_i$ uses relation $R' \in S_j$ in its body in a negated atom, then $i > j$.

Stratified negation is a syntactic restriction on Datalog programs, and it helps to keep T_P monotone, as it cannot happen that negation is used through recursion. Our uninitialized variables analysis in Figure 2.4 uses stratifiable negation, as it satisfies the above criteria. Note that there are also other ways to deal with negation in Datalog. For example, researchers have proposed the well-founded semantics [133] that gives meaning to Datalog programs with unrestricted use of negation.

Note on technical realization The ordering among strata defined by the above stratification criteria can also be combined with the fixpoint computation, in contrast to applying the entire set of rules in each fixpoint iteration. Specifically, for each strata S according to the computed order, we repeatedly apply $T_{P[S]}$ until fixpoint in S, where $T_{P[S]}$ denotes the set of rules that reference predicate symbols in S. This approach also works well for programs with stratified negation because at the time we negate an atom, the contents of the respective relation has already stabilized given that the relation belongs to a lower strata. Note that in case of non-recursive Datalog programs, all strata will simply consist of a single predicate symbol.

The way how we apply $T_{P[S]}$ gives room for optimization. A naive approach is to repeatedly apply $T_{P[S]}$ on the entire contents of a database instance. In contrast, in semi-naïve evaluation [14], when applying $T_{P[S]}$ at the ith step of the fixpoint computation for strata S, we only consider the rules that are actually affected by tuples that were newly derived in step $i-1$. This helps to reduce the amount of work, as we avoid applying rules that could not produce new tuples anyway. In the following, we focus on incremental fixpoint computation, as our goal is to reuse previously computed analysis results after program changes.

2.4 Incrementalizing Standard Datalog

Incrementally evaluating a Datalog program entails two steps. First, we run a non-incremental fixpoint computation based on an EDB instance to compute the IDB instance. Then, based on changes in the EDB instance in terms of fact deletions or insertions, we

```
Δ(CFlow(src, trg)) :- Δ(CSimple(src, trg)).
Δ(CFlow(src, trg)) :- Δ(CWhile(src, trg)).

Δ(CSimple(src, trg)):-Δ(PrecedingStatement(src,trg)),SimpleStatement(src).
Δ(CSimple(src,trg)):-PrecedingStatementᵛ(src,trg),Δ(SimpleStatement(src)).

Δ(CWhile(src, trg)):-Δ(PrecedingStatement(src, trg)), WhileStatement(src).
Δ(CWhile(src, trg)):-PrecedingStatementᵛ(src, trg),Δ(WhileStatement(src)).
Δ(CWhile(src, trg)):-Δ(WhileStatement(src)), FirstStatement(src, trg).
Δ(CWhile(src, trg)):-WhileStatementᵛ(src), Δ(FirstStatement(src, trg)).
Δ(CWhile(src, trg)):-Δ(WhileStatement(trg)), LastStatement(trg, src).
Δ(CWhile(src, trg)):-WhileStatementᵛ(trg), Δ(LastStatement(trg, src)).
```

Figure 2.6: Delta rules defined by the Counting algorithm for the control flow analysis.

incrementally update the contents of the IDB instance. We assume that the initial results are already computed with the techniques presented in Section 2.3. We review techniques for incrementally evaluating both non-recursive and recursive Datalog programs.

Counting for non-recursive Datalog A well-know algorithm for incrementally maintaining the results of non-recursive Datalog programs is called Counting [47]. At the core of the Counting algorithm, every tuple is associated with an incrementally maintained *support count*, which equals to the number of alternative derivations a tuple has. For example, in the results of the control flow analysis (Figure 2.3), every derived tuple has count 1. In the results of the uninitialized variable analysis, tuples `UninitializedBefore(3, last)` and `UninitializedBefore(3, err)` had count 2, as they were redundantly inferred again in Figure 2.5 at I_8. Incremental maintenance boils down to maintaining the support counts: A tuple insertion (deletion) increments (decrements) the count by 1. If the support count drops to 0, the tuple can be removed from the respective relation.

Counting uses *delta rules* to identify what changes in IDB relations in response to changes in EDB relations. With every rule r of the form $h :- a_1, ..., a_n$, Counting associates n delta rules where the ith ($1 \leq i \leq n$) rule is of the form

$$\Delta_i(r) : \Delta(h) :- a_1^v, ..., a_{i-1}^v, \Delta(a_i), a_{i+1}, ..., a_n.$$

$\Delta(h)$ represents the changes in relation h, and $h^v = h \uplus \Delta(h)$, that is, the signed union wrt. support counts of the previous relation contents and the delta in it. Figure 2.6 shows the delta rules of the control flow analysis from Figure 2.2. Using these delta rules for incremental maintenance is straightforward. We iterate over the predicate symbols in a topological order of the (singleton) strata. For each predicate symbol, we store the original relation contents, compute the delta by either using the changes in the respective EDB relation or the delta rules of the predicate, and then prepare the updated relations. There is no need to repeatedly reapply Datalog rules until the results stabilize, as there is no recursive dependency among the rules, so the computation reaches a fixpoint with a single application of the rules.

DRed for recursive Datalog Delete and Re-derive (DRed) [47] is an incremental algorithm to maintain the results of recursive Datalog programs. Given a change in EDB relations, the algorithm uses the following three phases to update the IDB relations. Each phase runs until fixpoint:

- *Delete*: This phase computes an over-estimate of the tuples that need to be deleted based on EDB tuple deletions. There is emphasis on the word "over-estimate", as DRed ignores alternative derivations during the delete phase. This means that irrespective of whether an alternative derivation remains after the deletion of a tuple, the tuple is still marked for deletion. For this reason, the set of marked tuples may be more than what actually needs to be deleted. This will be corrected later.

  With every rule r of the form $h :- a_1, ..., a_n$, DRed associates n delta rules where the ith ($1 \leq i \leq n$) rule is of the form:

  $$\Delta^{del}(h) :- a_1, ..., a_{i-1}, \Delta^{del}(a_i), a_{i+1}, ..., a_n.$$

  Assume a_i references predicate symbol p. Then, a_i represents the original contents of relation p, *without* incorporating any deletions. If p is an EDB relation, then $\Delta^{del}(a_i)$ consists of the set of facts deleted from p. If p is an IDB relation, then $\Delta^{del}(a_i)$ represents the set of tuples computed by the respective delta rules thus far in the fixpoint computation. Once all $\Delta^{del}(a_i)$ stabilizes, we obtain a_i^v by removing $\Delta^{del}(a_i)$ from the original contents of p. The next phase fixes the potential over-deletion of the delete phase.

- *Re-derive*: This phase puts back all those tuples that have alternative derivations based on the tuples that remained after the delete phase. For each rule r of the form above, DRed uses one re-derive rule:

  $$\Delta^{red}(h) :- \Delta^{del}(h), a_1^v, ..., a_n^v.$$

  During the fixpoint computation, we keep augmenting a_i^v obtained at the end of the delete phase by adding all those tuples back that get derived in $\Delta^{red}(a_i)$.

- *Insert*: The phase computes the tuples that need to be inserted based on EDB tuple insertions. DRed uses n delta rules for a rule r of the form above, and the ith rule looks as follows:

  $$\Delta^{ins}(h) :- a_1^v, ..., a_{i-1}^v, \Delta^{ins}(a_i), a_{i+1}^v, ..., a_n^v.$$

  Again, assume that a_i references predicate symbol p. If p is an EDB relation, then $\Delta^{ins}(a_i)$ consists of the set of facts inserted into p and a_i^v refers to the updated EDB relation with the insertions incorporated. If p is an IDB relation, then $\Delta^{ins}(a_i)$ represents the set of tuples computed by the respective delta rules thus far in the fixpoint computation, and a_i^v refers to the union of the tuples obtained for a_i at the end of the re-derive phase and $\Delta^{ins}(a_i)$. Technically, the re-derive and insert phases are often combined into a single phase, as both of these phases only ever infer new tuples and never delete any. In this case, the combined re-derive phase also needs to consider insertions in EDB relations.

```
int temp = readSensor(...);          (1)
int last, err = 0;                   (2)
while (outOfRange(temp)) {           (3)
  last = err;                        (3a)
  temp = readSensor(...);            (3b)
}
log("Last temperature %d", last);    (4)
```

Figure 2.7: The subject program we get after adding an initializer for `err` in Figure 2.1.

Let us now use DRed to update the results of the uninitialized variables analysis in response to a program change. Assume we update the subject program in Figure 2.1 as shown in Figure 2.7 by adding an initializer for `err`. In response to this change, the CFG of the subject program does not change, but `err` becomes initialized at all control flow locations after statement 2. In turn, `last` also becomes initialized at statement 3a and 3b, but it is still uninitialized at statement 3 and 4, as we do not know statically if the loop will be executed at all.[1]

We defer from explicitly spelling out all the delta rules used by DRed for our analysis, instead, we show the results of the delete and re-derive phases in Figure 2.8. There is no insert phase, as there were no fact insertions. The delete phase invalidates all but one tuple. On the one hand, this is good because we can invalidate all tuples reporting about the uninitialized state of `err`. On the other hand, we invalidate several tuples for `last` that need to be put back later in the re-derive phase. This is indicative of a behavior that happens frequently with DRed: We pay the price for correct incremental updates in the face of recursive dependencies with a potential over-deletion of tuples. At the end of the re-derive phase, we obtain the correct updated analysis results: `err` does not appear as uninitialized anywhere, while the initialized state of `last` does not span outside of the loop body.

Optimizing DRed with Counting An optimization for DRed is to combine it with Counting. Specifically, we can associate support counts with tuples just like with Counting and incrementally maintain them as tuples get inserted or deleted during the DRed phases [47]. The benefit of the support counts manifests in the re-derive phase, as all those tuples that still have a positive support count after the delete phase reached a fixpoint can be immediately put back into their respective relations. Without the support counts, we would need to go back to the EDB relations and start applying Datalog rules from there. However, care should be taken with the incremental maintenance of support counts during the delete phase because we can easily end up with incorrect analysis results due to *cyclic reinforcements* between tuples. Let us demonstrate the problem through the uninitialized variables analysis from Figure 2.4.

When we ran the uninitialized variables analysis on the subject program in Figure 2.1, we ended up inferring `UninitializedBefore(3, last)` and `Uninitialized-Before(3, err)` twice, that is, the support count of these tuples were two (as shown by I_8 in Figure 2.5). Now assume the program change in Figure 2.7. In response to this change, DRed first deletes `UninitializedAfter(2, err)`, which, in turn, invalidates

[1]We could employ a more sophisticated analysis that reasons about loop conditions and across function calls inter-procedurally, but this kind of analysis is out of scope here.

End of delete phase

Uninitialized Before	
stmt	var
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err
3	last
3	err

Uninitialized After	
stmt	var
2	last
2	err
3	last
3	err
3a	last
3a	err
4	last
4	err
3b	last
3b	err

Δ^{del}(Uninitialized Before)	
stmt	var
3	last
3	err
3a	last
3a	err
3b	last
3b	err
4	last
4	err

Δ^{del}(Uninitialized After)	
stmt	var
2	err
3	err
3	last
3a	err
3a	last
3b	err
3b	last
4	err
4	last

Uninitialized Before$^{\vee}$	
stmt	var

Uninitialized After$^{\vee}$	
stmt	var
2	last

End of re-derive phase

Δ^{red}(Uninitialized Before)	
stmt	var
3	last
3a	last
4	last

Δ^{red}(Uninitialized After)	
stmt	var
3	last
4	last

Uninitialized Before$^{\vee}$	
stmt	var
3	last
3a	last
4	last

Uninitialized After$^{\vee}$	
stmt	var
2	last
3	last
4	last

Figure 2.8: Results of DRed delete and re-derive phases applied to the uninitialized variables analysis in response to the program change in Figure 2.7.

one derivation of `UninitializedBefore(3, err)`. This means that we must decrement the support count of the tuple by one, still leaving one alternative derivation. Even though, the value of the support count tells us that we could stop with the propagation of the deletion, we must not do that because it would lead to incorrect incremental updates. We must realize that prior to the program change the very reason for deriving `Uninitialized-Before(3, err)` through the CFG edge `3b` $\longrightarrow$ `3` was that we already derived the same tuple through the CFG edge `2` $\longrightarrow$ `3`. Due to the loop in the CFG, there is cyclic reinforcement between the two derivations. After the program change, we must invalidate both of these derivations.

To correctly update the analysis result in face of cyclic reinforcements, we cannot rely on the support counts during the delete phase. A positive support count should not be used as evidence for a tuple being present in a relation. Instead, support counts must be temporarily ignored for deletions to spread along recursive dependencies, thereby invalidating all cyclically dependent derivations. Once the delete phase reached a fixpoint, we can start re-deriving tuples based on the remaining positive support counts. We refer the reader to the literature [47] for the correctness proofs of Counting and DRed.

Note on technical realization Both Counting and DRed are high-level algorithms in the sense that they prescribe how to perform incremental updates to IDB relations through delta rules and phases of updates, but real-world Datalog solvers still need to take care of other low-level implementation details. To see this, it is enough to go back to the description of a Datalog rule. "A rule is interpreted as a universally quantified implication: The substitutions of the variables in the body imply when the head holds." This entails relational algebra operations; e.g. joining, filtering, projecting of relations. In Chapter 3, we show how IncA solves these technical details.

2.5 Chapter Summary

We reviewed the syntax, semantics, and incrementalization of standard Datalog through concrete static analyses. We started off by implementing a simple control flow analysis, and we used the model-theoretic semantics to give meaning to our Datalog program. Then, we implemented a flow-sensitive uninitialized variables analysis that used the results of the control flow analysis. The analysis implementation was recursive, and it used stratified negation. We used the fixpoint-theoretic semantics to compute the result of our analysis on a concrete subject program. Finally, we turned our attention to the incrementalization of standard Datalog. We discussed both the Counting and DRed algorithms. The DRed algorithm is particularly interesting for the next chapter because we will use it as the incrementalization algorithm in the back end in the baseline version of IncA.

3 The IncA Incremental Analysis Framework

This chapter shares material with the ASE'16 paper "IncA: A DSL for the Definition of Incremental Program Analyses" [125].

Abstract — Program analyses support software developers, for example, through error detection, code-quality assurance, and by enabling compiler optimizations and refactorings. To provide real-time feedback to developers within IDEs, an analysis must run efficiently even if the analyzed code base is large.

To achieve this goal, we present the IncA framework for the definition of efficient incremental program analyses that update their result as the subject program changes. IncA comes with its own Datalog dialect, compiles analyses into graph patterns, and relies on existing incremental graph pattern matching algorithms. To scale IncA analyses to large programs, we describe optimizations that reduce caching and prune change propagation. Using IncA, we have developed incremental control flow and points-to analysis for C, well-formedness checks for DSLs, and 10 FindBugs checks for Java. Our evaluation demonstrates significant speedups for all analyses compared to their non-incremental counterparts.

3.1 Introduction

We demonstrated in Chapter 2 that Datalog is a good fit for implementing static analyses: Due to its declarative nature, the details of efficient evaluation can be left to a Datalog solver. For example, Whaley et al. develop highly precise points-to analysis in Datalog in the *bddbddb* framework [140]. The *bddbddb* solver uses binary decision diagrams to represent relations in the back end, which allows *bddbddb* to scale to large databases with better performance than hand-tuned versions of the functionally equivalent points-to analysis. As Whaley et al. write "*bddbddb* takes advantage of optimization opportunities that are too difficult or tedious to do by hand". Backes et al. perform network analysis on Amazon-scale using Datalog in seconds [13], while Smaragdakis et al. develop a range of increasingly precise points-to analysis for Java in the Doop framework using Datalog [114]. Both of these use cases are backed by the Soufflé Datalog solver [66]. Again, there is a range of optimizations that come out of the box from the Soufflé solver; ranging from partial evaluation for efficient index selection to multi-threaded execution.

Our work improves the performance of program analyses through incrementality: When part of the subject program changes (for example, through user edits), we only reanalyze the changed part, plus all the code whose analysis result depends on the changed results. This way, incremental program analysis can provide significant improvements compared to reanalyzing the whole code base from scratch. Our goal is to meet the timeliness requirement of analysis clients in IDEs, that is, to deliver sub-second update times for analyses on evolving subject programs.

We present IncA, a Datalog-based framework for the definition and efficient incremental evaluation of program analyses. Compared to other Datalog-based frameworks (such as Flix [79] or Soufflé [66]), IncA does not use a fact extractor to generate EDB relations from structured input data (e.g. an AST) prior to an analysis run. Instead, analyses developers in IncA can directly write Datalog code against the AST of a subject program. To this end, IncA uses its own Datalog dialect through several extensions over standard Datalog. Importantly, IncA Datalog uses virtual EDB relations to access types and structural links of the subject language. This has two benefits for incremental evaluation:

- The EDB relations are close to the original structure of the subject programs which makes incremental maintenance of EDB relations simpler in face of program changes, which is an important first step for the efficient incrementalization of analyses.
- Analysis developers can precisely capture what information is needed from a subject program. This is useful information for the incremental evaluation because changes to program elements not needed by an analysis can be safely ignored, as they cannot have an impact on an analysis result anyway. IncA exploits this by performing a *meta analysis* on IncA analyses to provide hints to the runtime system about which EDB changes can be ignored, thereby improving incremental performance.

The IncA compiler translates an IncA analysis into a set of *graph patterns* [101]. Graph patterns prescribe expected relationships between AST nodes, so, from this perspective, they are no different from an analysis using virtual EDB relations to access AST nodes. The reason why we use graph patterns is because there are existing incremental graph pattern matching libraries readily available. We use the Viatra Query library [129] in the back end of IncA.[1] Viatra Query takes care of the efficient incrementalization of the low-level relational algebra behind Datalog programs. We extend Viatra Query with DRed-style evaluation to support recursive analyses. The design and implementation of IncA is independent of a particular subject language or analysis, and we developed a generic architecture for the integration of IncA into IDEs. We instantiate IncA on top of the projectional JetBrains MPS IDE.[2]

To evaluate IncA, we implemented incremental analyses for C and Java programs and measured their runtime performance. For C, we developed incremental control flow analysis, flow-sensitive points-to analysis, and domain-specific well-formedness checks. For Java, we reimplement 10 FindBugs analyses [62]. Our evaluation shows that IncA-based incremental program analyses yield significant speedups without introducing unacceptable memory or initialization overheads.

Contributions In summary, we make the following contributions:

- We introduce IncA Datalog, which is tailored to writing static analyses directly against the AST of subject programs. We translate IncA Datalog code to graph patterns (Section 3.3).
- We implement compiler optimizations for IncA that reduce the memory required for incrementalization and the time required for change propagation (Section 3.4).
- We describe the runtime system of IncA as a generic architecture that allows the integration of IncA analyses into different IDEs. We instantiate this architecture on top of a projectional IDE (Section 3.5).
- We develop three incremental analyses for C, including an incremental flow-sensitive points-to analysis, and reimplement 10 FindBugs analyses for Java (Section 3.6).
- We evaluate the memory and runtime performance of IncA through our case studies on real-world C and Java projects (Section 3.7). We show that program analyses developed with IncA provide real-time feedback and scale to large code bases.

3.2 Incremental Program Analysis with IncA

This section explains the IncA approach to incremental program analysis. We demonstrate IncA Datalog and how we use it to encode analysis directly against the AST of a subject program. Then, we translate IncA Datalog code to graph patterns and use incremental graph pattern matching to evaluate analyses in response to program changes.

IncA Datalog We already provided an implementation of a control flow analysis in Section 2.2. We use that analysis as a running example here, too, and we recap what kind of EDB relations we used earlier. Our earlier implementation of the analysis (Figure 2.2) relied on EDB relations, such as `PrecedingStatement`, `FirstStatement`, and `LastStatement` that were prepared by a fact extractor prior to an analysis run. The goal of a fact extractor is to traverse the AST of a subject program and generate facts that represent the relevant relationships between AST nodes. This is a typical approach for Datalog-based frameworks

[1]Viatra Query was formerly know as EMF-IncQuery.

[2]`https://www.jetbrains.com/mps`

```
CFlow(src : Statement, trg : Statement) :- {
  CSimple(src, trg)
} alt {
  CWhile(src, trg)
}

CSimple(src : SimpleStatement, trg : Statement) :- {
  Statement.prev(src, trg)
}

CWhile(src : Statement, trg : Statement) :- {
  WhileStatement(src),
  Statement.prev(src, trg)
} alt {
  FirstStatement(src, trg)
} alt {
  LastStatement(trg, src)
}

FirstStatement(loop : WhileStatement, first : Statement) :- {
  WhileStatement.stmts(loop, first),
  not Statement.prev(_, first)
}
LastStatement(loop : WhileStatement, last : Statement) :- {
  WhileStatement.stmts(loop, last),
  not Statement.prev(last, _)
}
```

Figure 3.1: Control flow analysis for a subset of C in IncA Datalog.

because fact extractors help resolve a mismatch related to input representation: Datalog can only work with flat tuples in relations, while the data structures representing subject programs are typically hierarchical, just like the AST. Extracting EDB relations may simply require the traversal of single structural links in the AST, e.g. to extract parentship between nodes, but extracting `FirstStatement` and `LastStatement` require more complex reasoning. Statement `s` is the first/last statement of `l` if `l` is a loop, `s` is a statement in the body of `l`, and there is no other statement `q` whose predecessor/successor would be `s`. This is essentially pattern matching over the AST of the subject program. Supporting such patterns is fundamental to the design of IncA Datalog.

IncA analyses do not rely on extracted EDB relations, instead they work directly against the AST of subject programs (or any other hierarchical data structure used to represent the subject program). To this end, IncA uses Datalog for analysis specification, but it extends it with virtual relations that allow to access types and structural links from the subject language. We say “virtual” because these relations are not modeled explicitly like EDB relations produced by a fact extractor or IDB relations defined by the rules of an analysis. Instead, they are defined implicitly by the abstract syntax of the subject language. We demonstrate this by rewriting the control flow analysis in IncA Datalog as shown in Figure 3.1. We highlight key implementation details of this analysis:

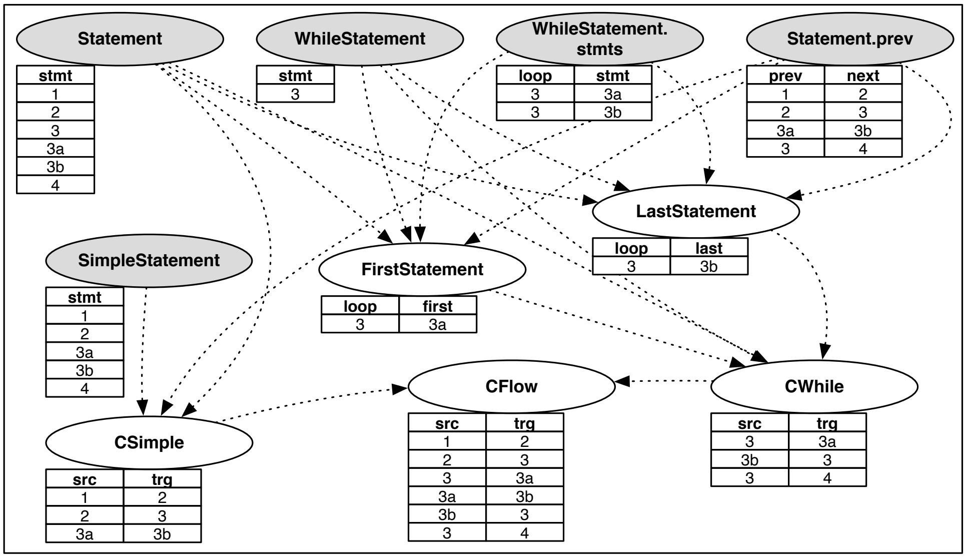
- Unary EDB relations, such as `Statement`, `SimpleStatement`, and `WhileStatement` represent types from the subject language.

- Binary EDB relations `WhileStatement.stmts` and `Statement.prev` represent structural links in the AST. Link `stmts` is defined on the type `WhileStatement` in the subject language, and it is the containment link holding the statements in the body of a while loop. Link `prev` is a virtual link that allows to access the previous sibling of a node (if any). IncA supports `prev` on all types, irrespective whether the link is defined explicitly in the abstract syntax of the subject language (which usually is not the case). It does not come up in this example, but `next`, `parent`, and `index` are also virtual links that IncA supports on all types to access the next sibling, parent of a node, and index of a node among its siblings, respectively. These virtual links are fundamental for navigating in the AST of a subject program.
- `FirstStatement` and `LastStatement` are now IDB relations, which is in contrast to using them as extracted EDB relations in Figure 2.2.
- IncA Datalog allows to use negation on EDB relations, such as `not Statement.prev(last, _)`. Intuitively, this atom constrains the `Statement` nodes that will substitute `last` by requiring that `last` does not precede another `Statement` node, i.e. it is the last among the sibling statements.
- In IncA, we do not repeat the same rule head for alternative bodies, instead, we define the head once, and connect the alternative bodies with the `alt` keyword. This will play a role when we define visibility for a rule, as we will discuss in Section 3.3.
- IncA allows to define optional type parameters for variables in the head atom. This is for convenience only. Defining type annotations has the exact same effect as using unary EDB relations with the respective types in the rule body.

Let us now look at how IncA evaluates this analysis on a concrete subject program. We start with the initial, non-incremental evaluation.

Evaluation with graph patterns To compute the tuples of the relations `CSimple`, `CWhile`, and `CFlow`, we must traverse the AST and discover the relevant structural relations between the AST nodes. A natural representation of such computations is a *graph pattern* [101]. Much like the IncA Datalog code in Figure 3.1, graph patterns describe sets of related entities through structural constraints on AST nodes and on instances of other graph patterns.

Given a set of interconnected graph patterns, we can compute their results by using a *computation graph*. A computation graph performs relational algebra operations, e.g. by filtering, joining, unioning, projecting related entities. First, we provide a high-level view on the computation graph for our control flow analysis in Figure 3.2. The figure abstracts away from the low-level relational algebra operations, and purely focuses on data dependencies defined by the Datalog rules. A computation graph consists of two kinds of nodes, each of which yields a set of related entities. *Input nodes* (gray) represent the AST structure directly; They do not perform any computation, and they do not depend on any other node. In Figure 3.2, input nodes `Statement`, `SimpleStatement`, and `WhileStatement` enumerate the different kinds of statement nodes, while `WhileStatement.stmts` and `Statement.prev` enumerate the statements in a loop body and the precedence relationship between nodes. *Computation nodes* (white) use the results of input nodes and other computation nodes to relate program entities. For example, node `CSimple` uses information from node `Statement`,



stmt
1
2
3
3a
3b
4

stmt
3

loop	stmt
3	3a
3	3b

prev	next
1	2
2	3
3a	3b
3	4

loop	last
3	3b

stmt
1
2
3a
3b
4

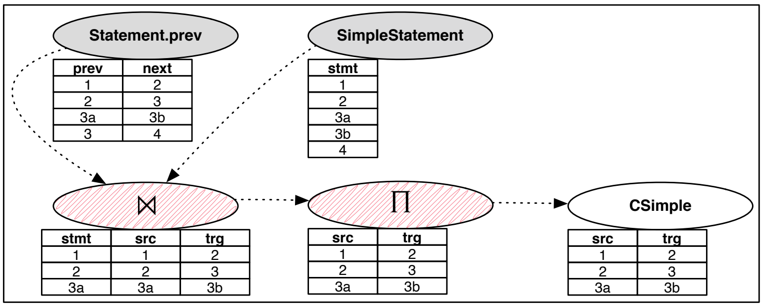
loop	first
3	3a

src	trg
1	2
2	3
3	3a
3a	3b
3b	3
3	4

src	trg
3	3a
3b	3
3	4

src	trg
1	2
2	3
3a	3b

Figure 3.2: Computation graph of the control flow analysis. Arrows show data dependencies.



prev	next
1	2
2	3
3a	3b
3	4

stmt
1
2
3a
3b
4

stmt	src	trg
1	1	2
2	2	3
3a	3a	3b

src	trg
1	2
2	3
3a	3b

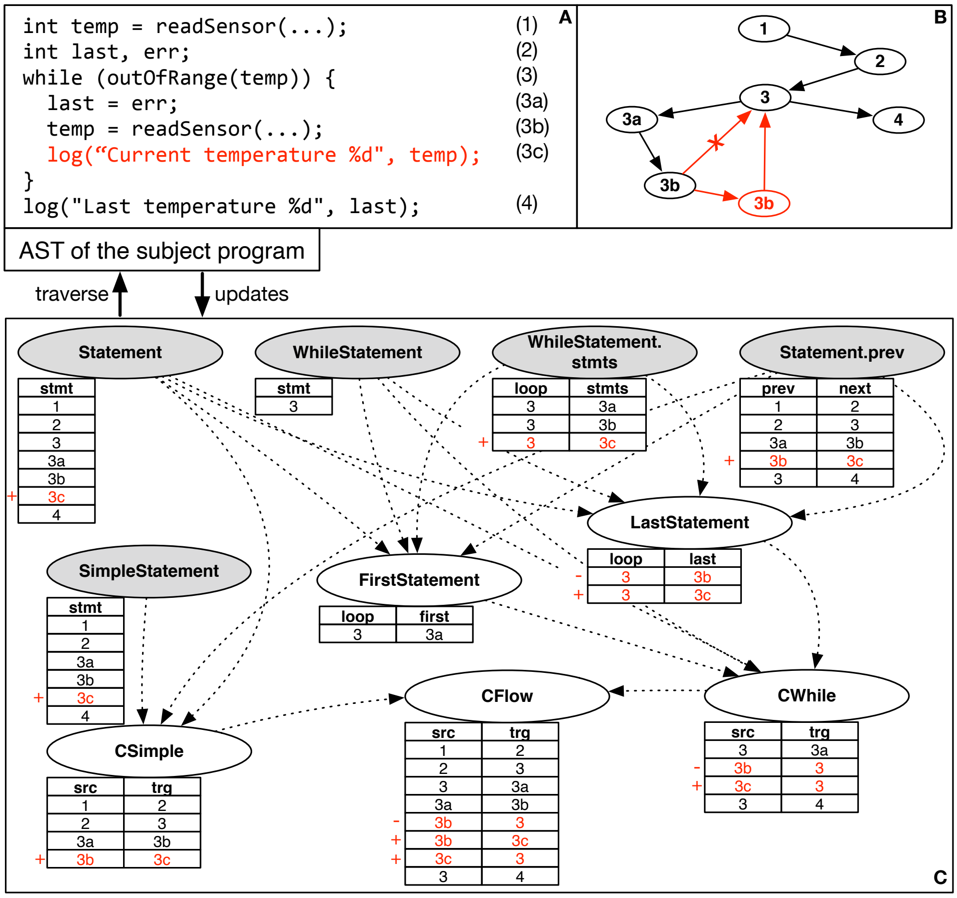
src	trg
1	2
2	3
3a	3b

Figure 3.3: Excerpt of the low-level computation graph for the control flow analysis. The figure shows the relational algebra operations used for computing the results of the `CSimple` rule from Figure 3.1.

`SimpleStatement`, and `Statement.prev` to identify statements related through simple control flow. Node `CFlow` combines the results of the two computation nodes `CSimple` and `CWhile` to produce the complete CFG.

Figure 3.3 shows the low-level relational algebra operations for the `CSimple` rule from Figure 3.1. First, we join the results of `Statement.prev` and `SimpleStatement` using a join node (denoted by $\bowtie$). Then, we project away the first column using a project node (denoted by Π), which is also the result of `CSimple`. Transforming a complete analysis to such low-level computation network is a complex task. In IncA, we use the graph pattern matching library VIATRA QUERY that turns graph patterns automatically into a computation network. We discuss how we translate IncA Datalog to graph patterns in Section 3.3.

Incremental evaluation with graph patterns We encode program analyses as graph patterns and computation graphs because this provides a good basis for incrementalizing the computation. Suppose the user modifies the subject program and adds a new statement



```
int temp = readSensor(...);                        (1)
int last, err;                                     (2)
while (outOfRange(temp)) {                         (3)
  last = err;                                      (3a)
  temp = readSensor(...);                          (3b)
  log("Current temperature %d", temp);             (3c)
}
log("Last temperature %d", last);                  (4)
```

Statement

	stmt
	1
	2
	3
	3a
	3b
+	3c
	4

WhileStatement

stmt
3

WhileStatement. stmts

	loop	stmts
	3	3a
	3	3b
+	3	3c

Statement.prev

	prev	next
	1	2
	2	3
	3a	3b
+	3b	3c
	3	4

LastStatement

	loop	last
-	3	3b
+	3	3c

SimpleStatement

	stmt
	1
	2
	3a
	3b
+	3c
	4

FirstStatement

loop	first
3	3a

CSimple

	src	trg
	1	2
	2	3
	3a	3b
+	3b	3c

CFlow

	src	trg
	1	2
	2	3
	3	3a
	3a	3b
-	3b	3
+	3b	3c
+	3c	3
	3	4

CWhile

	src	trg
	3	3a
-	3b	3
+	3c	3
	3	4

Figure 3.4: Incremental evaluation of the control flow analysis with the computation graph; (A) subject program after code change, (B) updates in the CFG, (C) computation graph incrementally updating the analysis result.

3c to the loop body of 3 as illustrated in Figure 3.4 A. In turn, the CFG of the subject program changes as shown in Figure 3.4 B. Using the computation graph, instead of recomputing a new CFG from scratch, we can *incrementally update* the existing CFG.

To this end, computation graphs use memoization and perform incremental graph pattern matching: When code gets changed, we send change events to the input nodes of the computation graph. The nodes then transitively propagate changes to all dependent computation nodes and trigger the reanalysis of changed program entities. This way, we avoid any reanalysis of unchanged parts of the program. In our example, adding the new statement to the loop body triggers the following changes in the computation graph (as also shown in Figure 3.4 C):

- Insert `3c` to `Statement` and `SimpleStatement`, `(3, 3c)` to `WhileStatement.stmts`, and `(3b, 3c)` to `Statement.prev`.
- Delete `(3, 3b)` and insert `(3, 3c)` to `LastStatement` because the last statement

in the loop body is now statement 3c, not 3b.

- Insert (3b, 3c) to CSimple because there is now a sequential control flow between statement 3b and 3c.
- Delete (3b, 3) and insert (3c, 3) to CWhile due to the change in LastStatement. Control now flows from 3c to the loop condition, not from 3b.
- Propagate the insertions and deletions from CSimple and CWhile to CFlow.

The change propagation goes on in the computation network until the result of a computation node changes. Computation nodes update their results in a topological ordering based on their data dependencies. For the low-level relational algebra, we use the VIATRA QUERY library in IncA. To support recursive analyses, we extended VIATRA QUERY with a DRed-style evaluation (see Section 2.4). Technically, this means that computation nodes are grouped into strongly connected components and change propagation follows a two step approach in a topological ordering of the components; first propagating deletions until fixpoint, and then running the re-derive and insert phases until fixpoint.

In our baseline implementation of IncA, we rely on projectional editing in the front end. In a projectional editor, we receive change events for user-issued AST changes directly from the IDE. Fine-grained AST change notifications align perfectly with incrementalization, and there is no need for a potentially costly parsing step to compute AST differences or to extract facts prior to the analysis. After an AST change, a projectional editor derives a new projection from the AST and displays it to the programmer. Later, in Chapter 7, we will also develop an efficient parser-based front end for IncA. Here, we continue our discussion with the complete syntax of IncA Datalog, and we show how we translate an IncA analysis to graph patterns.

3.3 Syntax and Compilation of IncA Datalog

In this section, we discuss IncA Datalog in greater detail. We describe the syntax of the language and explain how we translate its syntactic constructs to graph patterns.

3.3.1 Syntax of IncA Datalog

Figure 3.5 shows the syntax of IncA Datalog. A *module* groups related rules. Modules can import other modules to gain access to their rules. We write $\overline{a}$ for a sequence of a elements, which includes the empty list.

An IncA Datalog *rule* resembles a standard Datalog rule, but there are some key differences. An IncA rule has an optional visibility specification. By default, the visibility is public. *Public* and *protected* rules are visible to importing modules, but only the public rules are visible to the clients of an analysis. *Private* rules are only visible in the module that defines the rule. IncA allows to define a rule head only once, and rules can have one or more *alternative* rule bodies. Allowing to define a head only once is important because otherwise we could end up with conflicting visibilities for different rules defining the same head. Only variables can be used in the head of an IncA rule, and these variables can have an optional type annotation.

(module)	m	$::= \textbf{module}\ n\ \textbf{import}\ \overline{n}\ \{\overline{r}\}$
(rule)	r	$::= vis\ n(\overline{n : T})\ \text{:-}\ \overline{alt}$
(visibility)	vis	$::= \textbf{private} \mid \textbf{protected} \mid \textbf{public}$
(alternative)	alt	$::= \{\overline{a}\}$
(atom)	a	$::= n(\overline{n}) \mid \textbf{not}\ n(\overline{n}) \mid vr \mid \textbf{not}\ vr \mid n+(n,n) \mid t == t \mid t \mathrel{!=} t$ $\textbf{count}\ n(\overline{n}) \mid \textbf{count}\ vr$
(virtual EDB relation)	vr	$::= T.L(n,n) \mid T(n)$
(term)	t	$::= n \mid c$
(constant)	c	$::=$ number $\mid$ string $\mid$ enum $\mid$ boolean $\mid$ AST node
(type)	T	$::=$ AST node type (from subject language)
(link)	L	$::=$ link of an AST node type (from subject language) $\mid$ $\textbf{parent} \mid \textbf{prev} \mid \textbf{next} \mid \textbf{index}$
(name)	n	$::=$ name

Figure 3.5: The syntax of IncA Datalog.

A rule body consists of *atoms*. An atom is either a relation name with a list of variables possibly in a negated form, a virtual EDB relation name with a single or a pair of variables (depending on whether the EDB relation is unary or binary) possibly in a negated form, binary transitive closure of a relation, equality or inequality between terms, or a count construct. A count is a special form of aggregate that allows to count the number of tuples in an EDB or IDB relation.

The *virtual EDB relation*s are defined by the subject language. The *type*s defined in the subject language become unary relations, while the *link*s defined on types become binary relations. In addition to the explicitly defined links, IncA also supports `parent`, `prev`, `next`, and `index` as built-in links, available on all types.

3.3.2 Compilation to Graph Patterns

The theory of graph patterns is well established [101], and we define the semantics of IncA Datalog through translation to graph patterns. A graph pattern consists of pattern variables and constraints over these variables. The constraints can refer to other graph patterns, as well. The process of graph pattern matching is about finding sub-graphs in an input graph that satisfy the structural constraints prescribed by the graph pattern. We use the following set of constraints, as these are supported by the VIATRA QUERY library that we use for graph pattern matching:

- `Entity(v,T)` holds if variable `v` has type `T`.
- `Relation(l,v1,v2)` holds if there is an edge labeled `l` between variables `v1` and `v2`.
- `Eq(v1,v2)` holds if `v1` and `v2` are the same values. Here, `v1` and `v2` can either be a node or a constant value.
- `Neq(v1,v2)` holds if `v1` and `v2` are different values. Similar to `Eq`, `v1` and `v2` can either be a node or a constant value.
- `PC(p,`$\overline{\texttt{v}}$`)/NPC(p,`$\overline{\texttt{v}}$`)` holds if the pattern `p` accepts/rejects the tuple of values $\overline{\texttt{v}}$.

- `TC(p,v1,v2)` holds if the transitive closure of the binary pattern `p` contains the tuple `(v1,v2)`.
- `Alt(p̅,v̅)` holds if any of the patterns in `p̅` accepts the tuple `v̅`.
- `Count(p,v)` is a special kind of graph pattern that counts the number of matches of pattern `p` and stores the count in variable `v`.

We follow a bottom-up translation process as follows. For each IncA Datalog variable, we introduce a fresh graph pattern variable. Atoms referencing types from the subject language become `Entity` constraints, while atoms accessing links of types become `Relation` constraints. The built-in links `parent`, `prev`, `next`, and `index` require special runtime support, as these links are not modeled explicitly in the subject language. We discuss this later in Section 3.5. In case of negation applied on an EDB relation, we generate a helper graph pattern containing the (non-negated) access to the EDB relation and reference the helper pattern in an `NPC`. Atoms referencing IDB relations become `PC` constraints, if negation is applied, we directly use an `NPC` constraint. For transitive closure, we use the `TC` constraint, and equality/inequality is mapped to `Eq`/`Neq`. For a count aggregate, we use the pattern generated for the counted construct and embed that in a `Count` pattern.

We collect the rules from a module and its transitively imported modules and generate graph patterns for the rules and their alternatives. Type annotations on variables in head atoms become `Entity` constraints. Finally, given the variables `v̅` generated for a rule `r` and the patterns `p̅` generated for the alternatives of *r*, we generate a constraint `Alt(p̅,v̅)`.

The expressive power of IncA Datalog and of the graph patterns presented thus far is classified as FO(LFP) [63, 96], which stands for First Order logic extended with the Least Fixed Point operator. In Section 3.6, we demonstrate that this expressive power is already sufficient for several practically-relevant program analyses.

3.4 Compiler Optimizations for IncA

The performance of IncA depends on both the memory required for caching as well as the efficiency of change propagation in the computation graph. In this section, we describe compiler optimizations for IncA that improve both of them.

Our approach for performance improvement relies on the observation that the evaluation of a program analysis usually only depends on a relatively small part of an AST. For incremental analysis, this means that many AST changes do not affect the analysis result and can be safely ignored. We can use this observation to improve caching and change propagation. There is no need to write a changed element to the input nodes of the computation graph if it is known to be irrelevant for the analysis. This saves both memory and time, because, by discarding irrelevant changes in input nodes, we avoid subsequent change propagation in the computation graph, which, in turn, also avoids caching of unneeded results.

To make use of this observation in practice, we must be able to distinguish relevant from irrelevant changes. To this end, we have developed an analysis for IncA programs, that is, an *analysis of the program analysis* code. We use the term *meta analysis* to refer to this analysis in the remainder of this chapter. Our meta analysis inspects an IncA analysis to compute a conservative approximation of all changes that can affect the result of the

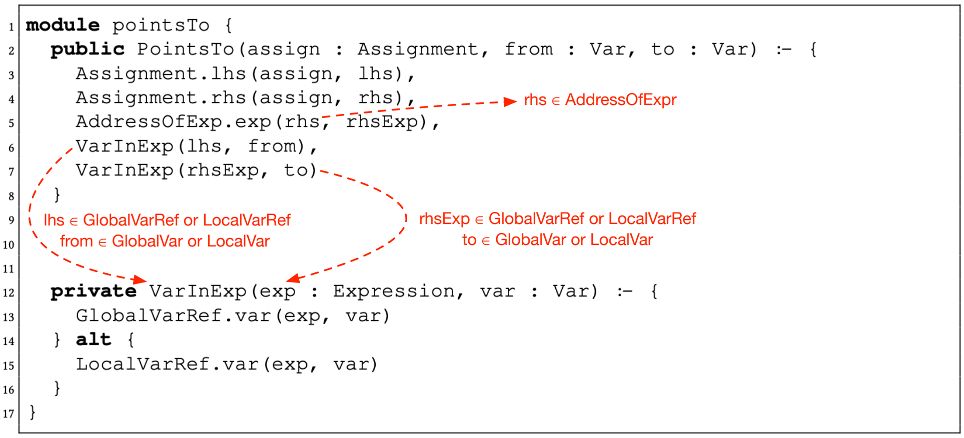
IncA analysis. We analyze each module of the IncA analysis with its transitive imports separately because the IncA compiler creates one computation graph for each module.

As a first approximation of the relevant changes, we can collect the *declared types* of all variables appearing in a head atom. In the absence of an explicit type annotation, we just treat a variable as it would have `Any` type, representing all types from the subject language. For example, consider again the control flow analysis in Figure 3.1. The IncA code refers to types `Statement`, `SimpleStatement`, and `WhileStatement`. Since type `Statement` is a supertype of the other two statement types, a sound approximation for the relevant changes is given by the set of changes that modify nodes of type `Statement`. Accordingly, when incrementally executing the control flow analysis in the IncA runtime system, we can ignore changes to expressions, function declarations, and others. We emphasise that the type annotations are controlled by the analysis developer and that the subtyping relationship between types is defined by the subject language. Without explicit type annotations, we cannot infer much from just looking at the head atoms, as this reasoning would yield `Any`, meaning that we must consider all changes at runtime.

Using the declared types of AST nodes is a good starting point for optimization; however, it is rather imprecise. To improve the effectiveness of the optimization, we also need to look into the rule bodies and take into account the constraints enforced by atoms referencing EDB relations. Consider an excerpt of a simple points-to analysis for C shown in Figure 3.6. Rule `PointsTo` computes pairs of C variables for assignments of the form `u = &v`. It uses rule `VarInExp` to reject expressions that are not variable references (such as additions or multiplications) and otherwise to extract the referenced variable. If we only consider the declared types and ignore the constraints enforced by rule bodies, we have to assume that a change to any node of type `Assignment`, `Var`, or `Expression` may affect the analysis result. However, we can apply the following reasoning to narrow the set of relevant changes in Figure 3.6:

- Initially, both variables `lhs` and `rhs` have type `Expression` because the target of the links `Assignment.lhs` and `Assignment.rhs` has type `Expression` based on the subject language.
- The atom in line 5 accessing `AddressOfExp.exp` constrains both `rhs` and `rhsExp`, as the former must be an instance of `AddressOfExp`, while the latter must be an instance of `Expression` (assuming that is type of the link target).
- The atom in line 6 references the `VarInExp` IDB relation and uses variables `lhs` and `from`. The rule `VarInExp` has two alternatives, the first one restricting `exp` to `GlobalVarRef`, while the second one restricts it to `LocalVarRef`. Expressions that satisfy neither restriction are rejected by `VarInExp`. In turn, `lhs` in rule `PointsTo` must be an instance of `GlobalVarRef` or `LocalVarRef`. `VarInExp` also constrains `var`, as it must be an instance of `LocalVar` or `GlobalVar`, as these are the types of the links `LocalVarRef.var` and `GlobalVarRef.var`. In turn, `from` in rule `PointsTo` must also be an instance of `LocalVar` or `GlobalVar`.
- The same reasoning applies to the atom in line 7, which constrains variables `rhsExp` and `to`.

This reasoning shows that we only need to observe changes to statement nodes of type `Assignment`, to expression nodes of type `AdressOfExp`, `GlobalVarRef`, and `LocalVarRef`,



```
module pointsTo {
  public PointsTo(assign : Assignment, from : Var, to : Var) :- {
    Assignment.lhs(assign, lhs),
    Assignment.rhs(assign, rhs),
    AddressOfExp.exp(rhs, rhsExp),
    VarInExp(lhs, from),
    VarInExp(rhsExp, to)
  }


  private VarInExp(exp : Expression, var : Var) :- {
    GlobalVarRef.var(exp, var)
  } alt {
    LocalVarRef.var(exp, var)
  }
}
```

rhs ∊ AddressOfExpr
lhs ∊ GlobalVarRef or LocalVarRef
from ∊ GlobalVar or LocalVar
rhsExp ∊ GlobalVarRef or LocalVarRef
to ∊ GlobalVar or LocalVar

Figure 3.6: Identifying types to filter irrelevant program changes in the IncA runtime system.

and to variable nodes of type `GlobalVar` and `LocalVar`. Let us put this result into perspective with a concrete C dialect. mbeddr [135] is a set of extensible languages based on C for embedded software development. The mbeddr languages contain more than 200 different kinds of expressions. This means that our optimization can yield significant improvements in memory and run time in case an IncA analysis only uses a small subset of these different kinds of expressions. We empirically confirm this in Section 3.7.

Technically, our meta analysis is an inter-procedural data-flow analysis. The input of the meta analysis is an IncA analysis, and its output is a set of types from the subject language. The high-level steps of our meta analysis are as follows:

1. *Traverse dependency chains:* We perform a depth-first traversal of the dependency graph defined by the IncA rules, starting at publicly visible ones. The meta analysis is inter-procedural, that is, we pass type information between rules. Different call chains may result in different type constraints for variables; thus we analyze all call chains separately.

2. *Compute intersection of type constraints per alternative:* During traversal, for each alternative of a rule, we collect the type constraints from atoms referencing EDB relations or other IDB relations for each variable. An alternative can only succeed if each variable satisfies all type constraints prescribed for the variable. This corresponds to constructing the intersection of all type constraints for each variable.

3. *Compute union of type constraints per rule:* A rule succeeds if at least one of its alternatives succeeds. Thus, it is sufficient if the variables in the head atom satisfy the type constraints of at least one alternative. We approximate this by constructing the union of the type constraints for the variables from all alternatives.

4. *Use type constraints for optimization:* The analysis result is the union of the type constraints for the different call chains. Only changes that affect nodes of these types can affect the analysis result. We prune the propagation of irrelevant changes, thus avoiding the computation and caching of irrelevant pattern matches.

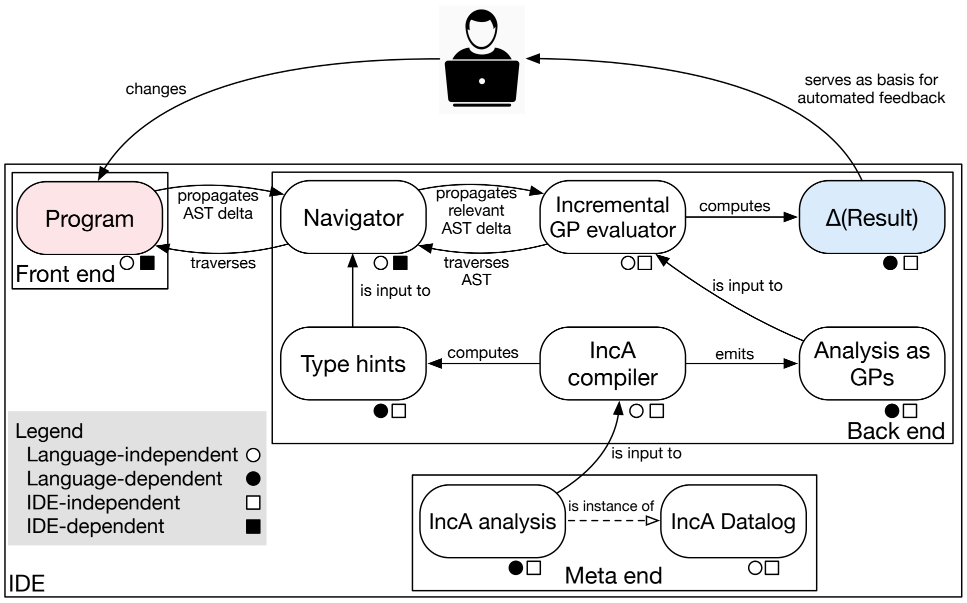


Figure 3.7: Architecture for integrating IncA into IDEs. Red color is used for analysis input, while blue is used for analysis output.

Note that using a declarative language like IncA Datalog for analysis specification is a key enabler in performing the outlined meta analysis. This is because the analysis code itself is analyzable, which would not be the case had we used a library-based approach for program analysis. We also emphasise that our meta analysis is not bound to IncA Datalog: It could also be applied on the graph patterns, as well.

3.5 Technical Realization and IDE Integration

We elaborate on the architecture of IncA's runtime system, identifying components that enable integration of incremental program analyses into IDEs and explain their interactions.

3.5.1 Architecture Overview

Figure 3.7 shows the architecture of IncA as integrated in an IDE. The figure is a refined version of the architecture shown in Figure 1.3. Throughout the discussion, we emphasise which component is independent of the subject language and/or the IDE itself.

As detailed in Section 3.5.3, our architecture requires that the front end in the IDE translates user edits into AST change notifications, which trigger the incremental analysis. In the front end, we intentionally wrote `Program` instead of Δ(`Program`) (which we used in Figure 1.3) to emphasise that it is also the runtime system's responsibility to compute the initial analysis result, which requires the whole subject program, not just a delta. The front end technology itself (not a concrete subject program) is language-independent, as it works with any language supported by the IDE. However, the front end needs to be aware of how the IDE represents a subject program in terms of data structures, so the front end is an IDE-dependent component.

In the meta end, we use IncA Datalog, which we introduced in Section 3.3. IncA Datalog is language-independent and IDE-independent. A concrete IncA analysis references types and links from the subject language, so it is language-dependent, but IDE-independent.

In the back end, the IncA compiler translates an IncA analysis into a set of interconnected graph patterns as discussed in Section 3.3. Additionally, the compiler uses the optimization described in Section 3.4 to compute type hints for the runtime system that help to filter our irrelevant program changes. The compiler itself is language- and IDE-independent, but the artifacts produced by the compiler are language-dependent.

The entry point to incremental analysis is the navigator, which acts as an adapter between the IDE and the incremental graph pattern evaluator. The navigator gets notified about AST changes by the IDE. It also allows the evaluator to traverse the input AST during initialization of the computation graph. Later, when the navigator receives an AST change notification, it notifies the incremental evaluator about *relevant* AST changes, as determined by our compiler optimization (Section 3.4). It is also the navigator's responsibility to incrementally maintain the `parent`, `prev`, `next`, and `index` links of AST nodes, as required by the IncA language (Section 3.3). Since the navigator knows the IDE-internal AST representation, it constitutes a language-independent but IDE-dependent component.

The incremental evaluator is responsible for the incremental maintenance of the analysis results. The component is independent of the IDE and of the subject language. The evaluator uses the navigator to navigate in the AST and to receive notifications about AST changes; this way, the evaluator does not depend on the internals of the IDE.

The incrementally maintained results of a program analysis can be used to inform the user about new errors or as part of a refactoring in the IDE. However, as the result delta is in a format used by the incremental evaluator, an IDE-dependent adapter component may be needed, which adapts the results to e.g. editor services or an error highlighter. This connection closes the loop between the user and IncA and shows how incremental analysis supports continuous feedback in IDEs.

3.5.2 Implementation for MPS

We instantiated the above architecture for the Meta Programming System (MPS). MPS is an IDE that uses *projectional editing* [136] instead of a parser-based approach. When editing the program in a projectional editor, every user edit (for example, inserting an operator) directly corresponds to an AST change. After an AST change, a projectional editor renders the new projection from the changed AST based on projection rules of the AST nodes and displays it to the programmer. Projectional editing is well-suited for incremental program analysis because the user's edits directly correspond to incremental AST changes and no incremental parsing is necessary.

Our system reuses the incremental graph pattern matching component of Viatra Query. This component realizes the computation graph presented in Section 3.2. We extended Viatra Query with support for DRed-style evaluation to correctly handle recursive analyses, as well. Viatra Query expects graph patterns to be specified using the Java API

PSystem.[3] We implemented the IncA compiler to emit PSystem code from an IncA analysis. After startup, Viatra Query uses the navigator to initialize its computation graph and to retrieve AST changes.

3.5.3 Applicability in other IDEs

The applicability of our solution is ultimately determined by the granularity of an IDE's incremental change notifications as the subject program changes. We see three kinds of IDEs where our solution can be applied efficiently. (i) Projectional IDEs (like MPS) where code manipulations directly correspond to incremental AST change events. (ii) Graphical IDEs, which, like projectional IDEs, also directly manipulate structures and can thus easily derive AST changes after a user edit. (iii) Finally, parser-based IDEs with a front end that can efficiently compute AST deltas. This may require a combination of incremental parsing [95, 139] and efficient tree diffing algorithms. In this case, the degree of incrementality that our approach can achieve depends on the granularity of the incremental AST differences the front end can provide. In Chapter 7, we elaborate on our own experiments with a parser-based front end for IncA.

3.6 Case Studies

To validate our approach, we have used IncA to implement three program analyses for mbeddr C [135] and one program analysis for Java. mbeddr C is an extensible C dialect and IDE for embedded software built on top of MPS. This section describes the program analyses and provides details about their implementation, while Section 3.7 presents the performance evaluation.

3.6.1 Control Flow Analysis

The introductory example in Section 3.2 already gave an intuition about control flow analysis in IncA Datalog. The incremental construction of a CFG is an important building block for incremental, flow-sensitive analyses such as the flow-sensitive points-to analysis described next. These two analyses combined enable further analyses such as uninitialized variables and unused assignment analysis [90].

We implemented an incremental control flow analysis that handles all of mbeddr C, including conditionals (`if` and `switch`), loops (`for`, `while`, and `do while`), and jumps (`break` and `continue`). The implementation follows the style of the introductory example, extending the `CFlow` rule with further alternatives to handle all control statements. The complete control flow analysis produces a CFG where the nodes not only represent statements of the program, but also other control flow points like the alternative `case` branches of a `switch` statement or the `else if` parts of an `if` statement.

[3]https://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/PSystem

$$\frac{}{u \rightarrow v}\ u = \&v \ ^{(1)} \quad \Bigg| \quad \frac{x \rightarrow v}{u \rightarrow v}\ u = x \ ^{(2)} \quad \Bigg| \quad \frac{x \rightarrow y,\ y \rightarrow v}{u \rightarrow v}\ u = {*x} \ ^{(3)} \quad \Bigg| \quad \frac{x \rightarrow u,\ y \rightarrow v}{u \rightarrow v}\ {*x} = y \ ^{(4)}$$

Figure 3.8: Andersen's rules for points-to analysis. The arrow $u \longrightarrow v$ means that variable u points to variable v.

3.6.2 Points-to Analysis

Our second case study is a points-to analysis for mbeddr C. Given a variable that stores a pointer, the goal of a points-to analysis is to identify the possible targets of the variable. There is a vast amount of research in this area [77, 106, 143] because the precision of points-to analysis directly benefits optimizations and other analyses.

Our analysis is an Andersen-style analysis [6], which is a well-known formulation for computing points-to information. The Andersen rules consider four basic kinds of assignments as shown in Figure 3.8 and derive the points-to relation for the whole program from them. Our points-to analysis in IncA builds on Andersen's rules but extends them in three ways. First, by implementing the analysis in IncA, we immediately improve the run time after code changes through incrementality. Second, we add flow-sensitivity by building on top of our incremental control flow analysis. This means that the analysis computes the points-to targets for variables per CFG node. Third, we do not require the code to only use the four kinds of assignments in Andersen's rules, rather support all of mbeddr C except pointer arithmetics.

We point out though that our analysis is *unsound*, as it is intra-procedural, and it ignores the effects of transitively reachable functions through function calls. In other words, our analysis will compute an under-approximation of the potential points-to targets for the program variables. This means that the analysis cannot be used for optimization purposes, but it can still be used for error reporting. Implementing the analysis this way was an intentional design choice; The baseline version of IncA is simply not powerful enough to support inter-procedural analyses efficiently. By the end of this dissertation, we will create a version of IncA that can efficiently support inter-procedural analyses, as well. In fact, we will keep revisiting points-to analysis as we extend IncA. For now, we settle with the unsound analysis, as it is still an expensive analysis due to flow-sensitivity, so it is interesting to see the performance of IncA.

3.6.3 Well-formedness Checks for mbeddr C

We implemented four well-formedness checks for mbeddr C and its language extensions. While the control flow analysis and the points-to analysis inspected each function declaration in separation, our well-formedness checks require global knowledge about the source code. The checks are as follows:

CYCLE mbeddr C provides modules for organizing code. This check detects cyclic module dependencies.

GLOBAL This check detects conflicting global variables with the same name across modules.

```
module FindBugs {
  public CI_CONFUSED_INHERITANCE(class : Class) :- {
    Class.isFinal(class, isFinal),
    isFinal == true,
    Class.member(class, member),
    Field(member),
    Field.visibility(member, visibility),
    ProtectedVisibility(visibility)
  }
}
```

Figure 3.9: FindBugs `CI_CONFUSED_INHERITANCE` in IncA.

REC This check detects recursive functions by construction and inspection of a call graph. In embedded systems with constrained memory, the stack space required for recursive functions is often unacceptable.

COMP mbeddr C supports interfaces and composable components. This check detects components that fail to implement all functions declared by their interfaces.

3.6.4 FindBugs for Java

FindBugs [62] is a suite of predefined patterns to detect potential bugs in Java code. To show that our system is independent of the subject language, we implemented 10 FindBugs analyses in IncA for MPS' Java dialect. Apart from a few language extensions, this Java language is identical to the original Java language. As an example, we show the implementation of the `CI_CONFUSED_INHERITANCE` rule in Figure 3.9. It detects `final` classes that have at least one protected field. Since the class is final, it cannot be subclassed, and the field should be private or public. The implementation runs incrementally thanks to IncA.

3.7 Performance Evaluation

This section presents the performance evaluation of IncA using the case studies from Section 3.6. We answer the following questions:

Run Time (Q 3.1): Are incremental IncA analyses significantly faster than their non-incremental counterparts? Does this come with an acceptable initialization time?

Memory (Q 3.2): Is the extra memory induced by incrementalization acceptable?

Optimization Impact (Q 3.3): How does our optimization (Section 3.4) affect the run time and memory requirements?

3.7.1 Evaluation Setup

For each case study, we start with an initial subject program (introduced below). After the initial, non-incremental run of the analysis, we programmatically make 100 updates in the subject program and run the analysis after each update. Each update consists of 1 to 20

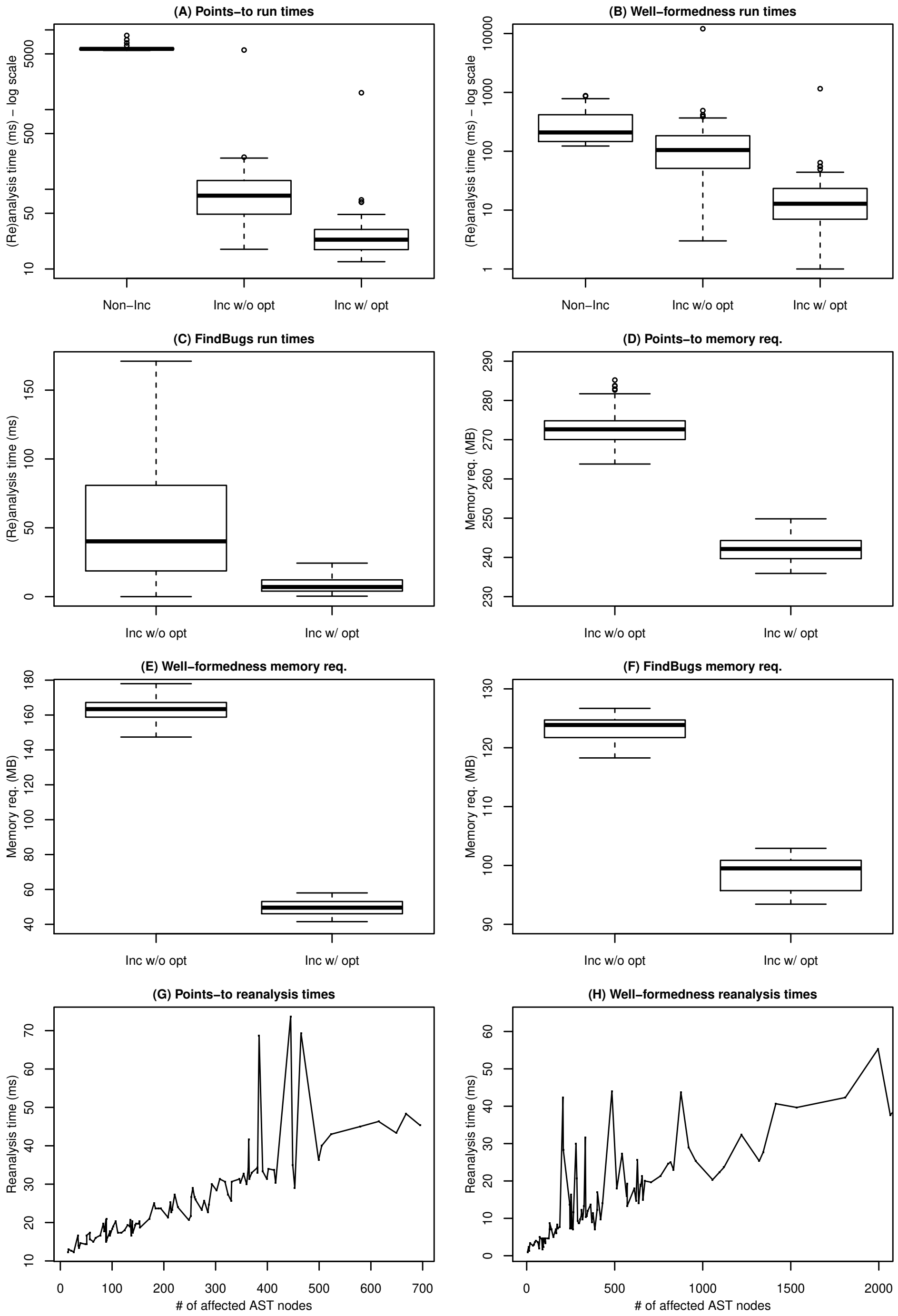


Figure 3.10: Run time and memory measurements for the case studies.

random code changes, such as duplicating a statement, deleting a function, renaming a variable, or introducing a new import. This approach allows us to imitate how a user would modify the source code.

We measure the wall-clock time of processing the initial code and of processing each update step. For the memory measurement, we call the garbage collector after each analysis run and measure the required heap memory. We subtract the heap memory used before running the first analysis to obtain the memory usage of IncA. We repeat each measurement five times and discard the results of the first and second run to account for JVM warm-up.

As the first benchmark, we run the control flow and points-to analysis on the Toyota ITC code,[4] a collection of C code snippets with intentional bugs to test the precision of static analysis tools. The code base comprises about 15,000 lines of C code. We compare the performance of the incremental analyses to a non-incremental flow-sensitive points-to analyses that was already available in mbeddr. The two analyses produce exactly the same results on the benchmark.

The second benchmark was running the well-formedness checks on a commercial Smart Meter software implemented in mbeddr C [137]. A smart meter is an electric meter that continuously records the consumption of electrical power, calculates derived quantities, and sends the data back to the utility provider for monitoring and billing. The whole project comprises about 44,000 lines of mbeddr C code. For comparison, we implemented non-incremental well-formedness checks in MPS Java that produce exactly the same results.

Running the 10 FindBugs checks constituted the third benchmark. We ran it on the Java implementation of the mbeddr importer which is responsible for migrating legacy C code to mbeddr C. The importer comprises about 10,000 lines of MPS Java code. Because of the MPS Java code base, we would need to generate textualized Java code after every code change to be able to use the original FindBugs tool. For our large code base this is impractical, and thus for this benchmark, we do not have a non-incremental counterpart.

We ran the benchmarks on a 64-bit OSX 10.10.3 machine with an Intel Core i7 2.5 GHz processor and 16 GB of RAM, using MPS version 3.3 and Java 1.8.0_65.

3.7.2 Evaluating Run Time (Q 3.1) and Optimization Impact (Q 3.3)

Figure 3.10 A-C show the results of our run time measurements. For points-to analysis and well-formedness checks, we show the results of the incremental and non-incremental solutions using a logarithmic scale. For FindBugs, we only show the incremental results on a linear scale.

Box plots A and B immediately show that incremental program analyses in IncA perform significantly better than non-incremental analyses. Box plots A - C also show that the optimization has a significant impact on the run time. The following table summarizes the median run time data:

[4] https://github.com/regehr/itc-benchmarks

	Non-inc.	Inc w/o opt		Inc w/ opt	
		init	update	init	update
Points-to (A)	5.8s	5.6s	83.2ms	1.6s	23.3ms
W.-form. (B)	209ms	12.1s	104.8ms	1.2s	12.8ms
FindBugs (C)	n/a	4.5s	40.2ms	2.3s	7ms

The initialization times, especially for the optimized versions, do not pose an unduly run time requirement. After initialization, incremental analysis without optimization achieves speedups of A=70x and B=2x compared to non-incremental analysis. With optimization, we even achieve speedups of A=249x and B=16x. Indeed, the optimization accounts for an additional speedup of 2 - 10x for initialization and of 3 - 8x for change-processing time. IncA and its optimization provide good runtime performance across analyses and for both subject languages.

Figure 3.10 G and H show the points-to and well-formedness analysis times as a function of the input change size (the number of deleted or added AST nodes). IncA scales well because the plots remain roughly linear with increasing input change sizes. We conclude:

Run Time (Q 3.1): Incremental program analysis with IncA provides significant speedups of up to 249x compared to non-incremental analysis. The analyses scale linearly with increasing input change sizes. The initialization times only take a few seconds at most, so we consider them acceptable.
Optimization Impact (Q 3.3): The IncA compiler optimization improves initialization time by 2–10x and change-processing time by 3–8x.

3.7.3 Evaluating Memory (Q 3.2) and Optimization Impact (Q 3.3)

Figure 3.10 D - F show the results of our memory measurements. We measured the memory required by incrementality for each of the 100 update steps and created the box plots from this data. We summarize the median memory requirements:

	Inc w/o opt	Inc w/ opt
Points-to (D)	273MB	242MB
Well-form. (E)	163MB	50MB
FindBugs (F)	124MB	100MB

The points-to analysis is the most complex case study, and it has the biggest memory requirement. Its computation graph caches a major part of the input AST to compute a complete CFG and to handle a large variety of assignments, including nested ones. The optimization helps to reduce the memory requirement with A=11%, B=69%, and C=19%. Based on our experience with real-world usage of MPS, the IDE typically requires around 1 – 2GB of memory. If we calculate with the optimized points-to analysis (as points-to has the highest memory requirement out of all the analyses) and 1GB for MPS, then the analysis adds an extra ~25% memory. We conclude:

Memory (Q 3.2): For real-world scenarios, incremental program analysis with IncA requires an acceptable amount of 25% additional memory for our most complex case study.
Optimization Impact (Q 3.3): The IncA compiler optimization reduces the required memory by 11 - 69%.

3.8 Chapter Summary

We presented a baseline version of the IncA framework. IncA uses its own dialect of Datalog for analysis specification, which allows to implement analysis directly over the AST of subject programs using virtual EDB relations for types and links from a subject language. IncA analyses get compiled into graph patterns, and we use incremental graph pattern matching to evaluate analyses efficiently. We presented a meta analysis that computes hints for the runtime system to tell relevant from irrelevant program changes apart, thereby improving incremental performance. We designed a generic architecture for integrating IncA into IDEs, and we instantiated this architecture for the projectional MPS IDE. We demonstrated the applicability of IncA by developing incremental program analyses for C and Java. Our performance evaluation shows that IncA provides significant speedups compared to non-incremental analyses, while incurring acceptable memory overhead.

Let us revisit our requirements from Section 1.4 to see where we stand with this baseline version of IncA. Our projectional editor-based front end satisfies the requirements Genericity (R4) and Efficiency (R2), as it is language-independent, and it delivers precise AST deltas for free. However, we note that projectional editing is not a widespread approach among IDEs. Ideally, we would like to make IncA available also for parser-based IDEs. We will revisit this later in Chapter 7. In the meta end, the IncA Datalog language satisfies Genericity (R4) and Declarativity (R5), as it is language-independent, and it hides the incrementalization details from analysis developers. However, IncA has limited expressive power at this point (Expressiveness (R3)), as it only supports analysis that relate existing program elements, but it cannot support analyses that also need to generate abstract values at runtime. We will discuss this problem in more details in the next chapter. In the back end, we use a generic incremental evaluator that works with any analysis the meta end can express (Genericity (R4)). The IncA back end delivers on Efficiency (R2), as it updates analysis results in milliseconds for our case studies. We did not talk about Correctness (R1) explicitly, as we reused an existing incremental evaluator, which we extended with the state-of-the-art DRed algorithm.

4 Incrementalizing Lattice-Based Program Analyses

This chapter shares material with the OOPSLA'18 paper "Incrementalizing Lattice-based Program Analyses in Datalog" [126].

Abstract — A key characteristic of existing incremental Datalog-based analysis frameworks, including the previously presented version of IncA, is that they only support the powerset lattice as the sole abstraction that an analysis can use. This means that analyses can only work with sets of tuples, but there is no way to generate values from custom lattices at runtime. This is a limitation because many practically relevant analyses require custom lattices to represent abstract domains (e.g. number intervals, string values), and they also aggregate lattice values at runtime. Moreover, aggregation is typically recursive, as subject programs frequently exhibit cyclic control flow, which severely complicates efficient incrementalization in face of program changes.

In this chapter, we present a novel algorithm called DRed_L that supports the efficient incremental maintenance of recursive lattice-based aggregation in Datalog. The key insight of DRed_L is to dynamically recognize increasing replacements of old lattice values by new ones, which allows us to avoid the expensive (over-)deletion of the old value. We integrate DRed_L into IncA, and we use IncA to realize incremental implementations of strong-update points-to analysis and string analysis for Java. As our performance evaluation demonstrates, both analyses react to code changes within milliseconds.

4.1 Introduction

Lattices are fundamental in program analyses because they are used to represent abstract domains. A lattice is a partially ordered set where any two elements from the set have a least upper bound and a greatest lower bound. These operations are important when analyzing programs with cyclic control flow (or recursive data structures), as they help to aggregate multiple lattices values (e.g. computed on different control flow paths) into a single one. This way an analysis has control over the approximations it uses, as opposed to keeping track of all possible values.

A key characteristic of existing *incremental* Datalog solvers, including $\mathrm{DRed_L}$ presented in Chapter 3, is that they only support the powerset lattice as the abstraction an analysis can use. This means that analyses can only work with sets of tuples, but it is not possible to use custom lattices, such as a lattice representing number intervals. Moreover, the powerset lattice only supports set union and intersection as the aggregation operator, but many practically relevant analyses require custom aggregation operators [90]. For example, an interval analysis uses union on number intervals to aggregate at control flow merge points. As subjects programs frequently exhibit cyclic control flow or recursive data structures, aggregation also becomes recursive. This severely complicates efficient incrementalization because correctly updating analysis results after a program change may require the complete unrolling of previous results that were computed through multiple fixpoint iterations.

In this chapter, we present a new algorithm called $\mathrm{DRed_L}$ that incrementally solves recursive Datalog programs using user-defined aggregations subject to the following two requirements: (i) aggregations operate on lattices and (ii) recursive aggregations are monotonic. Both requirements are readily satisfied by program analyses. The key insight of $\mathrm{DRed_L}$ is that the monotonicity of recursive aggregations allows for efficient handling of monotonic input changes. This is necessary for correctness and essential for efficiency. We have formally verified that $\mathrm{DRed_L}$ is correct and yields the exact same result as evaluating the Datalog program from scratch.

We add support for custom lattices to every component in IncA. First, we extend IncA Datalog to support user-defined lattices and aggregations. Then, we modify the IncA compiler to also support lattices. Finally, we integrate $\mathrm{DRed_L}$ into the incremental graph pattern evaluator in the back end of IncA. The front end remains the same in this chapter, as we reuse the integration with the MPS language workbench.

To evaluate the applicability and performance of IncA, we have implemented two Java analyses in IncA adapted from the literature: Strong update points-to analysis [73] as well as character-inclusion and prefix-suffix string analysis [29]. Our analyses are intra-procedural, the points-to analysis is flow-sensitive, while the string analyses are flow-insensitive. We ran performance measurements for both analyses on four real-world Java projects with up to 70 KLoC in size. We measured the initial non-incremental run time and the incremental update time in face of random code changes. Our measurements reveal that the initialization takes a few tens of seconds, while an incremental update takes a few milliseconds on average. Additionally, we also benchmarked the memory requirement of IncA because incrementalization relies on extensive caching. Our evaluation shows that the memory requirement can grow large but not prohibitive.

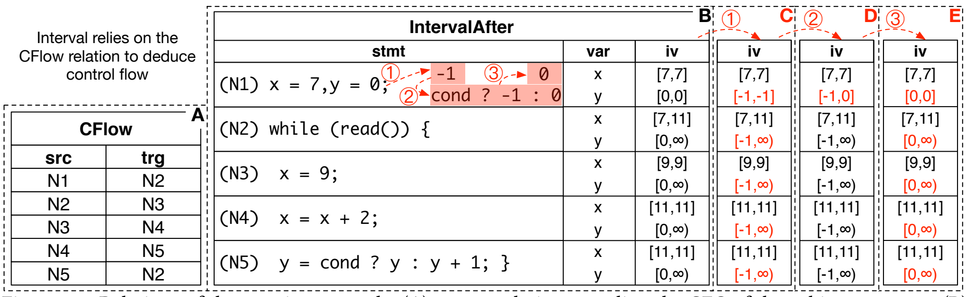
Contributions In summary, we make the following contributions:

- We identify and describe the key challenge for incremental lattice-based program analysis: cyclic reinforcement of lattice values (Section 4.2).
- We present the IncA approach for incremental lattice-based program analysis by embedding in Datalog, requiring recursive aggregation (Section 4.3).
- We develop DRed$_L$, the first algorithm to incrementally solve recursive Datalog rules containing user-defined aggregations (Section 4.4).
- We provide a formal treatment of DRed$_L$ and prove it correct (Section 4.5).
- We implement DRed$_L$ and the integration into IncA as open-source software (Section 4.6).
- We develop two lattice-based program analyses as case studies; strong-update points-to analysis and a collection of string analyses (Section 4.7).
- We evaluate the run time and memory performance of DRed$_L$ through our case studies on real-world Java subject programs (Section 4.8).

4.2 Challenges of Incrementalizing Lattice-based Program Analysis

In this section, we introduce a running example to illustrate the challenge of incremental lattice-based program analysis. The example is a flow-sensitive interval analysis for Java, which reports the possible value ranges of program variables. As a specification language, we use IncA Datalog, but we extend it with support for lattices. It will become clear in this section that incremental program analysis requires incremental recursive aggregation, which existing solvers fail to support.

Interval Analysis with Datalog An interval analysis reasons about the runtime values of variables in terms of number intervals. A flow-sensitive interval analysis keeps track of the value ranges of variables at each control flow location separately. To this end, we need


Interval relies on the CFlow relation to deduce control flow
A
B
C
D
E
①
②
③
-1
cond ? -1 : 0
0

CFlow

src	trg
N1	N2
N2	N3
N3	N4
N4	N5
N5	N2

IntervalAfter

stmt	var	iv (B)	iv (C)	iv (D)	iv (E)
(N1) x = 7,y = 0;	x	[7,7]	[7,7]	[7,7]	[7,7]
	y	[0,0]	[-1,-1]	[-1,0]	[0,0]
(N2) while (read()) {	x	[7,11]	[7,11]	[7,11]	[7,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N3) x = 9;	x	[9,9]	[9,9]	[9,9]	[9,9]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N4) x = x + 2;	x	[11,11]	[11,11]	[11,11]	[11,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)
(N5) y = cond ? y : y + 1; }	x	[11,11]	[11,11]	[11,11]	[11,11]
	y	[0,∞)	[-1,∞)	[-1,∞)	[0,∞)

Figure 4.1: Relations of the running example. (A) `CFlow` relation encoding the CFG of the subject program. (B) `Interval` relation showing the value ranges of variables per CFG node. (C) - (E) Updates in the `Interval` relation after changing the initializer of y through the series `0` ⟶ `-1` ⟶ `cond ? -1 : 0` ⟶ `0`. Red color shows the changes compared to previous values.

a control flow analysis for Java first. For now, we just assume that this analysis is already implemented in a style similar to the one we presented for C in Section 3.6, that is, we can assume that a relation `CFlow(src, trg)` is available. Each `(src, trg)` tuple represents an edge of the CFG. Figure 4.1 B shows an example subject program with statements `N1` to `N5`. Its CFG as computed by `CFlow` is shown in Figure 4.1 A.

We now construct a flow-sensitive interval analysis on top of this CFG. We define a relation `IntervalAfter(stmt, var, iv)`, where a tuple `(s, v, i) ∈ IntervalAfter` describes that after the execution of statement `s`, variable `v` must take a value in interval `i`. Terms `s` and `v` are existing program elements, and `i` is a computed value from the *interval lattice*. To compute the interval `i` of variable `v` after statement `s`, the analysis must consider two cases. If `s` (re)assigns `v`, we compute `v`'s interval based on the assignment expression. Otherwise `v`'s interval does not change, and we propagate the interval associated with `v` from before the statement:

```
IntervalAfter(stmt : Stmt, var : Var, iv : Interval) :- {
  AssignsNewValue(stmt, var),
  AssignedInterval(stmt, var, iv)
} alt {
  not AssignsNewValue(stmt, var)
  IntervalBefore(stmt, var, iv)
}
```

In order to figure out this previous interval, we query the CFG to find all control-flow predecessors of the current statement and collect the variable's intervals for each of the predecessors in a relation `PredecessorIntervals`. As the interval containment *partial order* forms a *lattice*, we can obtain the smallest interval containing all predecessor intervals of the current statement. It is computed by aggregating such predecessors using the *least upper bound* (`lub`) lattice operation:

```
PredecessorIntervals(stmt:Stmt, var:Var, pred:Stmt, iv:Interval) :- {
  CFlow(pred, stmt),
  IntervalAfter(pred, var, iv)
}
IntervalBefore(stmt : Stmt, var : Var, lub(iv : Interval)) :- {
  PredecessorIntervals(stmt, var, _pred, iv)
}
```

Note that `IntervalAfter` and `IntervalBefore` are *mutually recursive* and also induce *cyclic tuple dependencies* whenever `CFlow` is cyclic. This is typical for flow-sensitive program analyses. The term `lub(iv:Interval)` in the head atom of `IntervalBefore` is new syntax; IncA Datalog allows to use aggregation operators this way.

We obtain the final analysis result by computing the (least) fixpoint of the Datalog rules. The computation always terminates because the rules in our analysis are monotonic wrt. the partial order of the interval lattice and because we use widening to ensure that the partial order does not have infinite ascending chains [30]. For example, as shown in Figure 4.1 B, the interval analysis computes tuples `(N2, x, [7, 11])` and `(N2, y, [0, ∞])` for the loop head. Tuple `(N2, x, [7, 11])` is the result of joining `(N1, x, [7, 7])` and `(N5, x, [11, 11])`. Tuple `(N2, y, [0, ∞])` is the result of the fixpoint computation, joining all intermediate intervals `lub([0, 0], [0, 1], [0, 2], ...)` of `y` before `N2`.

Incremental Interval Analysis The analysis presented up to here is standard. We now look at incrementalization. Existing incremental Datalog solvers cannot handle this analysis because they do not support recursive aggregation over lattices. The root cause of the limitation is that realistic subject programs typically have loops or recursive data structures. Analyzing them requires fixpoint iteration over cyclic control flow or data-flow graphs, which, in turn, requires recursive aggregation.

The goal of an *incremental* analysis is to update its results in response to changes in subject programs with *minimal computational effort*. For illustrative purposes, we consider three changes ①, ②, and ③ of the initializer of `y` in Figure 4.1 B, and then we review what is expected from an incremental analysis in response to them. First, we change it from `0` to `-1`. Figure 4.1 C shows the updated `iv` column of relation `IntervalAfter`. While this change does not affect previous results of `x`, the intervals assigned to `y` require an update at all CFG nodes. After a subsequent change from `-1` to `(cond?-1:0)`, we expect that only the interval at `N1` gets updated as shown in Figure 4.1 D because the analysis of the loop already considered `y=0` before. Third, consider a final change from `(cond?-1:0)` to `0`. Now the analysis does not have to consider `y=-1` anymore, and we expect an update to all lower bounds of the intervals assigned to `y` as shown in Figure 4.1 E.

How can we update a previous result with minimal computational effort in response to changes ①, ②, and ③? Fundamentally, we propagate program changes bottom-up through the analysis' Datalog rules, starting at the changed program element. For example, a change to a variable's initializer directly affects the result of `AssignedInterval`, which is then used in the first rule of `IntervalAfter`. If the new initializer has a different interval than the old initializer, `AssignedInterval` propagates a derived change to `IntervalAfter`. A derived change leads to the deletion of the old interval and the insertion of the new interval into the relevant relation. In turn, `IntervalAfter` propagates such a change to `PredecessorIntervals`, which propagates it to the aggregation in `IntervalBefore`, which propagates it to the second rule of `IntervalAfter`, and so on. This way we can handle change ①, where we delete `(N1, y, [0, 0])`, insert `(N1, y, [-1, -1])`, and propagate these changes.

Handling change ② efficiently is more challenging: We must augment our strategy to avoid deleting `(N1, y, [-1, -1])` before inserting `(N1, y, [-1, 0])`, so that we can reuse previous results. Failing to do that would result in re-computing the exact same result, which is unnecessary excess work degrading efficiency.

The toughest problem is change ③, where we delete `(N1, y, [-1, 0])` and insert `(N1, y, [0, 0])`. The problem manifests at CFG node `N2` and is due to the loop in the CFG and the `lub` aggregation over it. Before change ③, `N2`'s CFG predecessors `N1` and `N5` report intervals `[-1, 0]` and `[-1, ∞)` for `y`, as shown in Figure 4.1 D. When propagating deletion `(N1, y, [-1, 0])` and insertion `(N1, y, [0, 0])` to `N2`, we expect to replace `(N2, y, [-1, ∞))` by `(N2, y, [0, ∞))`. However, the aggregation in `IntervalBefore` for `N2` also has to consider interval `[-1, ∞)` from `N5` and `lub([0, 0], [-1, ∞)) = [-1, ∞)`. However, note how this yields the *wrong* result for `N2` because the very reason for `[-1, ∞)` at `N5` is the old initializer at `N1`, which just got deleted. We call this situation *cyclic reinforcement* of lattice values. An incremental analysis must carefully handle cyclic reinforcements to avoid computing incorrect analysis results. The design of an algorithm for executing program analyses incrementally and correctly is the central contribution of this chapter.

Problem Statement Our goal is to incrementalize program analyses with custom lattices and recursive, user-defined aggregations. Given that we design an improved version of IncA in terms of a framework as a whole, all requirements from Section 1.4 apply. In ??, we defined Expressiveness (R3), as "The analysis framework must support a wide range of practically relevant analyses". We refine this requirement for this chapter to capture precisely what is our goal here. We require that our solution applies systematically to any program analyses with lattice-based recursive aggregation.

Note that we intentionally do not mention *precision* because that is the responsibility of the analysis developer when designing the lattices and their operations. There is no limitation on the design of those by our solution. However, a more precise analysis might take more time to compute (e.g. because we reach a fixpoint in more iterations), so it has an impact on performance, just like in a non-incremental analysis.

4.3 Incremental Lattice-Based Program Analysis with IncA

We extend the baseline version of IncA by adding support for lattices in all components of IncA. This section focuses on our extensions to the back end. We recap how we incrementalize analyses with IncA, and we add support for lattice-based aggregations in two steps; first supporting non-recursive aggregations and then recursive ones.

4.3.1 Incremental Execution of Non-Recursive Analyses

Recap how IncA incrementalizes non-recursive analyses IncA incrementalizes analyses encoded with the IncA Datalog language. An IncA analysis uses EDB relations to access the required elements (instances of types and links) from the AST of a subject program. The output of an IncA analysis is the tuples in the IDB relations defined by the IncA Datalog rules. IncA expects from the IDE that, in response to a change in the subject program, the IDE can efficiently update the contents of the EDB relations. We use a computation graph to track data dependencies between rules and perform relational algebra operations defined by an IncA analysis. In response to a change in the EDB relations, IncA propagates changes in the computation graph, updates the results of computation nodes, and memoizes the new results. The IncA compiler optimizes the subject analyses to avoid the propagation of irrelevant changes, as described in Section 3.4.

Adding support for non-recursive aggregation In contrast to other incremental systems (cf. survey on incremental solvers [46]), IncA supports aggregations over *recursive* relations. Here, however, we first review how IncA handles *non*-recursive aggregations, which, while much simpler, is an important stepping stone. Formally, an aggregating Datalog rule has the form:

```
Agg(t₁, ..., tₖ, α(v)) :- {
  Coll(t₁, ..., tₖ, _x₁, ..., _xₗ, v)
}
```

Here, the *aggregand* column v and *aggregate result* column α(v) are both lattice-valued, and α is an *aggregation operator*, i.e. a mapping from a *multiset* of lattice values to a single lattice value. Without loss of generality (as shown below), we assume this is the only rule with `Agg` in the head. We call `Agg` the aggregating relation and `Coll` the *collecting* relation; $\mathtt{t}_1,\ldots,\mathtt{t}_k$ are the *grouping* variables, while $\mathtt{\_x}_1,\ldots,\mathtt{\_x}_l$ are *auxiliary* variables.

For example, in the interval analysis from Section 4.2, `PredecessorIntervals` served the role of `Coll` whereas `IntervalBefore` served the role of `Agg`, while α was `lub`. We used `stmt` and `var` as grouping variables, and we used `pred` as an auxiliary variable to enumerate the interval values of all CFG predecessors.

For each substitution of the grouping columns, the aggregand values `v` in `Coll`, if any, are mapped by α into the aggregate result column of the single corresponding tuple in `Agg`. Given a set of grouping values $\mathtt{t}_1,\ldots,\mathtt{t}_k$, different sets of values $\mathtt{\_x}_1,\ldots,\mathtt{\_x}_l$ can have the same aggregand `v` associated with them, so α aggregates over a *multiset* of values, instead of just a simple set.

Note that the above form of the aggregating rule does not restrict expressiveness: IncA Datalog actually allows multiple alternative rule bodies for `Agg`. The compiler then introduces a helper relation `Coll` that is derived using these multiple rule bodies, and then aggregates `Agg` from `Coll`.

We can incrementalize non-recursive aggregations by (i) incrementally maintaining `Coll` as usual and (ii) incrementally maintaining the aggregate result α`(v)` of each group whenever an aggregand `v` is inserted or deleted from the collecting relation. IncA specifically supports the latter kind of incrementality for aggregation operators that are induced by associative and commutative binary operations, e.g. least upper bound or greatest lower bound (see Section 4.4.4).

The above form of aggregating rules is independent of whether the aggregation is recursive or not. An aggregation is recursive if `Coll` also depends on `Agg` (e.g. see dependencies between `PredecessorIntervals`, `IntervalAfter`, and `IntervalBefore` in Section 4.2). This leads us to the next part where we discuss recursive analyses.

4.3.2 Incremental Execution of Recursive Analyses

Recap how IncA incrementalizes recursive analyses In contrast to non-recursive relations, recursive relations require a fixpoint computation. That is, an insertion into a recursive relation can trigger subsequent insertions, which can trigger subsequent insertions, and so on. The difficulty comes with handling deletions due to cyclic reinforcement of tuples in the analysis result. We demonstrated this problem in Section 2.4. The essence of the problem is that if a tuple was derived multiple times within a cycle, deleting one derivation does not necessarily invalidate the other derivations, but invalidating all derivations is sometimes necessary to obtain correct results. This is similar to the cyclic reinforcement of lattice values we discussed in Section 4.2, with an important difference: The cyclic reinforcement of tuples does not involve aggregations.

To provide correct incremental maintenance for recursive analyses, we employed the DRed algorithm in IncA in Chapter 3. The basic strategy of DRed is to run in two phases, each until fixpoint, in response to program changes. First delete everything that transitively depends on a deleted tuple while temporarily ignoring alternative derivations. This is an over-approximation and, in general, will delete too much. After the deletion reaches a fixpoint, start a re-derive phase to insert back all those tuples that can be derived from the remaining ones that were left intact during the delete phase. The re-derive phase also uses the insertions as input to derive new tuples.

Adding support for recursive aggregation The key technical contribution of this chapter is to develop a novel algorithm for incrementalizing lattice-based recursive aggregation in Datalog. As we explained in Section 4.2, the main challenge is the cyclic reinforcement of lattice values (in contrast to cyclic reinforcement of *tuples*, as discussed above). Specifically, change ③ in Figure 4.1 B induced the deletion of tuple `(N1, y, [-1, 0])`, which should have triggered the deletion of `(N2, y, [-1, ∞))`. However, because `N2` occurred in a loop, there was cyclic reinforcement between the lattice values: `N5` still reported the interval `[-1, ∞)`, even though the very reason for that value is that the initial interval was `[-1, 0]` previously. At this point, we did not know how to proceed.

Why can we not just apply DRed here? How is cyclic reinforcement of lattice values different from cyclic reinforcement of tuples? The main difference is that updating an aggregate value induces both a deletion of the old value and an insertion of the new value. However, DRed first processes all deletions and postpones insertions until the re-derive phase. Hence, one issue is that lattice values require an interleaving of the delete and re-derive phases, which violates the contract of DRed. A second issue is more subtle. Let's say we allow interleaving. When deleting the old aggregate value, DRed's delete phase will delete all tuples derived from it. In particular, it will delete the new aggregate value, which we were just about to insert.

To resolve these issues, we developed a novel algorithm called DRed$_\mathrm{L}$. It generalizes DRed to support incremental computation of recursive *monotonic* aggregations over lattices. Given a partially ordered set $(\mathcal{M}, \sqsubseteq)$, an aggregation operator α is $\sqsubseteq$-monotonic if $\alpha(M) \sqsubseteq \alpha(M \cup \{m\}) \sqsubseteq \alpha(M \cup \{m'\})$ for all multisets of values $M \subseteq \mathcal{M}$ and all values $m, m' \in \mathcal{M}$ with $m \sqsubseteq m'$. That is, the aggregate result of a monotonic aggregation increases with the insertion of any new tuple or with the increasing replacement of any existing tuple in the collecting relation.

Crucially, our algorithm recognizes monotonicity at runtime when handling the update of an aggregate result. Given a $\sqsubseteq$-monotonic aggregator α, whenever DRed$_\mathrm{L}$ sees a deletion $(\mathtt{t}_1,\ \ldots,\ \mathtt{t}_k,\ \mathtt{old})$ from a relation and an insertion $(\mathtt{t}_1,\ \ldots,\ \mathtt{t}_k,\ \mathtt{new})$ to the same relation with $\mathtt{old} \sqsubseteq \mathtt{new}$, it recognizes that together they represent an increasing replacement of a lattice value. We call such a change pair a $\sqsubseteq$-*increasing replacement*, and deletions that are part of a $\sqsubseteq$-increasing replacement do not need to go through a full delete phase. This way, deletions of old aggregate results will not invalidate cyclically dependent lattice values and, in particular, the new aggregate values. This allows DRed$_\mathrm{L}$ to perform correct incremental maintenance even in the presence of recursive aggregation.

We describe the details of DRed$_\mathrm{L}$ in Section 4.4 focusing on correctness and efficiency. While in Section 4.4 we explain DRed$_\mathrm{L}$ in its full generality, its main application is the incrementalization of program analyses that use custom lattices. Program analyses routinely use lattices to approximate program behavior [90]. Without recursive aggregation, only the powerset lattice can be represented, which also limits us to using set union or intersection as the aggregation operator. DRed$_\mathrm{L}$'s support for recursive aggregation lifts this limitation and enables the incrementalization of program analyses over any user-defined lattice. In Section 4.7, we implement two lattice-based program analyses, and, in Section 4.8, we show

how DRed$_\mathrm{L}$ performs for these analyses on real-world subject programs.

4.4 Incremental Recursive Aggregation with DRed$_\mathrm{L}$

We design DRed$_\mathrm{L}$, an incremental algorithm for solving Datalog rules that use recursive aggregation over lattices. Given a set of Datalog rules, DRed$_\mathrm{L}$ efficiently and transitively updates IDB relations upon changes to EDB relations. DRed$_\mathrm{L}$ propagates changes to IDB relations in the four steps described below, which are similar to the earlier DRed algorithm. However, there are key differences at each step, necessitated by the support for lattice-based aggregation:

Change splitting: Split incoming changes into monotonic changes (increasing replacements and insertions) and anti-monotonic changes (deletions that are not part of increasing replacements). The crucial novelty over the older DRed algorithm is the lattice-aware recognition of increasing replacements at runtime, which heavily impacts the other three steps, as well.

Anti-monotonic phase: Interleaved with the previous step, we process anti-monotonic changes by transitively deleting the relevant tuples and everything derived from them. This is an over-approximation and, in general, will delete too much, but, importantly, over-deletions guarantee that we compute correct results in the face of cyclic reinforcements.

Re-derivation: Fix the over-approximation of the previous step by re-deriving deleted tuples from the remaining tuples.

Monotonic phase: Process monotonic changes. For insertions, we insert the new tuple and transitively propagate the effects. For increasing replacements, we simultaneously delete the old tuple and insert the new tuple and transitively propagate their effects. By propagating deletions and insertions of increasing replacements together, we ensure that dependent relations will in turn recognize them as increasing replacements and handle them accordingly.

In the remainder of this section, we summarize the assumptions of DRed$_\mathrm{L}$ on Datalog rules, introduce necessary data structures for DRed$_\mathrm{L}$, present DRed$_\mathrm{L}$ as pseudocode, and explain how DRed$_\mathrm{L}$ incrementalizes aggregations.

4.4.1 Assumptions of DRed$_\mathrm{L}$ on the Input Datalog Rules

In order to guarantee that DRed$_\mathrm{L}$ satisfies the requirements Correctness (R1) and Efficiency (R2), the input Datalog rules must meet the following assumptions:

Monotonic recursion (A 4.1): We call a set of mutually recursive Datalog rules a *dependency component*. DRed$_\mathrm{L}$ assumes that each dependency component respects a partial order $\sqsubseteq$ of each used lattice (either the natural order of the lattice, or its inverse). Given this order, the relations represented by the rules must only recur $\sqsubseteq$-monotonically: If they are updated by insertions or $\sqsubseteq$-increasing replacements, then *recursively* derived results may only change by insertions or $\sqsubseteq$-increasing replacements. This assumption has important implications: Abstract interpretation operators and aggregations must be monotonic within a dependency component wrt. the chosen partial order. The analysis developer must ensure that operators are monotonic. Additionally, recursion through negation is not allowed: IncA automatically rejects analyses that do not conform to this requirement (Section 4.6).

Aggregation exclusivity (A 4.2) Alternative rules deriving the same (collecting) relation must produce mutually disjoint results, if the relation is in a dependency component that uses aggregation at all. This is important for ruling out cyclic reinforcements.

Cost consistency (A 4.3) Assume that `R(`$\texttt{t}_1$`, ..., `$\texttt{t}_k$`, v)` is a non-aggregating relation, where `v` is a lattice-typed column. We require that columns `(`$\texttt{t}_1$`, ..., `$\texttt{t}_k$`)` uniquely determine column `v`.

No infinite ascending chains (A 4.4) DRed_L computes the least fixpoint of the Datalog rules. To ensure termination and as is standard in program analysis frameworks, DRed_L requires that the used lattices do not contain infinite $\sqsubseteq$ -ascending chains [11, Section 6] for any $\sqsubseteq$ that is chosen as part of Monotonic recursion (A 4.1). Fulfilling this requirement may require widening [30], as also used in our interval analysis in Section 4.2.

Monotonic recursion (A 4.1) allows a dependency component to use and aggregate over several lattices. For each of those, the component must be monotonic; this is the responsibility of the analysis developer. In many typical lattice-based analyses (e.g. where aggregators are idempotent like `lub`, `glb`), the IncA compiler can automatically guarantee Cost consistency (A 4.3) by transforming the IncA analysis code (see Section 4.6.3). In other cases, it is the responsibility of the analysis developer. For non-aggregating standard Datalog programs, Aggregation exclusivity (A 4.2) trivially holds.

To prove that DRed_L correctly updates analysis results, we present a proof sketch in Section 4.5. We also validated that these assumptions do not inhibit expressiveness (Expressiveness (R3)) for incremental program analyses by integrating DRed_L into IncA (Section 4.6) and developing lattice-based analyses as case studies (Section 4.7).

4.4.2 Support Data Structures

The anti-monotonic step of DRed_L performs an over-deletion, after which some tuples may need to be re-derived, and the monotonic phase performs monotonic deletions that do not always have to be propagated. To make these decisions, we adapt support counts for tuples from DRed, and we design a new data structure called support multisets for aggregate results:

- *Support count*: The support count of a tuple is equal to the number of alternative derivations a tuple has within a relation (across all alternative rules and local variables). $\texttt{Support}^{\#}_{R}$`(t)` represents the support count of tuple `t` in relation `R`.
- *Support multiset*: The support multiset of an aggregate result contains the individual aggregands that contribute to it. Formally, given an aggregating rule as in Section 4.3.1, the support multiset $\texttt{Support}^{MS}_{Agg}$ associates, to each substitution of

```
 1 procedure maintainIncrementally(all) {
 2     for C in top. order of dep. components {
 3         effect := immediateConsequences(C, all)
 4         (anti, mon) := changeSplitting(effect)
 5         deleted := ∅
 6         while (anti != ∅) { // anti-monotonic phase - fixpoint
 7             new := updateAnti(anti); deleted ∪= new
 8             newEffect := immediateConsequences(C, new)
 9             (anti, mon) := changeSplitting(newEffect ∪ mon)
10         }
11         red := directRederive(deleted) // re-derive phase
12         mon ∪= immediateConsequences(C, red)
13         all ∪= deleted ∪ red
14         while (mon != ∅) { // monotonic phase - fixpoint
15             new := updateMon(mon); all ∪= new
16             mon := immediateConsequences(C, new)
17         }
18     }
19 }
```

Figure 4.2: DRed$_L$ main procedure.

grouping variables of the aggregating relation Agg, the aggregands in the group from Coll. In other words, $\mathtt{Support}^{MS}_{Agg}(\mathtt{t}_1,\dots,\mathtt{t}_k)$ is the multiset of values v that satisfy Coll($\mathtt{t}_1$,...,$\mathtt{t}_k$,_$\mathtt{x}_1$,...,_$\mathtt{x}_l$,v) for some _$\mathtt{x}_1$, . . . ,_$\mathtt{x}_l$.

These support data structures are used and incrementally maintained by DRed$_L$. As $\mathtt{Support}^{\#}_{R}$ is no larger than R and $\mathtt{Support}^{MS}_{Agg}$ is no larger than Coll, there is no asymptotic overhead.

4.4.3 DRed$_L$ Algorithm

We present the DRed$_L$ algorithm as pseudocode in the following. The main procedure is shown in Figure 4.2. The entry point is procedure `maintainIncrementally`, which takes a changeset *all* as input and updates affected IDB relations. In the code, we use italic font exclusively for variables that store changesets.

As usual for incremental Datalog solvers, DRed$_L$ iterates over the dependency components of the analysis in a topological order (line 2). We exploit that recursive changes within each component are required to be monotonic by Monotonic recursion (A 4.1). We start in line 3 by computing the *immediate consequences* of changes in *all* on the bodies of the Datalog rules in the current component C. That is, for a Datalog rule h :– $\mathtt{a}_1,\dots,\mathtt{a}_n$ in C, we compute the consequences of the changes on $\mathtt{a}_1,\dots,\mathtt{a}_n$ first. We omit the details of `immediateConsequences`, but the implementation relies on relational algebra operations that we discussed in Chapter 3. Technically, the interim result `effect` is expressed on rule bodies and not yet projected to rule heads like h (the actual relations to be maintained). This allows us to maintain support counts and support multisets for alternative body derivations.

If component C uses aggregation, in line 4 we perform **change splitting** of changeset *effect*, according to the $\sqsubseteq$ of Monotonic recursion (A 4.1). We compute the set of *monotonic*

```
20 procedure updateAnti(body) {
21     head := ∅
22     foreach change in body  {
23         R := rel(change); h := $\pi_R$(change)
24         if (R is non-aggregating) {
25             Support$^{\#}_R$(|h|) -= 1
26             if (|h| ∈ R) {
27                 head ∪= {h}
28             }
29         } else { // aggregating: R($\bar{t}$, $\alpha$(v))
30             let R($\bar{t}$, v) = |h|
31             Support$^{MS}_R$($\bar{t}$) -= {v}
32             if (∃w. ($\bar{t}$, w) ∈ R) {
33                 head ∪= {-R($\bar{t}$, w)}
34             }
35         }
36     }
37     update stored relation contents by head
38     return head
39 }
```

Figure 4.3: DRed$_\mathrm{L}$ anti-monotonic phase.

changes mon from $effect$ by collecting all insertions and those deletions that are part of an increasing replacement:

$$\begin{aligned} mon \;=\; & \{+r(t_1,\dots,t_k) \mid && +r(t_1,\dots,t_k) \in effect\} \\ \cup\; & \{-r(t_1,\dots,t_k,c_{old}) \mid && -r(t_1,\dots,t_k,c_{old}) \in effect, \\ & && +r(t_1,\dots,t_k,c_{new}) \in effect, \\ & && c_{old} \sqsubseteq c_{new}\} \end{aligned}$$

Here and below we write $+r(t_1,\dots,t_k)$ for a tuple insertion and $-r(t_1,\dots,t_k)$ for a deletion. The set of *anti-monotonic changes* consists of all deletions not in mon. Note that for a component C that does not use aggregation, mon simply consists of all insertions and $anti$ consists of all deletions.

Next, we perform the **anti-monotonic phase** on changeset $anti$ iteratively until reaching a fixpoint (lines 6-10). In each iteration, we first use procedure updateAnti (discussed below) to update the affected support data structures and relations (line 7). This yields changes new to rule heads defined in C. We propagate the new changes to $deleted$ because they are candidates for a later re-derivation. We also propagate the new changes to C to handle recursive effects (line 8), yielding recursive feedback in $newEffect$. By design, the anti-monotonic phase within a dependency component only produces further deletions and never insertions. We merge the $newEffect$ changes with the monotonic changes mon and split them again because a new change may cancel out an insertion or form an increasing replacement. Note that this can be done efficiently by indexing the changesets for aggregating relations over the grouping variables. This way we can efficiently query relevant lattice values when deciding if a pair needs to be split up or formed.

Procedure updateAnti (Figure 4.3) processes anti-monotonic changes of rule bodies and projects them to changes of the rule heads while keeping support counts and support multisets up-to-date. We iterate over the anti-monotonic body changes, all of which are

```
40 procedure directRederive(deleted) {
41     red := ∅
42     foreach change in deleted {
43         R := rel(change)
44         if (R is non-aggregating) {
45             if ($Support^{\#}_R$(|change|) > 0) {
46                 red ∪= {+|change|}
47             }
48         } else { // aggregating: R($\bar{t}$,$\alpha$(v)):-...
49             let R($\bar{t}$,v) = |change|
50             if ($Support^{MS}_R$($\bar{t}$) != ∅) {
51                 red ∪= { +R($\bar{t}$, $\alpha$($Support^{MS}_R$($\bar{t}$)) ) }
52             }
53         }
54     }
55     update stored relation contents by red
56     return red
57 }
```

Figure 4.4: DRed$_L$ re-derive phase.

deletions by definition. For each change, we obtain the changed relation symbol `R`, and we project with π_R the body change to the corresponding change of the relation's head `h` (line 23). While `h` is a change, we write |`h`| to obtain the change's absolute value, that is, the tuple being deleted or inserted. If `R` does not aggregate, we decrease the support count of `h`, and we propagate a deletion of `h` if it is currently derivable in `R`. If instead `R` aggregates, we decompose `h` to obtain the grouping terms $\bar{t}$ and the aggregand `v`. We delete `v` from the support multiset of $\bar{t}$. Furthermore, if `R` currently associates an aggregation result `w` to $\bar{t}$, we propagate the deletion of said association from `R`. Note that the associated aggregation result `w` is unique by Cost consistency (A 4.3), and we delete it as soon as any of the aggregands is deleted. It is important to point out that a positive support count or non-empty support multiset after deletions is no evidence for the tuple being present in the relation, due to the possibility of a to-be-deleted tuple falsely reinforcing itself through cyclic dependencies. We will put back tuples that still have valid alternative derivations, and we will put back aggregate results computable from remaining aggregands in the re-derivation step of DRed$_L$. Finally, we update the stored relations and return *head* for recursive propagation in the main procedure `maintainIncrementally`.

Back in `maintainIncrementally`, we proceed with **re-derivation** to fix the over-deletion from the anti-monotonic phase (lines 11-12). To this end, we use procedure `directRederive` (Figure 4.4) to re-derive tuples that were deleted during the anti-monotonic phase but still have support. The input to the procedure is *deleted*, and we iterate over the deletions in the changeset. If `R` does not aggregate, we use the support count: A positive support count indicates the tuple is still derivable, and we propagate a re-insertion (lines 45-47). If instead `R` aggregates, we use the support multiset: A non-empty support multiset indicates that some aggregand values are left (line 50). In this case, we recompute the aggregation result by applying the aggregation operator α and propagating a re-insertion (line 51). Due to the support multiset, we do not need to re-collect aggregand values, saving precious time. In Section 4.4.4, we explain how we further incrementalize

```
58 procedure updateMon(body) {
59     head := ∅
60     foreach change in body  {
61         R := rel(change); h := $\pi_R$(change)
62         if (R is non-aggregating) {
63            Support$^{\#}_{R}$(|h|) += sign(h)
64             if (support changed to or from 0) {
65                 head ∪= {h}
66             }
67         } else { // aggregating: R($\bar{t}$,$\alpha$(v))
68             let R($\bar{t}$, v) = |h|
69             head ∪= {-R($\bar{t}$, $\alpha$(Support$^{MS}_{R}$($\bar{t}$)))}
70             if (sign(h) == -1) {
71                 Support$^{MS}_{R}$($\bar{t}$) -= {v}
72             } else {
73                 Support$^{MS}_{R}$($\bar{t}$) += {v}
74             }
75             head ∪= {+R($\bar{t}$, $\alpha$(Support$^{MS}_{R}$($\bar{t}$)))}
76         }
77     }
78     update stored relation contents by head
79     return head
80 }
```

Figure 4.5: DRed$_L$ monotonic phase.

the aggregation computation (blue highlighting in the code). We return to procedure `maintainIncrementally` by storing the re-derived tuples in *red*. Because these tuples, together with the previously deleted ones, can trigger transitive changes in downstream dependency components, we add *deleted* and *mon* to *all* (line 13). We perform a signed union, so ultimately if a tuple was deleted but then we could re-derive it, then that tuple will not change *all*. Note that re-derivation triggers insertions only and hence only entails monotonic changes that we handle in the final step of DRed$_L$.

Finally, DRed$_L$ runs the **monotonic phase** until a fixpoint is reached (lines 14-17). In each iteration, we use procedure `updateMon` (Figure 4.5) to compute the effect of the monotonic changes. Procedure `updateMon` is similar to `updateAnti`, but `updateMon` handles deletions as well as insertions due to increasing replacements. If `R` does not aggregate (lines 62-66), we update the support count of `h` according to the tuple being an insertion (`sign(h)=1`) or a deletion (`sign(h)=-1`). If instead `R` aggregates (lines 67-76), we delete the old aggregate result and insert the new one. To this end, we compute the aggregate result over the support multiset before and after the change to the support multiset. We collect all tuple deltas and return them to `maintainIncrementally`, which continues with the next fixpoint iteration. The implementation also checks if the produced tuple deltas may cancel each other out, which is quite common with idempotent aggregation functions such as `lub`.

Procedure `maintainIncrementally` executes the four steps of DRed$_L$ for each dependency component `C` until all of them are up-to-date. While it may seem that a change requires excessive work, in practice many changes have a sparse effect and only trigger relatively little subsequent changes. We evaluate the performance of DRed$_L$ in detail in

Section 4.8. One potential source of inefficiency in $\mathrm{DRed}_{\mathrm{L}}$ is computing the aggregation result over the support multiset (highlighted in blue in pseudo code). We eliminate this inefficiency through further incrementalization.

4.4.4 Incremental Aggregator Function

Procedures `updateAnti` and `updateMon` recompute the aggregate results based on support multisets. A straightforward implementation, that reapplies aggregation operator α on the multiset contents, will require $\mathcal{O}(N)$ steps to recompute the aggregate result from a multiset of N values. For example, a flow-insensitive interval analysis on a large subject program may write to the same variable N times. For large N, this can degrade incremental performance as computational effort will linearly depend on input size N, instead of the change size, which puts our Efficiency (R2) requirement in danger.

Given associative and commutative aggregation operators (like `glb` or `lub`), our idea is to incrementalize the aggregator functions themselves using the following approach. Independently from the partial order of the lattice we aggregate over, we take an arbitrary *total* order of the lattice values (e.g., the order of memory addresses). Using this order, we build a balanced search tree (e.g. AVL [108, Chapter 3.3]) from the aggregands. At each node, we store additionally the aggregate result of all aggregands at or below that node. The final aggregate result is available at the root node. Upon insertion or deletion, we proceed with the usual search tree manipulation. Then, we locally recompute the aggregate results of affected nodes and their ancestors in the tree. At each node along the path of length $\mathcal{O}(\log N)$ where N is the number of nodes in the tree, the re-computation consist of aggregating in $\mathcal{O}(1)$ time the locally stored aggregand with intermediate results. In sum, this way, we can incrementally update aggregate results in $\mathcal{O}(\log N)$ steps, while using $\mathcal{O}(N)$ memory.

4.5 Formal Semantics of $\mathrm{DRed}_{\mathrm{L}}$ and Correctness Proof

We provide a formal treatment of the theory behind $\mathrm{DRed}_{\mathrm{L}}$. Our approach relies on the formal semantics of recursive monotonic aggregation given by Ross and Sagiv [100]. We recap here the most important semantical aspects (originally introduced by Ross and Sagiv [100]), and then present a novel proof sketch for the correctness of $\mathrm{DRed}_{\mathrm{L}}$.

4.5.1 Semantics of Recursive Aggregation

A *database* or *interpretation* assigns actual instance relations to the relation names (*predicates*) appearing in the Datalog rules. A database is *cost consistent* if it satisfies the condition in Cost consistency (A 4.3), i.e. lattice columns are functionally determined by non-lattice columns in all relations. Let the set of variables that appear in the body (in any of the atoms) of a Datalog rule r be $Vars$. Then, given a concrete database, one can evaluate the body of a rule r to find all substitutions to $Vars$ that satisfy all subgoals in the body.

A *model* is a cost-consistent database that satisfies all Datalog rules, i.e. any given IDB relation is perfectly reproduced by collecting or aggregating the derivations of all those rules that have this relation in the head. A *Datalog semantics* assigns a unique model to a set of Datalog rules (where, technically, EDB relations are also encoded as rules that only use constants). In the following, we present the semantics of Ross and Sagiv by induction on dependency components, i.e. when considering the relations defined by a given component, we assume that we already know the semantics of all other components it depends on, so they can be equivalently substituted by constants and considered as EDB relations.

For such a single dependency component, by Monotonic recursion (A 4.1), we have a partial order $\sqsubseteq$ for each lattice used. We can thus lift this notation to define a partial order (shown to be a lattice as well) on databases; we say that $D_1 \sqsubseteq D_2$ iff for each tuple $t_1 \in D_1$, there is a tuple $t_2 \in D_2$ in the corresponding relation such that $t_1 \sqsubseteq t_2$. For cost-consistent databases, this is equivalent to saying D_2 can be reached from D_1 by tuple insertions and $\sqsubseteq$-increasing replacements.

It has been shown [100] that if a set of Datalog rules satisfy Monotonic recursion (A 4.1) and Cost consistency (A 4.3), then there is a unique *minimal model* M_{min}, i.e. all models M have $M_{min} \sqsubseteq M$. Thus the minimal model is considered as the semantics of the Datalog rules with recursive aggregation.

4.5.2 Correctness of the Algorithm - Proof Sketch

We give an informal sketch to prove the proposed DRed$_\text{L}$ algorithm correct (Correctness (R1)), given the assumptions from Section 4.4.1.[1] We do not include proofs for the correctness of well-known techniques used, such as algebraic differencing [46] or semi-naïve evaluation [44].

Specifically, we will show that DRed$_\text{L}$ terminates and produces the minimal model, while keeping its support data structures consistent, as well. As usual, we will conduct the proof using (i) induction by update history, i.e. we assume that DRed$_\text{L}$ correctly computed the results before given input changes were applied, and now it merely has to maintain its output and support data structures in face of the change; as well as (ii) componentwise induction, i.e. we assume that the result for all lower dependency components have already been correctly computed (incrementally maintained), and we consider them EDB relations.

Preliminaries Let B^{old} be the EDB relations before a changeset Δ, inducing minimal model M^{old}_{min}, which DRed$_\text{L}$ correctly computed by the induction hypothesis; let $B = B^{old} \cup \Delta$ be the new input and M_{min} the new minimal model, which DRed$_\text{L}$ shall compute. Let $D^{\cap}$ be the $\sqsubseteq$-greatest lower bound of the two minimal models (exists since databases themselves form a lattice [100]); essentially it is the effect of applying the anti-monotonic changes only.

Let we denote by $D_{init} = D^0_{anti}$ the current database (state of EDB and IDB relations) when DRed$_\text{L}$ starts to process the dependency component (with changes in EDB relations already applied), by D^i_{anti} after iteration i of the anti-monotonic phase, by D^{final}_{anti} at the end of the anti-monotonic phase, by $D_{red} = D^0_{mon}$ after the immediate re-derivations, by D^i_{mon} after iteration i of the monotonic phase, and finally by D^{final}_{mon} after the monotonic phase.

[1]To prove general fixpoint convergence, Ross and Sagiv required each lattice to be a *complete lattice*. We have dropped this assumption in favor of No infinite ascending chains (A 4.4), which will also suffice for our goals. Note that we require the finite height of *ascending* chains only, thus strictly speaking our assumption is neither stronger nor weaker than the original one.

Monotonicity Iterations of the anti-monotonic phase only delete tuples, so $D_{init} \sqsupseteq D^i_{anti} \sqsupseteq D^{final}_{anti}$. Re-derivations are always insertions, and each re-derived tuple t_{red} compensates for a tuple $t_{anti} \sqsupseteq t_{red}$ ($t_{anti} = t_{red}$ for non-aggregating relations) that was deleted during the anti-monotonic phase; therefore we also have $D^{final}_{anti} \sqsubseteq D_{red} \sqsubseteq D_{init}$. Finally, by induction on i and by Monotonic recursion (A 4.1), each iteration i of the monotonic phase performs monotonic changes, thus $D_{red} \sqsubseteq D^i_{mon} \sqsubseteq D^{final}_{mon}$.

Termination The anti-monotonic phase must terminate in a finite number of steps as there are finite number of tuples to delete. The immediate re-derivation affects at most as many tuples as the anti-monotonic phase, thus it terminates. The monotonic phase must reach its convergence limit in a finite number of steps, as well, for the following reasons. Given a finite database of EDB relations, a finite amount of non-lattice values are available. Due to Cost consistency (A 4.3), it follows that the output of the component is also finite. As the monotonic phase never deletes tuples, it can only perform a finite number of insertions to reach this finite output size. Finally, it may only perform a finite number of $\sqsubseteq$-increasing replacements due to No infinite ascending chains (A 4.4).

Upper bound and support consistency We now show that after the anti-monotonic phase, intermediate database states are upper bounded by the desired output, and that support data structures are consistent: A tuple t is (i) contained in the database or (ii) has a positive support count or nonempty support multiset only if $\exists t' \in M_{min}$ with $t \sqsubseteq t'$.

The anti-monotonic phase deleted all tuples that, in any way, depended transitively on the anti-monotonic part of input deletions, therefore those that remained must have had support not impacted by the deletions, thus $D^{final}_{anti} \sqsubseteq D^{\cap} \sqsubseteq M_{min}$. As rules are monotonic by Monotonic recursion (A 4.1), and the support data structures are correctly maintained (using algebraic differencing [46] by `immediateConsequences`), any tuple t that has support in state D^{final}_{anti} must have a t' that has support in (and thus contained in) M_{min}. This means that re-derivation only inserts tuples upper bounded by the minimal model, so $D_{red} \sqsubseteq M_{min}$.

A similar argument applies during the monotonic phase. Assume by induction that $D^{i-1}_{mon} \sqsubseteq M_{min}$. All tuples t that are to be inserted in iteration i due to having support (as computed by `immediateConsequences` and reflected in the support data structures by `updateMon`), must have (by Monotonic recursion (A 4.1) and the induction hypothesis) a corresponding $t' \in M_{min}$ with $t \sqsubseteq t'$, thus $D^i_{mon} \sqsubseteq M_{min}$ still holds. Eventually, $D^{final}_{mon} \sqsubseteq M_{min}$.

Cost consistency We will show that the output D^{final}_{mon} is cost consistent. EDB relations are considered cost consistent due to Cost consistency (A 4.3) and the assumed correctness of DRed$_L$ for lower dependency components. Therefore it is sufficient to check the cost consistency of IDB relations, which we do by indirect proof. By the induction hypothesis, the state of the algorithm before the change was the cost-consistent model M^{old}_{min}. The change in EDB relations did not directly affect IDB relations and the anti-monotonic phase only deleted tuples, so D^{final}_{anti} is cost-consistent as well. During the re-derive and monotonic phase, tuples are only inserted if they have support, so cost-consistency is only violated if tuples $t \neq t'$, agreeing on all non-lattice columns, both had support. Let us assume such a situation arises, and we prove that it can actually never happen.

Due to Cost consistency (A 4.3), the Datalog rules themselves are specified to obey cost consistency, so t and t' both can't be transitively derivable from EDB relations at the same time. Thus one of them, say t, must be an error in the support data structure, cyclically reinforcing itself. Such cyclic reinforcement can form for t if it was previously present, then one of its derivations is deleted (which is possible in the monotonic phase via an increasing replacement), but a positive support count still remained, so it was left in the database.

Because of the way how change splitting is performed, increasing replacements in the monotonic phase are only permitted for dependency components that use aggregation. By Aggregation exclusivity (A 4.2), alternative rules for the same relation must produce disjoint results, so all derivations of t must stem from the same rule, and may only differ in local variables of the rule. Therefore an increasing replacement that has removed one such derivation must have removed all of them, decreasing the support count of t to zero. This contradiction concludes the indirect proof.

Model The monotonic phase acts as a standard bottom-up Datalog evaluation using semi-naïve (differential) iterations [44]. Each iteration i takes the Datalog rules that can be applied to D_{mon}^{i-1} in order to derive new facts or increase aggregate results to obtain D_{mon}^{i}. If a rule can be applied to a database, its result will contribute to the computed immediate consequence due to the correctness of differential evaluation. At the final fixpoint, the immediate consequence is empty; this means that no more rules can be applied to D_{mon}^{final}, therefore the state thus reached (shown above to be cost-consistent) is a model.

Minimality By definition, $M_{min} \sqsubseteq M$ for any model M over the same EDB relations. However, we have shown above that $D_{mon}^{final} \sqsubseteq M_{min}$ which implies that $D_{mon}^{final} = M_{min}$, i.e. the algorithm is correct.

4.6 Integrating Lattices into the IncA Framework

We extend the baseline version of IncA presented in Chapter 3 with support for custom lattices and aggregation. Our extensions span four layers: IncA Datalog, compiler, runtime system, and IDE integration. We briefly review each of these extensions.

4.6.1 Extensions to IncA Datalog

We designed a DSL for the definition of lattices, and we extended IncA Datalog, so that analysis developers can use lattices in their analyses, together with custom aggregation operators. Figure 4.6 shows the syntax of the new IncA Datalog language, where we highlight our extensions compared to the version of IncA Datalog presented in Figure 3.5.

The new syntax enforces a clear separation between relational code defined by Datalog rules and lattice code defined by our Java-based DSL for implementing lattices. A lattice definition `l` declares the name of the lattice, algebraic data-type constructors for lattice values, and lattice operations. Each lattice operation has a name and is implemented in Java extended with calls to lattice constructors and operations. Each lattice must implement the lattice operations `leq`, `lub`, and `glb`, but other helper logic can also be implemented.

To use a lattice in Datalog rules, first the module of the lattice must be imported. Then, the type annotation of head variables can also use lattice types in addition to AST node types. The new term $\mathtt{L}_n\mathtt{.n}(\bar{t})$ allows to call lattice operations; e.g. constructors to create

(module)	m	$::=$ **module** n **import** $\overline{n}$ $\{\overline{mc}\}$
(module content)	mc	$::=$ l | r
(lattice)	l	$::=$ **lattice** $L_n\{$**constructors**$\{\overline{ctor}\}\ \overline{lop}\}$
(lattice name)	L_n	$::=$ name
(type in lattice definition)	T_{lat}	$::=$ T_{lang} | L_n | Java type
(constructor)	$ctor$	$::=$ $n(\overline{T_{lat}})$
(lattice op)	lop	$::=$ **def** $n(\overline{n : T_{lat}}) : \overline{T_{lat}} = lopb$
(lattice op body)	$lopb$	$::=$ Java code + lattice constructors and operations
(rule)	r	$::=$ $vis\ n(\overline{hv})$:- $\overline{alt}$
(head variable)	hv	$::=$ $n : T_{rel}$ | $lop(n : L_n)$
(visibility)	vis	$::=$ **private** | **protected** | **public**
(alternative)	alt	$::=$ $\{\overline{a}\}$
(atom)	a	$::=$ $n(\overline{n})$ | **not** $n(\overline{n})$ | vr | **not** vr | $n + (n, n)$ | $t == t$ | t != t | **count** $n(\overline{n})$ | **count** vr
(virtual EDB relation)	vr	$::=$ $T_{lang}.L_k(n, n)$|$T_{lang}(n)$
(term)	t	$::=$ n | c | $L_n.n(\overline{t})$
(constant)	c	$::=$ number | string | enum | boolean | AST node
(type in relational code)	T_{rel}	$::=$ T_{lang} | L_n
(type from subject language)	T_{lang}	$::=$ AST node type (from subject language)
(link)	L_k	$::=$ link of an AST node type (from subject language) | **parent** | **prev** | **next** | **index**
(name)	n	$::=$ name

Figure 4.6: Syntax of IncA Datalog with lattices. The highlighted parts show the additions in comparison to the prior version of the IncA Datalog from Figure 3.5.

lattice values or other helper operations defined on lattices. Finally, if a head variable has lattice type $\mathtt{L}_n$, then an analysis developer can also make use of any aggregation operator `lop` defined on $\mathtt{L}_n$ by wrapping the head variable with a call to `lop`. IncA only allows to use lattice operations with the right signature for aggregation: The operation must take two lattice values and produce one lattice value. However, IncA does not verify that the aggregator operator is indeed monotonic, ensuring this is the responsibility of the analysis developer. Note that in contrast to Section 4.3.1, there is no need to define separate rules that compute relations `Coll` and `Agg`. As we discuss later, those are automatically generated by the IncA compiler based on the type annotations on head variables. We demonstrate the use of the extended IncA Datalog through case studies in Section 4.7.

4.6.2 Runtime System

IncA still relies on the VIATRA QUERY library for the incremental evaluation. However, the library did not support incremental lattice-based aggregation before. To this end, we integrated $\mathrm{DRed}_{\mathrm{L}}$ and our optimization for aggregators based on AVL trees into VIATRA QUERY. We also introduced a new graph pattern constraint `Agg(p,c,`$\tau_{L.n}$`)`, where `p` is a graph pattern, c is a lattice-typed column of p, and $\tau_{L.n}$ is the AVL tree used for aggregating values in column c with aggregation operator *L.n*. We make use of this new pattern in the compiler next.

4.6.3 Compiler

Compilation to graph patterns We translate a lattice definition into a Java class, where lattice constructors become class constructors and lattice operations become static methods. We allow relations to use instances of these classes as values. For each aggregation operation `L.n` used in the IncA program under compilation, we generate an AVL tree implementation $\tau_{L.n}$ specialized for `L.n`, as described in Section 4.4.4.

We use the above aggregating graph pattern `Agg(p,c,`$\tau_{L.n}$`)` for translating IncA Datalog rules that yield an aggregated lattice value. Specifically, for every aggregating rule `r`, we generate two graph patterns: An auxiliary pattern $\mathtt{r}'_p$ that is the result of compiling `r` without any aggregation annotation (cf. relation `Coll` in Section 4.3.1), as well as the main pattern $\mathtt{r}_p$ that aggregates the tuples of $\mathtt{r}'_p$ using our new `Agg` graph pattern (cf. relation `Agg` in Section 4.3.1). This translation naturally extends to more than one aggregating column.

The compilation of the terms $\mathtt{L}_n\mathtt{.n}(\bar{t})$ require special runtime support, as these get translated to ordinary Java method calls on the class generated for the respective lattice. Viatra Query supports a special graph pattern that can yield the result of Java expressions, which we use during compilation.

Checks on IncA analyses The IncA compiler also performs static analyses on IncA subject analyses to ensure that the assumptions from Section 4.4.1 hold. These analyses include (i) checking the consistent usage of aggregations and (ii) enforcing stratifiable negation as per Monotonic recursion (A 4.1), and (iii) Aggregation exclusivity (A 4.2). These analyses are all implemented in IncA itself, and they all rely on analyzing the strongly connected components in the call graph of the rules in a subject analysis. The results of the analyses are used by the compiler, which reports errors to the analysis developer. Currently, there is no check verifying that a subject analysis satisfies Cost consistency (A 4.3), but we implemented a transformation (in Java) in the IncA compiler that can automatically rewrite subject analyses in certain cases to satisfy cost consistency.

Transformation for ensuring Cost consistency (A 4.3) The transformation assumes that within a single rule body, the lattice-typed variables are already uniquely determined by the non-lattice typed variables (local or not). This can typically be verified by the functional dependency solver already implemented in Viatra Query. If the assumption is satisfied for all bodies of the aggregating rule, the transformation runs in two steps. First, in order to prevent multiple derivations of the same aggregation group within the same rule body, we lift up all local variables from the rule body to the head. This way, we can ensure that the lattice-typed head variables are uniquely determined by the non-lattice typed head variables (since we have just made all variables head variables). Second, for rules with multiple rule bodies, we introduce a new intermediate rule per body, and we discriminate them with a unique (integer) constant value in the rule head. Each such intermediate rule performs an intermediate aggregation over the “lifted” local variables, so its head will match that of the original aggregating rule, plus the unique discriminator. The final transformed rule will then aggregate the union of these intermediate aggregates.

It is important to point out that this transformation preserves the semantics of the original IncA Datalog rules only because IncA analyses exclusively use lattice aggregators derived from idempotent, associative, and commutative binary operations (`glb`, `lub`). In contrast, for a generic aggregating rule set (e.g. with summation), lifting would indeed change the semantics (as some parts would be included in the sum multiple times). We emphasise that the transformation is specific to IncA, not the underlying DRed_L algorithm, which assumes that the input Datalog rules satisfy Cost consistency (A 4.3).

4.6.4 IDE Integration

We kept the integration with MPS for the new version of IncA, including all the typical IDE features, such as syntax highlighting, type system, generators, and validations. Technically, the new language extension for defining lattices is a DSL in MPS, and we reused MPS' Java language for the implementation of the bodies of lattice operations.

4.7 Case Studies

To validate our approach, we have used IncA to implement two lattice-based program analyses for Jimple, which is an intermediate program representation for Java. Our first analysis is a lattice-based points-to analysis, while the second one is a collection of analyses statically approximating the runtime values of string-typed variables. Both of these analyses use lattice-based aggregation. This section describes the program analyses and provides details about their implementation, while Section 4.8 presents the performance evaluation.

4.7.1 Jimple as Subject Language

The subject language for our evaluation is Jimple, an intermediate Java representation defined in the Soot framework [19]. As we discuss in Section 4.8, we analyze Java subject programs with our analyses. To resolve the gap between the two languages, we use the Java (bytecode) to Jimple transformer from Soot, which produces functionally equivalent Jimple code from Java code.

We chose Jimple as subject language because it only uses `if`s and `goto`s to express control flow, which relieves us from reasoning about more complex control flow. This is beneficial because it helps with the construction of the CFG, which we need later, as our points-to analysis is flow-sensitive. Constructing the CFG for a general purpose language like Java is a complex task, as there are a large number of control constructs that would need to be considered, plus there is also dynamic dispatch. We already experimented with implementing control flow analysis for C in Section 3.6, and to deal with the complexity of the control flow analysis, we also needed to introduce simplifications, such as ignoring function pointers and pointer arithmetics. Jimple helps us to deal with the complexity of the control flow analysis for Java, but it does not simplify the follow-up analyses themselves (like points-to), as the Jimple code produced by the Soot transformer is functionally equivalent to the original Java subject programs. Addressing control flow analysis for Java itself is an interesting research challenge, but it was beyond the scope of this dissertation.

4.7.2 Strong-update Points-to Analysis

Our first analysis adapts the strong update points-to analysis for C introduced by Lhoták and Chung [73] to Jimple. We chose this analysis because of its practical relevance; Points-to analyses are the basis of many other analyses. The strong update points-to analysis is at the sweet spot between the cheap flow-insensitive and the precise flow-sensitive analysis: It is flow-sensitive for a variable or field (given that we analyze Jimple) as long as the target points-to set consists of a singleton object, and then gives up on flow-sensitivity immediately when the target set has more than one object. At this point, it turns to a flow-insensitive analysis to answer points-to questions. This makes sense because dereferencing a non-singleton points-to set does not allow strong updates (that would overwrite the previous results) anyway, and so in most of the cases the precision gained through flow-sensitivity is limited. In addition to the mixed flow-sensitivity, our analysis is field-sensitive, as it reasons about the points-to sets of fields in objects. The analysis is intra-procedural, as it does not analyze across function calls. The original paper [73] provides details on the corner cases and precision characteristics. Here, we focus on the implementation in IncA.

First, we start with the flow-insensitive part of the points-to analysis. We use the following two rules to compute the points-to sets of variables and object fields, respectively:

```
VarPT(var : Var, obj : Obj)
FieldPT(base : Obj, field : Field, obj : Obj)
```

Here, the types `Var`, `Field`, and `Obj` all appear in the Jimple subject language, that is, the flow-insensitive analysis does not use lattices or aggregation. The rule bodies (omitted here) collect assignments and interpret the left and right hand sides to collect the pointer variables and the pointed objects. The rules recursively depend on each other.

Next, we define the rules for the flow-sensitive part of the points-to analysis as follows:

```
VarPTBefore(stmt : Stmt, var : Var, lub(obj : SObj))
VarPTAfter(stmt : Stmt, var : Var, lub(obj : SObj))
FieldPTBefore(stmt : Stmt, base : Obj, field : Field, lub(obj : SObj))
FieldPTAfter(stmt : Stmt, base : Obj, field : Field, lub(obj : SObj))
```

These rules use the singleton-set lattice `SObj` to decide when to give up on tracking the points-to set of a variable. An excerpt of the lattice definition showing the constructors, less or equals operation, and least upper bound operation is as follows:

```
lattice SObj {
    constructors { Bot | S(Obj) | Top }

    def leq(l : SObj, r : SObj) : bool =
        return match (l, r) with {
            case (S(o1), S(o2)) => o1 == o2
            case ...
        }

    def lub(l : SObj, r : SObj) : SObj =
        return match (l, r) with {
            case (S(o1), S(o2)) => o1 == o2 ? l : Top
            case ...
        }
}
```

The rules for the flow-sensitive analysis follow the same approach as the interval analysis in Section 4.2 in that they recursively depend on each other to propagate the points-to information along the subject program's CFG. The implementation builds on a control flow analysis for Jimple. The rules all aggregate with `lub` on the lattice `SObj`, making sure that the analysis gives up on tracking non-singleton sets (see first `case` in the body of `lub`).

The final ingredient of the strong update analysis is responsible for combining the results of the previous two:

```
VarPT_SU(stmt : Stmt, var : Var, lub(obj : PSObj)) :- {
  VarPTAfter(stmt, var, trg),
  trg != SObj.Top,
  obj == PSObj.fromSObj(trg)
} alt {
  VarPTAfter(stmt, var, topTrg),
  topTrg == SObj.Top,
  VarPT(var, trg),
  obj == PSObj.fromObj(trg)
}

FieldPT_SU(stmt:Stmt, base:Obj, field:Field, lub(obj:PSObj)) :- {
  ...
}
```

The analysis returns the result of the flow-sensitive analysis in the first alternative if that returns a non-Top value. The second alternative turns to the flow-insensitive analysis. Additionally, these two functions aggregate the points-to targets into a single set (a value in the powerset lattice `PSObj`). The reason for this is that the flow-sensitive analysis gives up on tracking non-singleton targets to save memory, so we must not store separate tuples per target objects here either, otherwise we would lose the memory saving. To query the points-to information, clients would interact with rules `VarPT_SU` and `FieldPT_SU`.

4.7.3 String Analyses

Our second case study concerns string values. Costantini et al. proposed several abstract domains to keep track of the properties of string values [29]. We implemented with IncA three distinct string analyses. First, the character inclusion analysis that keeps track of the definitely-contained and maybe-contained set of characters for a string variable. This, in turn, can be used to decide whether the return value of operations like `indexOf(s)` can be negative or not, which is interesting because a negative value causes operations like `split` or `substring` to throw an exception. We also implemented analyses that compute the longest common prefix and suffix of the strings a variable take; this is useful to check well-formedness, for example, to ensure that an SQL statement always starts with a keyword and terminates with a semicolon. Compared to the points-to analysis, the string analyses are *flow-insensitive*.

The interesting part about these analyses is the implementation of the lattices, as they are more involved than the singleton set lattice used by the points-to analysis. For example, an excerpt of the prefix lattice implementation is as follows:

```
lattice Prefix {
    constructors { Bot | Pre(string) }

    def top() : Prefix = Pre("")

    def leq(l : Prefix, r : Prefix) : boolean =
        return match (l,r) with
            case (Pre(p1),Pre(p2)) => p1.startsWith(p2)
    def lub(l : Prefix, r : Prefix) : Prefix =
        return match (l, r) with
            case (Pre(p1), Pre(p2)) => Pre(lcp(p1, p2))
    def interpretSubstring(v : Prefix, s : int, e : int) : Prefix =
        return match v with
            case Pre(p) => {
                final int l = p.length();
                if (s <= e && e <= l) {
                    return Pre(p.substring(s, e));
                } else if (s <= e && s < l && l < e) {
                    return Pre(p.substring(s, l));
                } else {
                    return top();
                }
            }
    ...
}
```

The `Prefix` lattice [2] has two kinds of elements: `Bot`, the bottom of the lattice, is never used by the analysis, it marks a “failure”. The `Pre` element wraps a prefix string. The `leq` operator performs a prefix-check using Java’s `String.startsWith` method, while `lub` uses a helper function, `lcp`, that computes the longest common prefix of two strings. The pattern matching syntax allows us to encode the lattice structure in just a few lines of code.

The `Prefix` lattice defines a series of functions that abstractly interpret the different kinds of string manipulation operations in Java. For example, function `interpret-Substring` interprets the result of `substring` calls on strings, in the `Prefix` abstract domain. It performs various checks using the start and end indices and returns a slice of the prefix accordingly, or resorts to the lattice top in case of an empty result.

4.8 Performance Evaluation

This section presents the performance evaluation of IncA using the case studies from Section 4.7. We answer the following questions:

Run Time (Q 4.1): Is the update time of IncA fast enough for interactive applications in IDEs, and does this come with an acceptable initialization time?

[2]This lattice is an interesting example, as it has infinite descending chains (any prefix can be continued to an even longer prefix), but no ascending ones (any given prefix has a finite number of truncations), hence No infinite ascending chains (A 4.4) is still satisfied.

Memory (Q 4.2): Is the extra memory requirement induced by incrementalization acceptable?

4.8.1 Evaluation Setup

We analyze the following Java code bases: (i) Google Truth, an assertion framework (9 KLOC Java code), (ii) Google Gson, a JSON serialization library (14 KLOC), (iii) PGSQL JDBC, a PostgreSQL-Java binding (45 KLOC), and (iv) BerkeleyDB, an embedded database (70 KLOC). We selected these because they represent widely used real-world applications. We imported the code into MPS and transformed it to Jimple, which is functionally equivalent to the original Java programs.

The actual evaluation is as follows. For each subject program, we start the analysis with an initial, non-incremental run. We then introduce 1,000 incremental program changes on the Jimple code, simulating the edits of a developer. We use two kinds of changes. First, we randomly perform generic changes such as copying and deleting expressions, statements, and methods, or renaming variables. Second, we perform random changes tailored to each analysis: We change assignments for the points-to analysis and modify string values for the string analysis. In all cases, we measure the wall clock time of the initial run and the run time of the incremental updates to answer Run Time (Q 4.1). To measure memory consumption for Memory (Q 4.2), we subtract MPS' total memory use before the initialization of the analyses from the value after the analyses are initialized. For the points-to analysis, we vary the subject code base when running the measurements. For the string analyses, we also vary the number of tracked string properties to find out how much more expensive it is to track not only one string property but two or three at the same time. We do this in order to see if a developer can simply turn on and off tracked properties to control the amount of information the analysis provides without incurring excessive performance overhead.

To validate that our analyses compute correct results, we compared the results of the incremental analysis with the results of a full re-analysis after every code manipulation, and we verified that they are the same. We ran the benchmarks on an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 10.12.6, Java 1.8.0_121 and MPS version 2017.3.5. We now show the results of our performance evaluation.

4.8.2 Evaluating Run Time (Q 4.1)

We start the discussion with the points-to analysis. The *initialization times* of the analysis are as follows: Google Truth - 6.5s, Google Gson - 9.4s, PostgreSQL JDBC - 57.8s, and BerkeleyDB - 64.3s. In contrast to these numbers, loading a larger project in MPS easily takes a few tens of seconds, so even the initialization on BerkeleyDB is not prohibitively long. We compare the numbers to the MPS project loading time because it is better to initialize the analyses during start-up, otherwise developers would observe a longer pause when an analysis is queried first. Figure 4.7 A captures the *incremental update times* of IncA for the four subject programs in a box plot (outliers removed). These times are very fast, exactly the kind of numbers we want in interactive applications. Figure 4.7 B reinforces this observation because it shows the scaling of the system on BerkeleyDB in terms of

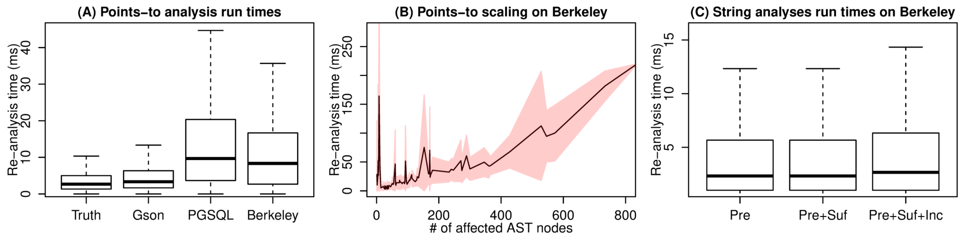


Figure 4.7: Run time measurement results.

update time wrt. the size of the program change. We had multiple data points per change sizes: The line connects the mean values, and we obtained the red area by subtracting and adding the standard deviation to the mean. We can also see on this graph that the update times are fast, even for larger changes. For example, changes affecting ~800 AST nodes were typically the duplication or removal of complete methods.

Let us now look at the results for the string analyses. We use the notation `Pre` to refer to the string analysis that keeps track of the longest common prefix, `Pre+Suf` tracks the suffix in addition, while `Pre+Suf+Inc` also tracks character inclusion. The initialization times for BerkeleyDB are as follows: `Pre` - 13.5s, `Pre+Suf` - 13.8s, and `Pre+Suf+Inc` - 20.4s. Figure 4.7 C shows the incremental update times on BerkeleyDB. Note that the string analyses are flow-insensitive, compared to the flow-sensitive points-to analysis, and this also shows on the run time because the string analyses are cheaper to compute, even incrementally. We also observe that (i) these run times meet the requirements of interactive applications, (ii) activating one (`Suf`) or two more string properties (`Suf+Inc`) has only very minor impact on incremental performance.

Run Time (Q 4.1): IncA achieves our goals for run time performance. It requires an acceptable amount of initialization time even on our largest subject program, and its update times are a few milliseconds on average.

4.8.3 Evaluating Memory (Q 4.2)

The memory consumption of IncA for the points-to analysis is as follows: Google Truth - 670MB, Google Gson - 1GB, PostgreSQL JDBC - 4.5GB, and BerkeleyDB - 5GB (these numbers essentially remain constant as the program changes). To put these numbers into perspective: (i) At the time of measurements MPS used around 2GB of memory, so the analysis on BerkeleyDB uses 2.5X of that, and (ii) the points-to analysis is a flow-sensitive analysis, and the Jimple representation of BerkeleyDB has a few hundred thousand CFG nodes. To provide a ballpark for the number of tuples we note that the relations of the points-to analysis on the BerkeleyDB together contain roughly 57 million tuples. These are high numbers, and they are because of extensive caching, but, in return, we have fast incremental updates.

The memory requirement of the string analyses on BerkeleyDB are as follows: `Pre` - 1.6GB, `Pre+Suf` - 1.7GB, `Pre+Suf+Inc` - 2.2GB. The overall lower memory use compared to the points-to analysis is because of flow-insensitivity. An interesting observation is that maintaining the longest common suffix in addition to the prefix requires only a 100MB extra memory, but activating the character inclusion induces an extra 500 MB compared to `Pre-Suf`. This is because the character inclusion lattice wraps two sets of characters, which is expensive in terms of memory for the large code base.

Memory (Q 4.2): We pay the price for the fast update times with memory. The memory requirement of IncA can grow large, but it is not prohibitive. In practice, developers would typically work with smaller parts of projects, reducing the absolute amount of memory needed. In addition, we have concrete future plans for reducing the memory requirement (see Section 9.2).

4.8.4 Discussion

On Jimple as the subject language While choosing Jimple as the subject language of our analyses made the construction of the CFG simpler, it also made our incremental analysis pipeline (as depicted in Figure 1.3), in fact, *non-incremental*. This is because producing the Jimple AST from textual Java code requires several components that are non-incremental. First, we produce Java bytecode from the textual source code of our subject programs with the `javac` compiler. This is a non-incremental process. Then, we use the Soot Jimple transformer to create the Jimple (in-memory) AST from Java bytecode. This is yet another non-incremental component. Finally, we use an AST-to-AST transformation to turn a Soot Jimple AST into an MPS Jimple AST. This is again a non-incremental transformation. Having all these non-incremental components was the reason why we programmatically introduced program changes in the MPS Jimple AST throughout our performance evaluation. We acknowledge that this is not an ideal setup in terms of aiming for an entirely incremental pipeline. However, for the purposes of our evaluation, specifically to show the incremental performance of IncA and to measure the memory overhead, this setup was sufficient already. Additionally, there is a long line of research work required to efficiently incrementalize all of the above mentioned non-incremental components, and this was beyond the scope of this dissertation.

On inter-procedurality We have benchmarked IncA on intra-procedural analyses here because these are relatively easy to express while also requiring significant computation time as the size of procedures grow. The goal of incrementalization is to make the computational effort for (re-)analysis proportional to the size of the change, not to the size of the entire subject program. As we demonstrated above, we have achieved this goal with IncA. However, as we will demonstrate in the next chapter, the performance of IncA is still not sufficient to efficiently support inter-procedural analyses. They could already be expressed with the IncA Datalog language, but the back end is simply not efficient enough to deliver the update times that applications in IDEs need.

4.9 Chapter Summary

In this chapter, we added support for custom lattices and recursive aggregation in IncA. This required extensions in every component of the IncA architecture. We first developed the DRed_L algorithm that incrementalizes recursive aggregation over lattices in the back end. Then, we extended IncA Datalog with a DSL to define lattices and with new language constructs that allow to use lattices in rules. We made our compiler lattice-aware, and we introduced a new aggregating graph pattern in Viatra Query. We kept the integration with MPS and implemented full IDE support for lattice-based analyses. Our evaluation on real-world subject programs shows that IncA provides fast update times for incremental lattice-based analyses.

We revisit again our requirements from Section 1.4. We did a major step in this chapter by improving the Expressiveness (R3) of IncA, as custom lattices and aggregation are fundamental to static analyses. We provided a correctness proof for our DRed_L algorithm (Correctness (R1)). We still meet the requirements Declarativity (R5) and Genericity (R4), as the extended IncA Datalog is still a declarative language and also independent of a particular subject language or analysis. Regarding Efficiency (R2), IncA clearly delivers the performance interactive applications in IDEs need, as it delivers millisecond update times for lattice-based analyses. However, while the current back end efficiently supported the presented benchmark analyses, later (wrt. to the timeline of IncA development), it became clear to us that it cannot efficiently support inter-procedural analyses. We work out a solution to this problem in the next chapter.

5 Incrementalizing Lattice-based Inter-procedural Analyses

This chapter shares material with a paper that is under submission at PLDI'21.

Abstract — Inter-procedurality is key in improving the precision of static analyses, as it allows to reason across function calls instead of analyzing functions in isolation. However, efficient incrementalization of inter-procedural lattice-based analyses is particularly challenging for two reasons. First, in an inter-procedural setting, even a small program change can have a large impact on the overall analysis result, thereby hindering the potential for reusing much of the previously computed results. Second, a recursively computed aggregate result may need to be completely unrolled after a program change, and the incurred computational overhead is exacerbated by inter-procedurality, as well.

In this chapter, we present an approach that can efficiently incrementalize lattice-based inter-procedural analyses. We use IncA Datalog for analysis specification, and we develop a novel incremental solver called LADDDER. Compared to prior approaches including DRed$_L$, LADDDER uses a non-standard aggregation semantics which allows to loosen monotonicity requirements on analyses and to improve the performance of lattice aggregators, enabling more efficient analysis implementations. In our evaluation, we also take a novel approach, as we quantify changes by their impact on the overall analysis result. We show that most of the time the impact is low, so there is potential for incremental reuse, but prior approaches cannot exploit this. In contrast, LADDDER can utilize the potential and deliver update times on the ballpark of milliseconds on average, with manageable memory overhead.

5.1 Introduction

Each previous chapter in this dissertation was a stepping stone for improving the expressive power of IncA. We built the baseline version of IncA with support for recursive analyses in Chapter 3. Then, we added support for lattices and recursive aggregation in Chapter 4 given that these are fundamental to static analyses. An important precision aspect of static analyses is whether they reason about subject programs *intra-procedurally* or *inter-procedurally*. We have not considered inter-procedural IncA analyses as case studies so far. This was not because of an expressivity limitation: As we will show in this chapter, IncA Datalog can express inter-procedural analyses already. The reason for this is related to the back end: Efficient incrementalization and inter-procedurality are at odds with each other.

The idea behind incrementalization is that small program changes typically cause small changes in the analysis result, so updating the previous result promises significant speedups over re-computing a completely new result. In reality though, it is difficult to deliver on this idea for lattice-based inter-procedural analyses. First, in case of *inter-procedural* analyses, even a small program change can have a large effect on the previous analysis result. We define the notion of *impact* to represent the size of the difference between the analysis results before and after a change. Clearly, high-impact changes can destroy the performance gains of incrementality. Second, lattice-based analyses often yield results that are computed over several fixpoint iterations due to recursive dependencies in the analysis. For example, this is the case when analyzing loops or recursive data structures. When a subject program changes, these fixpoint iterations may be completely unrolled and a new result may be re-computed with new fixpoint iterations. Such cyclic dependencies severely complicate efficient incrementalization, and inter-procedurality exacerbates this.

The incremental analysis community has long been looking to find the right balance between efficiently handling high-impact changes and expressiveness in terms of supporting custom lattices and aggregation. For instance, several frameworks can efficiently deal with inter-procedural analyses, but, in exchange, they limit the kinds of supported analyses, as is the case for e.g. incremental set-based [68, 93], compositional [24], or IFDS/IDE [9] analysis (see Section 8.1 for more details).

Datalog-based frameworks make the opposite trade-off. Plenty of examples demonstrate that Datalog is expressive for a wide range of analyses, as demonstrated by lattice-based analyses with IncA in Chapter 4, or other examples including points-to analyses [112], analysis of concurrent [123] or distributed systems [27]. However, state-of-the-art incremental Datalog solvers struggle to deliver good performance for inter-procedural analyses. For example, when we benchmarked IncA with an inter-procedural analysis, we discovered two major scalability problems affecting DRed$_L$ in the back end:

- DRed$_L$ frequently performs unnecessary re-computation after certain deletions, which is exacerbated by high-impact changes: We frequently measured incremental update times comparable to from-scratch times.
- DRed$_L$ imposes overly strict monotonicity requirements on lattice-based analyses, which requires inefficient encodings of important analysis patterns: We observed that such analyses do not scale inter-procedurally at all.

The limitations of DRed$_L$ and the other prior approaches show that efficient incrementalization of analyses with custom lattices and recursive aggregation in an inter-procedural setting is still a significant problem.

In this chapter, we prepare the back end of IncA to efficiently incrementalize lattice-based inter-procedural analyses. We reuse IncA Datalog (including lattices) for specifying analyses, but we provide a new incremental Datalog solver specifically for this problem. We call our incremental solver LADDDER, which is short for Lattice Aggregating Differential Dataflow-based Datalog EvaluatoR. LADDDER is based on *differential dataflow* (DDF)[82], a generic computational model for incrementalizing dataflow-based fixpoint computations. Prior work [51, 102] demonstrated the use of DDF as an incremental fixpoint algorithm for Datalog, but it is limited regarding aggregation: (i) lattice aggregation is inefficiently incrementalized, and (ii) just like DRed_L, the monotonicity requirements on analysis definitions are too strict to allow an efficient encoding of the analysis. LADDDER improves on earlier approaches with novel aggregation semantics that allows for (i) efficient incrementalization of aggregations, as well as (ii) termination and correctness guarantees that are valid under more relaxed monotonicity assumptions. What makes efficient incrementalization of aggregators challenging in DDF is that, while other incremental fixpoint algorithms [47, 88] coalesce results into a single “current” state, DDF maintains a partially ordered set of intermediate computation states. Incremental aggregators must thus collect aggregands from, and yield results at, multiple such states. We designed data structures that enable efficient incrementalization in LADDDER by exploiting the algebraic properties of lattice-based aggregations.

We answer two separate questions in our evaluation. First, whether the domain of lattice-based inter-procedural program analyses is at all amenable to efficient incrementalization. We are the first to formalize a notion of *incremental impact*: As noted above, high-impact changes are those where the old and the new analysis result differ significantly, which entails slower update times. Efficient incrementalization is only possible if high-impact changes occur infrequently. The second empirical question is whether LADDDER can deliver efficient incremental updates in this domain. Both experiments measured the effects of a series of synthetic code changes on an inter-procedural lattice-based points-to analysis with code from the Qualitas Corpus [128]. We observed that the vast majority of the random program changes have low impact. This means that there is potential for incremental reuse of previous results. However, as we will show, prior approaches cannot exploit this. In contrast, LADDDER manages to handle more than 99% of such changes in sub-second time, which is exactly the ballpark that applications in IDEs need. We also show that LADDDER significantly outperforms DRed_L. The results confirm that IncA equipped with LADDDER in the back end provides a practical solution for incrementalizing inter-procedural lattice-based analyses.

Contributions In summary, we make the following contributions:

- We show why existing Datalog-based incremental fixpoint algorithms are not fit to support inter-procedural lattice-based analyses (Section 5.2).
- We introduce LADDDER, a novel technique based on DDF to evaluate Datalog with lattice-based recursive aggregation (Section 5.3).
- We design optimizations for LADDDER that exploit the algebraic properties of lattices to improve aggregation performance (Section 5.4).

- We formally define the semantics of LADDDER and provide correctness proofs (Section 5.5).
- We evaluate the performance of our approach with a lattice-based inter-procedural points-to analysis on real-world subject programs (Section 5.6).

5.2 Prior Work and Problem Statement

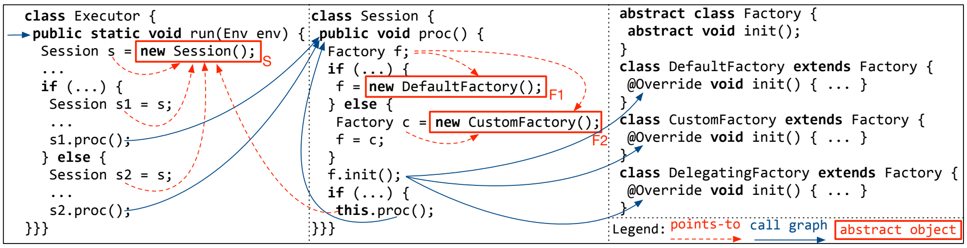
We introduce a running example to illustrate the challenges of incremental lattice-based inter-procedural analyses. We develop a lattice-based points-to analysis, which reports on the set of objects a variable can point to at runtime. We use IncA Datalog for the analysis specification. We review how DRed_L falls short on efficiently supporting our points-to analysis. We discuss other non-Datalog-based approaches in Section 8.1.

Running example: singleton points-to analysis We introduce a running example first, which we will use for illustration purposes throughout the chapter. Our running example is an inter-procedural singleton points-to analysis that distinguishes three kinds of points-to targets using a lattice: no target (bottom), a single target, or any target (top).[1] This approximation makes the analysis cheaper to compute because it limits the amount of information we keep track of precisely. Later in Section 5.6, we will consider a version of the analysis that keeps track of a set k number of points-to targets precisely. Points-to is the basis of many other analyses. For example, using this analysis, an IDE can perform approximate call graph construction (without precisely reasoning about call receivers with large points-to sets), or a compiler may perform inlining of virtual calls if the number of call targets is below a certain number.

Figure 5.1 illustrates how the analysis works on a subject program. The analysis starts in method `Executor.run`. It has two virtual calls to `proc`, using variables `s1` and `s2` as receivers. The singleton points-to analysis determines that `s1` and `s2` point to a single object `S` via variable `s`. Thus, we obtain precise call-graph edges for the calls of `proc` inside `run`, targeting `Session.proc()`. To derive the points-to targets of `this` within method `proc`, we join the points-to targets of the call receivers `s1` and `s2`. Since `s1` and `s2` point to the same object `S`, we conclude `this` points to the single target `S`, which allows us to resolve the recursive call of `proc` precisely. In contrast, variable `f` in `proc` can point to two different targets, which our singleton lattice approximates as `Top`. When resolving `f.init()`, we therefore must consider all `init` methods with compatible receiver types.

Points-to analysis in IncA Datalog We use IncA Datalog for the implementation of our running example. However, to simplify the presentation, we start off with a set-based points-to analysis and then rewrite the analysis later to use a lattice for controlling the approximation. Consider the following rules of a points-to analysis implementation in IncA Datalog motivated by a points-to analysis from the DOOP framework [112]:

[1]There is similarity between this analysis and the strong-update points-to analysis presented in Section 4.7, as both analyses use the singleton set lattice. However, the strong-update points-to analysis mixes flow-insensitivity and flow-sensitivity to find the right precision-performance trade-off, as it tracks points-to sets per CFG node as long as a set is singleton. The analysis presented in this section is different in this regard because it is an inter-procedural flow-insensitive analysis that never even tries to precisely track non-singleton points-to sets. The two analyses could be combined, but we do not consider that here.



```
class Executor {
 public static void run(Env env) {
   Session s = new Session();
   ...
   if (...) {
    Session s1 = s;
    ...
    s1.proc();
   } else {
    Session s2 = s;
    ...
    s2.proc();
 }}}
```

```
class Session {
 public void proc() {
  Factory f;
  if (...) {
   f = new DefaultFactory();
  } else {
   Factory c = new CustomFactory();
   f = c;
  }
  f.init();
  if (...) {
   this.proc();
 }}}
```

```
abstract class Factory {
 abstract void init();
}
class DefaultFactory extends Factory {
 @Override void init() { ... }
}
class CustomFactory extends Factory {
 @Override void init() { ... }
}
class DelegatingFactory extends Factory {
 @Override void init() { ... }
}
```

Figure 5.1: Example subject program used as input to the singleton points-to analysis.

```
PT(var, obj) :- {
  Reach(meth), Alloc(var, obj, meth)
} alt {
  Move(var, from), PT(from, obj)
} alt {
  Resolve(_, _, var, obj)
}

Resolve(invo, meth, this, obj) :- ...

Reach(meth) :- {
  Resolve(_, meth, _, _)
} alt {
  FuncName(meth, "run")
}
```

The PT rule computes tuples `(var, obj)` where `var` may point to `obj` at runtime. The first alternative considers heap objects allocated to variables, the second alternative transitively computes pointed objects through assignments between variables, while the third alternative computes the points-to targets of `this` variables. `Resolve` relates a virtual call `invo`, its target method `meth`, the `this` variable in the target method, and its points-to target(s) `obj`. We will show the rule body later. Finally, `Reach` enumerates reachable methods. First it collects all methods from `Resolve` that can be the target of a virtual call. Second, the `run` method (in Figure 5.1) is considered reachable by default. The other relations, such as `Alloc`, `Move`, or `FuncName`, appearing in the code are virtual EDB relations enumerating facts about the subject program.

The previous points-to analysis precisely keeps track of the points-to targets. In our example, we only track singleton points-to targets precisely, but approximate to `Top` otherwise using a singleton lattice. Such lattice-based computations cannot be efficiently encoded in standard Datalog operating only with sets of tuples. To this end, we reuse the lattice extensions of IncA Datalog introduced in Chapter 4.

Figure 5.2 shows the implementation of our singleton points-to analysis. The main difference compared to the analysis from before is that the points-to sets are now represented with a singleton lattice value. To compute the lattice value, we use the $\mathrm{PT}_{Collect}$ collecting rule and the PT aggregating rule. We use lattice aggregator `lub` (least upper bound) in PT: If a variable points-to different singleton sets, the aggregation result will be `Top`. This is where the analysis over-approximates. Note that the aggregation is *recursive*, as $\mathrm{PT}_{Collect}$ and PT recursively call each other through `Resolve`. This is important to point out, as this

```
PT_Collect(var, oSet) :- {
  Reach(meth), Alloc(var, obj, meth),
  oSet == SingletonSet(obj)
} alt {
  Move(var, from), PT(from, oSet)
} alt {
  Resolve(_, _, var, oSet)
}

PT(var, lub(oSet) ) :- {
  PT_Collect(var, oSet)
}

Resolve(invo, toMeth, this, oSet) :- {
  VCall(base, sig, invo, inMeth), Reach(inMeth),
  PT(base, oSet), oSet ⊏ Top ,
  obj == unwrapSingletonSet(oSet) , HeapType(obj, type),
  Lookup(type, sig, toMeth), ThisVarInFunc(toMeth, this)
} alt {
  VCall(base, sig, invo, inMeth), Reach(inMeth),
  PT(base, oSet), oSet ⊒ Top ,
  Lookup(_, sig, toMeth), ThisVarInFunc(toMeth, this)
}

Reach(meth) :- {
  Resolve(_, meth, _, _)
} alt {
  FuncName(meth, "run")
}
```

Figure 5.2: Excerpt of the singleton points-to analysis implementation in IncA Datalog. Lattice operations are highlighted with blue color.

will lead to cyclic reinforcement among aggregate results, which is known to complicate efficient incrementalization as already discussed in Section 4.2.

We now provide the implementation of `Resolve`, which must be lattice-aware. We use two alternative rule bodies. The first one handles the case when the points-to value of the receiver is a singleton set: `oSet ⊏ Top`. In this case, we look up the target method of the call with the sought signature on the type of the receiver object. The singleton points-to value also gets associated with the `this` variable in the method. The second alternative handles the `Top` case: `oSet ⊒ Top`.[2] As we do not have a specific type, we look up all methods with the right signature, and we also associate `Top` with the `this` variable. We pay the price for giving up the precise tracking of the target objects. Figure 5.1 illustrated this precision loss for the virtual call `f.init()`.

Given a set of Datalog rules and facts, a Datalog solver computes the *minimal* set (least fixpoint) of tuples that satisfy all rules. The solving process is an iterative fixpoint computation where rules are applied repeatedly until no new tuples can be inferred. To ensure *termination* and minimality, the rules must be *monotonic*. In standard Datalog, without custom lattices, this means that each fixpoint iteration can only ever produce new

[2]Note that we implement the comparisons ⊏ and ⊒ as lattice operations in the real analysis code. Here, we use graphical symbols for brevity.

tuples, never retract previously inferred ones. With lattices, the standard requirement is $\sqsubseteq$-monotonicity according to the semantics by Ross and Sagiv [100]. Throughout the fixpoint computation, $\sqsubseteq$-monotonic rules must either infer new tuples or increasingly replace lattice values (in tuples), which means that a lattice value is replaced by a larger one wrt. a chosen $\sqsubseteq$ order of the respective lattice. This was also the guiding principle when designing DRed_L in Section 4.4. Meeting this requirement will pose challenges during the incrementalization of our analysis.

Incremental singleton points-to analysis We consider now an incremental version of our singleton points-to analysis. Imagine a change in method `run` in Figure 5.1, where we replace the initializer of `s` by `s = new EncryptedSession()`. An incremental singleton points-to analysis must propagate the effect of this change to update the previously computed analysis result. However, since our analysis is inter-procedural, the effect can have a large impact and may affect information in multiple methods. In our example, we have to fix the points-to targets of `s`, `s1`, `s2`, and `this`, as well as the targets of all three `proc` calls.

Our goal was to use IncA with DRed_L for the incrementalization of our analysis. This is an interesting experiment because we did not benchmark inter-procedural analyses in previous chapters. We set out to measure the performance of IncA on the source of the minijavac compiler.[3] With 6.5 KLoC, the code base is relatively small, but it uses JRE library methods, so the transitively reachable code size is considerable. Our plan was to first perform a non-incremental analysis and then introduce program changes to measure the update times. We set a 10 minute timeout for the initial analysis, and IncA timed out. Further investigation revealed that DRed_L would not have been able to finish with any timeout, as the fixpoint computation could not terminate.

We investigated the problem more, and we found that we violated a requirement of DRed_L regarding Monotonic recursion (A 4.1). Specifically, the problem is with the `Resolve` rule in Figure 5.2. When the second points-to target of `var` is found, `oSet` switches from a singleton set to `Top`. This makes the second alternative body of `Resolve` applicable. But, this also means that all call targets of `invo` inferred in the first alternative now need to be retracted. However, removing these call targets may again invalidate one of the points-to targets of `var`, meaning the first alternative is actually the relevant one. DRed_L prohibits analysis formulations like this exactly because they lead to non-termination. We observe:

Observation 1: Violations of $\sqsubseteq$-monotonicity naturally occur in definitions of static analysis when giving special treatment to lattice values such as `Top`.

In an attempt to obtain a baseline performance estimate with DRed_L anyway, we rewrote the `Resolve` rule to satisfy $\sqsubseteq$-monotonicity. For ease of reference, we name the original formulation of `Resolve` (in Figure 5.2) **branching** and the new formulation **non-branching**. The **non-branching** formulation uses a single alternative to handle both singleton and `Top` points-to values:

```
Resolve(invo, toFunc, this, oSet) :- {
  VCall(base, sig, invo, inFunc), Reach(inFunc),
  PT(base, oSet), Heap(obj),
  SingletonSet(obj) ⊑ oSet , HeapType(obj, type),
  Lookup(type, sig, toFunc), ThisVarInFunc(toFunc, this)
```

[3] `https://github.com/mtache/minijavac`

```
}
```

The idea of **non-branching** is to enumerate *all* heap objects and filter the relevant ones with `SingletonSet(obj) ⊑ oSet`. If `oSet` is a singleton set then this boils down to an equality check between the contained object and `obj`, but, if `oSet` is `Top`, then the check always holds. With this formulation, we can ensure that the analysis never retracts previous inferences.

Again, we tried to measure the performance of IncA on minijavac, now with the **non-branching** `Resolve` formulation. Unfortunately, IncA again timed out after 10 minutes. Through performance profiling, we learned that our **non-branching** `Resolve` formulation computed tens of millions of intermediate tuples, which led to the time out. What happens is that the rule queries the points-to set of a variable `PT(base, oSet)`, joins the tuples with all heap objects `Heap(obj)`, and then filters the tuples with `SingletonSet(obj) ⊑ oSet`. The join between `PT` and `Heap` is intractable and must be avoided. Note that our **branching** formulation of `Resolve` did not need to compute this join because we provided special treatment for `Top`.

Observation 2: Encoding computations to satisfy ⊑-monotonicity can make them intractable.

As a last attempt to obtain a performance baseline, we reverted to the set-based points-to analysis from Doop (without singleton sets). This analysis trivially adheres to ⊑-monotonicity required by DRed_L since there is no lattice aggregation. This measurement is interesting because even in a lattice-aware computation, most operations are set-based. For example, Figure 5.2 is lattice-aware but contains only 5 lattice operations, while the rest is set-based joins, selections, and projections. We again ran IncA on minijavac, now using the Doop analysis. To measure the performance, we randomly deleted and re-added assignments to alter the points-to targets. We recorded the following running times:

Initialization:	35.24s	
Incremental insertion:	0.02s mean (±0.01s with 95 % confidence)	3.40s max
Incremental deletion:	9.14s mean (±0.57s with 95 % confidence)	21.75s max

The initialization time is a one-off cost, so we deem 35 s unproblematic. More interesting are the times required for incremental processing. When inserting a new assignment statement into the code, IncA yields the new results within 20 ms on average. This clearly is fast enough for generating continuous feedback in an IDE. However, deleting an assignment reveals poor incremental performance, requiring 9.14 s on average. We found that the performance problem is actually due to DRed, not even DRed_L. As discussed in Section 2.4, in response to a deletion, DRed invalidates *all* results that (transitively) depend on the deleted code, and then it re-derives results that are still valid after the deletion. This is to ensure correct updates in face of *cyclic dependencies* in the analysis result, however it makes incremental handling of deletions prohibitively slow. For intra-procedural analyses this problem was not apparent when benchmarking IncA in Section 4.8.

Observation 3: DRed appears unsuitable for incrementalizing inter-procedural analyses.

We are left in a situation where none of our attempts for an incremental singleton points-to analysis worked out. IncA could support the **non-branching** encoding, but the analysis is prohibitively expensive. The **branching** encoding would avoid the expensive join, but IncA (through DRed_L) requires $\sqsubseteq$-monotonicity. Even though, we considered only IncA here, the problem is actually more general because other frameworks that support lattice-based aggregation also face a similar restriction, as we discuss in Section 8.1. We also tried the set-based analysis, and we found that it is unlikely that any DRed-based framework can scale to inter-procedural analyses in general due to the excessive over-deletion of tuples. However, there is also a non-DRed-specific problem with the set-based analysis. By giving up the lattice abstraction, the analysis also precisely tracks the non-singleton points-to sets. We examined the results for minijavac, and we found that 30% of the variables have non-singleton points-to sets. This accrues considerable unnecessary work when we are only interested in singleton points-to sets. We revisit this observation in Section 5.6.6.

Problem statement Our goal in this chapter is to develop the first solution that can efficiently incrementalize inter-procedural analyses with recursive aggregation over lattices. Given that we improve the IncA framework as a whole in this chapter, all of our requirements from Section 1.4 are relevant. We provide additional details to Expressiveness (R3) and Efficiency (R2) though. Regarding Expressiveness (R3), this section demonstrated that IncA Datalog can already express inter-procedural analyses, even ones that are not $\sqsubseteq$-monotonic. This is desired; Our solution shall apply to all inter-procedural analyses with custom lattices. In particular, it should support the **branching** analysis implementation of the case study. Regarding Efficiency (R2), our solution shall provide the kinds of update times that interactive applications in IDEs need, even for inter-procedural analyses.

In the following, we develop a new incremental solver for IncA. Our solver not only supports recursive lattice-based aggregations but also imposes a weaker monotonicity requirement compared to prior approaches, including DRed_L, thus permitting the **branching** implementation from above. We prove correctness and termination guarantees for our new solver, and we empirically validate that it provides sub-second update times on average for inter-procedural lattice-based analyses.

5.3 Incremental Lattice-Based Program Analysis with LADDDER

While IncA Datalog with lattices is a good fit for a wide range of static analyses, Section 5.2 revealed two main obstacles to efficient incrementalization, especially in the face of high-impact changes: cyclic reinforcement among partial analysis results and overly strict monotonicity requirements on the analysis definition. Our approach to solve these challenges is based on a new incremental Datalog solver called LADDDER. We use a non-standard aggregation semantics in LADDDER, which makes it possible to loosen the strict monotonicity requirement enforced by existing incremental solvers including DRed_L, thereby allowing for more efficient analysis definitions (cf. **branching**). LADDDER builds on differential dataflow [82] (DDF), which is a generic computational model to incrementally maintain any iterative dataflow computation. Instead of coalescing the partial results obtained in each iteration of a fixpoint computation, DDF maintains an ordered set of computation states together with the partial results they compute. Such states are placed along multiple “time axes”: One axis stands for the epochs of changes in the subject program, and another one for the fixpoint iterations.

5.3.1 Initial Analysis with Laddder

We review how Laddder initializes the analysis result, and we discuss the incremental maintenance in Section 5.3.2. The input to Laddder is an analysis encoded as IncA Datalog rules plus facts encoding the subject program. Laddder repeatedly applies rules until a fixpoint is reached, thereby computing the tuples that the relations (defined by the rules) consist of. Laddder follows a semi-naïve evaluation strategy [44]: In each iteration of the fixpoint computation, Laddder only considers new tuples from the previous iteration instead of re-applying rules on the whole set of tuples computed thus far. Laddder breaks up the analysis into dependency components and applies rules according to a topological ordering of these components: Only after the fixpoint iterations finished in all *upstream* components, will Laddder start evaluating a *downstream* component.

Figure 5.3 shows the evaluation trace of Laddder on the subject program from Figure 5.1 focusing on the dependency component that consists of the rules defined in Figure 5.2. The trace goes from top to bottom along the fixpoint *iteration time axis*. An increasing *timestamp* (denoted by `T`) value is associated with every iteration, and the figure shows all tuples inferred at a specific timestamp. The fixpoint computation starts from the tuples produced by upstream dependency components, which, for our example, are all singleton components consisting of rules enumerating facts. These facts all appear at timestamp `0`. The iteration time axis in the figure is not a mere visualization aid to follow the fixpoint computation: DDF-style approaches [51, 89, 102], also including Laddder, actually remember which timestamp each tuple was derived at. This is in contrast to other incremental solvers (like DRed_L) that coalesce the sets of tuples into a single “current” state.

Remembering timestamps is key to efficient incrementalization in Laddder because they help to rule out cyclic reinforcement between partial results, as we will show in Section 5.3.2. For example, Laddder derives tuple `Reach(proc)` both at timestamp `7` and at timestamp `10`. The first derivation is due to the resolution of the `s1.proc()` and `s2.proc()` calls, while the second one is due to the recursive call in `proc` itself. The latter is a consequence of the former, therefore it is derived at a later iteration, i.e. with a higher timestamp. Generally, we must increment timestamps (i.e. postpone for the next iteration) at least once as we go around a dependency cycle among rules to unroll the recursion. We chose a simple way to achieve this: Each inferred head tuple gets a timestamp that is one higher than the highest timestamp of the tuples used in the rule body.

Laddder also maintains a support *count* for each tuple *per timestamp*: The count is equal to the number of alternative derivations a given tuple has at a specific timestamp. For example, `Reach(proc)` has count 2 at timestamp `7` (note the 2× symbol) because it can be derived in two alternative ways by using the two `Resolve` tuples from timestamp 6. Support counts play an important role in avoiding unnecessary re-computation during incremental maintenance because they tell Laddder if alternative derivations remain for a tuple after deletions. We demonstrate this in detail in Section 5.3.2.

Based on support counts, Laddder considers different forms of *timelines* for each tuple.

T	Tuples produced by T_i
0	*facts* (VCall, Move, Alloc, ...)
1	Reach(run)
2	$PT_{Collect}$(s, {S})
3	PT(s, {S})
4	$PT_{Collect}$(s1, {S}), $PT_{Collect}$(s2, {S})
5	PT(s1, {S}), PT(s2, {S})
6	Resolve(s1.proc(), proc, $this_{Session}$, {S}), Resolve(s2.proc(), proc, $this_{Session}$, {S})
7	2×$PT_{Collect}$($this_{Session}$, {S}), 2×Reach(proc)
8	PT($this_{Session}$, {S}), $PT_{Collect}$(f, {F1}), $PT_{Collect}$(c, {F2})
9	Resolve(this.proc(), proc, $this_{Session}$, {S}), PT(f, {F1}), PT(c, {F2})
10	$PT_{Collect}$(f, {F2}), $PT_{Collect}$($this_{Session}$, {S}), Reach(proc), Resolve(f.init(), $init_{Def.Factory}$, $this_{Def.Factory}$, {F1})
11	PT(f, Top), PT($this_{Session}$, {S}), Reach($init_{Def.Factory}$)
12	Resolve(f.init(), $init_{Def.Factory}$, $this_{Def.Factory}$, Top), Resolve(f.init(), $init_{Cus.Factory}$, $this_{Cus.Factory}$, Top), Resolve(f.init(), $init_{Del.Factory}$, $this_{Del.Factory}$, Top)
13	Reach($init_{Cus.Factory}$), Reach($init_{Del.Factory}$)

Figure 5.3: LADDDER evaluation trace for the singleton points-to analysis on the subject program from Figure 5.1. {O} represents a singleton set containing abstract object O.

Figure 5.4 shows the timelines of tuple `Reach(proc)`; for now, ignore the dashed lines in the figure. Cumulative count shows the total number of alternative derivations as a function of timestamps: There are 2 alternative derivations at timestamp 7, and one more at timestamp 10. From the cumulative count timeline, a cumulative existence timeline can be inferred: Its value is 1 if the tuple exists at a specific timestamp, 0 otherwise. Ultimately, the cumulative existence is the important information for the fixpoint computation: a Datalog rule can use a tuple only when it "exists". This is straightforward during the initial semi-naïve evaluation because consecutive iterations infer new tuples based on the tuples from previous iterations, but will gain significance in an incremental setting (see Section 5.3.2). To maintain the cumulative timelines during semi-naïve evaluation, LADDDER uses differential timelines that reflect the changes in the cumulative ones. The differential count signals the changes in support counts at specific timestamps, while the differential existence is either +1 or -1 to signal the appearance or disappearance of a tuple.

The aggregation semantics of LADDDER vastly differs from previous approaches, influencing how timelines are maintained. All existing DDF-based frameworks [51, 82, 102], as well as non-DDF approaches like $DRed_L$ or Flix [79], use an aggregation semantics that was introduced by Ross and Sagiv [100]. In this semantics, a change in an aggregate result

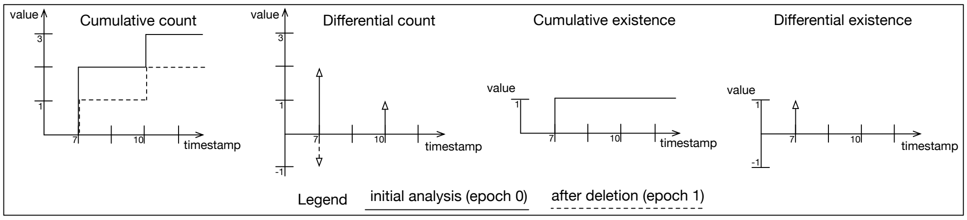


Figure 5.4: Timelines of `Reach(proc)` as maintained by LADDDER.

is represented with a deletion of the old and an insertion of the new result. In contrast, LADDDER uses a non-standard *inflationary* semantics [48]. This means that LADDDER never retracts old aggregate results, it only ever inflates the set of aggregate results with the newly computed ones along the iteration time axis. Formally, for an aggregation group $\overline{g}$ (set of grouping variables), timestamp t, and aggregation operator α, LADDDER computes the set of tuples $\{(\overline{g}, \alpha(M[t])), (\overline{g}, \alpha(M[t-1])), \ldots (\overline{g}, \alpha(M[0]))\}$ where $M[T]$ yields the multiset of aggregands at timestamp T. This is also visible in Figure 5.3 because LADDDER derives `PT(f, {F1})` at timestamp 9 (from $M[9] = \{F1\}$) and then `PT(f, Top)` at timestamp 11 (from $M[11] = \{F1, F2\}$), without retracting the former. As we detail in Section 5.3.3, this semantics allows us to loosen the monotonicity requirements on analysis definitions compared to the stricter requirements of existing solutions, which we demonstrated in Section 5.2. With inflationary semantics, once a tuple gets derived at a timestamp, it will exist at all subsequent timestamps along the iteration axis. In other words, the existential timelines can actually be represented by a single timestamp, which signals the first moment when a tuple appears during the fixpoint computation. For example, `PT(f, {F1})` has a validity interval `[9, ∞)` according to inflationary semantics, while it would have `[9, 11)` with other DDF-based approaches.

However, with inflationary semantics, the relations of the analysis will contain additional, intermediate, aggregate results. Typically, this is unwanted by downstream components or analysis clients. Similarly, timestamps do not carry useful information for downstream components; they are important inside the current component to unroll recursion. To this end, LADDDER performs two steps of postprocessing on the output of each component. First, it provides a *timeless* view of the tuples by simply not writing timestamps to the output. Second, LADDDER filters out intermediate results and only writes the final aggregate result to the output for each aggregation group. The final result is either the largest or smallest value according to the aggregation direction. Let us now take a look at how all this helps to incrementally update analysis results after input changes.

5.3.2 Incremental Analysis with LADDDER

Orthogonal to the iteration time axis, LADDDER uses a separate time axis to represent epochs when the input changes. Assume that LADDDER already computed and stored the results of a semi-naïve evaluation based on the input at epoch e_1 and a change happens at e_2, where $e_2 > e_1$. The goal of incrementalization is to correct the states of the previous evaluation based on the diff between the inputs at e_1 and e_2, so that they become as if the input at e_2 was evaluated from scratch. LADDDER uses the input diff to trigger a new fixpoint evaluation that infers which tuples need to be inserted or deleted at higher timestamps. Let us look at an example.

Assume that we associate epoch 0 with the initial code in Figure 5.1, and we delete `s2.proc()` at epoch 1. The input diff is the deletion of a virtual call fact, and LADDDER uses this to update (compensate) its state to epoch 1:

T	Tuples produced by T_i
0	-VCall(s2, proc, s2.proc(), run)
6	-Resolve(s2.proc(), proc, $\text{this}_{Session}$, {S})
7	$-\text{PT}_{Collect}$($\text{this}_{Session}$, {S}), -Reach(proc)

Concepts presented before, such as timestamps, support counts, and timelines, all come into play. The compensation starts with the diff appearing at timestamp 0 as a deletion, hence the minus (-) symbol. The deletion of this fact invalidates a `Resolve` tuple at timestamp 6, originally derived by the first `Resolve` rule in Figure 5.2. Among the tuples used in the rule body during the initial derivation, `PT(s2, {S})` was inferred latest at 5, hence the timestamp 6 for the head. Propagating this, LADDDER deletes *one* derivation each of both $\texttt{PT}_{Collect}$`(this`$_{Session}$`, {S})` and `Reach(proc)` at timestamp 7. The support count of both tuples gets decremented to 1 (c.f. Figure 5.3 at timestamp 7), but this means that an alternative derivation still remains for each tuple. There is no existential diff, so the compensating propagation terminates.

The example is indicative of a scenario where DRed_L would not stop in only three steps, but would rather over-delete and re-derive much of the previous result. The problem is that DRed_L cannot tell apart the different derivations of the `Reach(proc)` tuples, having coalesced them into a single state. Cyclic dependency is possible; a positive support count remaining after deleting a derivation of a tuple is insufficient evidence for its continued existence. This is easily demonstrated by deleting the `s1.proc()` call in `run`; then the only justification for `proc` being reachable is the recursive call in itself, but that recursive call would not be executed at all if there is no other function calling `proc`.

While incrementalizing simple relational algebra operations in DDF is a solved problem [82, 89], the efficient incrementalization of aggregation along two (epoch and iteration) time axes is challenging. In Section 5.4, we propose novel data structures that can efficiently maintain lattice aggregates in such a setting. But first, we review the assumptions of LADDDER on analysis definitions.

5.3.3 Monotonicity, Assumptions, and Guarantees of LADDDER

A novel aspect of our work is that LADDDER imposes looser monotonicity requirements on analysis definitions compared to prior approaches. We first discuss the restrictions in the state of the art by revisiting the IncA Datalog code in Figure 5.2.

Recall that for the subject program from Figure 5.1 the points-to value associated with `f` is updated from `{F1}` to `Top` in iteration 11 (c.f. Figure 5.3). The standard non-inflationary Ross and Sagiv semantics [100] would represent the change in the aggregate result as a deletion-insertion pair: `-PT(f, {F1})` and `+PT(f, Top)`. These tuples have the same grouping values, and the inserted lattice value dominates the deleted one (`{F1} ⊑ Top`). In Ross and Sagiv semantics, this is considered a ⊑-increasing change of the entire relation; termination and minimality of the fixpoint computation is guaranteed by the recursion itself being ⊑-monotonic (cf. Monotonic recursion (A 4.1)).

Existing approaches can only work with ⊑-monotonicity if it is guaranteed that when a deletion appears at *some* stage of the fixpoint computation, then the dominating insertion also appears at the *same* stage. This can exactly go wrong with the **branching** analysis encoding. The first alternative body of the `Resolve` rule in the **branching** pattern is especially problematic because it retracts previous inferences once a points-to set switches to `Top`. This was immediately a problem for the old IncA back end with DRed_L because the solver could not guarantee that the second alternative of `Resolve` would produce the dominating insertion at the same stage of the fixpoint computation. As we showed in Section 5.2, this resulted in a diverging fixpoint computation.[4] However, this problem also affects other DDF-based approaches, as it is crucial that the two parts of a ⊑-monotonic change always line up at the *same* timestamp, otherwise there is a potential for non-termination, as well. In practice, we found this hard to guarantee: Very careful control is required on behalf of both the solver and the analysis specifier regarding when timestamps are incremented. While the easiest option would be to increment timestamps after each rule application, this could easily break up the alignment between the two `Resolve` rules in the above example. If the ⊑-decreasing rule finishes evaluating the change at a lower timestamp than the rule deriving the dominating increase, then there is a timestamp range where the results are ⊑-decreasing. This violates the Ross and Sagiv semantics and can lead to non-termination.

Solving this non-termination challenge, if possible at all, requires either very clever preprocessing of the Datalog rules, or input from the analysis developer. The goal is finding the set of maintained relations where timestamps are increased in a way that (i) avoids the above mentioned non-⊑-monotonicity, and yet (ii) the select set of relations is a *cut* of the recursion, i.e. each dependency cycle goes through at least one timestamp increase, in order to avoid cyclic dependencies in the analysis result. Furthermore, such a clever choice of cut points will restrict the ability of the solver to internally optimize the rules, e.g. by extracting common sub-rules into auxiliary relations that are separately maintained. However, this is a frequently used query optimization strategy (cf. higher-order view maintenance [5]). To this end, LADDDER uses the following looser assumption, enabled by inflationary aggregation:

Eventual ⊑-monotonicity (A 5.1): Our more relaxed assumption also requires ⊑- monotonicity, but only for one cut, and only in an *eventual* sense. Each involved rule may derive the corresponding tuple at any timestamp independently of each other. The insertion of the ⊑-dominating tuple may be derived at an arbitrary timestamp, potentially later than the deletion it dominates.

Furthermore, the analysis developer does not need to inform the query engine of a cut, as in, the developer can just focus on defining the rules of the analysis. As long

[4]Interestingly, we only learned later on when researching LADDDER that our strong-update points-to analysis case study in Section 4.7 faced a similar problem, but there the formulation of the Datalog rules happened to ensure that such deletion-insertion pairs lined up during fixpoint computation.

as an eventually monotonic cut exists, the solver can choose any other cut and still get correct results. This allows larger freedom for the analysis developer, but also for the solver to perform optimizations. In particular, the analysis developer only has to check that for each non-$\sqsubseteq$-monotonic rule, another rule exists that will eventually dominate the decrease, so that the end result is $\sqsubseteq$-monotonic. See in Figure 5.2; the first alternative body of `Resolve` is non-$\sqsubseteq$-monotonic, but the second alternative body eventually dominates it.

Ladddder imposes two further assumptions as well:

Well-behaving aggregators (A 5.2): We call an aggregator well-behaving if it satisfies the following requirements: (i) it is an associative and commutative binary operation, (ii) it respects a partial order $\sqsubseteq$, that is, when applied to a (multi)set of aggregands, the result must $\sqsubseteq$-dominate the aggregands, (iii) it guarantees a stationary output in a finite number of repeated applications even in case of infinite lattices (i.e. is a widening [30]). For lattices with finite ascending chains, the lattice operations `lub`/`glb` immediately satisfy these criteria.

Stratified recursion (A 5.3): Non-monotonic recursion is forbidden, which entails two constraints. First, stratified negation is required. Second, for each lattice that is *produced* in a dependency component, all well-behaving aggregators applied on a specific lattice must agree on the same $\sqsubseteq$ ordering direction. We emphasize that the requirement applies per lattice per component. Note the word *produced*, as it can happen that a lattice is produced in an upstream component, and the current component just treats is relationally without aggregating on it. If a lattice is not produced in the current component, then the requirement does not apply. Both of these requirements are standard in Datalog solvers, e.g. DRed$_L$ also expects that analyses meet these conditions.

5

Guarantees If an analysis satisfies these requirements, then Ladddder guarantees that the fixpoint computation terminates and yields the minimal model, i.e. the smallest relations that are compatible with the rules (Correctness (R1)). Ladddder supports any lattice-based analysis that is eventually $\sqsubseteq$-monotonic, which also includes $\sqsubseteq$-monotonic analyses (Expressiveness (R3)). We emphasise that, in case the analysis is $\sqsubseteq$-monotonic according to the Ross and Sagiv semantics, our eventually $\sqsubseteq$-monotonic semantics yields the *exact same* results as the traditional one. In Section 5.6, we also evaluate Efficiency (R2). Note that we informally refer to our abstract domains as lattices (as is typical in program analysis), but technically we only require a partial order with a specific kind of aggregation operator. A formal treatment of the assumptions, semantics, correctness properties, and their proof sketches are available in Section 5.5.

5.4 Incremental Aggregation in Ladddder

As introduced in Section 5.3, DDF is incremental along multiple time axes: input epochs and iteration rounds. Therefore an aggregator data structure must yield the aggregate value (per each aggregation group) as a function of iteration timestamp, and then incrementally amend this function for each new epoch. We present a number of possible architectures.

Naive aggregation The most straightforward approach is to maintain a (multi)set of aggregands for each iteration round - possibly sparsely, omitting timestamps where the multiset has not changed. It is easy to maintain the invariant that each aggregand is present in exactly those of the multisets that correspond to timestamps where the cumulative existence timeline has the value 1. If this timeline changes for a tuple, the lattice value has to be added to or removed from a range of timestamps; for each affected multiset, the new aggregate value has to be computed. For large multisets of aggregands, re-aggregating them upon each change may take a long time, therefore we have looked into incremental aggregation techniques.

Parallel incremental aggregation While traditional incremental aggregation algorithms do not support multiple time axes, they can be employed to maintain the aggregate value of a single multiset valid at a given timestamp (or, sparsely, an interval of timestamps where the multiset is unchanging). The difference to the naive version is that, upon updating the multiset, the aggregate value is incrementally updated rather than recomputed from scratch.

The original paper describing DDF [82] uses a similar architecture for simple aggregations, such as `sum`, where the sums are maintained directly instead of the multisets. For Laddder, we need incremental lattice aggregation though. The approach we used for DRed_L in Section 4.4.4 is also applicable here. Given an associative and commutative binary aggregation operation, we can maintain (i) the multiset as a balanced binary tree (e.g. AVL tree [108, Chapter 3.3]), and (ii) at each node the aggregate of the subtree rooted there. Copies of a single aggregand will be held in all such trees that are covered by its existential timeline, hence the word *parallel*. As this can lead to excessive memory use, we propose a novel architecture.

Sequential incremental architecture To eliminate the content duplication across the aggregator states (trees), we take advantage of (i) the binary aggregation operator being associative and commutative as per Well-behaving aggregators (A 5.2) and (ii) Laddder being inflationary so that all aggregands present at a given iteration round are also present in all subsequent iterations. The multiset of all aggregands valid at a timestamp can therefore be split into a multiset of "new" aggregands inserted at that timestamp, and the multiset of "old" aggregands inserted at any previous timestamp (none of which are removed); the total aggregate would then be the result of the binary aggregation of these two collections. Since the aggregate of "old" values is known anyways (as the aggregate value at the previous timestamp), it is sufficient to maintain the "new" aggregands in an incremental aggregator data structure.

This architecture is depicted in Figure 5.5: At each timestamp, we maintain (as a tree) the aggregate of values inserted in that iteration, if there is any (**A**). α denotes the aggregation operator, and small r_i denotes the aggregate result at the root of a tree. Capital R_i depicts the total aggregate value, which is computed as a *sequential* roll up of all tree root aggregates r_i up to a specific timestamp. When a new epoch updates one of these trees (**B**), its local aggregate is maintained as usual, and the total aggregate at the timestamp is recomputed. Then the new total aggregate will roll up to each later timestamp to recompute the totals, stopping early if there is no change at some point (**C**).

Lazy folding optimization In each architecture, propagating effects to later timestamps

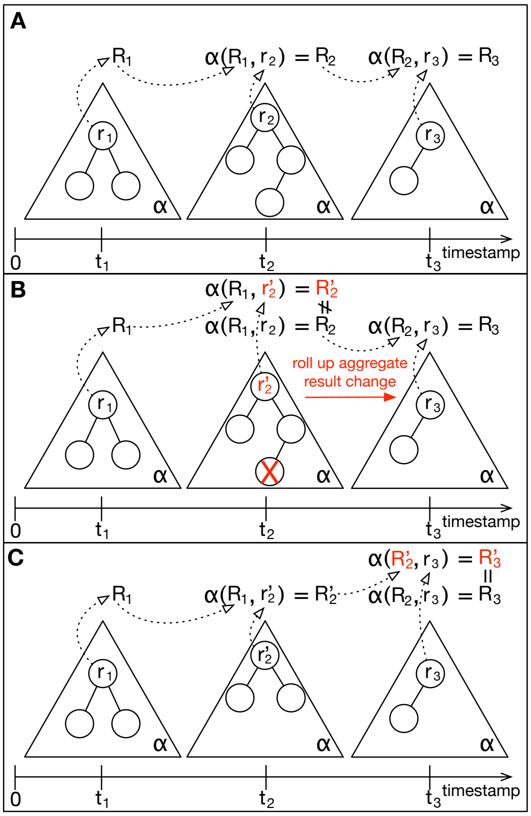


Figure 5.5: Sequential architecture. Triangles represent balanced trees maintaining (intermediate) aggregates.

can be delayed until LADDDER finishes processing all updates at earlier timestamps, so that each aggregation group is rolled up at most once per epoch.

5.5 Formal Semantics of LADDDER and Correctness Proof

We provide formal treatment of the theory behind LADDDER. After introducing the necessary terminology, we provide a more detailed description of the assumptions of LADDDER on input analyses. Then, we spell out correctness properties about LADDDER, and we prove that LADDDER satisfies them, as required by Correctness (R1).

5.5.1 Concepts

Chapter 2 provides an overview on standard Datalog and its incrementalization. Here, we introduce other advanced or non-standard terminology.

Predicates The predicates in a dependency component are divided into *exported* and *private* predicates, depending on whether they are directly used in downstream dependency components or user-facing results. The former group is denoted $Exp(D)$ for dependency component D. The Datalog solver is free to introduce new private predicate symbols for storing auxiliary results and rearrange rules in a way that leaves the meaning of exported predicates intact. A *cut* of a dependency component is a subset of its predicates with the property that all recursive dependency loops in the component must intersect the cut. An *interpretation* is an assignment of actual relations to Datalog predicate symbols.

Immediate consequence Fixing the interpretation of the predicates in a cut (as well as any upstream input predicates) allows the non-recursive evaluation of all rules belonging to the component. For a cut c of a component D, the *immediate consequence* operator $T_c(I,J_c)$ takes I as an interpretation of upstream dependency components and J_c an interpretation of predicates in the cut, and directly applies the Datalog rules in the component to derive a collection of tuples. These tuples will form a new interpretation of all predicates in D. By recursion, this includes the relations in the cut; we denote by $T_c(I,J_c)[c]$ the restriction of the results to the predicates in the cut.

In addition to the standard Datalog immediate consequence operator $T_c(I,J_c)$, Laddder also makes use of an alternative immediate consequence operator $\hat{T}_c(I,J_c)$ referred to as *inflationary consequence*, which is obtained by modifying the behavior of aggregators. In addition to the aggregate value of the current iteration, $\hat{T}$ also returns (derives as additional tuples in the aggregating relation) each aggregate result obtained at any earlier iteration timestamp.

Laddder iteratively applies the inflationary consequence operator. We denote as $\hat{T}_c^{(k)}$ the effect of k iterations, with $\hat{T}_c^{(0)}(I,J_c) = J_c$ and $\hat{T}_c^{(k+1)}(I,J_c) := \hat{T}_c(I, \hat{T}_c^{(k)}(I,J_c)[c])$., while $\hat{T}_c^{\omega}(I,J_c)$ consists of all tuples derived in any number of iterations: $\hat{T}_c^{\omega}(I,J_c) := \hat{T}_c^{(1)}(I,J_c) \cup \hat{T}_c^{(2)}(I,J_c) \cup \ldots$.

Lattices and ordering Datalog rules with aggregation or expression evaluation can compute new values beyond those present in EDB relations. These values belong to appropriate *abstract domains*, which are often (practically) infinite. In our use case of program analysis, such domains are typically lattices. Note, however, that for Laddder to work correctly, we only actually require partial orders equipped with binary operators having certain properties (see Section 5.5.2).

Given that Datalog rules can simply use lattice values produced in upstream dependency components without aggregating them, we explicitly say that a component *produces a lattice* if the value gets derived in the component by expression evaluation or aggregation.

Taking any one of the two partial orderings $\sqsubseteq$ for each *produced lattice* of a dependency component, they can be naturally extended [100] to: (i) tuples derived by a Datalog rule as $t \sqsubseteq t'$, if t and t' are $\sqsubseteq$-related on all variables of computed lattices, and agree elsewhere; (ii) entire interpretations $I \sqsubseteq I'$ if all $t \in I$ have a $t' \in I'$ with $t \sqsubseteq t'$. The latter relationship is a *preorder* (transitive but not antisymmetric), as it permits $I \sqsubseteq I' \sqsubseteq I$, denoted as $I \simeq I'$, even if $I \neq I'$; for finite interpretations this means agreement in a subset of tuples that $\sqsubseteq$-dominate all differences.

5.5.2 Refined Assumptions of Laddder on the Input Datalog Rules

We refer to Well-behaving aggregators (A 5.2) and Stratified recursion (A 5.3) from Section 5.3.3. However, Eventual $\sqsubseteq$-monotonicity (A 5.1) was introduced in Section 5.3.3 only informally, so a more formal treatment is needed here. We introduce the notion of a *well-cut recursion.*

Well-cut recursion (A 5.1): We require that each dependency component D has at least one such cut c that is *well-cut*, i.e. has the following properties:

Aggregated cut (A 5.1.1): The cut is placed after aggregations. More precisely, each predicate in c is the aggregation of a collecting relation along all of its produced lattice variables.

Observable monotonicity (A 5.1.2): c contains all the exported predicates of D, i.e. $Exp(D) \subseteq c$. In other words, predicates that may permanently $\sqsubseteq$-decrease (when $\sqsubseteq$-increasing the input), and hence not part of the well-cut, must not be directly externally observable, only through their effect on the well-cut. Note that this also implies that only aggregated predicates are visible externally.

Eventually $\sqsubseteq$-monotonic cut (A 5.1.3): For each lattice produced in it, we require that the dependency component D be associated with one of the ordering directions $\sqsubseteq$ of the lattice such that for the above mentioned cut also demonstrates *eventually* $\sqsubseteq$-monotonic recursion for the inflationary consequence operator of any other aggregated cut. All tuples that will eventually be derived for the predicates in the well-cut, in any number of iterations, must be $\sqsubseteq$-dominated by at least one tuple eventually derivable from each possible interpretation that $\sqsubseteq$-dominates the original starting interpretation. However, the two tuples need not be derived in the same iteration. Formally, for well-cut c and any aggregated cut d, $J_d \sqsubseteq J'_d$ implies $\hat{T}_d^{\omega}(I,J_d)[c] \sqsubseteq \hat{T}_d^{\omega}(I,J'_d)[c]$.

The above definition of Well-cut recursion (A 5.1) supersedes the informally described Eventual $\sqsubseteq$-monotonicity (A 5.1) from here on. Clarifying Well-behaving aggregators (A 5.2), the aggregators of the dependency component are expected to agree with the order $\sqsubseteq$ of the corresponding lattice as mentioned in Eventually $\sqsubseteq$-monotonic cut (A 5.1.3).

5.5.3 Semantics

Note while the existence of a well-cut is assumed, the semantics and the implementation are based on a more general kind of cuts: an *eligible cut* is any aggregated cut (as in Aggregated cut (A 5.1.1)) that contains all of $Exp(D)$ (as in Observable monotonicity (A 5.1.2)). As eventual monotonicity is not required, it is not necessarily a well-cut. Laddder performs an iterative fixpoint computation that computes the analysis result according to the following semantics. For a dependency component D, take an arbitrary eligible cut and start at an empty J_c to compute $D_c^{\text{raw}}(I) := \hat{T}_c^{\omega}(I,\emptyset)[c]$ the least fixpoint of $\hat{T}_c$. Since the inflationary variant of the immediate consequence operator was used, the result may contain aggregate results from older iterations, as well. They can be discarded by pruning the results at the cut by taking the $\sqsubseteq$-maximal (or, equivalently, latest result from each aggregation group). The result of this pruning is denoted $D_c^{\text{prune}}(I) := Prn(D_c^{\text{raw}}(I))$. The LADDDER semantics of the dependency component, as far as downstream components or the end user is concerned, is the interpretations of the exported predicates $Exp(D)$ (all contained in c by Observable monotonicity (A 5.1.2)), i.e. $D^{\text{exp}}(I) := D_c^{\text{prune}}(I)[Exp(D)]$.

5.5.4 Correctness Properties

If the assumptions stated above are met, LADDDER guarantees the following correctness properties:

Termination (P 5.1): The fixpoint computation of $\hat{T}_c$ completes in a finite number of iterations for any aggregating cut c.

Stability (P 5.2): For any well-cut c, the results are stable under the consequence operations: both the raw results $D_c^{\text{raw}}(I)$ and their pruned form $D_c^{\text{prune}}(I)$ are fixpoints of the inflationary consequence operator $\hat{T}_c$; while the latter is also a fixpoint of the conventional immediate consequence operator T_c. Informally, this means that the Datalog rules are satisfied in the final state.

Minimal model (P 5.3): For any well-cut c, among all possible $\hat{T}_c$-stable interpretations, $D_c^{\text{raw}}(I)$ and $D_c^{\text{prune}}(I)$ are both $\sqsubseteq$-minimal (which is only unique up to $\simeq$, due to the preorder property). Moreover among such $\sqsubseteq$-minimal interpretations, $D_c^{\text{prune}}(I)$ is set-minimal. In practice, this implies the absence of recursively self-reinforcing false tuples in the results.

Well-defined semantics (P 5.4): The semantics $D^{\text{exp}}(I)$ is independent from the choice of the eligible cut used to compute it. In fact, as long as a well-cut exists, any other cut that satisfies Aggregated cut (A 5.1.1) and Observable monotonicity (A 5.1.2) (but not necessarily Eventually $\sqsubseteq$-monotonic cut (A 5.1.3) and Observable monotonicity (A 5.1.2)) will yield the same final result as the well-cut. As eligible cuts are easy to find algorithmically (e.g. just include all aggregations), there is no need for the user to point out a cut to the evaluator, merely promise that a well-cut exists. Furthermore, as any eligible cut is acceptable, the evaluator may freely choose one based on performance considerations.

Compatible semantics (P 5.5): If the standard Ross and Sagiv aggregation semantics is also defined for a well-cut c of the component (and thus T_c is immediately monotonic), its least fixpoint (minimal model) is $D^{\text{exp}}(I)$.

5.5.5 Proofs of Correctness Properties

Correctness of incremental maintenance between epochs is guaranteed by DDF. Here, we only discuss the correctness of from-scratch derivation.

Proof sketch for Termination (P 5.1) Since recursion through negation is forbidden by Stratified recursion (A 5.3), the inflationary consequence operator $\hat{T}$ has the property that $J_c \subseteq \hat{T}_c(I,J_c)[c]$ (*inflationary growth*). This means that each tuple in the interpretation of the cut predicates is derived again by $\hat{T}$, so that it will be present in the next iteration of the fixpoint search. Thus each subsequent fixpoint iteration must set-monotonically grow. This iteration will terminate as it is upper bounded by a finite envelope set: There is a finite number of objects in the upstream relations to serve as aggregation groups for the cut, and (as the collection of aggregands only grow) the aggregate value of each group can only $\sqsubseteq$-monotonically grow, and only a finite number of times (given that Well-behaving aggregators (A 5.2) requires aggregators to be widenings [30]). This is true regardless of the chosen aggregating cut.

Proof sketch for Stability (P 5.2) Let c be a well-cut. By definition, $D_c^{\text{raw}}(I)$ is a fixpoint of $\hat{T}_c$. $D_c^{\text{prune}}(I)$ is a fixpoint of $\hat{T}_c$ because (i) aggregators are consistent with $\sqsubseteq$ (Stratified recursion (A 5.3)), so all obsolete aggregate results removed by pruning are $\sqsubseteq$ -dominated by the latest aggregate result tuple, so $D_c^{\text{raw}}(I)[c] \simeq D_c^{\text{prune}}(I)$, (ii) by the fixpoint nature of the former and by applying Eventually $\sqsubseteq$-monotonic cut (A 5.1.3) in both directions, this implies $D_c^{\text{raw}}(I) = \hat{T}_c^{\omega}(I, D_c^{\text{raw}}(I))[c] \simeq \hat{T}_c^{\omega}(I, D_c^{\text{prune}}(I))[c]$, so altogether (iii) $D_c^{\text{prune}}(I) \simeq \hat{T}_c^{\omega}(I, D_c^{\text{prune}}(I))[c]$. For set-monotonically growing $\hat{T}_c$, this implies that each iteration step is unchanging: $D_c^{\text{prune}}(I) \simeq \hat{T}_c(I, D_c^{\text{prune}}(I))[c]$.

Note however that $\hat{T}_c(I, D_c^{\text{prune}}(I))[c] = T_c(I, D_c^{\text{prune}}(I))[c]$ in this case, as in each aggregation group, it leaves the single dominant aggregate value alone. Thus we have a fixpoint of the non-inflationary T_c as well.

Proof sketch for Minimal model (P 5.3) Any interpretation K has $\emptyset \sqsubseteq K$. If K is a fixpoint of $\hat{T}_c$ for well-cut c, then Eventually $\sqsubseteq$-monotonic cut (A 5.1.3) gives by induction $D_c^{\text{raw}}(I) = \hat{T}_c^{\omega}(I, \emptyset)[c] \sqsubseteq \hat{T}_c^{\omega}(I, K)[c] = K[c]$. Thus $D_c^{\text{raw}}(I)$ is $\sqsubseteq$-minimal, and so is $D_c^{\text{prune}}(I) \simeq D_c^{\text{raw}}(I)$. Among the $\simeq$-equivalence class, $D_c^{\text{prune}}(I)$ is set-minimal, as it only contains the topmost aggregate values.

Proof sketch for Well-defined semantics (P 5.4) Since different cuts c, d are just *re-arrangements* of the same recursive structure, $T_d^{(2)}(I, J_d)[c]$ can be expressed in the form $T_c(I, J_c)[c]$ for some combination J_c of $J_d[c]$ and $T_d(I, J_d)[c]$. The same goes for $\hat{T}$. Consequently, the cuts must share fixpoints, so that for each fixpoint K of $\hat{T}_c$, $K[d]$ is a fixpoint of $\hat{T}_d$, and vice versa.

Let c be a well-cut and d be an eligible cut. Since $\hat{T}_d^{\omega}(I, \emptyset)$ exists by Termination (P 5.1), it must also be a fixpoint of $\hat{T}_c$, and thus by Minimal model (P 5.3) it must dominate the least fixpoint: $\hat{T}_c^{\omega}(I, \emptyset)[c] \sqsubseteq \hat{T}_d^{\omega}(I, \emptyset)[c]$.

On the other hand, the argument in the proof for Minimal model (P 5.3) can be repeated with $\hat{T}_d$, for which Eventually $\sqsubseteq$-monotonic cut (A 5.1.3) applies equally well: for fixpoint $K := \hat{T}_c^{\omega}(I, \emptyset)$, we have $\hat{T}_d^{\omega}(I, \emptyset)[c] \sqsubseteq \hat{T}_d^{\omega}(I, K[d])[c]$. As discussed above, the two cut-specific consequence operators share fixpoints, so the latter further equals $K[c] = \hat{T}_c^{\omega}(I, \emptyset)[c]$.

Putting the two inequalities together, we get $\hat{T}_c^{\omega}(I, \emptyset)[c] \simeq \hat{T}_d^{\omega}(I, \emptyset)[c]$. By Observable monotonicity (A 5.1.2), this c-projection contains $Exp(D)$, thus $\hat{T}_c^{\omega}(I, \emptyset)[Exp(D)] \simeq \hat{T}_d^{\omega}(I, \emptyset)[Exp(D)]$.

While $\simeq$ allows differences, pruning removes them all, so $Prn(\hat{T}_d^{\omega}(I, \emptyset)[Exp(D)])$ must equal $Prn(\hat{T}_c^{\omega}(I, \emptyset)[Exp(D)])$ which further equals $Prn(\hat{T}_c^{\omega}(I, \emptyset)[c])[Exp(D)] = D^{\text{exp}}(I)$.

Proof sketch for Compatible semantics (P 5.5) Assume T_c is $\sqsubseteq$-monotonic, i.e. $J_c \sqsubseteq J_c'$ implies $T_c(I, J_c)[c] \sqsubseteq T_c(I, J_c')[c]$. By definition of the inflationary operator, we have $T_c(I, J_c)[c] = Prn(\hat{T}_c(I, J_c)[c]) \simeq \hat{T}_c(I, J_c)[c]$, so by induction $T_c^{\omega}(I, J_c)[c] \simeq \hat{T}_c^{\omega}(I, J_c)[c]$. The latter further equals $D_c^{\text{raw}}(I) \simeq D_c^{\text{prune}}(I)$. Since the last form is a result of pruning, there is at most one tuple per aggregation group, just like in $T_c^\omega(I, J_c)[c]$; the two are $\simeq$-related, and must therefore be equal. Hence they will give the same result for $Exp(D)$.

5.5.6 Additional Notes

The perceptive reader may notice that the inflationary consequence operator $\hat{T}$, as defined above, is only realizable as a function of I and J_c if the second argument contains *all* aggregating cuts (even those outside c). This is required so that the previous results in these relations are available as input, which the inflationary aggregators must emit on top of the current aggregate result. Thus J_c would have to be extended to a broader interpretation that contains extra relations on top of the cut c, and can be produced as a projection $[c \cup c']$ for extra aggregating predicates c'. This way, $\hat{T}(I, J_c)$ is expressible as $T(I, J_c[c]) \cup J_c$. For the sake of simplicity, we omitted this complicating detail from the above formalizations, as it is ultimately inconsequential: it does not affect termination or any other property. Note that the condition for eventual monotonicity becomes $J_c[c] \sqsubseteq J_c'[c] \longrightarrow \hat{T}_c^\omega(I, J_c)[c] \sqsubseteq \hat{T}_c^\omega(I, J_c')[c]$.

The above formal arguments were slightly simplified by requiring eligible cuts to contain all exported predicates. In practice, this requirement can be dropped; it is sufficient for the well-cut to satisfy Observable monotonicity (A 5.1.2).

5.6 Performance Evaluation

In this section, we evaluate the performance of LADDDER with real-world subject programs. The first question concerns the whole problem domain rather than a specific solution:

Incrementalizability (Q 5.1): Is the domain of lattice-based inter-procedural analyses incrementalizable in principle, i.e. are high-impact changes (that affect major parts of the results) rare?

We evaluate three other research questions related to Efficiency (R2):

Run Time (Q 5.2): Can LADDDER or DRed$_L$ provide quick feedback for lattice-based inter-procedural analyses?

Memory (Q 5.3): Is the extra memory consumption of LADDDER acceptable for IDE usage?

Optimization Impact (Q 5.4): How does the optimized aggregator architecture (Section 5.4) affect the run time and memory use of LADDDER?

5.6.1 Evaluation Setup

Analysis We evaluate an inter-procedural k-update points-to analysis that over-approximates to Top only if a points-to set grows beyond a fixed size k. The running example from Section 5.2 was a special case with $k = 1$. The rationale behind this analysis is that setting a low k value makes the analysis cheaper to compute, while still being useful in practice (as explained in Section 5.2). Here, we use $k = 5$, which allowed to infer non-Top points-to value for around 40% of the program variables. Tuning k to find the best

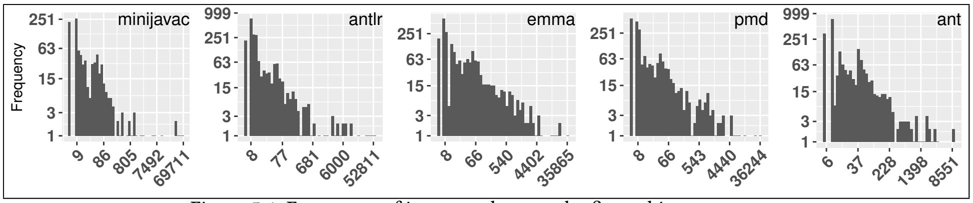


Figure 5.6: Frequency of impact values on the five subject programs.

precision-performance trade-off is an interesting research question, but we do not consider it here. Our benchmark analysis is based on an inter-procedural points-to analysis from Doop [112], which precisely keeps track of points-to sets of any size, reasons about fields and arrays, and constructs a call graph. We imported this analysis to IncA and imposed k-approximation by lattice aggregation. We implemented the k-update analysis in two functionally equivalent ways: The **branching** and **non-branching** variants are described in Section 5.2. We ensured that both analyses soundly approximate program behavior in case Top is associated with a variable. We emphasise that the design of the analysis is not our contribution and that the precision aspect of the analysis is not of interest for the purpose of our evaluation. The analysis incrementalized by Laddder or DRed$_L$ (when applicable) has the exact same precision and result as the non-incremental version.

Subject program We used code from the Qualitas Corpus [128], which is a collection of real-world Java code bases that is also used for benchmarking Doop [40]. We selected four projects: antlr (22 KLoC), emma (26 KLoC), pmd (61 KLoC), and ant (105 KLoC). We also added the much smaller minijavac (6.5 KLoC) to compare the performance of DRed$_L$ and Laddder. We used the Doop fact extractor to produce facts that describe the program elements (e.g. function signatures) and their relationships (e.g. parameters of function) from the subject program and *the JRE*.

Testbed We performed the benchmarks on an Intel Core i7-6820HQ at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 10.15.4, Java 1.8.0_121 and MPS version 2019.1. We performed each benchmarking scenario 4 times, dropped the result of the first one to account for JVM warmup, and report numbers as the average over the remaining three.

5.6.2 Evaluating Incrementalizability (Q 5.1)

We must understand the impact of program changes on realistic subject programs. We first ran a from-scratch analysis on each subject program, and then we programmatically triggered changes. We randomly selected an allocation instruction in the subject program (not the JRE) to delete and then re-insert it into the set of facts. We consider all 880 allocations in minijavac itself; from each of the larger subject programs, we randomly sample 2,000 allocations.

We define the *impact metric* of a change as the size (in tuples) of the symmetric difference of the pre- and post-state analysis result relations. Impact is a solver-independent metric because it defines a lower bound on the amount of work that any solver must perform to update an analysis result, and it does not take into account the number of tuples changed internally by the solver. Contrast this with DRed$_L$ over-deleting far more tuples than the

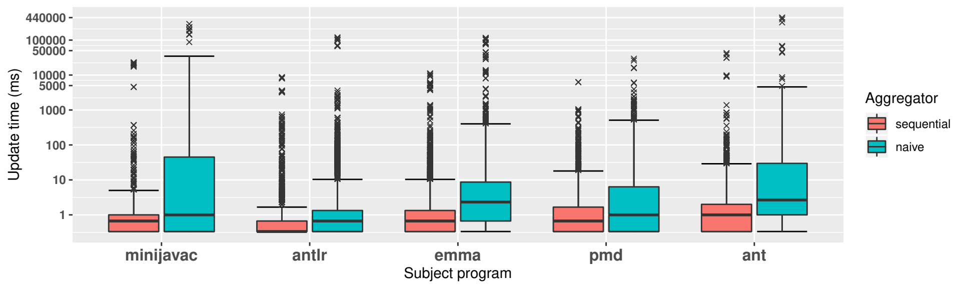


Figure 5.7: Comparison of update times between naive aggregation and sequential incremental aggregation.

impact of the changes would demand (see Section 5.2).

Figure 5.6 shows the frequency (number of occurrences) of the different impact values for the five subject programs. For all subject programs, we found that around 98 % of the deletions have an impact below 1,000, and only around 1 % above 10,000. In contrast, the overall number of tuples in the relations is in the ballpark of few millions. These numbers show that there is a lot of potential for reusing previous results.

Incrementalizability (Q 5.1): We find that lattice-based inter-procedural analyses are indeed amenable to incrementalization.

5.6.3 Evaluating Run Time (Q 5.2)

Non-branching with LADDDER and DRed$_L$ We first tried out the **non-branching** analysis implementation with both LADDDER and DRed$_L$. We set a 10 minutes timeout for the initial analysis. This time frame is too long for developers to wait at IDE startup, but we did not want to discard a solver too quick given that we analyze over the entire JRE. Unfortunately, both solvers timed out. We even tried the analysis with a preprocessed minijavac subject program, where we only generated facts for the reachable parts of the subject program (including JRE). This is not a realistic scenario because it requires to build a call graph, which, in turn, requires a points-to analysis prior to our benchmark points-to analysis. Still, it helps to reduce the number of facts significantly. However, even with the perprocessed subject program, the solvers timed out.

Branching with LADDDER We then used the **branching** analysis to measure the performance of LADDDER. We emphasise that the **branching** analysis computes the exact *same* result as the **non-branching** analysis would do, but the analysis formulation requires eventual monotonicity, which immediately disqualifies DRed$_L$. Note that we do *not* perform any preprocessing on the subject programs. We first measured the initialization time. Then, we triggered the (i) deletion and (ii) re-insertion of each allocation instruction, and we measured the time it takes to update the results for LADDDER after each change.

The initialization times of the analysis in seconds on the five code bases are as follows: minijavac - 59.50, antlr - 57.46, emma - 62.16, pmd - 171.93, and ant - 101.21. Though such delay causes a noticeable break in the development flow, we argue that they are acceptable because they are (i) one-off costs only and (ii) possibly precomputed. Figure 5.7 shows the incremental update times of LADDDER on a log scale in a box plot. For now, focus on the

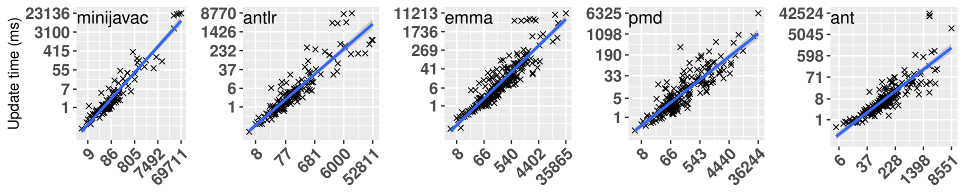


Figure 5.8: Update times as a function of impact on the five subject programs.

red (sequential) boxes and ignore the blue (naive) boxes. Fewer than 1 % of the outliers fall beyond 1 second, while the average update time is on the millisecond ballpark. These are the numbers we expect for interactive applications in IDEs. However, we also acknowledge that there are a few outlier update times reaching up to 50 seconds. To better analyze these update times, we performed a different measurement. Figure 5.8 shows update time as a function of impact on a log-log plot for the subject programs. We verify that the outlier update times indeed belong to the high-impact changes. We fit a linear regression on the log-log plot, and we found that the relationship `time` ~ $impact^{1.5}$ approximately holds for all subject programs. We believe that dealing with the rare cases of high update times in practice would be to send the analysis task to the background, while diagnostic markers in the IDE could either become stale or hidden temporarily. This problem is something that IDEs already encounter even with non-Datalog-based analyses.

Run Time (Q 5.2): Based on the observation that high-impact changes happen rarely, we argue that LADDDER can deliver good incremental run times with acceptable initialization times. It is essential that we use **branching** encoding (requiring eventual monotonicity), without that, it is hopeless to scale inter-procedural lattice-based analyses.

5.6.4 Evaluating Memory (Q 5.3)

We measured the memory use of LADDDER by taking the reachable JVM heap size after initializing the **branching** analysis and subtracting the size from before the analysis. Throughout the program changes, the memory use of LADDDER remained roughly the same. The memory use of the analysis in GBs is as follows: minijavac - 3.70, antlr - 4.07, emma - 3.94, pmd - 8.65, and ant 5.71. MPS used around 2 GB before running the analysis, which means the largest overhead is ~ 4.5 times that. These values may seem high, but we emphasize that the analysis also runs on the JRE. Furthermore, we have concrete plans for memory optimizations, which we discuss in Section 9.2.

Memory (Q 5.3): The memory consumption of LADDDER can get large, but not prohibitive. We have concrete plans on our research agenda to reduce the memory overhead.

5.6.5 Evaluating Optimization Impact (Q 5.4)

We conducted experiments to compare the performance effects of the naive and sequential aggregator architectures (Section 5.4). We following table summarizes our results:

Aggregator	Metric	Subject Program				
		minijavac	antlr	emma	pmd	ant
Sequential	initialization (s)	59.50	57.46	62.16	171.93	101.21
	mean update time (ms)	102.62	16.30	36.60	7.83	49.42
	max update time (s)	23.14	8.77	11.21	6.33	42.52
	memory (GB)	3.70	4.07	3.94	8.65	5.71
Naive	initialization (s)	52.44	66.28	64.20	159.58	105.44
	mean update time (ms)	938.07	234.62	395.37	57.10	485.83
	max update time (s)	289.36	119.93	116.38	29.34	439.90
	memory (GB)	4.06	4.20	4.06	8.58	5.98

The table shows that there is no significant difference between the initialization times and memory use for the two solutions. However, sequential aggregation is an order of magnitude faster both in terms of mean and maximum update times compared to naive aggregation (see also Figure 5.7).

5

Optimization Impact (Q 5.4): The sequential aggregator architecture updates an order of magnitude faster than the naive one. There is no significant difference between the initialization times and memory use.

5.6.6 Discussion on the Set-based Analysis

Given that our experiment for comparing the performance of LADDDER and DRed$_L$ failed with the **non-branching** analysis, we decided to go back to the original DOOP analysis and use that for comparison. For this scenario, we do *not* use any preprocessing, but we only use the smallest minijavac subject program. Even though, we give up on k-approximating, this is an interesting comparison because both LADDDER and DRed$_L$ build on the same VIATRA QUERY library, so any performance difference will be due to the differences in the fixpoint algorithms. Note that without custom lattices, DRed$_L$ is essentially a DRed implementation, while LADDDER is essentially a DDF implementation. The results of this comparison are shown in the following table:

		DRed	DDF
Initialization (s)		35.24	62.79
Inc. insertion (s)	mean	0.02 (±0.01 w/ 95 % confidence)	0.13 (±0.10 w/ 95 % c.)
	max	3.40	42.93
Inc. deletion (s)	mean	9.14 (±0.57 w/ 95 % confidence)	0.15 (±0.10 w/ 95 % c.)
	max	21.75	40.48
Memory (GB)		2.43	5.35

These numbers show some interesting insights. DRed has lower memory use, smaller initialization time, and smaller maximum update times compared to DDF. However, DDF can deliver consistently low average update times, while the over-deletion problem causes unacceptable average update times for deletions with DRed. The DDF numbers also reveal why the k-approximation is actually practically relevant: Compared to the results for the k-update analysis on minijavac (see table in Section 5.6.5), the set-based analysis adds an extra 1.65 GB (+45%) memory use and doubles the maximum update time. We looked at the points-to results and found that roughly 15% of the program variables in minijavac has a points-to set larger than 5 targets. Those are all the variables whose points-to sets the k-update analysis would approximate with Top, thereby avoiding work compared to the set-based analysis.

We see mixed results with the set-based analysis. DDF is the clear winner in terms of average update time, but DRed possesses better memory use and initialization time. We also see that the set-based analysis adds considerable overhead compared to the k-update analysis, essentially trading off performance for precision.

5.7 Chapter Summary

We presented an approach for the efficient incrementalization of lattice-based inter-procedural analyses. We used IncA Datalog with lattices in the meta end and projectional editing in the front end. Crucially, our new solver Laddder uses inflationary aggregation semantics to loosen the monotonicity requirements on analysis definitions compared to prior approaches. Laddder also combines the DDF computation model with efficient aggregator architecture for improved performance. In our evaluation, we first verified that lattice-based inter-procedural analyses are amenable to incrementalization because high-impact changes happen rarely throughout a random series of changes on real-world code bases. Then, we showed that Laddder delivers the performance interactive applications in IDEs need, as it updates results in sub-second time for more than 99 % of all changes, typically in a few milliseconds. We pay the price for the fast updates with memory: The overhead can get large, but not prohibitive.

We revisit again our requirements from Section 1.4. In terms of Expressiveness (R3) in the meta end, we did not make any improvements, as IncA Datalog could already express inter-procedural lattice-based analyses. However, we extended the back end to efficiently support such analyses as per Efficiency (R2). We provided a correctness proof for our Laddder algorithm to satisfy Correctness (R1). Our solution still satisfies Declarativity (R5) and Genericity (R4), as we did not change IncA Datalog, and Laddder is also a generic solver algorithm. In the following, we focus on a re-design of IncA Datalog to provide better language abstractions tailored to program analyses.

6 A DSL for Incremental Program Analysis

This chapter shares material with the ASE'16 paper "IncA: A DSL for the Definition of Incremental Program Analyses" [125] and with the FTfJP'18 paper "Incremental Overload Resolution in Object-oriented Programming Languages" [127].

Abstract — IncA Datalog with its relational data model provides a good basis for efficient incrementalization. As we demonstrated with case studies before, the language itself is also sufficiently expressive for a range of practically relevant analyses. However, throughout the development of those case studies, we frequently asked the question if relational operations are indeed the best abstraction we can provide for analysis implementation in IncA. Based on the development experience with a large-scale case study on type checking Featherweight Java, we witnessed that IncA Datalog is often inconvenient to use due to its verbosity and because it lacks certain language features that would support recurring patterns in the analysis implementation.

In this chapter, we *propose* a new analysis DSL called IncA$_{fun}$. We create core IncA$_{fun}$ by re-designing the syntax of IncA Datalog, and we introduce language abstractions that are familiar from traditional programming languages: functions, statements, and expressions. Then, we design a number of language extensions in IncA$_{fun}$, such as pattern matching, control statements, cast expression, that help to improve readability and conciseness of analysis implementations. We report on two case studies where we used IncA$_{fun}$ for analysis implementation. First, we implement a type checker for Featherweight Java, and we argue about the improved readability of the implementation by comparison to the functionally equivalent implementation in IncA Datalog. Second, we report on a Master's thesis that used IncA$_{fun}$ to implement a type checker and borrow checker for Rust. These Rust analyses constitute the largest IncA case study implemented so far, as they comprise 4 KLoC in size.

6.1 Introduction

From the perspective of incrementalization, IncA Datalog with its relational data model provides a good foundation. Starting from well-known techniques for the incrementalization of relational algebra with computation networks as discussed in Chapter 3, and coupled with our contributions in the back end, we managed to provide sub-second update times for IncA analyses. Moreover, we demonstrated throughout the previous chapters that IncA Datalog is expressive enough for a wide range of program analyses. However, based on the development experience throughout those case studies, we frequently wondered if reasoning over bare relations with relational operations, such as joins, filters, or projections is indeed the best abstraction we can provide for the implementation of program analyses. Typically, program analyses follow a backward or forward style with functions that take some input values and produce output(s) [90]. Following this style is not possible with IncA Datalog. Given that there are only minor syntactical differences between IncA Datalog and standard Datalog (not considering the DSL for lattice definition in IncA Datalog), we started to formulate a more general question: Is there a better DSL for implementing incremental static analyses than Datalog?

To better study the adequacy of Datalog as a specification language, we present a case study on overload resolution for Featherweight Java (FJ). This analysis is interesting because we have not considered an analysis from the type checking domain before. But, more importantly, we show that while we can express overload resolution in IncA Datalog, the language is often inconvenient to use for the implementation due to several reasons. We demonstrate these reasons in detail in Section 6.3, but we highlight a few of them upfront. Some of the problems are inherent in the syntax of Datalog: (i) verbosity because all intermediate variables must be explicitly defined, (ii) code duplication due to the lack of control structures, and (iii) non-directional analysis definition because there is no difference between input and output values. Others are missing language features in Datalog that would support recurring patterns in the analysis code: (i) pattern matching to deconstruct AST nodes, (ii) loop and if statement, or (iii) cast expression.

In this chapter, we propose a new DSL for incremental program analysis that fixes the above described shortcomings of Datalog. We call the new DSL IncA$_{fun}$. We design IncA$_{fun}$ in two steps. First, we re-design the syntax of Datalog to create the core IncA$_{fun}$ language. The word *"fun"* in IncA$_{fun}$ stands for *functional*, as in, IncA$_{fun}$ uses functions (instead of rules) as the main abstraction to specify analyses. In function bodies, IncA$_{fun}$ uses constructs that are familiar from typical programming languages; statements, expressions, input and output parameters of functions. Second, we design language extensions to concisely support the above described missing language features. We use IncA$_{fun}$ as an alternative to IncA Datalog in the meta end of our IncA analysis framework. We compile the IncA$_{fun}$ language extensions to core IncA$_{fun}$ and then that to IncA Datalog.

We evaluate IncA$_{fun}$ through two case studies. First, we use IncA$_{fun}$ to reimplement overload resolution for FJ. We showcase interesting implementation details from the FJ type checker and compare the implementation to the functionally equivalent IncA Datalog code. Second, as part of a master thesis [21], IncA$_{fun}$ was used to implement a type checker for Rust and the borrow checker used in the Rust compiler to verify ownership of values. We briefly report on this work, as well.

Contributions In summary, we make the following contributions:

- We implement overload resolution for FJ with IncA Datalog. We report on the developer experience and identify what makes the language inconvenient to use for the analysis implementation (Section 6.3).
- We introduce the syntax of our new IncA$_{fun}$ DSL, including our language extensions, and we compile IncA$_{fun}$ to IncA Datalog (Section 6.4).
- We present two case studies using IncA$_{fun}$ as the specification language. First, we implement overload resolution for FJ, and then we briefly discuss the IncA$_{fun}$ implementation of a Rust type checker and borrow checker (Section 6.5).

6.2 Background: Overload Resolution in Featherweight Java

This section provides background material on overload resolution because we will use this analysis as our motivating case study throughout this chapter. Static method overloading is an essential feature of object-oriented programming languages. Method overloading allows developers to provide the same method name to multiple method implementations. Developers typically use method overloading for one of two reasons. First, they use it to provide a more flexible interface to a single functionality by accepting different kinds of parameter types. A common example of this is constructor overloading, which allows users of a class to construct a class instance in different ways:

```java
class Node {
  public Node(int weight, Color color) { ... }
  public Node(int weight) { this(weight, Color.BLACK); }
  public Node() { this(1); }
}
```

Second, developers use method overloading to select one of multiple functionalities by dispatching over the number and types of arguments. For example, method overloading is a key component for the visitor pattern:

```java
interface GraphVisitor {
  void visit(Node node);
  void visit(Edge edge);
  void visit(SubGraph graph);
  // add specialized handlers for other elements as needed
}
```

The downside of method overloading is that it becomes difficult for developers to reason about method calls. In particular, the name of the called method does not provide sufficient information to understand which implementation will be invoked. While this is also true for dynamic dispatch via inheritance and overriding, overloaded methods are not governed by the Liskov Substitution Principle, such that a developer has to inspect the exact dispatch target to anticipate the effect of a call. To resolve a call to an overloaded method `e.m(`a_1`,...,`a_n`)` in FJ, a developer has to take all of the following information into account [16, Section 2]:[1]

- The compile-time type `C` of `e`: All methods named `m` in `C` are candidates for the resolution.
- The superclasses of `C`: All methods named `m` in the superclasses of `C` are candidates for the resolution.
- The number of arguments a_1`,...,`a_n: Only methods with `n` parameters are candidates for the resolution.
- The compile-time types C_1`,...,`C_n of the arguments a_1`,...,`a_n: Only methods that accept C_1`, ..., `C_n or superclasses thereof are candidates for the resolution.
- The distance in the class hierarchy between the argument types C_1`,...,`C_n and the parameter types D_1`,...,`D_n of candidate methods: Select the unique candidate with minimal aggregated distance or report an error if no unique candidate with minimal distance exists.

Since this is a lot of information for a developer to trace manually, IDEs mirror the compiler's behavior and resolve overloaded methods automatically. Most editors handle overload resolution together with name resolution during type checking, since the target method determines the type of the method invocation. Like all IDE services, overload resolution has to be incremental and react to changes in subject programs efficiently. That is, when a code change occurs, the IDE has to incrementally update previously computed overload resolution results. Our goal is now to implement overload resolution for FJ with IncA Datalog to automatically incrementalize the analysis.

6.3 Overload Resolution with IncA Datalog

In this section, we discuss our implementation of overload resolution for FJ with IncA Datalog. On the one hand, our goal is to show the most important building blocks of the analysis implementation. We do not aim to present the full implementation because it consists of a few hundred lines of IncA Datalog code. On the other hand, we point out why we found IncA Datalog inconvenient to use for the implementation, motivating the need for a new analysis DSL.

Analysis Implementation We start our implementation by defining the entry point, which is a `TypeOf` rule that computes the compile-time type of an FJ expression. The input expression can be of several kinds; e.g. variable reference, field access, cast, method call, just to name a few. Method call is particularly interesting because the receiver of a method call is also an expression, and, as we detailed above, the whole overload resolution process starts from the compile-time type of the receiver. This already suggests that we will need a recursive analysis implementation.

Figure 6.1 shows an excerpt of the `TypeOf` implementation, showing three alternatives. The rule has two head variables, the `exp` is the expression in question, while `type` in our

[1]Note that the rules for overload resolution for standard Java are rather different: `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html`.

```
 1 TypeOf(exp : Exp, type : Type) :- {
 2   Cast(exp),
 3   Cast.exp(exp, innerExp),
 4   Cast.type(exp, type),
 5   TypeOf(innerExp, innerExpType),
 6   ContainingModule(exp, module),
 7   ResolveClass(innerExpType, module, innerExpClass),
 8   ResolveClass(type, module, castClass),
 9   IsSubtype(innerExpClass, castClass, module)
10 } alt {
11   Cast(exp),
12   Cast.exp(exp, innerExp),
13   Cast.type(exp, type),
14   TypeOf(innerExp, innerExpType),
15   ContainingModule(exp, module),
16   ResolveClass(innerExpType, module, innerExpClass),
17   ResolveClass(type, module, castClass),
18   IsSubtype(castClass, innerExpClass, module)
19 } alt {
20   ...
21 } alt {
22   MethodCall(exp),
23   MethodCall.receiver(exp, receiver),
24   TypeOf(receiver, receiverType),
25   ContainingModule(exp, module),
26   ResolveClass(receiverType, module, class),
27   MinimalMethod(exp, class, method),
28   MethodDec.returnType(method, type)
29 }
```

Figure 6.1: Excerpt of the main `TypeOf` rule which computes the type of an expression.

implementation represents the name of a class. The first two alternatives compute the type for cast expressions, while the last one handles method calls. The complete implementation has much more alternatives, as we must handle all kinds of expressions.

We start with cast expressions. Given an expression `e = (C) e`$_0$ where `e`$_0$ has type `D`, FJ allows either upcast when `D<:C`, or downcast when `C<:D`. The first alternative handles upcasts. We start by computing the type `innerExpType` of the inner expression `innerExp` using `TypeOf` recursively (lines 2-5). FJ organizes classes into modules, and we use the container module to look up the classes `innerExpClass` and `castClass` for `innerExpType` and `type`, respectively (lines 6-8). Finally, we check that `innerExpClass` is a subtype of `castClass` (line 9). The second alternative uses the exact same logic except for the last atom because that checks that the expression is a downcast (line 18).

The last alternative computes the type of a method call. We first recursively use `TypeOf` on `receiver` to obtain the `receiverType` (lines 22-24). We use the container `module` of `exp` to look up the `class` that the `receiverType` represents (lines 25-26). Then, we use the `MinimalMethod` (explained next) rule to obtain the `MethodDec` with the minimum distance (line 27). Finally, the type of the method call is the `returnType` of the found `MethodDec` (line 28).

Let us now look at the implementation of `MinimalMethod` through an excerpt shown in Figure 6.2. `MinimalMethod` defines three head variables; `call` represents the method

```
MinimalMethod(call : MethodCall, class : ClassDec, method : MethodDec) :- {
  count MinimalMethodLookup(call, class, _) == 1,
  MinimalMethodLookup(call, class, method)
}

MinimalMethodLookup(call:MethodCall, class:ClassDec, method:MethodDec) :- {
  CandidateMethod(call, class, method),
  MethodDistance(call, method, distance),
  MinMethodDistance(call, class, minDistance),
  distance == minDistance
}

MinMethodDistance(call:MethodCall, class:ClassDec, glb(distance:Nat)) :- {
  CandidateMethod(call, class, method),
  MethodDistance(call, method, distance)
}

MethodDistance(call:MethodCall, method:MethodDec, sum(distance:Nat)) :- {
  MethodDec.parameters(method, param),
  MethodCall.arguments(call, arg),
  Parameter.index(param, pindex),
  Argument.index(arg, aindex),
  pindex == aindex,
  ArgParamDistance(arg, param, distance)
}
```

Figure 6.2: The core overload resolution logic in IncA Datalog.

call, `class` is the class of the receiver, and `method` is the target of the call. According to the overload rules in Section 6.2, `method` must be the single method with the minimal distance for the resolution to succeed. To this end, `MinimalMethod` counts the number of candidate methods with minimal distance, and only yields a result if the count is one. It uses `MinimalMethodLookup` which yields all methods with minimal distance. The helper rule `CandidateMethod` (whose implementation is omitted) only enumerates methods that have the required name and the appropriate number of parameters. `MethodDistance` zips arguments and parameters and sums up the distances between their types, while `MinMethodDistance` computes the minimum distance out of all distances. To perform computation on numbers, we define the `Nat` lattice, which represents natural numbers. An excerpt of its implementation is shown in Figure 6.3.

Perceived Developer Experience While implementing the above analysis in IncA Datalog, we experienced that certain characteristics of the language make the implementation overly verbose or difficult to read. We also identified certain features that could have helped to make the implementation more readable or easier to understand. We summarize our findings as follows:

Non-directional code The analysis implementation is ultimately non-directional. There is no difference between input and output parameters. This is a not ideal because static analyses typically reason about subject programs in a forward or backward style [90]. This is entirely missing from our implementation, as we simply define relations with the Datalog rules. It would be natural to implement `TypeOf` as a

```
lattice Nat {
  constructors { Value(int) | Top }
  def bot() : Nat = return Value(0)
  def top() : Nat = return Top
  def leq(l : Nat, r : Nat) : boolean = {
    match (l, r) with {
      case (Value(v1), Value(v2)) => return v1 <= v2
      case _ => return false
    }
  }
  def glb(l : Nat, r : Nat) : Nat = {
    match (l, r) with {
      case (Value(v1), Value(v2)) => return Value(Math.min(v1, v2))
      case (_, Top) => return l
      case (Top, _) => return r
    }
  }
  def sum(l : Nat, r : Nat) : Nat = {
    match (l, r) with {
      case (Value(v1), Value(v2)) => return Value(v1 + v2)
      case _ => return Top
    }
  }
  ...
}
```

Figure 6.3: A lattice representing natural numbers as implemented in IncA Datalog.

function that takes an expression as input and computes a type, or `MinimalMethod` as a function that takes a call and a class as input and returns the resolved target method.

Lack of support for pattern matching IncA Datalog uses virtual EDB relations to access the necessary parts of the subject program's AST. However, all links and types need to be traversed individually, and there is no language feature that would allow to deconstruct entire subtrees, while introducing variables to refer to relevant parts of the tree.

Lack of control structures Currently, the only way to provide alternative implementations for the same rule is to define multiple rule bodies. We ended up duplicating almost the same rule body twice to handle upcasts and downcasts in Figure 6.1. Of course, we could have factored out the common atoms into a separate helper rule, but this obscures the implementation, and it is also inconvenient in general, as the helper rule may end up having a large number of head variables that are only relevant for certain rule bodies.

Verbosity In IncA Datalog, every local variable must be declared explicitly. For example, when implementing `TypeOf` for cast expression, we wrote:

```
Cast(exp),
Cast.exp(exp, innerExp),
...
TypeOf(innerExp, innerExpType),
```

(module)	m	$::= \textbf{module}\ n\ \textbf{import}\ \overline{n}\ \{\overline{mc}\}$
(module content)	mc	$::= l \mid f$
(lattice)	l	$::= \textbf{lattice}\ L_n\{\textbf{constructors}\{\overline{ctor}\}\ \overline{lop}\}$
(lattice name)	L_n	$::=$ name
(type in lattice definition)	T_{lat}	$::= T_{lang} \mid L_n \mid$ Java type
(constructor)	$ctor$	$::= n(\overline{T_{lat}})$
(lattice op)	lop	$::= \textbf{def}\ n(\overline{n : T_{lat}}) : \overline{T_{lat}} = lopb$
(lattice op body)	$lopb$	$::=$ Java code + lattice constructors and operations
(function)	f	$::= vis\ \textbf{def}\ n(\overline{n : T_{in}}) : \overline{T_{out}} = \overline{alt}$
(visibility)	vis	$::= \textbf{private} \mid \textbf{protected} \mid \textbf{public}$
(alternative)	alt	$::= \{\ \overline{s}\ \}$
(statement)	s	$::= \overline{n} := e \mid \textbf{assert}\ cond \mid \textbf{yield}\ e$
(condition)	$cond$	$::= e == e \mid e\ != e \mid e\ \textbf{instanceOf}\ T_{lang} \mid$
		$e\ \textbf{not instanceOf}\ T_{lang} \mid \textbf{def}\ e \mid \textbf{undef}\ e$
(expression)	e	$::= n \mid c \mid e.L_k \mid n(\overline{e}) \mid n+(e) \mid \textbf{count}\ n(\overline{e}) \mid L_n.n(\overline{e})$
(constant)	c	$::=$ number | string | enum | boolean
(type of input parameter)	T_{in}	$::= T_{lang}$
(type of output parameter)	T_{out}	$::= T_{lang} \mid L_n \mid L_n/lop$
(type from subject language)	T_{lang}	$::=$ AST node type (from subject language)
(link)	L_k	$::=$ link of an AST node type (from subject language) |
		parent | **prev** | **next** | **index**
(name)	n	$::=$ name

Figure 6.4: Syntax of core IncA$_{fun}$.

Here, `innerExp` is a variable that we only refer to once in the rule body. This may seem like a minor issue, but we emphasise that the full analysis implementation is a few hundred lines of IncA Datalog code, so all of this verbosity adds up and negatively affects readability. Being able to write something like `exp.innerExp` would help to avoid the declaration of variables that are only referred to once.

Based on these findings, we set out to experiment with an alternative design for a new specification language in IncA. We present our new approach next.

6.4 Syntax and Compilation of IncA$_{fun}$

In this section, we present our new IncA$_{fun}$ language and its compiler. First, we discuss the syntax in two steps: We define a core language by re-designing the syntax of IncA Datalog, and then we present a number of language extensions. Second, we discuss how we compile IncA$_{fun}$ to IncA Datalog.

6.4.1 Syntax of IncA$_{fun}$

Syntax of core IncA$_{fun}$ Figure 6.4 shows the syntax of core IncA$_{fun}$. We reuse the DSL for lattice definitions from IncA Datalog, so we do not discuss that again here. It is important to emphasise that IncA$_{fun}$ is still a declarative language, and it has the same expressive power as IncA Datalog. However, there are several syntactic differences in IncA$_{fun}$ compared to IncA Datalog, and we highlight those in the following.

IncA$_{fun}$ uses functions to organize the analysis implementation. A function takes a set of input values and produces output values. A function can have multiple alternative bodies, and its result is the union of the results of the individual alternatives. An important requirement for functions is that they are only allowed to use lattice types or aggregation on output types. This requirement merely guides analysis developers to think about lattices as computed values produced by functions. Input parameters must use types from the subject language.

Instead of atoms accessing relations, the body of a function consists of statements. Statements can be of three kinds; variable declarations, assertions, and yield statements. Assertions define constraints that must be satisfied in order for the function to yield an output. If any of the assertions is violated for a given set of inputs, the function returns an empty set of results. The conditions in assertions can be one of the following: equality, inequality, `instanceOf` check against an AST node type, `not instanceOf`, definedness and undefinedness testing. The latter two are used to check if a function has an output or does not have an output given an input. IncA$_{fun}$ supports the following expressions: variable reference, constant value, path expression, function call, counting the number of outputs of a function, transitive closure, and lattice operation call.

Syntax of IncA$_{fun}$ language extensions Figure 6.5 shows the syntax of our language extensions. These extensions are purely syntax sugar, as they will be reduced to core IncA$_{fun}$. However, as we will show in Section 6.5, they improve readability and conciseness of the analysis implementation. We introduce four new kinds of statements. (i) Switch is used to define alternative statements, potentially in a nested fashion. (ii) Foreach allows to iterate over a collection of nodes or all instances of an AST node type. (iii) If statement is a simple conditional statement, optionally with else if and else branches. (iv) Match is used for pattern matching. Match patterns can be of the following kinds: constant value, named pattern, named pattern with a nested pattern, AST node type with nested patterns for select links, and wildcard pattern which matches any node. We also add a new kind of expression; A cast expression allows to cast an expression to an AST node type. If a cast fails, the function does not yield an output for the surrounding alternative.

6.4.2 Compilation of IncA$_{fun}$

We use a sequence of compilation steps to produce graph patterns from IncA$_{fun}$ code:

- We compile IncA$_{fun}$ language extensions to core IncA$_{fun}$.
- We compile core IncA$_{fun}$ to IncA Datalog.
- Finally, we compile IncA Datalog to graph patterns, as described in Section 3.3.2.

$$
\begin{array}{lll}
\text{(statement)} & s & ::= \ldots \mid \textbf{switch}\ \{\overline{s}\}\ \overline{alt} \mid \\
 & & \quad \textbf{foreach}\ var\ \textbf{in}\ e\ \{\overline{s}\} \mid \textbf{foreach}\ var\ \textbf{in}\ T_{lang}\ \{\overline{s}\} \mid \\
 & & \quad \textbf{if}\ (cond)\ \{\ \overline{s}\ \}\ \overline{elseif}\ \textbf{else}\ \{\ \overline{s}\ \} \mid \\
 & & \quad \textbf{match}\ (\overline{e})\ \textbf{with}\ \{\ \overline{case}\ \} \\
\text{(else if branch)} & elseif & ::= \textbf{else if}\ (cond)\ \{\ \overline{s}\ \} \\
\text{(match case)} & case & ::= \textbf{case}\ (\overline{pat})\ \{\ \overline{s}\ \}\ \mid \textbf{default}\ \{\ \overline{s}\ \} \\
\text{(match case pattern)} & pat & ::= c \mid n \mid n : pat \mid T_{lang}(\overline{L_k = pat}) \mid \_ \\
\text{(expression)} & e & ::= \ldots \mid e\!:\!T_{lang}
\end{array}
$$

Figure 6.5: Syntax of IncA$_{fun}$ language extensions.

Compilation of IncA$_{fun}$ Language Extensions to Core IncA$_{fun}$

We discuss the compilation of the language extensions (Figure 6.5) through their implementation in Scala. First, we show an excerpt of the abstract syntax of IncA$_{fun}$ in the form of Scala case classes. We use these case classes in the compiler implementation.

```
// excerpt of the abstract syntax of the core language
case class Alt(stmts: Seq[Stmt])

case class AssignStmt(v: Var, exp: Exp) extends Stmt
case class AssertStmt(cond: Cond) extends Stmt

case class Equality(left: Exp, right: Exp) extends Cond
case class Inequality(left: Exp, right: Exp) extends Cond
case class InstanceOf(exp: Exp, typ: Type) extends Cond

case class Var(name: String) extends Exp
case class NumLit(v: Int) extends Exp
case class PathExp(exp: Exp, link: Link) extends Exp

// abstract syntax of the language extensions
case class StmtList(stmts: Seq[Stmt]) extends Stmt
case class SwitchStmt(alternatives: Seq[Alt]) extends Stmt
case class ForeachStmt(itr: Var, exp: Exp, stmts: Seq[Stmt]) extends Stmt
case class ForeachStmt(itr: Var, typ: Type, stmts: Seq[Stmt]) extends Stmt
case class IfStmt(cond: Cond, thenClause: Seq[Stmt],
  elseIfClauses: Seq[ElseIfClause],
  elseClause : Option[Seq[Stmt]]) extends Stmt
case class MatchStmt(exps: Seq[Exp], cases: Seq[MatchCase])

case class ElseIfClause(cond: Cond, stmts: Seq[Stmt])

case class MatchCase(patterns: Seq[MatchPattern], stmts: Seq[Stmt])

trait MatchPattern
case class ConstantPattern(value: Exp) extends MatchPattern
case class PatternVariable(name: String) extends MatchPattern
case class NamedPattern(name: String, pattern: MatchPattern)
    extends MatchPattern
case class NodePattern(typ: Type, bindings: Seq[(Link, MatchPattern)])
    extends MatchPattern
case class WildcardPattern() extends MatchPattern

case class Cast(exp: Exp, typ: Type) extends Exp
```

StmtList We start with `StmtList`, which is a temporary construct used in the compilation logic of several of the language extensions. It is used in cases when the compiler produces a collection of statements from a single statement. The compiler replaces a `StmtList` with the contained `Stmts` in the surrounding `Alt`:

```
def compileStmtList(alt : Alt) : Alt = {
  Alt(
    alt.stmts.flatMap(s => {
      s match {
        case StmtList(stmts) => stmts
        case _ => Seq(s)
      }
    })
  )
}
```

SwitchStmt A `SwitchStmt` compiles into a series of `Alts`, where each switch alternative contributes one `Alt` containing the statements preceding the `SwitchStmt`, the statements of the respective case, and the statements succeeding the `SwitchStmt`. The following code shows the compilation of a single `SwitchStmt` in a single `Alt`. However, the real implementation is more involved because it also needs to handle cases when multiple `SwitchStmts` are used in the same `Alt` or when `SwitchStmts` are nested.

```
def compileSwitchStmt(alt: Alt): Seq[Alt] = {
  alt.stmts.find(s => s.isInstanceOf[SwitchStmt]) match {
    case Some(SwitchStmt(alts)) =>
      val prevStmts = alt.stmts.takeWhile(s => !s.isInstanceOf[SwitchStmt])
      val nextStmts = alt.stmts.reverse.takeWhile(s =>
        !s.isInstanceOf[SwitchStmt]).reverse
      alts.map(alt => Alt(prevStmts ++ alt.stmts ++ nextStmts))
    case _ => Seq(alt)
  }
}
```

ForeachStmt A `ForeachStmt` compiles into a `StmtList` containing either an assignment to the respective iterator variable or a type constraint on the iterator variable, plus the original statements from the body.

```
def compileForeachStmt(stmt: Stmt): StmtList = {
  stmt match {
    case ForeachStmt(itr, exp: Exp, stmts) =>
      StmtList(Seq(AssignStmt(itr, exp)) ++ stmts)
    case ForeachStmt(itr, typ: Type, stmts) =>
      StmtList(Seq(AssertStmt(InstanceOf(itr, typ))) ++ stmts)
  }
}
```

IfStmt An `IfStmt` compiles into a `SwitchStmt` with the following alternatives:

- The first alternative represents the `then` clause. The statements in the alternative consist of an `AssertStmt` with the `then` condition, plus the original statements from the body of the `then` clause. See lines 3-4.

- Each `else if` clause contributes an alternative where the statements in an alternative consist of the negated conditions for the preceding clauses (including the `then` clause), the condition of the respective `else if`, plus the statements from the body of the clause. See lines 5-17. `negateCondition` is a helper function that accepts a condition and returns its negated form; e.g. equality becomes an inequality, `instanceOf` becomes a `not instanceOf`, `def` becomes an `undef`, and so on.
- The `else` clause contributes one alternative where the statements consist of the negated conditions for all preceding clauses, plus the statements from the body of the clause. See lines 18-26.

```
 1 def compileIfStmt(stmt: IfStmt): SwitchStmt = {
 2   SwitchStmt(
 3     // then
 4     Seq(Alt(Seq(AssertStmt(stmt.cond)) ++ stmt.thenClause)) ++
 5     // else if
 6     stmt.elseIfClauses.zipWithIndex.map(t => {
 7       val elseif = t._1
 8       val index = t._2
 9       val previousConditions = stmt.elseIfClauses.take(index).map(_.cond)
10       Alt(
11         Seq(AssertStmt(negateCondition(stmt.cond))) ++
12         previousConditions.map(prevCond =>
13           AssertStmt(negateCondition(prevCond))) ++
14         Seq(AssertStmt(elseif.cond)) ++
15         elseif.stmts
16       )
17     }) ++
18     // else
19     stmt.elseClause.map(stmts => {
20       Alt(
21         Seq(AssertStmt(negateCondition(stmt.cond))) ++
22         stmt.elseIfClauses.map(elseif =>
23           AssertStmt(negateCondition(elseif.cond))) ++
24         stmts
25       )
26     })
27   )
28 }
```

MatchStmt A `MatchStmt` compiles into a `SwitchStmt`. Each match case contributes one alternative to the `SwitchStmt`:

```
def compileMatchStmt(stmt: MatchStmt): SwitchStmt = {
  stmt match {
    case MatchStmt(exps, cases) =>
      SwitchStmt(cases.map(compileMatchCase(_, exps)))
  }
}
```

A `MatchCase` first declares new variables for each matched expression (from the `MatchStmt`). To this end, it uses the `genName` helper function which generates fresh unique names. Each `MatchPattern` contributes a collection of statements to the alternative. The call to `compileMatchPattern` gets as argument the pattern and the variable name introduced earlier for the matched expression. Finally, we append the statements originally contained in the `MatchCase`:

```
def compileMatchCase(matchCase: MatchCase, exps: Seq[Exp]): Alt = {
  val newNames = exps.map(e => genName(e))
  Alt(
    // variable declarations for matched expressions
    exps.zipWithIndex.map(t => AssignStmt(Var(newNames(t._2)), t._1)) ++
    // statements contributed by the match case patterns
    matchCase.patterns.zipWithIndex.flatMap(t =>
      compileMatchPattern(t._1, newNames(t._2))) ++
    // statements of the match case itself
    matchCase.stmts
  )
}
```

`MatchPatterns` contribute the following collections of `Stmts`:

- `ConstantPattern` simply contributes an equality check between the constant expression and the variable name.
- `PatternVariable` introduces a new name for the matched expression.
- `NamedPattern` is like `PatternVariable`, but it also contributes more statements by recursively compiling the nested `MatchPattern`.
- `NodePattern` yields a type constraint, introduces new variable declarations through path expressions accessing the links from the bindings, and recursively contributes statements from the nested `MatchPatterns`. Importantly, the call to `compile-MatchPattern` now gets the freshly generated names as arguments because these represent the new matched expressions.
- `WildcardPattern` does not contribute any statements, as it matches “anything”.

```
def compileMatchPattern(pattern: MatchPattern, name: String): Seq[Stmt] = {
  pattern match {
    case ConstantPattern(v) =>
      Seq(AssertStmt(Equality(v, Var(name))))
    case PatternVariable(n) =>
      Seq(AssignStmt(Var(n), Var(name)))
    case NamedPattern(n, p) =>
      Seq(AssignStmt(Var(n), Var(name))) ++
        compileMatchPattern(p, name)
    case NodePattern(typ, bindings) =>
      val newNames = bindings.map(b => genName(b))
      // type constraint for the outer variable
      Seq(AssertStmt(InstanceOf(Var(name), typ))) ++
      // variable declarations for the path expressions
      bindings.zipWithIndex.map(t =>
        AssignStmt(Var(newNames(t._2)), PathExp(Var(name), t._1._1))) ++
      // statements produced by the inner patterns
      bindings.zipWithIndex.flatMap(t =>
        compileMatchPattern(t._1._2, newNames(t._2)))
    case WildcardPattern() =>
      Seq()
  }
}
```

CastExp A `CastExp` compiles into a variable reference, but it also introduces new statements to declare and constrain the referenced variable. To this end, it must capture the `Stmt` context where it is used and replace the `Stmt` with a `StmtList`. The following snippet shows two examples of this rewriting, but the actual implementation is generic, as it handles all cases:

```
def compileCastExp(stmt: Stmt): StmtList = {
  stmt match {
    case AssertStmt(Equality(Cast(exp, typ), right)) =>
      val newName = genName(exp)
      StmtList(Seq(
        AssignStmt(Var(newName), exp),
        AssertStmt(InstanceOf(Var(newName), typ)),
        AssertStmt(Equality(Var(newName), right))
      ))
    case AssignStmt(v, PathExp(Cast(exp, typ), link)) =>
      val newName = genName(exp)
      StmtList(Seq(
        AssignStmt(Var(newName), exp),
        AssertStmt(InstanceOf(Var(newName), typ)),
        AssignStmt(v, PathExp(Var(newName), link))
      ))
    ...
  }
}
```

Compilation of Core IncA$_{fun}$ to IncA Datalog

We describe the compilation of core IncA$_{fun}$ to IncA Datalog in a top-down fashion. We collect the functions from a module and generate an IncA Datalog rule for each one of them. Function input parameters become Datalog variables $\overline{\mathtt{v}_i}$, and we create designated variables $\overline{\mathtt{v}_o}$ for the output tuple. The type annotations on the input parameters and the return types become type annotations on the respective variables. Each alternative of a function becomes an alternative of the respective rule.

For statements, we proceed as follows. An assertion simply promotes the compilation target of its condition. Every expression will be represented with a designated variable, and an assignment matches up the variables on the left-hand side with the variables that result from the expression on the right-hand side. We generate atoms with equality for the pairs of variables from the two sides. To represent the output tuple of a function, we use designated variables. We handle a yield statement as an assignment to these variables.

Next, we translate IncA$_{fun}$ conditions. Equality and inequality straightforwardly translate to atoms with equality and inequality. An `instanceOf` condition becomes an atom accessing the virtual EDB relation with the respective AST node type. A `not instanceOf` is similar, but uses negation in the atom. `Def` (`undef`) also naturally maps to an atom accessing the rule generated for the respective function (in a negated form).

We proceed with expressions. An IncA$_{fun}$ variable becomes a Datalog variable. Every other expression gets a fresh variable that represents the result of the expression. The variable introduced for a constant is constrained by an equality. A path expression $\mathtt{e.L}_k$ is mapped into an atom accessing the virtual EDB relation describing the respective link

L_k. If a path expression traverses multiple links, then it gets broken up into single path expressions with intermediate variables. A function call `f(ē)` becomes an atom `f(ēv̄)`, where `v̄` are fresh and represent the result of the call (the notation `ēv̄` means concatenation). Transitive closure, count, and lattice operation call naturally map to their IncA Datalog counterpart.

This translation process shows that there is roughly a one-to-one mapping between core IncA$_{fun}$ and IncA Datalog, the differences are purely syntactic. The last step of our compilation pipeline reduces IncA Datalog to graph patterns, which we have already discussed in Section 3.3.

6.5 Case Studies with IncA$_{fun}$

In this section, we present two case studies that used IncA$_{fun}$ for analysis implementation. First, we discuss overload resolution as implemented in IncA$_{fun}$, and we highlight how the language helps to improve the readability and conciseness of the implementation. Second, a collection of Rust program analyses have been re-implemented with IncA$_{fun}$ in the context of a master thesis, and we briefly report on that case study, as well.

6.5.1 Overload Resolution for Featherweight Java with IncA$_{fun}$

We already discussed the implementation of the FJ type checker with IncA Datalog in Section 6.3. We re-implemented the type checker with IncA$_{fun}$, as well. The complete IncA$_{fun}$ implementation comprises roughly 600 LoC. Here, we only discuss the re-implementation of the two main rules that we discussed in Section 6.3, the complete code is available online.[2] Figure 6.6 shows the implementation of the `getTypeOf` function in IncA$_{fun}$, while Figure 6.7 shows the core overload resolution logic in IncA$_{fun}$. The implementation follows the same high-level logic as in Section 6.3, but there are several syntactic differences compared to the IncA Datalog implementation:

Directional analysis implementation Functions in IncA$_{fun}$ help differentiate between inputs and outputs instead of computing relations. For example, the `getTypeOf` function takes the expression as input and computes the type as output. Similarly, the functions related to computing the minimal distance method in Figure 6.7 clearly separate what serves as input and what is a computed value. Importantly, lattice values always appear as outputs of functions. The bodies of the functions use statements which clearly define an order of computation steps that derive the output from the input through the declared local variables.

Pattern matching We use a `match` statement in the body of `getTypeOf` to pattern match on the input `exp`. We define multiple match cases to distinguish the different kinds of expressions and to deconstruct the input AST node. The first case handles cast expressions, while the last one handles method calls.

Control structures We use a `switch` statement in the first case body in `getTypeOf` to define the two checks for up- and downcasts, while reusing the rest of the case body.

[2] `https://github.com/seba--/inca-experiments`

```
 1 def getTypeOf(exp : Exp) : Type = {
 2   match (exp) with {
 3     case Cast(type : castType, exp : innerExp) =>
 4       innerExpType := getTypeOf(innerExp)
 5       module := getContainingModule(exp)
 6       innerExpClass := resolveClass(innerExpType, module)
 7       castClass := resolveClass(castType, module)
 8       switch {
 9         assert def isSubtype(innerExpClass, castClass, module)
10       } alt {
11         assert def isSubtype(castClass, innerExpClass, module)
12       }
13       yield castType
14     case ...
15     case MethodCall(receiver : receiver, name : name) =>
16       receiverType := getTypeOf(receiver)
17       module := getContainingModule(exp)
18       class := resolveClass(receiverType, module)
19       method := getMinimalMethod(exp, class)
20       yield method.returnType
21   }
22 }
```

Figure 6.6: Computing the type of an FJ expression in IncA$_{fun}$.

`Yield` statements are used to specify the output(s) of a function. We use `foreach` extensively in Figure 6.7, as we select the output of the functions from a collection of elements. Finally, `if` statements help to improve the readability of the conditions that govern when an output must be returned. It is important to note that the control flow of an IncA Datalog function does not match that of a traditional programming language with similar constructs. For example, the `yield` statement in line 11 in Figure 6.7 does not return from the function to the caller after the first `method` found, instead the function will produce the whole set of methods that satisfy the constraints. Similarly, it can happen that the if condition in `getMethodDistance` in Figure 6.7 may not be satisfied at all, but there is no code that would return a value in that case. This is not a problem for an IncA$_{fun}$ function, as, in this case, an empty set of result will be returned.

Path expressions Line 20 in Figure 6.6 yields `method.returnType` as the result of the match case. The expression `method.returnType` is called a path expression. A path expression takes an expression on the left hand side and allows to traverse (multiple) links (of AST node types), without the need to explicitly declare local variables for the (intermediate) results. Another example is in line 26 in Figure 6.7, where we directly formulate the constraint for zipping without the need to explicitly declare variables for the indices of `param` and `arg`. Path expressions play an important role in reducing the code size for larger analyses.

We argue that the resulting implementation is more readable and easier to understand compared to the IncA Datalog implementation. After all, our design decisions were directly governed by our experience with the IncA Datalog implementation. Although, we also note that this is only our experience as the developers of IncA. We did not perform an empirical

```
def getMinimalMethod(call : MethodCall, class : ClassDec) : MethodDec = {
  if (count lookupMinimalMethod(call, class) == 1) {
    yield lookupMinimalMethod(call, class)
  }
}

def lookupMinimalMethod(call : MethodCall, class : ClassDec) : MethodDec ={
  minDistance := getMinMethodDistance(call, class)
  foreach method in getCandidateMethod(call, class) {
    distance := getMethodDistance(call, method)
    if (distance == minDistance) {
      yield method
    }
  }
}

def getMinMethodDistance(call : MethodCall, class : ClassDec) : Nat/glb = {
  foreach method in getCandidateMethod(call, class) {
    yield getMethodDistance(call, method)
  }
}

def getMethodDistance(call : MethodCall, method : MethodDec) : Nat/sum = {
  foreach param in method.parameters {
    foreach arg in call.arguments {
      if (param.index == arg.index) {
        yield getArgParamDistance(arg, param)
      }
    }
  }
}
```

Figure 6.7: The core overload resolution logic in IncA Datalog.

user study about the usability of the new language design as part of this dissertation, but a Master's student actually used IncA$_{fun}$ to implement another large case study, which we discuss next.

6.5.2 Rust Program Analyses with IncA$_{fun}$

As part of a Master's thesis [21], we conducted experiments with IncA$_{fun}$ and the Rust programming language [69]. Rust promotes safety and speed as its main goals, and it accomplishes these goals with a number of language abstractions that incur zero runtime overhead. The Rust type system and compiler enforces strong guarantees about the safety of Rust programs to ensure that they are free of concurrency problems (e.g. data races) or memory access violations (e.g. buffer overflows, dangling pointers, or accessing uninitialized variables). Given that these analyses are all non-trivial and target a real-world language, we set out to use IncA$_{fun}$ to implement them to see if the language is expressive enough for this use case. We briefly report on our experiences here.

We started off with implementing a type checker for Rust that handles e.g. functions, let bindings, structs, enums, and generics with explicit type annotations. This was straightforward to implement with IncA$_{fun}$, and the new syntax of IncA$_{fun}$ with functions and the language extensions allowed a natural encoding of the typing rules. A problematic case was the exhaustiveness checking of pattern matching, which the Rust compiler performs to ensure that the cases of a pattern matching cover all possible inputs. The problem here with IncA$_{fun}$ was that many times the exhaustiveness checker logic must perform inference (e.g. when a wildcard pattern is used), and this is not possible with IncA because an IncA analysis can only work with existing AST nodes. The problem is actually more general, as standard Datalog would face the exact same problem. An idea here was to make use of lattices in IncA$_{fun}$ to represent inferred data, but ultimately we failed to handle this scenario. We refer the interested reader to the Master's thesis for further details [21].

A second major part of this work revolves around the various analyses that constitute the Rust *borrow checker*, which entails three main aspects:

Ownership In Rust, every value has exactly one owner, and the notion of ownership plays an important role in ensuring the memory safety of Rust programs. When a variable is assigned a value, that variable becomes the owner of the value. Ownership of the value can be moved to a new variable when the former variable is assigned to the new variable. However, at this point, reading from the old variable is not permitted anymore, as it is not the owner of the value anymore. Reasoning about ownership is more complicated in the general case, as values can be also passed along e.g. fields of structs.

Borrows Ownership of a value can be borrowed temporarily either through an immutable reference of a mutable one. There may be an arbitrary number of immutable references at any one time or exactly one mutable reference (and no immutable reference). There is a number of checks related to borrows. For example, the scope of a borrow cannot outlive the scope of the original owner, or no writes may happen to an immutable reference.

Lifetime The notion of a lifetime defines the scope that a reference is valid for. Lifetimes appear as generic annotation on types, and the Rust compiler enforces that values with the same lifetime cannot outlive each other.

With the assumption of certain simplifications such as explicit lifetime annotations or explicit generic type annotations, we managed to implement all of the analyses in the borrow checker with IncA$_{fun}$. For the ownership checking, we implemented a flow-sensitive uninitialized read analysis. Our analysis keeps track of the sets of uninitialized access paths per CFG location. An access path is in the form `var.fld1.fld2` where `var` is a variable and `fld1` and `fld2` are field accesses. When an access path gets assigned to a variable, the access path is added to the set of uninitialized ones, as that path is not the owner of the value anymore. The result of the analysis helps to mark reads from uninitialized access paths erroneous. We implemented a data-flow analysis for the borrow and lifetime checking, which keeps track of how values flow inside and across functions. However, the analysis is not inter-procedural per se because we can just rely on the signature of the called function when analyzing a function call.

We also performed a benchmark with the IncA$_{fun}$ implementation using synthesized Rust subject programs. The following table summarizes some key metrics:

Subject program	5 KLoC (19000 AST nodes)
IncA$_{fun}$ analysis implementation	4 KLoC
IncA initialization time	47.4 s
IncA incremental update time	33.6 ms (avg) / 434 ms (max)
Official non-inc. Rust analysis	6.3 s

With 4 KLoC in size, the Rust analysis is the biggest IncA$_{fun}$ analysis implemented so far. Interestingly, the incremental update times of the IncA$_{fun}$ implementation are also really fast, but we pay the price for this with an order-of-magnitude higher initialization time compared to the run time of the official Rust analysis. In sum, we argue that our experiments were ultimately successful: We managed to implement a large part of the official Rust analyses with IncA$_{fun}$, and we obtained fast incremental performance out of the box with IncA.

6.6 Chapter Summary

We proposed and evaluated a new analysis DSL called IncA$_{fun}$ in this chapter. We first re-designed the syntax of IncA Datalog to create core IncA$_{fun}$. The new DSL uses language constructs familiar from typical programming languages; functions, statements, and expressions. Then, we also designed a number of language extensions that cover recurring patterns in analysis implementations: pattern matching, control constructs, and cast expression. We used IncA$_{fun}$ to implement two case studies: overload resolution for FJ and a type system and borrow checker for Rust.

The major focus of this chapter was centered around language design, which relates to our Expressiveness (R3) requirement. However, the goal was not to actually improve the expressive power, but rather to improve the readability and conciseness of IncA analyses. We argue that we achieved this goal with IncA$_{fun}$, and our design was directly guided by the experiences with two large-scale case studies. However, we also emphasise that this claim is based on our own developer experience. We leave a more systematic user study assessing the design of our new DSL as future work. Just like IncA Datalog, IncA$_{fun}$ also satisfies the requirements Declarativity (R5) and Genericity (R4).

7 Textual Front End for Incremental Program Analysis

The contents of this chapter has not been published yet at a peer-reviewed conference.

Abstract — In the front end of IncA, we relied on projectional editing so far. This aligned perfectly with incrementalization, as AST differences can be computed with zero computational overhead in a projectional editor. The goal of this chapter is to enable IncA in textual IDEs, as well, given that they can be considered mainstream in comparison to projectional IDEs. However, computing AST differences in a textual IDE incurs computational overhead. Given that IncA analyses also incur a sub-second update time, we only have at most a few tens of milliseconds for computing AST differences after program changes, otherwise developers will notice pauses in their workflow when using an incremental IncA analysis.

Computing AST differences in a textual IDE entails two steps: parsing and AST differencing. Perhaps surprisingly, we found that existing parsers can deliver millisecond re-parse times. However, most of the existing AST differencing tools are not satisfactory for our purposes, as they are either slow, only work with a particular subject language, or encode AST differences imprecisely. Only recently did Miraldo et al. develop an AST diffing tool called *hdiff* that does not have any of these three limitations. We adopt *hdiff* in this chapter and use it to build a textual front end for IncA. Our benchmarking with real-world Python code reveals that our approach can compute AST differences in a few tens of milliseconds, thereby enabling IncA analyses in textual IDEs.

7.1 Introduction

The main goal of IncA is to enable the use of static analyses for interactive applications in IDEs. So far, in the front end of IncA, we used projectional editing. This aligned perfectly with our goal, as IncA relies on fine-grained AST change notifications, and a projectional editor can deliver these with zero computational cost after program changes. However, projectional editing can be considered a niche editor technology compared to the widespread use of text editors in IDEs. Given that we would like to make our contributions widely applicable, we explore in this chapter how to use IncA with a textual front end.

Unfortunately, our back end contributions do not apply immediately to IDEs with textual front ends. The problem is that the IncA back end only knows how to react to AST differences, but developers in a textual IDE modify the contents of a text buffer. This means that there is a gap between what IncA expects and how the IDE represents a subject program. To turn textual code changes into AST differences, an IDE must first parse the contents of the text buffer and then compute the differences between the old and new version of the AST. Unlike in a projectional editor, this process incurs extra computational cost. If the process is too slow, then that puts the performance of the entire incremental analysis pipeline in danger, which ultimately can lead to the dreaded noticeable pauses for the developer working in the IDE. One may immediately think that there is no challenge here at all, as both parsing and tree diffing are well studied in the literature. We found that state-of-the-art parsers are indeed really efficient, as they can re-parse files in milliseconds. However, we also found that most state-of-the-art tree diffing tools yield unsatisfactory results because of one or a combination of the following reasons:

- The tool is slow at computing diffs. We experimented with tree diffing tools and found that some of them require more than a second to compute the diff for a single file. This is a problem because AST differences must be computed in the ballpark of milliseconds, as together with an incremental analysis, the update time must remain in the sub-second ballpark (cf. Efficiency (R2)).
- The tool describes AST changes only in terms of insertions and deletions. This is unsatisfactory because moved subtrees must be completely deleted and re-inserted. For example, inserting a statement at the beginning of a method body would yield a large AST change, where all subsequent statements are deleted and then re-inserted. In turn, this may result in excess work in the followup incremental analysis, which then increases the update time (cf. Efficiency (R2)).
- The tool is tailored to a particular subject language. This is a problem because IncA is a language-independent analysis framework, so its front end must also be independent of the subject language (cf. Genericity (R4)).

Only recently, did Miraldo et al. present a tree diffing algorithm called *hdiff* that can also find and encode moved subtrees [85]. The key idea of *hdiff* is to assign cryptographic hashes to subtrees in order to identify clones efficiently. Importantly, *hdiff* is a generic algorithm that can compute diffs for arbitrary typed trees. In this chapter, we port *hdiff* from Haskell to Scala. We call our implementation $hdiff_S$. We show how to communicate the output of $hdiff_S$ to the back end of IncA, thereby enabling efficient incremental program analysis with a textual front end. Given that the work presented in this chapter is still in a preliminary state, we also discuss a number of next steps that revolve around the specialization of *hdiff*$_S$ to incremental analysis pipelines.

To evaluate our approach, we build a complete textual front end, and we benchmark its performance with Python code. We use an existing parser called fastparse[1] to parse Python files, and we employ *hdiff*$_S$ to compute AST differences. Then, we use a series of commits from the popular django web framework[2] written in Python and compute AST differences between consecutive versions. We find that our approach (parsing + diffing) can compute AST differences in a few milliseconds, which is exactly the kind of run time we need in the front end of IncA. We have already demonstrated in previous chapters that the IncA back end can deliver sub-second update times. Based on these two results, we conclude that we can deliver continuous feedback in textual IDEs with IncA.

Contributions In summary, we make the following contributions:

- We identify requirements for a textual front end in order to enable efficient incremental IncA analyses. We review if and how state-of-the-art parsers and tree diffing tools can meet these requirements (Section 7.2).
- We port the original *hdiff* algorithm from Haskell to Scala. We use our implementation *hdiff*$_S$ to demonstrate the key ideas of the original *hdiff* algorithm (Section 7.3).
- We translate the output of *hdiff*$_S$ to updates in EDB relations, thereby enabling incremental IncA analyses on top of the new textual front end (Section 7.4).
- We benchmark the performance of our new textual front end based on *hdiff*$_S$ with real-world Python code (Section 7.5).

7.2 Requirements for Textual Front Ends and Prior Work

In this section, we revisit the architecture of IncA, but, this time, with a textual front end. We discuss the high-level components of the new front end, and we formulate requirements that these components must satisfy in order to enable efficient incremental analyses. We review state-of-the-art techniques that could be used to build a textual front end and discuss if and how they satisfy our requirements.

Textual Front End for IncA Figure 7.1 shows an excerpt of the IncA architecture; We already introduced the complete high-level architecture in Figure 1.3. Here, we only show the front- and back end, and we drill into the front end to show its main components. Imagine a scenario where the IncA back end is incrementally maintaining the result of a program analysis. Instead of working with a projectional editor, developers now interact with a text buffer in the IDE to edit a subject program. Assume that there is an old version of the text and a new version after a program change. We can describe the change between the old and new version of the code in terms of text diffs, e.g. inserted/deleted lines or ranges of characters. However, the IncA back end works with AST diffs and not with textual diffs. To bridge this gap, we use a combination of a parser and an AST diffing tool. The parser takes the contents of the text buffer and constructs the AST of the subject

[1]`https://github.com/lihaoyi/fastparse`
[2]`https://github.com/django/django`

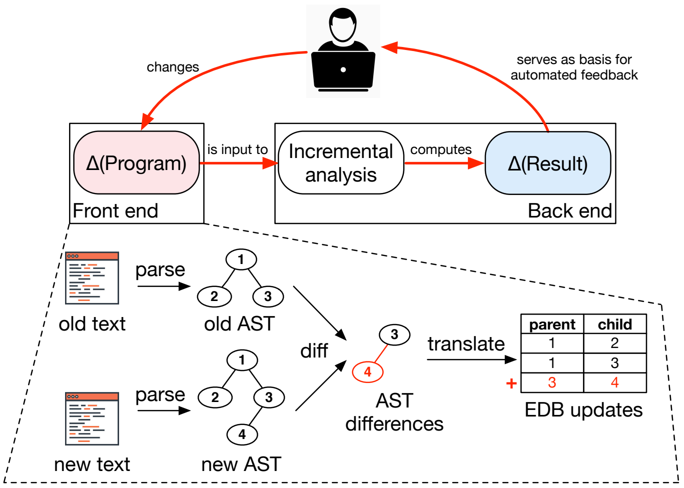


Figure 7.1: Textual front end with its components integrated into the architecture of IncA.

program. For now, we are not interested in how this AST is constructed after a program change; Whether the old AST is incrementally updated via an incremental parser or the new AST is created from scratch. We just assume that there is an old and a new AST. Then, the AST differ computes the differences between the two ASTs to yield a change set that consists of e.g. node/edge insertions/deletions, moves, or updates. We use this changeset and translate it to insertions and deletions in EDB relations that describe the subject program and the changes therein for IncA analyses.

Requirements for Textual Front Ends In Figure 1.3, we associated two requirements with the front end of IncA: Efficiency (R2) and Genericity (R4). We revisit and refine what they mean for a textual front end.

Genericity (R4): The textual front end (both parser + AST differ) must be independent of any particular subject program or subject language. This is to ensure that IncA itself remains generic, as the back end already satisfies this requirement.

Efficiency (R2): In Section 1.4, we stated that we expect to deliver the sub-second update times with incremental analyses, as we target applications in IDEs. This entails two specific criteria for the textual front end:

- It must compute AST diffs efficiently with a run time that is in the ballpark of at most a few tens of milliseconds. So far, with the projectional front end, we did not need to worry about this criteria, as a projectional editor derives AST diffs with zero overhead. This is not true anymore in case of a textual front end because both parsing and AST diffing incur computational overhead. If the time required to compute the AST diffs is too long, we cannot meet the timeliness requirement. As we demonstrated in previous chapters, the IncA back end is efficient, as it provides sub-second update times on average, but to preserve this, the front end must efficiently compute AST diffs.
- The AST diffs that the front end computes must be as precise as possible and free of redundant change operations. This is an important requirement to ensure the

efficiency of the follow-up incremental analysis. For example, imagine that we insert a statement at the beginning of a method body, and we incrementalize an analysis with IncA. If an AST diffing tool only reasons about insertions and deletions, then the resulting AST diff would be large, as all subsequent statements and their subtrees are deleted and then re-inserted. In turn, the IncA analysis must invalidate and then re-compute a large part of the analysis result. If instead, the diff also reasons about moves, then it could precisely describe the shifting of the statements without touching any of the subtrees rooted at the statements.

Next, we review state-of-the-art parsing and AST diffing techniques to see if they satisfy these requirements. Our goal is to find existing tools that we could use to build a textual front end for IncA. We performed small experiments with many of the following tools. We performed the benchmarks on an Intel Core i7-6820HQ at 2.7 GHz with 16 GB of RAM and Java 11.0.5.

State-of-the-art parsers Parsing is a widely researched area, and there is a myriad of different frameworks readily available. We sample here only a select few frameworks. First, we consider the following non-incremental parsers:

- ANTLR[3] is a parser generator framework that is widely used both in industry and academia. Parr et al. benchmarked the performance of ANTLR for various languages (e.g. Erlang, Lua, Verilog), and they found that (i) the parse time is on the ballpark of a few tens of milliseconds for practically relevant file sizes, and (ii) the relationship between file size and parse time is linear [91].
- fastparse is a parser generator framework for Scala. We used fastparse version 2.2.2 with a Python grammar to parse the source of the django web framework, and we found that the average parse time is around 10 ms. See more details in Section 7.5.
- JavaParser[4] is a parser for Java source code. We benchmarked the performance of JavaParser version 3.15.17 by parsing the sources of JavaParser itself (as the framework is written in Java). We found that the average parse time is around 6 ms, while the maximum parse time is 330 ms. A limitation of this tool is that it is tailored to Java, while the previous tools are generic frameworks.

Based on these results, we conclude that state-of-the-art parser frameworks are already fast enough to be used in a textual front end for IncA.

Next, we consider incremental parsing. In contrast to re-parsing the textual subject program repeatedly, an incremental parser constructs the whole AST once for an initial subject program, and then it incrementally updates parts of the AST in response to text changes. Assuming that the parser lets us extract the AST updates amid incremental parsing, this approach aligns perfectly with our use case in IncA. A popular incremental parsing framework is tree-sitter.[5] According to its authors, tree-sitter is generic, as it can generate an incremental parser from a user-defined grammar, and it so fast that it can

[3]https://www.antlr.org/
[4]https://javaparser.org
[5]https://github.com/tree-sitter/tree-sitter

simply re-prase files after every keystroke without interrupting the development flow in the IDE. Let us briefly look at the C APIs tree-sitter offers for incremental parsing:[6]

```
TSTree *ts_parser_parse(
  TSParser *self,
  const TSTree *old_tree,
  TSInput input
);

void ts_tree_edit(TSTree *self, const TSInputEdit *edit);
```

`ts_parser_parse` parses some text input and creates an AST. If the `old_tree` parameter is `null`, then tree-sitter constructs an AST from scratch. If the `old_tree` is not `null`, then incremental parsing takes place, and the resulting AST shares unchanged parts with the old AST. Before triggering an incremental parse, the `ts_tree_edit` function can be used to communicate changes in the input text with tree-sitter. An important next step is to extract the AST diffs after an incremental parse. However, to our surprise, we found that there is no way to extract this information, as tree-sitter supports only the following API:

```
TSRange *ts_tree_get_changed_ranges(
  const TSTree *old_tree,
  const TSTree *new_tree,
  uint32_t *length
);
```

The `ts_tree_get_changed_ranges` function takes the old and the new ASTs and returns the ranges of positions in the *input* text for which the AST structure has changed. Note that this information in itself is useful, as it is not necessarily the case that every text change communicated with `ts_tree_edit` actually results in an AST change. However, this also means that tree-sitter does not tell what the actual AST diffs are. Unfortunately, this is not sufficient for our purposes in IncA. Given that we must find a way to extract AST diffs in the front end, we now look at AST diffing techniques.

State-of-the-art AST diffing tools Just like parsing, AST diffing has also attracted a lot of research attention. We experimented with a number of frameworks, and we briefly report on our experience.

JSON Patch[7] is a standard defined by the Internet Engineering Task Force that defines how to represent diffs between two JSON documents, so that the diff can be an input to an HTTP PATCH operation to apply modifications to a resource. This standard is also interesting for our purposes because JSON is a popular choice for describing structured data, including ASTs, so we could use JSON patches to encode AST diffs. Importantly, the standard specifies not only insertion or deletion as potential change operation in a patch, but also moves.

The tool json-patch[8] is an actual Java implementation of the JSON Patch standard. We used version 1.13 for our experiments. We found that a limitation of json-patch is that it cannot always identify moves effectively, and often it just resorts to reasoning about

[6] `https://github.com/tree-sitter/tree-sitter/blob/master/lib/include/tree_sitter/api.h`
[7] `https://tools.ietf.org/html/rfc6902`
[8] `https://github.com/java-json-tools/json-patch`

insertions and deletions. Consider the following two versions of the same JSON document describing the AST of a function. The old version is shown on the left, and the new version is on the right where a new statement was inserted:

```
{
  "name" : "foo",
  "stmts" : [
    {
      "kind" : "VarDecl",
      "name" : "a",
      "value" : {
        "kind" : "PlusExpr",
        "lhs" : "1",
        "rhs" : "2"
      }
    },
    {
      "kind" : "Print",
      "exp" : {
        "kind" : "VarRef",
        "name" : "a"
      }
    }
  ]
}
```

```
{
  "name" : "foo",
  "stmts" : [
    {
      "kind" : "VarDecl",
      "name" : "b",
      "value" : {
        "kind" : "NumLit",
        "value" : "3"
      }
    },
    {
      "kind" : "VarDecl",
      "name" : "a",
      "value" : {
        "kind" : "PlusExpr",
        "lhs" : "1",
        "rhs" : "2"
      }
    },
    {
      "kind" : "Print",
      "exp" : {
        "kind" : "VarRef",
        "name" : "a"
      }
    }
  ]
}
```

json-patch computes the following diff between the above two JSON trees:

```
1  replace; path: "/stmts/0/name"; value: "b",
2  remove; path: "/stmts/0/value/lhs",
3  remove; path: "/stmts/0/value/rhs",
4  add; path: "/stmts/0/value/value"; value: "3",
5  replace; path: "/stmts/0/value/kind"; value: "NumLit",
6  remove; path: "/stmts/1/exp",
7  add; path: "/stmts/1/name"; value: "a",
8  add; path: "/stmts/1/value";
9       value: {"kind":"PlusExpr","lhs":"1","rhs":"2"},
10 replace; path: "/stmts/1/kind"; value: "VarDecl",
11 add; path: "/stmts/-";
12      value: {"kind":"Print","exp":{"kind":"VarRef","name":"a"}}]
```

Lines 1-5 update the subtree rooted at the first statement through a series of insertions, deletions, and replacements (which also translate to deletion-insertion pairs). Lines 6-10 do the same for the second statement, while lines 11-12 append the entire subtree rooted at the print statement to the list of statements. There is a lot of excess change operations in this diff; If we were to run an IncA analysis on the output of json-patch, IncA would essentially invalidate most of the analysis result as most of the nodes get deleted, and then it would re-compute the results based on the new AST. Ideally, the shifting of the original two statements should be described with two moves and no change at all to the nodes in their subtrees, thereby allowing efficient incrementalization.

Next, we considered AST differs that reason about moves, as well. To gauge the performance of the tools, we did a simple experiment. We took the source code of JavaParser, and we computed diffs for each file against itself, without introducing a change. First, we considered GumTree, which is a generic AST diffing framework [37]. We used version 2.1.2, and we found that the average diffing time is around 1.4s without any actual changes in the files, which is not fast enough for our purposes. Then, we looked at two other tools that were tailored to Java to see if the specialization brings any performance advantage. We considered GumTree Spoon (version 1.24), which is a version of GumTree that is specialized for Java,[9] and we experimented with ChangeDistiller [38].[10] We found that ChangeDistiller is fast, as the average diffing time is 5ms, while the maximum time is 230ms. GumTree Spoon was slightly slower with an average time of 164ms and a maximum time of 1.1s. Even though, these numbers are promising, we continued looking for other language-independent tools.

Finally, we considered *hdiff* from Miraldo et al. [85],[11] which is a fairly new tool, as it was introduced in 2017. *hdiff* is an AST diffing library applicable to arbitrary tree-like data structures, and it can also reason about moves. Miraldo et al. benchmarked the performance of *hdiff* on lua software projects, and they reported max 200ms diffing times for ASTs with up to 10,000 nodes. This number is particularly promising because, when we benchmarked fastparse with Python on django, we found that the average AST size per file is around 2,366 nodes. This means that *hdiff* has the potential to satisfy both our requirements.

In sum, AST diffing seemed to be more challenging when it came to meeting our requirements for a generic and efficient front end for incremental program analysis. *hdiff* seemed to be the only approach that we could use for our purposes. We decided to employ *hdiff* and build a textual front end for IncA with it. In the following, we port *hdiff* from its original Haskell implementation to Scala. We call our implementation *hdiff*$_S$. We introduce the main algorithmic concepts through our *hdiff*$_S$ implementation. Then, we explain how to translate the output of *hdiff*$_S$ to the back end of IncA. Our experiment was actually successful: As we will report later in this chapter, our approach can compute AST diffs for Python files in a few tens of milliseconds.

7.3 AST Differencing with *hdiff*$_S$

In this section, we present our own implementation of the original *hdiff* algorithm designed by Miraldo et al.. The original algorithm was implemented in Haskell. We re-implemented the algorithm in an object-oriented style in Scala, and we call our implementation *hdiff*$_S$.

Diff representation *hdiff*$_S$ is a generic algorithm that can compute AST differences between arbitrary tree data structures. We use a running example of 2-3 trees throughout this section. In this data structure, every intermediate node has 2 or 3 child nodes, and

[9] `https://github.com/SpoonLabs/gumtree-spoon-ast-diff/`

[10] Built from sources with the last commit being *feee5be* in `https://bitbucket.org/sealuzh/tools-changedistiller/src/master/`.

[11] `https://github.com/VictorCMiraldo/hdiff`

leaves simply store a string value. The definition in Scala looks as follows:

```
trait Tree23
class Leaf(value : String) extends Tree23
class Node2(t1 : Tree23, t2 : Tree23) extends Tree23
class Node3(t1 : Tree23, t2 : Tree23, t3 : Tree23) extends Tree23
```

Figure 7.2 A shows two instances of `Tree23`. We are interested in computing the differences between these two trees. The diff computed by *hdiff$_S$* is determined by the common subtrees that need to be copied over from source to target, potentially with extra permutation or duplication. Concretely, a diff is represented as a pair of deletion-insertion contexts, as shown in Figure 7.2 B. The deletion context binds subtrees of the source AST to so called *meta variables*. Intuitively, the deletion context can be thought of as a pattern to be matched against the source AST. The insertion context yields the target AST when its meta variables are substituted by the respective subtrees bound by the deletion context. In Figure 7.2 B, the deletion context binds meta variable 0 to `Leaf(‘‘c’’)` and meta variable 1 to `Node2(Leaf(‘‘a’’), Leaf(‘‘b’’))`. The insertion context then reuses these meta variables to efficiently encode the swap of the two subtrees. Had we used only insertions and deletions, the diff would have deleted and re-inserted the complete subtrees.

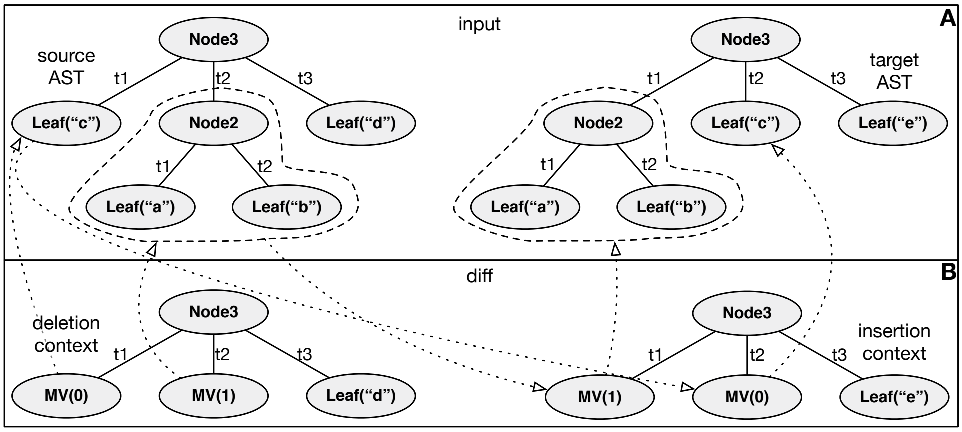


Figure 7.2: (A) Example 2-3 trees used as input to *hdiff$_S$* and (B) the resulting diff. The dashed arrows show how the deletion context binds meta variables to subtrees in the source AST and how the meta variables are substituted with those subtrees in the insertion context. Edges in the AST are labeled with the name of the containment link.

We model meta variables in Scala by introducing a “hole” for them in the 2-3 tree. A context is a 2-3 tree which also allows meta variable holes. Finally, a diff consists of a pair of contexts:

```
type Context[T] = T // type def for convenience
class Tree23MetaVarHole(mv : MetaVar) extends Tree23
class MetaVar(id : Int)
trait Tree23Diff(delCtx : Context[Tree23], insCtx : Context[Tree23])
```

The type def for `Context` may seem useless, but it helps in the following to identify what kind of data a piece of the diffing logic deals with. Using this API, the diff between the source and target ASTs from Figure 7.2 A is described as follows:

```
Tree23Diff(
  Node3(
    Tree23MetaVarHole(MetaVar(0)),
    Tree23MetaVarHole(MetaVar(1)),
    Leaf("d")
  ),
  Node3(
    Tree23MetaVarHole(MetaVar(1)),
    Tree23MetaVarHole(MetaVar(0)),
    Leaf("e")
  )
)
```

Next, we discuss how to identify common subtrees among the source and target ASTs.

Identifying common subtrees *hdiff$_S$* decorates ASTs with cryptographic hashes in the form of a Merkle tree [84]. Figure 7.3 shows this for an example 2-3 tree. The hash of a node is composed of the hash of its constructor kind plus the hash of its children. In case of leaf nodes, we also hash the string value. Finding common subtrees then boils down to finding nodes with the same hash value in the source and target Merkle trees, as they correspond to the roots of identical subtrees.

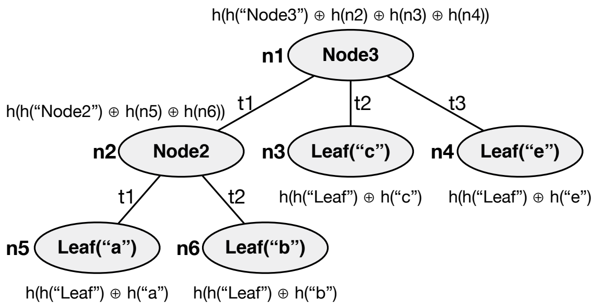


Figure 7.3: AST decorated by *hdiff$_S$* with cryptographic hashes. Nodes are labeled with a unique identifier to refer to them in the hash computation in the parent node. Symbol ⊕ represents an operator used to compose hashes.

To predict which subtrees are common between two trees, we implement an oracle as shown in Figure 7.4. `HashingOracle` takes the source and target ASTs as constructor arguments. It uses a trie data structure to provide efficient lookups of subtrees based on hashes. The map `srcTrie` is used to map hashes to subtrees from the source AST, while `intersectTrie` will contain mappings for the common subtrees. In lines 5-8, we iterate over all nodes of the source AST, and we insert a mapping to the trie where the key is the hash of the node and the value is a fresh meta variable. Note that we extended here the class `MetaVar` to not only store the integer id, but also a reference to the root of the subtree it represents. This will be important later when we substitute meta variables with subtrees in the insertion context. In lines 9-15, we iterate over the nodes of the target AST to find all those nodes whose hashes are already mapped in `srcTrie`. These nodes will be stored in the `intersectTrie`, as they represent the common subtrees. The `predict` function takes a subtree and returns the associated meta variable if the subtree is shared among the two ASTs, otherwise it returns `None`. The perceived reader may wonder if hash collisions

```
class HashingOracle(src: Tree23, trg: Tree23) {
  val srcTrie = Trie[MetaVar]()
  val intersectTrie = Trie[MetaVar]()
  var count = 0
  src.allNodes.foreach(node => {
    srcTrie.put(node.hash, MetaVar(count, node))
    count += 1
  })
  trg.allNodes.foreach(node => {
    val key = node.hash
    val mv = srcTrie.get(key)
    if (mv != null) {
      intersectTrie.put(key, mv)
    }
  })

  def predict(node: Tree23): Option[MetaVar] = {
    Option(intersectTrie(node.hash))
  }
}
```

Figure 7.4: The implementation of the oracle that predicts which subtrees are common between two trees.

can ever happen with this approach: Miraldo et al. propose to use strong cryptographic hash functions (e.g. SHA-256), so the chance for a potential collision becomes negligible.

Computing diffs Using the above oracle, we use a `diff` function to compute the diffs between two trees as shown in Figure 7.5. In lines 5-6, we prepare the deletion and insertion contexts using the oracle. Function `extract` is a polymorphic function that is overridden by the different `Tree23` subclasses. For example, for `Node2`, it looks as follows:

```
override def extract(oracle: HashingOracle): Context[Tree23] =
  oracle.predict(this) match {
    case Some(mv) => Tree23MetaVarHole(mv)
    case _ => Node2(t1.extract(oracle), t2.extract(oracle))
  }
```

```
1  class HashingDiffer {
2    def diff(src : Tree23, trg: Tree23): Tree23Diff = {
3      val oracle = HashingOracle(src, trg)
4
5      val delCtx = src.extract(oracle)
6      val insCtx = trg.extract(oracle)
7
8      val commonMetaVars = delCtx.metaVars intersect insCtx.metaVars
9      val postDelCtx = delCtx.retainMetaVars(commonMetaVars, src)
10     val postInsCtx = insCtx.retainMetaVars(commonMetaVars, trg)
11
12     Tree23Diff(postDelCtx, postInsCtx)
13   }
14 }
```

Figure 7.5: Implementation of the `diff` function that computes the differences between two trees.

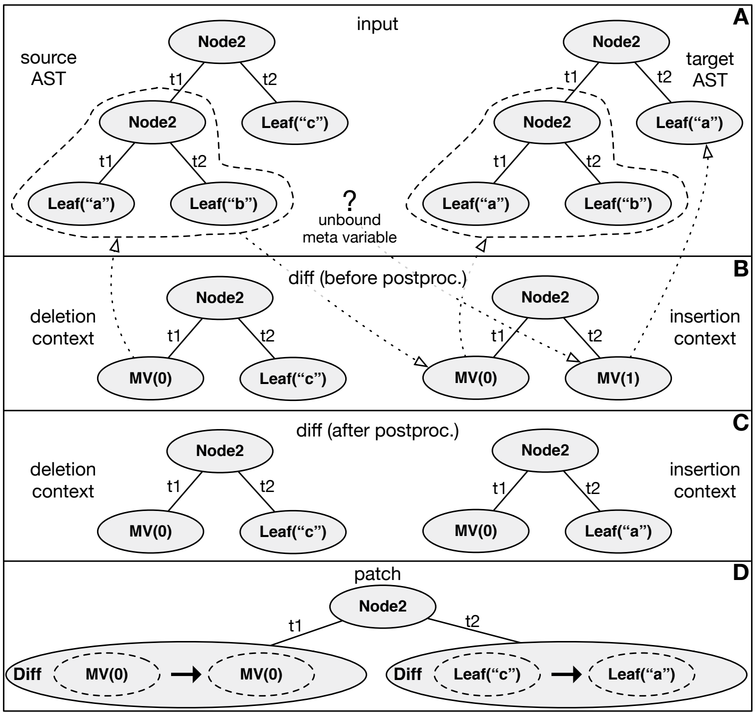


Figure 7.6: Steps of obtaining a patch. (A) Input ASTs. (B) Diff before postprocessing the deletion and insertion contexts. (C) Postprocessed diff where the unbound meta variable `MV(1)` is replaced by its subtree. (D) Patch with common spine extracted as prefix. Diff holes display their two contexts as trees with dotted lines.

If the oracle predicts that the current subtree is common between the source and target ASTs, then `extract` returns a meta variable hole with the respective meta variable, otherwise it continues looking for shared subtrees recursively among the children.

In lines 8-10 in Figure 7.5, we perform postprocessing on the two contexts. This step is necessary because the extracted contexts may contain meta variables that only appear in one of the contexts. A meta variable only appearing in a deletion context corresponds to a subtree that will not actually be copied, while a meta variable only appearing in an insertion context is an unbound variable. Consider the example in Figure 7.6 A-B. The oracle associates meta variable `MV(0)` with the `Node2` subtree and meta variable `MV(1)` with `Leaf(‘‘a’’)`, as these appear both in the source and target ASTs. However, meta variable `MV(1)` only appears in the insertion context. This is because the `Node2` subtree contains `Leaf(‘‘a’’)` in the source AST. In turn, the deletion context only binds meta variable `MV(0)`. To eliminate unbound meta variables, line 8 in Figure 7.5 computes the intersection of the variables that appear in the two contexts. The calls to `retainMetaVars` in lines 9-10 make sure that all variables that are not common in the two contexts get replaced by the subtrees that the oracle associated with them. For our example, this replaces `MV(1)` with `Leaf(‘‘a’’)` in the insertion context as shown in Figure 7.6 C.

Computing patches to minimize diffs Although the diff shown in Figure 7.6 C captures the differences between the two trees through the two contexts, it contains redundant information. Specifically, the constructor `Node2` appears at the roots of both the deletion and insertion contexts. For real-world subject programs where the ASTs are much larger, the amount of redundant parts can be significantly larger. To make the diff representation more compact, *hdiff*$_S$ computes a so called *patch* from the diff. The main idea is to extract the common constructors as a prefix forming the *spine* of the patch and then leave the actual diffs to the leaves of the patch. Figure 7.6 D shows the patch computed from the diff in Figure 7.6 C. The `Node2` constructor forms the spine, the left leaf encodes the identity change, while the right leaf is a replacement.

First, we model patches in the form of Scala classes and then briefly discuss how to compute them:

```scala
type Patch[T] = T // type def for convenience
class Tree23DiffHole(diff : Tree23Diff) extends Tree23
```

`Tree23DiffHole` is a `Tree23` in itself (just like `Tree23MetaVarHole`), and it wraps a diff. We introduce a polymorphic function called `createPatch` which is overridden by all `Tree23` subclasses. For example, the implementation for `Node2` looks as follows:

```scala
1  override def createPatch(other: Context[Tree23]): Patch[Tree23] =
2    other match {
3      case Node2(t1, t2) =>
4        try {
5          val p1 = this.t1.createPatch(t1)
6          val p2 = this.t2.createPatch(t2)
7          Node2(p1, p2)
8        } catch {
9          case e: UnboundMetaVarException =>
10           Tree23DiffHole.create(this, other)
11       }
12     case _ => Tree23DiffHole.create(this, other)
13   }
```

If the two trees (`this` and `other`) share the same constructor, then we extract that, and we recursively build the spine in the children nodes. If the constructors are different, we create a diff hole with `Tree23DiffHole.create`, which wraps the two trees that represent the deletion and insertion contexts. These contexts may contain `Tree23MetaVarHoles`. Lines 9-10 catch `UnboundMetaVarException`, which is an important sanity check. The exception may be thrown by nested calls to `Tree23DiffHole.create` in children nodes. The function throws the exception if the insertion context contains a meta variable that does not appear in the deletion context. This is a problem because such variables do not get bound to a concrete subtree, so the insertion context will not be able to substitute the meta variable. Note that we compute the patch from the postprocessed diff, so there cannot be any unbound meta variables when considering the complete (outermost) contexts. However, as the patch creation recursively descends to children nodes, the contexts shrink, and this may break bindings for meta variables. For example, consider the following patch which has an unbound meta variable in the `t2` child of `Node2`:

```scala
Node2(
  Tree23DiffHole(
    Tree23Diff(
      Tree23MetaVarHole(MetaVar(0)),
      Tree23MetaVarHole(MetaVar(0))
    )
  ),
  Tree23DiffHole(
    Tree23Diff(
      Leaf("a"),
      Tree23MetaVarHole(MetaVar(0)) // unbound meta variable
    )
  )
)
```

To fix the broken binding, the diff must be lifted up, and the spine must shrink to yield the following patch:

```
Tree23DiffHole(
  Tree23Diff(
    Node2(
      Tree23MetaVarHole(MetaVar(0)),
      Tree23MetaVarHole(MetaVar(0))
    ),
    Node2(
      Leaf("a"),
      Tree23MetaVarHole(MetaVar(0)) // bound in lifted context
    )
  )
)
```

The exception handler in `createPatch` lifts up the diff creation as needed, potentially recursively by re-throwing the exception. To make use of `createPatch`, we can simply modify `diff` in Figure 7.5 to not simply return `Tree23Diff(postDelCtx, postInsCtx)` but `postDelCtx.createPatch(postInsCtx)`.

In the original paper describing *hdiff*, Miraldo et al. also describe how to apply patches on an AST, as their primary use case is in computing and merging diffs in a version control system. In IncA, this is different, as our goal is to use the computed patches to incrementally maintain analyses. After computing a patch between a source and target AST, the target AST can simply be considered as the source AST for the next program change. In the following, we explain how to translate a patch computed by *hdiff*$_S$ to updates in EDB relations that IncA analyses use an input.

7.4 Integrating *hdiff*$_S$ with IncA

In this section, we integrate *hdiff*$_S$ with the back end of IncA. First, we recap how IncA represents subject programs with virtual EDB relations. Then, our goal is to use patches computed by *hdiff*$_S$ to incrementally update EDB relations. To this end, we define requirements that our solution must satisfy to perform correct and efficient updates. Finally, we present an actual algorithm that performs the updates while satisfying our requirements.

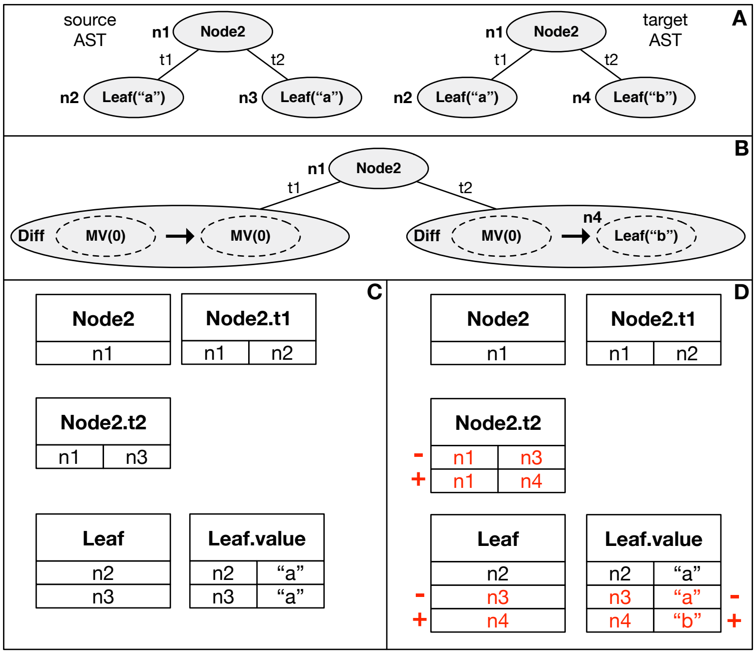


Figure 7.7: Updates in virtual EDB relations as triggered by a patch. (A) Source and target ASTs, (B) patch representing the diffs between them, (C) initial contents of EDB relations, and (D) updated contents of EDB relations after applying the patch. Symbols +/- show which tuples get inserted/deleted. This example demonstrates the problems that lead to requirements Differentiate subtrees of a meta variable (R 7.1) and No dangling trees (R 7.2).

7.4.1 Subject Program as Virtual EDB Relations

As we discussed in Section 3.3, IncA uses virtual EDB relations derived from the AST of a subject program. Unary EDB relations enumerate the instances of AST node types, while binary EDB relations enumerate the source and target nodes of structural links. Consider the 2-3 tree shown on the left of Figure 7.7 A where each node is labeled with a unique identifier. The same AST is encoded as a collection of EDB relations in Figure 7.7 C. Unary relations enumerate the nodes of the 2-3 tree per node type, while each binary relation enumerates the source and target nodes of a containment link in the AST. We initialize these relations through a traversal of the source AST before any program analysis runs.

7.4.2 Requirements for Correct and Efficient EDB Updates

In response to a textual program change, we re-parse the new contents of the text editor in the IDE to obtain a target AST. We take the source and target ASTs and use *hdiffs* to compute a patch between them. Finally, we use the patch to update EDB relations by deleting and inserting affected tuples. However, the original design of *hdiffs* does not immediately lend itself to correct and efficient EDB updates. On the one hand, meta variables make *hdiffs* itself efficient because they help to concisely represent structurally equivalent subtrees. On the other hand, exactly the fact that a single variable is used to represent multiple subtrees makes it difficult to perform EDB updates because we must be able to tell those subtrees apart when inserting or deleting tuples. For example, `n2` and `n3` are represented with the same meta variable in Figure 7.7 B, yet, there are different tuples describing their structure in the EDB in Figure 7.7 C. This observation leads to three requirements that our update solution must satisfy:

Differentiate subtrees of a meta variable (R 7.1): Assume that after a textual program change, we obtain the target AST shown in the right of Figure 7.7 A from the AST shown in the left. $hdiff_S$ computes the patch shown in Figure 7.7 B. The patch uses `MV(0)` to represent three occurrences of `Leaf(‘‘a’’)`, including `n3` in the source AST. The deletion context in the right prescribes the detachment of the subtree represented by `MV(0)`. We intentionally wrote here *detachment* instead of *deletion* because the purpose of meta variables is exactly to reuse subtrees. However, `MV(0)` itself does not tell which occurrence of the subtree should be detached by this deletion context. Both `n2` and `n3` are candidates, but the right one for this context is `n3`. One solution of course is to detach *all* subtrees represented by `MV(0)` when a deletion context prescribes *a* detachment. The insertion context then can just attach the necessary amount of detached subtrees. However, this is inefficient, as this way we may unnecessarily detach and attach some of the subtrees, as would be the case for `n2`, given that it is part of an identity change. Instead, what we need is to differentiate structurally equivalent but otherwise different subtrees represented by the same meta variable. In other words, we must be able to tell the different subtrees represented by `MV(0)` apart. In turn, the tuple `(n1, n3)` must be deleted from `Node2.t2`, as shown in Figure 7.7 D.

No dangling trees (R 7.2): Now that we detached `n3` in the previous step, we are ready to attach it somewhere else. However, we cant, as the insertion context in the right contains `n4`. This means that the subtree represented by `n3` is a dangling tree, which must be garbage collected by our solution. This entails the deletion of `(n3)` from `Leaf` and the deletion of `(n3, ‘‘a’’)` from `Leaf.value`. The other insertions appearing in Figure 7.7 D are due to the insertion of `n4`.

No double attach (R 7.3): Consider now the example in Figure 7.8, which is split up into subfigures just like Figure 7.7. The difference compared to the previous example is that `Leaf(‘‘b’’)` now appears in the source AST, and the target AST contains `Leaf(‘‘a’’)` as both leaves. Like before, `MV(0)` here also represents three occurrences of `Leaf(‘‘a’’)`. The deletion context detaches `n2` and deletes `n3`. However, the problem now is that the insertion context would attach two subtrees represented by `MV(0)`, but there is only one available. We must not attach the same subtree twice, instead a copy of `n2` must be attached. This copy is represented by `n4`.

We also formulate a non-requirement which is related to the changes affecting `n1` and `n1’` in Figure 7.8. Node `n1’` appeared in the patch because the diff must be lifted up to the `Node2` level to avoid `MV(0)` being unbound, and $hdiff_S$ copied `n1`. Ideally, we should preserve `n1` here, but we consider this as an optimization, which we leave to future work. The end result is correct, but it comes with unnecessary changes.

7.4.3 Updating EDB Relations

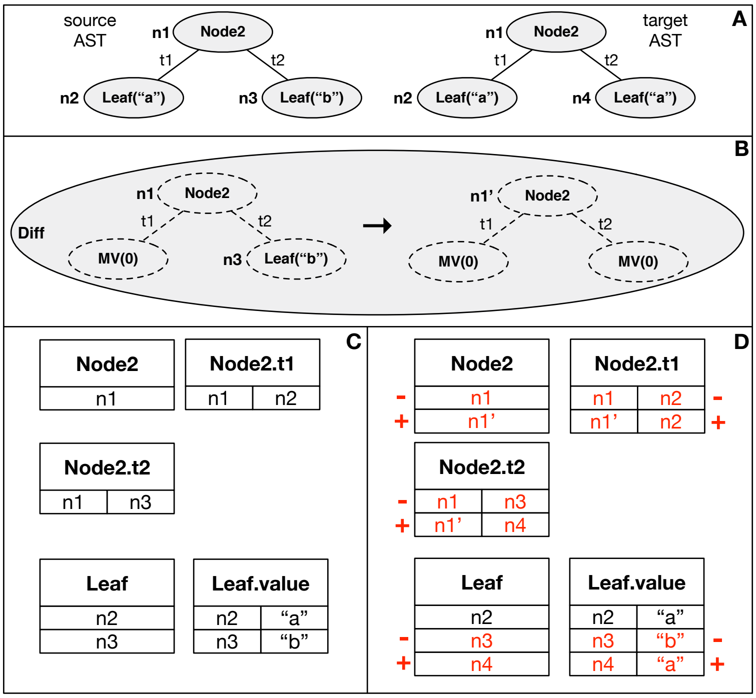
We now present our solution for translating patches to updates in EDB relations. We introduce an EDB trait with the following API for accepting updates:

```
trait EDB {
  def insertUnary(ty : Type, instance : Tree23)
  def deleteUnary(ty : Type, instance : Tree23)
  def insertBinary(link : Link, source : Tree23, target : Tree23)
  def deleteBinary(link : Link, source : Tree23, target : Tree23)
}
```

Type represents an AST node type in the 2-3 tree, while Link is a link in the tree.

The main function of the algorithm that updates EDB relations is shown in Figure 7.9. Function updateEDB takes an EDB instance and a source and target AST and deletes and inserts tuples from EDB relations. There are three main steps of the algorithm: (i) compute the patch with *hdiffs* between the source and target ASTs, (ii) translate the patch to EDB updates, and (iii) clean up dangling trees.

***hdiffs* extensions** We already discussed how *hdiffs* computes patches in Section 7.3. However, in order to integrate *hdiffs* with IncA, we needed to add a key extension to our implementation: Unique identifiers to AST nodes. As we discuss later, being able to uniquely identify nodes is important to tell apart different instances of the structurally equivalent subtrees. Technically, identifiers are implemented as randomly generated Java



Node2
n1

Node2.t1	
n1	n2

Node2.t2	
n1	n3

Leaf
n2
n3

Leaf.value	
n2	“a”
n3	“b”

	Node2
-	n1
+	n1’

Node2.t1		
n1	n2	-
n1’	n2	+

	Node2.t2	
-	n1	n3
+	n1’	n4

	Leaf
	n2
-	n3
+	n4

Leaf.value		
n2	“a”	
n3	“b”	-
n4	“a”	+

Figure 7.8: Example demonstrating the problem that leads to requirement No double attach (R 7.3).

```
1  def updateEDB(instance : EDB, src: Tree23, trg: Tree23): Unit = {
2    // compute patch between src and trg
3    val (patch, mvs) = HashingDiffer.diff(src, trg)
4    // apply patch to EDB instance
5    // (null, null) is the "no context" for root nodes
6    patch.applyPatchTo(instance, null, null)
7    // clean up dangling trees
8    mvs.foreach(mv => {
9      mv.detachedTrees.foreach(tree => {
10       tree.delete(instance, null, null)
11     })
12   })
13 }
```

Figure 7.9: Main function of the update algorithm.

`URIs` set when a `Tree23` node is instantiated. We also make sure that *hdiffs*$_S$ carries along the identifiers of nodes when extracting contexts or computing spines of patches. We use function `diff` to compute a patch in line 3 in Figure 7.9. The API of this function is slightly different than presented in Figure 7.5, as the function now also returns the meta variables from the intersection trie because we use them later in the postprocessing step.

Translating a patch to EDB updates The high-level steps of the patch translation are as follows:

- *Depth-first traversal of the spine until diff holes:* Traverse the patch depth first until diff holes are reached. During traversal, keep track of the parent node and the containment link of a node, as this information will be key to satisfy Differentiate subtrees of a meta variable (R 7.1). When reaching a diff hole, extract its deletion and insertion contexts and update EDB relations based on them.
- *Delete/insert EDB tuples based on deletion/insertion context:* Delete/insert all tuples from EDB relations that describe the structure of the deletion/insertion context. We traverse the contexts depth first.
- *Tree reuse through meta variable holes:* Meta variables represent reused subtrees, so we must only detach/attach the root of the subtrees they represent. When processing a meta variable in a deletion context, we ensure to detach the correct instance of the subtree represented by the meta variable based on the position in the tree, as required by Differentiate subtrees of a meta variable (R 7.1). When substituting meta variables with subtrees in the insertion context, we pay attention to attach the right subtree and to avoid double attaching, as required by No double attach (R 7.3).

We implement this logic now in actual code. We first extend the `Tree23` trait with three new functions, which will be overridden by the different subclasses:

```
trait Tree23 {
  def applyPatchTo(instance: EDB, parent: Tree23, link: Link)
  def delete(instance: EDB, parent: Tree23, link: Link)
  def insert(instance: EDB, parent: Tree23, link: Link)
}
```

`applyPatchTo` performs the depth-first traversal of the spine. The function `delete`/`insert` is responsible for recursively deleting/inserting the tuples represented by a concrete `Tree23`. Line 6 in Figure 7.9 calls `applyPatchTo` with the EDB instance and two null values representing the absence of a parent and containment link of the root node. We review the implementation of the new functions for concrete `Tree23` classes. We start with `Node2`:

```
case class Node2(t1: Tree23, t2: Tree23) extends Tree23 {

  override def applyPatchTo(instance: EDB, parent: Tree23, link: Link) {
    this.t1.applyPatchTo(instance, this, Link(classOf[Node2], "t1"))
    this.t2.applyPatchTo(instance, this, Link(classOf[Node2], "t2"))
  }

  override def delete(instance: EDB, parent: Tree23, link: Link) {
    instance.deleteUnary(Type(classOf[Node2]), this)
    instance.deleteBinary(link, parent, this)
    this.t1.delete(instance, this, Link(classOf[Node2], "t1"))
    this.t2.delete(instance, this, Link(classOf[Node2], "t2"))
  }

  override def insert(instance: EDB, parent: Tree23, link: Link) {
    ... // same as the logic in delete, but with insertions
  }

}
```

`applyPatchTo` descends into the left and right subtrees of `Node2`. In the recursive call, we update the parent node and containment link, so that the children also work with the right containment information. Function `delete` first updates the EDB instance by removing the tuple reporting that `this` is an instance of `Node2`, then it removes the containment link between `parent` and `this`. The contract in our implementation is that the containment link always gets deleted by the child. Finally, we recursively call `delete` on the two subtrees. The implementation of `insert` is similar, but it uses insertions.

Next, we show the implementation for `Leaf`:

```
case class Leaf(value: String) extends Tree23 {

  override def applyPatchTo(instance: EDB, parent: Tree23, link: Link) {
    // noop - end of recursion
  }

  override def delete(instance: EDB, parent: Tree23, link: Link) {
    instance.deleteBinary(link, parent, this)
    instance.deleteUnary(Type(classOf[Leaf]), this)
    instance.deleteBinary(Link(classOf[Leaf], "value"), this, this.value)
  }

  override def insert(instance: EDB, parent: Tree23, link: Link) {
    ... // same as the logic in delete, but with insertions
  }

}
```

There is no need for recursive descend in any of the functions, as `Leaf` represents a leaf node in the AST. Importantly, tuples describing primitive values associated with nodes must be deleted/inserted by the node itself, as we cannot override these methods on the primitive values themselves.

The implementation for `Tree23DiffHole` is special, as it represents a boundary between the spine and a diff:

```scala
class Tree23DiffHole(diff : Tree23Diff) extends Tree23 {

  override def applyPatchTo(instance: EDB, parent: Tree23, link: Link) {
    val dc = this.diff.delCtx
    val ic = this.diff.insCtx
    if (dc.isInstanceOf[Tree23MetaVarHole] && dc == ic) {
      // noop - identity change
    } else {
      dc.delete(instance, parent, link)
      ic.insert(instance, parent, link)
    }
  }

  override def delete(instance: EDB, parent: Tree23, link: Link) {
    throw ApplyPatchFailed("Should not reach this point")
  }

  override def insert(instance: EDB, parent: Tree23, link: Link) {
    throw ApplyPatchFailed("Should not reach this point")
  }

}
```

Function `applyPatchTo` switches from the traversal of the spine to the recursive deletion/insertion of subtrees, recursively passing along the containment information. A special case is when the deletion and insertion contexts represent the same meta variable. This is an identity change, which does not require any processing. Functions `delete` and `insert` will never be called on a `Tree23DiffHole`, as diff holes cannot be nested, and exactly diff holes are the ones that initiate `delete` and `insert` in the first place.

The implementation for `Tree23MetaVarHole` is also special, as we must correctly and efficiently reuse moved subtrees:

```scala
class Tree23MetaVarHole(mv : MetaVar) extends Tree23 {

  override def applyPatchTo(instance: EDB, parent: Tree23, link: Link) {
    throw ApplyPatchFailed("Should not reach this point")
  }

  override def delete(instance: EDB, parent: Tree23, link: Link) {
    val occurrence = mv.findOccurrence(parent.uri, link)
    instance.deleteBinary(link, parent, occurrence)
    mv.detachedTrees.push(occurrence)
  }

  override def insert(instance: EDB, parent: Tree23, link: Link) {
    val (tree, isReused) = this.mv.makeTreeAvailable()
    if (isReused) {
      instance.insertBinary(link, parent, tree)
    } else {
      tree.insert(instance, parent, link)
    }
  }

}
```

First, applyPatchTo will never be called on Tree23MetaVarHoles, as they are contained inside diffs, and we switch to recursive delete/insert calls in diff holes. Function delete finds the right instance of the subtree represented by the meta variable based on the identifier of the parent and the containment link (Differentiate subtrees of a meta variable (R 7.1)). We discuss later how the occurrences are registered. We detach the found subtree and register it on the meta variable as a subtree that can be re-attached elsewhere. Function insert consults the meta variable for a subtree to work with and considers two cases. First, if there is a previously detached subtree (through a previous call of delete), then we only attach the returned subtree. Second, if there is no detached subtree available, then makeTreeAvailable copies the subtree represented by the meta variable, and then we insert the complete subtree recursively. This is to satisfy No double attach (R 7.3).

As a last piece to our algorithm, we show the extensions to the class MetaVar, as it is now responsible for managing multiple subtrees:

```
class MetaVar(id : Int, representedTree : Tree23) {
  val treeOccurrences = mutable.Map[(URI, Link), Tree23]()
  val detachedTrees = mutable.Stack[Tree23]()

  def makeTreeAvailable(): (Tree23, Boolean) = {
    if (detachedTrees.nonEmpty) {
      (detachedTrees.pop, true)
    } else {
      (representedTree.copy, false)
    }
  }

  def registerOccurrence(parent : URI, link : Link, tree : Tree23) {
    treeOccurrences.put((parent, link), tree)
  }

  def findOccurrence(parent : URI, link : Link) : Tree23 = {
    treeOccurrences.get((parent, link)).get
  }
}
```

A meta variable does not only store an identifier and the subtree it represents, but also the concrete structurally-equivalent occurrences of its subtree from the source AST. To this end, we use a map that is keyed by the pair of the parent URI and the containment link. Function registerOccurrence is called by HasingOracle when preparing the source trie. Function findOccurrence is used in the delete function of Tree23MetaVarHole above to look up a concrete occurrence. Finally, makeTreeAvailable gives priority to already detached subtrees, if there is none available in the stack, then it copies the subtree that the meta variable represents and returns that.

Perceived readers may wonder what guarantees that findOccurrence will actually find a subtree occurrence given a parent URI and a link. We reason as follows:

- HasingOracle registers all occurrences of the structurally equivalent subtrees from the source AST under the same meta variable.

- Function `findOccurrence` is only called from `delete` on a `Tree23MetaVarHole`. This can only happen if the algorithm processes a deletion context of a diff hole. Deletion contexts capture the changes in the source AST.
- Given that both previous steps concern the source AST, it is guaranteed that there is an occurrence registered in `treeOccurrences` for a parent URI and link that are passed to the function `findOccurrence`.

Cleaning up dangling trees The last step in our algorithm is the postprocessing in lines 8-12 in Figure 7.9. Here, we delete all those detached trees that have not been reattached when applying a patch, as required by No dangling trees (R 7.2). This can happen if there are more occurrences of a meta variable in deletion contexts than in insertion contexts. Technically, we iterate over the trees in the `detachedTrees` stack of each `MetaVar` and call `delete` on them to recursively delete the tuples that represent their structure. Given that these trees have already been detached, we can safely use null for the parent and link in the initial call.

7.4.4 Language-independent Implementation

Throughout the presentation of *hdiff*$_S$ and our integration, we focused on 2-3 trees. We presented a number of functions (e.g. `extract`, `applyPatchTo`, `delete`, and `insert`) that must be overridden by the different `Tree23` case classes. The implementation simply followed the structure of the respective `Tree23` subclass. In our concrete implementation, we avoid all this, and we support any kind of subject language that is encoded with Scala case classes. To this end, we make use of Scala macros, which are functions called by the Scala compiler at compilation time. In our integration, macros manifest as annotations that analysis developers can apply on the case classes of the language definition. When macros get expanded by the compiler, we rewrite the original Scala classes and inject the implementation of all the required functions. The actual implementation of all *hdiff*$_S$-specific classes, such as meta variable, context, and patch, is in fact parametric over the subject language, as well.

7.5 Evaluation

In this section, we present a benchmark where we employ *hdiff*$_S$ for a concrete programming language. We answer the following research questions:

Ease of Integration (Q 7.1): Can *hdiff*$_S$ be used for a custom subject language with low manual effort?

Run Time (Q 7.2): Can *hdiff*$_S$ deliver AST differences in a few milliseconds? Can it serve as the basis for a textual front end for incremental static analyses?

The research questions primarily concern *hdiff*$_S$ and not a complete textual front end. The reason for this is that we plan to reuse existing parsers, and we already demonstrated in

Section 7.2 that they are already fast enough for our purposes. To this end, we focus on benchmarking *hdiffs*$_S$ itself in this section. Our first question is simply concerned about its general applicability, as the whole reason why we started experimenting with a textual front end was to enable IncA in textual IDEs in a language-independent manner. The second question is about performance, which is of paramount importance, as the time required to compute AST differences directly has an impact on the overall performance of IncA as a whole. We do not measure the memory overhead of *hdiffs*$_S$, as it does not use any stationary memory given that it is a non-incremental algorithm.

There is a number of questions that we do not consider in our evaluation; e.g. the compactness of the computed diffs or the performance of IncA as a whole with a concrete analysis consuming the computed diffs. The reason for this is that the work presented in this chapter is still in an early state: We discuss our planned next steps later. For the benchmark presented in this section, we used a machine with an Intel Core i7 at 2.7 GHz and 16 GB of RAM, running 64-bit OSX 10.15.4, Java 11.0.5, and Scala version 2.13.1.

7.5.1 Evaluating Ease of Integration (Q 7.1)

To experiment with the applicability of our approach, we integrated *hdiffs*$_S$ with Python. The choice for Python had practical reasons: We found the fastparse library which readily implements a Python parser in Scala. As we will show later, the parser is fast out of the box, which is important for our use case, allowing us to focus on the integration with *hdiffs*$_S$. Additionally, the parser represents the Python grammar as Scala case classes, which allows us to directly apply our macro annotations (see Section 7.4.4). Consider the following code snippet from the Python grammar definition with our macro annotations applied:

```
@DiffableType  trait mod
object mod {
  case class Module(body: Seq[stmt]) extends mod
  case class Interactive(body: Seq[stmt]) extends mod
  case class Expression(body: Seq[stmt]) extends mod
}

@DiffableType  trait stmt object stmt {
  case class FunctionDef(...) extends stmt
  case class ClassDef(...) extends stmt
  case class Return(...) extends stmt
  ...
}
```

Indeed, the highlighted annotations are the only changes we performed on the grammar definition. We apply the `DiffableType` annotation on the traits, and the compiler expands the macro on all subclasses, thereby generating all boilerplate code that *hdiffs*$_S$ relies on (as explained in Section 7.4). Considering Ease of Integration (Q 7.1), we conclude that it is straightforward to integrate *hdiffs*$_S$ with Python. However, we acknowledge that this was only possible because (i) a fast parser was readily available and (ii) the parser implementation used Scala case classes to represent the grammar, so we could apply our macro annotations. This may be considered a limitation, but we note that parsers are generally fast enough for our purposes (see Section 7.2) and that Scala is actually a popular choice for implementing parsers, as demonstrated by the fastparse library or the other widely used scala parser combinator framework.[12] As we discuss in Section 7.6, we also have concrete plans to enable easy integration of *hdiff*$_S$ with ANLTR grammars, thereby further improving the applicability of our approach.

Ease of Integration (Q 7.1): Integrating *hdiff*$_S$ with a Scala-based language implementation is as easy as applying macro annotations on the case classes of the grammar.

7.5.2 Evaluating Run Time (Q 7.2)

To evaluate the performance of *hdiff*$_S$, we computed AST differences from textual Python source code. We used the sources of django,[13] which is a popular web framework implemented in Python. The framework comprises roughly 250 KLoC. As a preparatory step, we took the latest 500 commits of the repository, checked out each one of them, and saved a snapshot of all the Python source files in the repository. We flattened out the folder hierarchy in our snapshots by creating file names based on the containing folder hierarchy; e.g. a file `root/a/b/c.py` gets saved as `root/a_b_c.py` in the snapshot. This makes sure that even if there is name clash between files, their original container path distinguishes them. Our actual evaluation happens as follows:

- We take the oldest snapshot and use fastparse to produce a Scala AST from each textual Python file. We store a mapping from file name to the pair of textual file content and the AST.
- We start iterating over the commits from older to newer. For each commit snapshot, we check which files have changed wrt. their textual content. To obtain the old textual file contents, we use the previously introduced map. We only reparse the files that have changed textually. This way we imitate a simple form of incremental parsing. Every time we parse a file with fastparse, we measure how long that takes.
- Given the new AST and the old AST looked up from the map, we use *hdiff*$_S$ to compute the AST differences between them. We measure the time *hdiff*$_S$ takes. Finally, we update the map, so that we carry forward the new AST and textual content.

The following table summarizes our benchmark results:

total number of diffed files	819	
avg. number of diffed files per commit	1.64	
avg./max nodes in the AST	2366.22 (±292 with 95 % confidence)	33116
avg./max parse time (ms)	10.24 (±1.29 with 95 % confidence)	140.00
avg./max diffing time (ms)	1.93 (±0.31 with 95 % confidence)	85.24

The table shows that quite a number of files have changed throughout the 500 commits, but, on average, a commit only affected 1.64 files. The AST sizes are on the same magnitude as the sizes reported by Miraldo et al. in the original *hdiff* paper, as their benchmarking with lua code also showed that the vast majority of the ASTs consisted of at most 10000 nodes [85].

[12]`https://github.com/scala/scala-parser-combinators`
[13]`https://github.com/django/django`

Fastparse is efficient, as it re-parses entire Python files in 10 ms on average. Finally, *hdiffs*$_S$ also delivers the performance IDEs need for interactive applications, as it computes AST differences in a few milliseconds on average. Taking the sum of the maximum parsing and diffing times and multiplying that with the average number of changed files per commit still yields only 369 ms, which shows that our approach can indeed support efficient textual front ends.

Run Time (Q 7.2): We find that *hdiffs*$_S$ computes AST differences in a few milliseconds. Coupled with an efficient parser, *hdiffs*$_S$ is a good fit for building an efficient textual front end for incremental IncA analyses.

7.5.3 Discussion

On the compactness of the AST diffs Our evaluation did not consider the quality of the *hdiffs*$_S$ output, that is, if the AST differences are actually as small or as precise as possible. On the one hand, as explained in Section 7.2, reasoning about moves and efficiency were the primary reasons why we considered adapting *hdiff* in the first place. Many other techniques either failed to deliver on efficiency and/or on the support for moves. This means that *hdiff* (and by that *hdiffs*$_S$) is at the forefront of what is possible in terms of AST diffing. On the other hand, it is not obvious how we could actually reason about the quality of the output of an AST diffing algorithm. We will look at the algorithms behind other AST diffing tools in Section 8.5, but we can already say upfront that different tools use different representations for AST diffs, which makes it difficult to compare outputs. This observation motivates one of our main directions for future work: As we explain in Section 7.6, we plan to design a language for describing AST differences that could be used by different diffing tools as a common format, thereby allowing to compare their outputs.

On benchmarking the entire analysis pipeline In the presented benchmark, we did not actually use an incremental Python analysis consuming the AST diffs computed by *hdiffs*$_S$. We acknowledge that to fully verify the efficiency of the whole incremental analysis pipeline, including AST diffing and incremental maintenance of the analysis, we would need to perform a complete benchmark. However, we already showed throughout this dissertation that the IncA back end is efficient for a variety of incremental program analyses (including inter-procedural analysis), delivering sub-second update times on average. Here, with the textual front end, we add a small overhead of a few tens of milliseconds (including parsing and diffing) on average to the update times of the analysis, so we argue that the overall update time will still be in the sub-second ballpark.

7.6 Future Work

We already implemented a preliminary version of *hdiffs*$_S$ and reported about that in this chapter. Nevertheless, we also identified concrete next steps for the development of *hdiffs*$_S$. We provide a brief overview on our ideas and the planned solution approach.

Adding support for virtual links IncA Datalog provides access to the structure of a subject program's AST through virtual EDB relations. See Figure 3.5 for an overview on the language syntax. The EDB relations enumerate instances of node types and links in the AST. However, there are also virtual links, such as `prev`, `next`, `parent`, and `index`, that are not modeled explicitly in the subject language. IncA still makes these links available because they are fundamental for navigating in the AST. The incremental maintenance of these links required special support also in case of the projectional front end (see Section 3.5), and this is also the case with the textual front end. We plan to extend the logic introduced in Section 7.4 so that when we translate an $hdiff_S$ patch to EDB updates, we also maintain these virtual links between AST nodes. Adding support for `parent` is straightforward, as we can simply extract the old parent from the deletion context and the new parent from the insertion context when processing a diff hole, so we can just update the corresponding two tuples in the EDB instance. The links `prev` and `next` are only applicable for list-typed fields, but then they can be handled similarly as `parent`. The maintenance of the `index` link is expensive (just like with the projectional front end), as inserting or deleting a node in the front of a list shifts the index of all consecutive elements. Just like with the projectional front end, we plan to add support for `index` on demand, meaning that the link would only be maintained in the EDB instance if an analysis actually uses it.

Integration with ANTLR grammars When designing and implementing $hdiff_S$ in Section 7.3 and then the integration with IncA in Section 7.4, we made sure to introduce interfaces for all operations that are specific to the abstract syntax of a particular subject language. Recall all the operations related to extracting meta variables, creating patches, or deleting and inserting tuples representing complete subtrees. Even though, we showed concrete implementation details for a 2-3 tree, $hdiff_S$ can actually generate all of this boilerplate code with Scala macros. In order to extend the applicability of our approach, we plan to implement a similar integration with ANTLR [91], which is a popular parser generator framework. Given the grammar definition of a language, ANTLR can generate all the machinery needed to parse and process textual programs in that language. The generated parsing infrastructure uses Java classes to represent the different terms of the grammar, and we plan to extend the ANTLR generator to also emit all $hdiff_S$-specific boilerplate code. This way we could use $hdiff_S$ to compare ASTs produced by ANTLR. This is an exciting direction for IncA, as ANTLR grammar implementations are readily available for many languages online.[14]

Changeset language with linear type system In an attempt to create a common format to describe the output of AST diffing tools, we plan to design a language describing changesets. We envision the following set of change commands:

```
trait ChangeCmd
class Detach(parent: URI, link: Link, node: URI) extends ChangeCmd
class Attach(parent: URI, link: Link, node: URI) extends ChangeCmd
class Unload(parent: URI, link: Link, node: URI) extends ChangeCmd
class Load(node: URI, children: Iterable[(Link, Any)]) extends ChangeCmd
```

`Detach` and `Attach` correspond to detaching a `node` from and attaching a node to a `parent` in an AST, without any changes to the subtree of `node`. `Unload` is a destructive operation, as it detaches a `node` from its `parent`, plus it also prescribes the removal of the entire subtree rooted at `node`. `Load` is a change command that makes a node available (e.g. by

[14]https://github.com/antlr/grammars-v4

loading it from disk), lazily loading its immediate children. Children are contained under a specific `Link` and they may either be represented by another `URI` (that must be loaded again) or by a literal value. The result of a `Load` must be attached afterwards to the AST. The integration of *hdiff*$_S$ with IncA in Section 7.4 revolved exactly around these kind of operations, so this language is a natural fit to act as an intermediate representation between a patch and the concrete EDB updates.

We formulated three requirements in Section 7.4 that our patch translation algorithm must satisfy. First a requirement concerning efficiency:

1. Differentiate subtrees of a meta variable (R 7.1): Differentiate subtrees of the same meta variable to precisely know which instance to detach/attach.

Then, two safety requirements concerning subtrees that we attach and detach:

2. No dangling trees (R 7.2): No dangling trees to ensure that a detached tree is actually attached later, otherwise a detached tree must be unloaded.
3. No double attach (R 7.3): No double attach to ensure that a detached tree can only be attached once, otherwise it first needs to be loaded.

There is another safety requirement that we did not explicitly spell out before, as we just relied on the correctness of *hdiff* itself. This concerns the resulting AST after applying change commands on the source AST:

4. A changeset must yield a well-formed AST, that is, when the change commands are applied on the source AST, then the resulting AST must conform to the grammar of the subject language wrt. types and the cardinalities of containment links.

Our goal is to design a type system for the changeset language, so that we can automatically verify if these requirements are satisfied by a concrete set of change commands. Our idea is to use a linear type system, which ensures that resources are used exactly once. In our approach, the resource the type system reasons about would be a containment slot (the target of a link) in the AST.

Using an incremental parser to speed up AST differencing Even though, we demonstrated in Section 7.2 that non-incremental parsers are efficient enough to support textual front ends for incremental static analyses, incremental parsers could take our approach to the next level. Even with an incremental parser that reports just about the changed parts of the AST (not the actual changes) after a re-parse, we could speed up our AST differencing further. This is because *hdiff*$_S$ does not actually require the complete ASTs as input, it could also work with two companion subtrees from the source and target ASTs. This way we could significantly reduce the sizes of the compared trees for realistic programs with several thousand AST nodes. We plan to experiment with this idea in the future.

7.7 Chapter Summary

This chapter focused on designing and implementing a textual front end for IncA. We found that we can readily use existing parsers, as they are already efficient enough to re-parse files after program changes in a live setting. For AST differencing, we considered the *hdiff* tool because of its efficiency and because it can also reason about moves. We ported *hdiff* from Haskell to Scala and created $hdiff_S$. We integrated $hdiff_S$ with the IncA back end by translating AST differences to updates in EDB relations. Our benchmarking with real-world Python code revealed that $hdiff_S$ is efficient at computing AST differences. Coupled with an efficient parser, $hdiff_S$ provides a practical solution for building a textual front end for IncA.

From the requirements we formulated in Section 1.4 for IncA, two apply to our new textual front end: Efficiency (R2) and Genericity (R4). Our solution satisfies both of these requirements. Our $hdiff_S$-based front end can deliver AST differences in a few tens of milliseconds on average after textual program changes. Even though, we used Python in our benchmarking, the design of $hdiff_S$ allows easy integration with other subject languages, as well.

8 Related Work

Our discussion of related work is directly governed by our thesis from Section 1.3. We identify three main categories of related work in the context of the first part of our thesis which states *"Incrementalization can significantly improve the performance of sophisticated static analyses"*:

- We survey how other researchers used incrementality to speed up analyses. We look at both one-off incremental algorithms and other incremental analysis frameworks in Section 8.1.
- As incrementalization is only one approach for speeding up static analyses, we also look at other techniques that can help to achieve speedups in Section 8.2.
- Given that Datalog plays a central role in this dissertation, we review state-of-the-art incremental fixpoint algorithms, solvers, and applications of Datalog in Section 8.3.

The second part of our thesis claims *"We can achieve this automatically while shielding analysis developers from the technical details of incrementalization."*. We survey two categories of related work:

- In IncA, we use declarative DSLs to hide the technical details of incrementalization. In Section 8.4, we review what kind of other specification languages researchers proposed for program analyses.
- Incrementalization of static analyses entails an efficient front end that can deliver program deltas efficiently. To this end, we survey tree diffing algorithms in Section 8.5.

Figure 8.1 provides an overview on a number of tools/approaches and the techniques they use for speeding up static analyses. The first row shows IncA, which we characterize as an approach primarily using incrementalization for speeding up static analyses. Additionally, we presented a meta analysis in Section 3.4 which is based on partial evaluation to compute hints for the runtime system about irrelevant program changes. In the last column, we use a full circle, as IncA is an analysis framework that incrementalizes whatever analysis can be expressed with its specification language.

Approach	*Incrementality*	*Graph reachability*	*Compositionality*	*Demand-driven eval.*	*Mixed-precision eval.*	*Partial eval.*	*Parallelism*	*Framework*
IncA (this dissertation)	●	○	○	○	○	●	○	●
Inc. CFLR points-to [77]	●	●	○	○	○	○	○	○
Inc. alias analysis [143]	●	○	○	○	○	○	○	○
Inc. demand-driven points-to [106]	●	○	○	●	○	○	○	○
Inc. MOD analysis [28]	●	○	○	○	○	○	○	○
Inc. model validations [22, 34, 45]	●	○	○	○	○	○	○	●
Task engine in Spoofax [138]	●	○	○	○	○	○	○	●
Set-based analysis frameworks [68, 93]	●	○	○	○	○	○	○	●
Magellan [35]	●	○	○	○	○	○	○	●
Reviser [9]	●	●	○	○	○	○	○	●
Co-contextual type checking [36, 72]	●	○	○	○	○	○	○	●
Inc. attribute grammars [31, 55, 116]	●	○	○	○	○	○	○	●
i3QL [87]	●	○	○	○	○	●	○	●
Self-adjusting computation [4, 52, 53]	●	○	○	○	○	○	○	●
Giga-scale points-to [32]	○	●	○	○	○	○	○	○
Analyses via graph reachability [97]	○	●	○	○	○	○	○	●
Refinement-based points-to [119]	○	●	○	●	●	○	○	○
Cauliflower [61]	○	●	○	○	○	●	●	●
IFDS [98]	○	●	○	○	○	○	○	●
IDE [105]	○	●	○	○	○	○	○	●
Soot [19, 131]	○	●	○	○	○	○	○	●
Heros [19]	○	●	○	○	○	○	●	●
Facebook Flow [25]	○	○	●	○	○	○	●	○
Facebook Infer [24]	○	○	●	○	○	○	●	●
Boomerang [118]	○	●	○	●	○	○	○	○
Just-in-time analysis [33]	○	○	●	●	○	○	○	●
SUPA [121]	○	●	○	●	●	○	○	○
Client-driven points-to [49]	○	○	○	○	●	○	○	○
Combined points-to [145]	○	○	○	○	●	○	○	○
Mixed type inference [92]	○	○	○	○	●	○	○	○
Precision-guided points-to [74]	○	●	○	○	●	○	○	○
Soufflé [66]	○	○	○	○	○	●	●	●
Parallel points-to [83]	○	○	○	○	○	○	●	○
Replication-based points-to [94]	○	○	○	○	○	○	●	○

Figure 8.1: Techniques used for speeding up static analyses. Rows list concrete solution approaches. Columns except for the last one show techniques used for speeding up analyses. Full (empty) circle in a cell means that a given approach uses (does not use) a specific technique. Approaches are grouped by their single fundamental technique, but it is often the case that an approach uses a number of techniques. The last column shows if an approach targets a specific analysis (empty circle) or if it is a framework applicable to a class of analyses (full circle).

8.1 Incremental Static Analysis

This section reviews how researchers use incrementality to speed up one-off static analyses and to build analysis frameworks. It will become clear that the state-of-the-art has limitations either because existing approaches only support specific classes of analyses (e.g. type checking or well-formedness validations) or because they do not support sophisticated analyses that would use a combination of (i) recursive dependencies, (ii) custom lattices and recursive aggregation, or (iii) inter-procedurality.

Incrementalizing individual analyses Lu et al. encode points-to analysis as a CFLR problem and use incrementality to speed up the analysis [77]. CFLR was introduced by Yannakakis [142], and its general idea is as follows. Assume there is an input graph with a set of nodes and edges, where the edges have labels encoding some kind of semantic information relevant to the problem at hand. Paths in this graph yield a sequence of words when we read out the labels of the edges in order. We can encode interesting analysis problems if we only allow paths which yield sentences that are accepted by a context-free grammar. The goal of CFLR is then to find such paths in the input graph via graph reachability. Lu et al. derive an auxiliary graph data structure that encodes information about assignments between variables and heap objects. Then, they define a CFLR problem over this data structure so that accepted sentences encode when a variable can point to an object. After a program change, they incrementally recompute the validity of affected paths to update the solutions of the CFLR problem.

Yur et al. incrementalizes a flow- and context-sensitive alias analysis for C [143]. An alias analysis computes if two pointer variables can point to the same object (which is equivalent to checking if their points-to sets intersect). Their approach assumes that the inter-procedural control flow graph (ICFG) of a subject program is given, and upon program changes, their analysis gets notified about changes in the ICFG. Their approach uses a worklist [90, Chapter 6.1.1] to govern the fixpoint computation of alias information along the ICFG.

Saha and Ramakrishnan define points-to analysis as a logic program using Datalog, and they build on top of the DRed algorithm to incrementalize its execution [106]. Their analysis also employs demand-driven evaluation, which means that it only computes points-to results for the parts of the subject program that are requested by analysis clients, and it defers from eagerly analyzing entire subject programs.

Cooper and Kennedy develop an algorithm for the incremental maintenance of inter-procedural and flow-insensitive MOD analysis [28]. A MOD analysis reports about which variables are affected by which statements in the program. This approach uses a so called “restarting iteration” technique to derive the new fixpoint of the data-flow information after a program change. This means that instead of retracting potentially invalid results after a program change, the analysis starts a new fixpoint computation using the old results as is. It is guaranteed that the newly computed result yields a safe over-approximation of the program behavior, but the result may also exhibit a large number of false positives, as well.

All of the above approaches incrementalize a specific static analysis. In contrast, IncA is an incremental analysis framework that incrementalizes any analysis that can be expressed with its specification language. We look at other incremental analysis frameworks next.

Incremental analysis frameworks Several systems in model-driven development use incrementality to speed up the execution of validation rules on models in face of model changes [22, 34, 45]. However, these approaches can be considered incremental only on a coarse-grained level, as validation rules that are affected by model changes are re-run *non-incrementally* on the *entire* input. This approach is also frequently called selective recomputation. This form of incrementalization works well for this use case, as, typically, validation rules are computationally not that expensive because they can just re-run on select parts of the input model. Contrast this to data-flow analyses that need to run on the whole subject program, where selective re-computation would not be sufficient to provide good incremental performance, and a more fine-grained dependency tracking is needed (like in IncA).

Wachsmuth et al. design a framework for incremental name binding and type checking [138]. The framework works with any subject language implemented in the Spoofax language workbench. The evaluation starts by building an index data structure about information relevant for name binding and type checking, such as scopes, imports, and static type annotations. This data structure is used to execute analysis tasks. The approach is similar to the previous model validations in terms of the degree of incrementality, as tasks are re-executed on a per file basis after program changes.

There is a number frameworks that incrementalize static analyses that operate with the powerset lattice [35, 68, 93]. It is not surprising that this category of analyses attracted a lot of attention, as many analyses having practical applications in IDEs and compilers can be encoded this way. Examples include liveness, very busy expressions, or uninitialized variables analysis [90]. Frameworks using the powerset lattice as the sole abstraction only support set union and intersection as a way to aggregate results, which is in contrast to supporting custom lattices and aggregation as in IncA. Pollock and Soffa present two approaches for incrementalizing powerset-based analyses [93]. First, the IMMEDIATE analysis is used to maintain the results of *may analysis* problems (i.e. when a data-flow value can occur on any control flow path). Second, the TWO_PHASE algorithm is used to compute the results of *must analyses* (i.e. all paths). This algorithm is more complex than IMMEDIATE, and it is analogous to DRed in Datalog because it considers cyclic reinforcements that can appear among the nodes of a strongly connected component in the CFG.

Khedker provides a formal treatment of monotone data flow frameworks and defines requirements against incremental analyses performed on them [68]. Khedker's thesis provides concrete algorithms for the incremental maintenance of analyses using *bitvector-represented* (boolean) lattices, which can also be used to encode powersets. However, it is not possible to efficiently represent arbitrary lattices as bitvectors, as is the case for example for the string prefix lattice we used in Section 4.7 or the k-approximating points-to set in Section 5.6. The potential for generalization is mentioned in Khedker's thesis [68, Chapter 13.2.1], but it is largely left as future work.

The Magellan system encodes program analyses in Prolog, and it uses incremental tabled evaluation to update the analysis result in response to changes in the subject program [35]. The framework is integrated into Eclipse, and it can automatically infer input facts based on the single static assignment form of Java programs. Clients of Magellan analyses can also mark analyses that are not used anymore, and the framework automatically throws away the caches for the affected relations, thereby freeing up memory. Even though, the specification language is Prolog, Magellan actually uses a subset of the language to ensure termination. In terms of the kinds of supported analyses, Magellan targets traditional set-based data-flow analyses. The paper describing the Magellan system also reports about the incremental performance of the system using a real-world subject program. The numbers show that Magellan can deliver sub-second update times for FindBugs-style linter rules or intra-procedural data-flow analyses. The subject program used as input can be represented with 400.000 facts. In contrast, we used subject programs in Section 5.6 that are represented with tens of millions of facts, and we benchmarked an inter-procedural points-to analysis. This shows that our approach significantly improves the performance of sophisticated analyses on large code bases compared to the state of the art.

The Reviser framework incrementalizes IFDS/IDE data-flow analyses [9]. IFDS [98] is a generic framework for expressing inter-procedural, finite, distributive subset problems. The main idea of IFDS is to define flow functions that transfer and compute data-flow facts along the edges of the inter-procedural CFG of a subject program. Technically, evaluating an IFDS analysis boils down to solving a graph reachability problem over the CFG. A requirement of IFDS is that flow functions are distributive over the merge operator; Given values `x` and `y` from an abstract domain and flow function `f`, it must hold for a distributive analysis that $\mathtt{f(x)} \sqcup \mathtt{f(y)} = \mathtt{f(x} \sqcup \mathtt{y)}$. Note that typically we only require from a static analysis that $\mathtt{f(x)} \sqcup \mathtt{f(y)} \sqsubseteq \mathtt{f(x} \sqcup \mathtt{y)}$ holds, so distributive analyses must fulfill a stronger requirement. Given the graph reachability-based formulation, IFDS allows to summarize the effects of flow functions in a procedure by caching reachability information. Such summaries can be reused based on calling contexts of the procedure, thereby improving performance. One caveat though is that summarization only works efficiently if summaries capture only procedure-local information, which is often difficult to achieve [20]. In case of IFDS, the data flow facts computed by the flow functions are elements of the powerset lattice. An extension of IFDS is the IDE [105] framework, which stands for Inter-procedural Distributive Environments (not to be confused with Integrated Development Environment). IDE allows flow functions to compute values from custom lattices, as well. Upon a change at a CFG node, Reviser re-computes the transitive reachability in the CFG to update analysis results. There are a number of differences between IncA and Reviser:

- IFDS (thus Reviser) *requires* that the CFG is provided as input to the analysis. It is often not straightforward to construct and incrementally maintain the (inter-procedural) CFG of a subject program. In many of our case studies, we first needed to use IncA Datalog to define a control flow analysis.
- IFDS/IDE is limited to distributive data-flow analysis problems. IncA imposes no such restriction.
- The primary application area of Reviser is to provide feedback at code reviewing time on pull requests. The update times reported in the Reviser paper are not sufficient for real-time feedback in IDEs, unlike with IncA.

Erdweg et al. design an approach for systematically incrementalizing type checkers [36]. Traditional type checkers typically traverse an AST top down while passing along and extending a typing context with new bindings during the recursive descend. Then, at variables in leaf nodes, the type checker uses the typing context to look up bindings, and then start propagating computed types upwards in the AST. However, this approach can easily hamper efficient incremental execution, as it imposes a lot of dependencies between terms in the syntax tree: A parent expression can only be typed once all of its sub-expressions are typed. The idea of co-contextual type checking as designed by Erdweg et al. is to reverse this approach. A co-contextual type checker actually starts at leaf nodes, and, instead of looking up bindings, it generates context requirements and constraints and propagates them upwards. After a program change, a co-contextual type checker only has to reconsider whether the new context satisfies the existing context requirements and constraints. Based on synthesized benchmarks, Erdweg et al. show that their new approach can achieve an order-of-magnitude speedup compared to the traditional formulation of the type checker.

Unfortunately, co-contextual type checking does not always yield good performance. Problems typically arise when a type checker would require global information (over the whole program) to make a decision, but co-contextual type checking only provides local information when checking a node. For example, this is the case for overload resolution, as typing a function call requires global knowledge about the class table. As we detail in our FTfJP'18 paper [127], we tried out a co-contextual formulation of overload resolution for FJ, and we witnessed that the type checker performs considerably worse than the type checker implemented in IncA. In a followup line of work [72], Kuci et al. actually show that there is a range of other features used in object-oriented languages (e.g. subtype polymorphism, nominal typing, or implementation inheritance) that also pose significant challenges in a co-contextual type checker. They achieve mixed results in terms of incremental performance with their co-contextual type checker. On synthesized benchmarks they report significant slowdowns in initialization time compared to the non-incremental type checker, while, on a real-world benchmark, the incremental performance of the co-contextual type checker degrades compared to the non-incremental type checker.

Attribute grammars [70] are a common way to derive semantic information about a subject program. Attribute grammars use declarative rules to compute values of attributes associated with nodes of the subject program's AST. Attribute values can be of two kinds: synthesized values are computed during traversal, while inherited values are passed down from ancestor nodes in the tree. An early work discussing the incremental evaluation of attribute grammars is from Demers et al. [31]. They show how to incrementalize non-recursively defined attribute grammars. Since then, attribute grammars and their incremental evaluation have seen a number of extensions. For example, Hedin have introduced the notion of reference attribute grammars [57]. References allow to use the values of attributes that originate from outside of the immediate surrounding context (e.g. its parent) of a node. The benefit of reference attribute grammars is the improved expressive power. Söderberg and Hedin also show how to incrementally evaluate reference attribute grammars [116]. Recently, Harkes et al. presented the incremental IceDust system [55], which is based on attribute grammars that use attributes defined through recursive aggregation. A limiting factor of attribute grammars when it comes to using them for static analyses is that ultimately the evaluation is governed by the AST of the subject program. This is different in IncA: It is true that relations computed by derived Datalog rules also relate AST nodes (or lattice values in IncA), but we can define other intermediate representations of the subject program this way. For example, in the case study on the flow-sensitive strong update points-to analysis in Section 4.7, we first implemented an analysis that computed a CFG for the subject program because we needed that for flow-sensitivity, and the original AST is not enough for this purpose. The analysis result then reported about points-to information per CFG node.

i3QL [87] incrementalizes programs developed with an SQL-like language embedded in Scala. There is a number of other similarities between IncA and i3QL:

- i3QL also uses a relational data model and incrementalizes relational algebra operations (e.g. joins, selections, filters).
- Beyond simple aggregation operators like count, min, avg, i3QL supports custom aggregation operators, as well, but only the evaluation of the simple ones are incrementalized. This is different in IncA, as custom aggregation operators are also incrementalized.
- i3QL also uses a DRED-style evaluation schema for recursive computations, just like the baseline version of IncA presented in Chapter 3.
- i3QL also constructs a computation network similar to the one used in Viatra Query in the back end of IncA. IncA uses a meta analysis to optimize incremental evaluation by computing hints about irrelevant program changes, as discussed in Section 3.4. In contrast, i3QL uses partial evaluation to perform optimizations in the computation network, e.g. to tune index selection, push down filter functions, or to share common sub-computations.

Based on all these feature in i3QL, we could have actually used it as the incremental evaluator in the *baseline* version of IncA when we started our research project. However, the original paper about i3QL only reported about benchmarks on small programs, while the performance of Viatra Query was already documented on large inputs [129]. Our back end contributions presented in this dissertation would be applicable to i3QL, as well.

Researchers have also designed systems that use general-purpose programming languages and automatically incrementalize computations developed with those languages. Self-adjusting computation falls into this category [4]. The key language abstraction in self-adjusting computation is a *modifiable reference*, which marks a variable whose contents may be read by the computation and updated by a mutator. Then, self-adjusting computation is about building and maintaining a dynamic dependency graph and computation traces based on the modifiable references. This is in contrast to the computation network used in IncA, which can be built purely based on an analysis definition. The idea of self-adjusting computation has already been implemented as an extension of C in the CEAL system [52] or in the ML-based Adapton system [53]. While, the authors of CEAL and Adapton demonstrated that self-adjusting computation can bring significant speedups compared to a from-scratch execution, the presented benchmarks are typically small in size compared to the subject programs we used in this dissertation. The limitation of self-adjusting computation lie in its generality, as it is difficult to devise optimizations based on a dynamic dependency graph. In contrast, IncA uses relational algebra, for which, well-known optimization techniques exist (e.g. pushing down selections).

8.2 Techniques for Speeding up Static Analyses

While IncA incrementalizes static analyses, we show that this is only one technique out of many that helps to speed up static analyses. As we will see throughout the discussion, many of the presented approaches use a combination of techniques, so we try to group related approaches based on the fundamental technique they use.

Graph reachability We argue that graph reachability and, in particular, computing transitive closure can be considered a technique for speeding up analyses because transitive closure of a directed graph is a fundamental problem in all areas of computer science. There is a range of specialized algorithms and data structures readily available (cf. surveys [23, 64]). In turn, if an analysis can be reduced to a graph reachability problem, then it can benefit from all those algorithms and data structures. We already introduced the idea of context-free language reachability (CFLR) as a way of formulating analyses in Section 8.1. Even outside the context of incrementality, CFLR has attracted a lot of attention.

Dietrich et al. use CFLR for field-sensitive, flow-insensitive, inter-procedural points-to analysis for Java [32]. In this case, the input graph is a points-to graph where the nodes represent the variables and (abstract) heap objects of the subject program, and the edge labels encode relationships like object allocation, variable assignment, and field loads and stores. The grammar of the CFLR problem is formulated in a way that it encodes the semantics of points-to analysis, that is, when can a variable point to a heap object at runtime. Coupled with specialized data structures for computing transitive closure, Dietrich et al. analyze the OpenJDK with the points-to analysis in around a minute.

Reps demonstrate how to encode a range of program analyses for C as a CFLR problem [97]; starting from traditional gen/kill data-flow analyses such as constant propagation, through program slicing, to points-to analysis. This work is not so much focused on improving the performance of static analyses, rather it is a foundational work showing how to formulate the context-free grammar to capture these analyses.

Sridharan and Bodík combine CFLR, demand-driven evaluation, and mixed precision evaluation in a context-sensitive points-to analysis [119]. First, they define the context-free grammar as the language of balanced parentheses; e.g. a method call edge denoted with "(" must be later closed by a return to call site edge denoted with ")" to represent a realizable control flow path. Second, the demand-driven aspect means that realizable paths are only sought when requested by the client of an analysis, e.g. by querying the points-to set of a specific variable. Third, to improve scalability, they also make use of mixed-precision evaluation. The analysis first computes an over-approximating result by skipping parts of the points-to graph that likely yield unrealizable paths. Then, upon request from the analysis client, the analysis tries to improve the precision of the results by inspecting previously skipped parts of the graph.

Cauliflower is a solver generator for CFLR problems [61]. Cauliflower comes with a DSL that allows to define the context-free grammar of the analysis problem, and it emits a parallel C++ executable that solves the CFLR problem given an input graph. Cauliflower specializes the executable with partial evaluation by selecting an efficient execution plan at code generation time.

Another large class of analyses based on graph reachability is IFDS/IDE, which we introduced in detail in Section 8.1. Soot is a widely used IFDS/IDE-based static analysis framework for Java [19]. Soot serves as an optimization framework due to the variety of intermediate representations it provides. Examples include; (i) Jimple, which is a simplified representation for Java, as every Jimple statement must have at most three components, or (ii) Shimple, which is Jimple in single static assignment form. We also used Jimple as the subject language in Section 4.7. Soot also comes with a handful of built-in inter-procedural analyses, such as call-graph construction, points-to analysis, or def/use analysis. Heros is a parallel IFDS/IDE solver for the Soot framework that is available open-source.[1]

Compositionality The idea is to analyze parts of a subject program in isolation and then compose the individual results into the full analysis result. The benefit of this approach is that components can be re-analyzed in isolation after a program change, which can yield significant speedups if components are small. The difficulty is that many practically relevant analyses do not compose well based on small units of code (e.g. individual statements).

Facebook Flow is a type checker for JavaScript [25]. Assuming that JavaScript modules have explicitly typed signatures, Flow analyzes the code per module, and stores a summary for each module. When code gets changed, Flow only re-analyzes the transitively affected modules bottom up, also parallelizing as much of the computation as possible, and then composes the results from the individual summaries. Unfortunately, there is no documented performance numbers on Flow, but the authors argue that it delivers the performance needed for interactive type checking in IDEs, plus Flow is really popular among developers according to GitHub statistics.[2]

Facebook Infer [24] is a static analysis framework that comes with a range of built-in analyses (e.g. analysis to find null pointer dereferences, finding resource leaks, race condition analysis) for a variety of languages (C, Objective-C, and Java). The theoretical foundation behind Infer is separation logic, which facilitates reasoning about mutations to the heap. Separation logic uses triplets of the form `{pre}prog{post}` where `{pre}` is a precondition describing the state of the heap before executing a program element `prog`, while `{post}` captures the postcondition. The analysis of an entire program is about finding the pre- and postconditions for all program elements. However, defining all these triplets is not the job of an analysis developer. Infer uses a theorem prover that can automatically infer pre- and postconditions based on some initial set of triplets. To this end, Infer uses bi-abduction, which is a form of logical reasoning for separation logic. The power of this technique lies in its compositionality, as inferences only need to be re-computed for program elements affected by a program change, the rest can be reused. Importantly, functions only need to be analyzed once, and their effects on the heap can be efficiently summarized. This allows Infer to efficiently scale to Facebook-scale subject programs and provide timely feedback on pull requests at code review time.

Demand-driven evaluation The idea of demand-driven evaluation is to only compute information that is needed by analysis clients and defer from eagerly analyzing entire subject programs. For example, Boomerang is a highly-precise points-to analysis that allows analysis clients to query points-to information for a given calling context, which can be answered much faster than an analysis on the whole program [118]. Boomerang uses a distributive formulation based on the IFDS framework.

Do et al. present the idea of just-in-time (JIT) static analysis [33], which is about

[1] `https://github.com/Sable/heros`
[2] `https://github.com/facebook/flow`

prioritizing the analysis of parts of a subject program that are close to the currently edited part over the rest of the subject program. This idea builds on the observation that software developers are “in context” at the currently edited part when working in an IDE, so showing error messages relevant to the edited part is more important than computing the analysis result for the entire subject program. This is essentially a form of demand-driven evaluation. The JIT approach employs a layered architecture for analysis where each layer expands the scope of the code part being analyzed and the results computed for each layer can be composed. First, the method of the currently edited code part is analyzed, then the container class, file, package, and so on. The goal is to quickly compute the results for the currently edited method, and, while the developer inspects the analysis results, the analysis can carry on with larger units of code. This kind of layered execution only works with data-flow analyses that are distributive. This is because distributive analyses allow to analyze each data flow separately, in any order, and they are also insensitive to the order in which the merge operator (e.g. lub) is applied, which is important for combining the results obtained in different layers. Do et al. instantiate the idea of JIT analysis for an Android taint analysis.

SUPA [121] is another demand-driven points-to analysis, which targets C programs. SUPA is an involved analysis that combines many techniques presented so far. It starts with a cheap pre-analysis to compute an over-approximating value flow graph encoding def-use chains. A def-use chain encodes a definition of a variable and all the uses reachable from the definition, without any other intervening definition. A points-to query is then formulated as a graph reachability problem over this graph data structure. SUPA tries to refine the precision of the value flow graph, thereby improving the quality of its results, if a time budget allocated by the analysis client allows for that. To this end, it uses a multi-stage approach; It first starts with a more expensive flow-sensitive and context-sensitive refinement, but failing to finish with that in the allocated budget, it downgrades its precision by giving up on context-sensitivity for a second refinement attempt.

Mixed-precision evaluation Giving up on (some) precision typically makes analyses cheaper to compute, but giving up on too much precision can easily result in a high amount of false results. Mixed-precision evaluation tries to find the right balance between precision and performance by automatically tuning the precision characteristics of analyses depending on user needs or heuristics. Note that mixed-precision analyses compute results for the entire subject program (potentially by sacrificing precision), but demand-driven evaluation only runs an analysis on code parts relevant to an analysis client.

Guyer and Lin present a mixed-precision points-to analysis [49]. The analysis runs in two phases with tight coupling between the points-to analysis and its analysis client. First, there is a fast context-insensitive, flow-insensitive pass of the points-to analysis on the subject program. Then, the analysis client examines the results, and it can report back to the analysis the memory locations or control flow locations where the analysis result is not sufficiently precise for its purposes. In turn, the analysis does a second pass with flow-sensitivity and/or context-sensitivity enabled at the requested locations.

A related approach is from Zhang et al., also concerning mixed-precision points-to analysis [145]. The key idea is to analyze different parts of a subject program with points-to analyses of different precision characteristics in terms of flow- and context-sensitivity. Subject programs get divided into program segments based on the aliasing effects of assignments. Then, they use heuristics to decide what kind of points-to analysis to use for each segment.

Plevyak and Chien present a type inference algorithm for dynamically typed object-oriented languages that tunes its precision at runtime [92]. Their approach encodes the type inference problem in the form of set constraints. The idea is to start with a fast type inference that is context-insensitive in two ways. First, it is call-site insensitive, as it collapses the information about different call sites of the same method. Second, it is object-insensitive, as it collapses the information about the different allocation sites of the receiver objects of method calls. If this imprecision leads to a typing violation, the algorithm selectively turns on call-site sensitivity and/or object-sensitivity for affected program variables.

Li et al. present an approach that selectively applies context-sensitivity for a Java points-to analysis [74]. From the technical perspective, this is also a two phase approach. First, they run a context-insensitive points-to analysis on the subject program. Based on its results, they build up a precision flow graph that helps to identify critical value flows that lead to imprecision without context-sensitivity. Instead of relying on heuristics, they identify three patterns for these value flows and find matches of these patterns via a graph reachability problem on the precision flow graph. The result of the pattern matching is a set of precision-critical methods in the subject program where context-sensitivity shall be enabled. They demonstrate that with this mixed-precision analysis they can retain almost all of the precision of the fully context-sensitive analysis while achieving considerable run time saving.

Partial evaluation A common optimization technique is to partially evaluate an analysis at compilation time, thereby allowing specializations of data structures and evaluation logic in the generated executable. We use a similar idea in IncA to compute hints about the relevant node types, as demonstrated in Section 3.4.

Soufflé is an analysis framework that heavily relies on partial evaluation [66]. Soufflé defines its own Datalog dialect for the specification of analysis. Soufflé synthesizes an efficient C++ executable from a set of Datalog rules. The Soufflé compiler partially evaluates the analysis specification to optimize index selection, specialize data structures, tune join orders, and more. The generated C++ code also has OpenMP annotations to utilize multi-threaded execution. The authors of Soufflé have reported good scalability; Highly-precise security and points-to analyses expressed in Soufflé scale to the entire JDK.

Parallelism An immediate idea for speeding up analyses is to exploit multi-core architectures, which is commonly used in every kind of hardware nowadays. However, many practically relevant program analyses do not lend themselves to efficient parallelization due to complex dependencies in analysis results.

Méndez-Lojo et al. present an approach for executing an Andersen-style points-to analysis in parallel [83]. The key idea is to encode the different kinds of points-to relevant instructions (e.g. assignment, allocation, dereferencing) in a graph data structure and treat Andersen's rules (see Figure 3.8) as graph rewrite rules. Parallelization of the analysis is not as straightforward as just applying rewrite rules in parallel because Méndez-Lojo et al. show that the application of rewrite rules can conflict with each other, which makes the analysis an irregular computation. This means that the computation depends on runtime data, and it is not possible to produce a parallel schedule statically. To this end, they make use of a speculative execution model, and they also develop optimizations to reduce the number of backtracking needed in face of conflicting rewritings. It is reported that the approach achieves up to 3X performance improvement compared to sequential execution.

A related approach is designed by Putta and Nasre, which is also based on the parallel execution of rewrite rules [94]. Their key insight is to reduce the conflicts between rewrite rules by keeping multiple copies of the points-to set of the same variable, thereby improving parallelism. Then, the local copies of the points-to sets must be merged after each rule application. However, as Putta and Nasre argue, care should be taken because this approach only works with analyses that are unordered in the sense that the order in which statements are executed in the subject program is not relevant for the analysis. Specifically, for points-to analysis, this means that flow-insensitivity is required, the approach does not work with a flow-sensitive analysis.

8.3 Datalog and Its Applications

Given that Datalog plays a central role in this dissertation, we have dedicated Chapter 2 to introduce its key concepts. In this section, we survey more advanced techniques. We first look at top-down evaluation of Datalog programs. Then, we discuss advanced incrementalization techniques. We present state-of-the-art Datalog solvers and their interesting technical details. Finally, we discuss a few industrial applications of Datalog.

Top-down evaluation So far in this dissertation, we only considered bottom-up evaluation for Datalog programs. With this approach, we start from a set of facts in EDB relations and repeatedly apply Datalog rules until fixpoint, thereby computing the minimal model of the Datalog rules. In contrast, top-down evaluation makes use of a specific user query and walks backwards from rule head to rule body, only trying to infer tuples that are actually required to answer the query. Contrast this to deriving all tuples, potentially redundantly, in bottom-up evaluation.

A widely used top-down approach for evaluating Datalog is called query/subquery (QSQ) [134]. The evaluation starts with a distinguished rule that describes the user query. An example is the rule `CFlowQuery(n1, trg) :- CFlow(n1, trg)` where `CFlow` is a relation computing CFG edges for some subject program. The rule head `CFlowQuery(n1, trg)` specifies that we are only interested in CFG edges where the source is a specific `n1` CFG node (e.g. a specific statement). Intuitively, QSQ tries to find a proof tree for CFG tuples where the source is `n1`. To this end, QSQ pushes variable bindings top-down from rule head to rule body and sideways among atoms in the same rule body. For example, QSQ would start pushing down a binding `{src → n1}` to the `CFlow` rule, thereby limiting the state space that the proof tree exploration would need to consider. In turn, in the body of `CFlow`, the incoming binding would be pushed sideways among atoms, potentially resulting in more bindings for other variables. QSQ is an iterative process and, for our example, the result of it would be the set of all those `(n1, trg)` tuples which can be derived given a set of facts.

Top-down evaluation has the potential of avoiding redundant derivations and improving performance by only considering derivations relevant to a user query. In contrast, bottom-up evaluation computes full results, which is important e.g. for program analyses if we are interested in finding all bugs in a program. Researchers proposed techniques that help to combine these two evaluation techniques. Magic sets [15] is family of rewriting techniques for Datalog programs that help to introduce auxiliary rules to make use of bindings relevant from a user query. This way, an otherwise bottom-up evaluation of the Datalog program can be governed by the user query, thereby combining the best of the two evaluation approaches.

Incremental Datalog Gupta and Mumick provide an overview on incremental Datalog maintenance in their survey [46]. We already discussed the Counting and DRed algorithms in Chapter 2; Counting is used to incrementally maintain the results of non-recursive Datalog programs, while DRed is used for recursive programs. When explaining the inner working of DRed, we emphasized that there is a potential for over-deletion in the delete phase, as DRed temporarily ignores alternative derivations to transitively mark all those tuples for deletion that are affected by deletions in EDB relations. DRed follows this approach to ensure correct results in face of recursive dependencies among rules.

Motik et al. propose an approach to mitigate the potential performance problems of the over-deletion in DRed. The technique is called backward/forward chaining (B/F for short) [88]. The key insight is that instead of "blindly" marking tuples for deletion during the delete phase, B/F tries to find evidence about to-be-deleted tuples still being derivable from remaining facts, thereby avoiding the deletion altogether. The backward step starts to build a proof about the existence of a tuple by walking backwards from rule head to rule body, towards facts in EDB relations (cf. QSQ from above). However, due to potential recursive dependencies, the backward chaining may end up in a cycle. To remedy this, the backward chaining merely collects tuples whose existence need to be proved and only visits each tuple once. Then, the forward chaining tries to prove the existence of the marked tuples by starting from remaining facts and repeatedly applying rules. The authors argue that the reason why B/F can be more efficient than DRed is because the set of derivable tuples is often smaller than the set of over-deleted tuples computed by DRed, so it pays off to interleave this kind of proof searching amid the delete phase.

Another line of work aims to avoid expensive over-deletions with the use of provenance information. Provenance can be thought of as meta information associated with tuples that reasons about why and how a certain tuple was derived during the fixpoint computation. Liu et al. propose to associate logical expressions as conditions of validity with each derived tuple [75]. In response to a deletion, the logical formulae can be quickly checked to verify if the deletion of a certain tuple can be avoided. The authors describe how to actually derive logical expressions when evaluating different relational algebra operations.

Notes on DRed$_\mathrm{L}$ We designed a new incremental fixpoint algorithm called DRed$_\mathrm{L}$ in Chapter 4 to compute the results of Datalog programs with recursive aggregation over lattices. Our proposed solution DRed$_\mathrm{L}$ generalizes DRed for recursive lattice aggregation, so the two algorithms share many features. In particular, the four steps of DRed$_\mathrm{L}$ introduced in Section 4.4 correspond to the analogous steps of DRed. However, some notable differences stem from aggregation support:

- For aggregations, tuple insertions may result in increasing replacements, which DRed cannot correctly handle, as they are not monotonic according to standard set containment.

- As its key novelty, DRed_L solves this problem by splitting tuples at runtime according to $\sqsubseteq$-monotonicity as opposed to simply separating deletions and insertions. This has far-reaching consequences on all aspects of the algorithm.
- The monotonic phase of DRed_L must thus incrementally process monotonic deletions (in addition to insertions). This requires a mechanism to determine how to propagate such deletions that were not present in DRed. For this purpose, we have adapted support counts and introduced support multisets, neither of which were part of DRed where only insertions were considered monotonic.
- The maintenance of these support data structures required DRed_L to enumerate each derivation exactly once; this is achieved through semi-naïve evaluation.
- As a side effect, such support data structures also enabled the re-derive phase of DRed_L to avoid recomputation in contrast to DRed.

An alternative to maintaining recursive aggregations would be to rewrite them into a form supported by DRed [81, 110]: Replicate tuple t with lattice value c to tuples t' with c' for each $c' \sqsubseteq c$, thus mapping $\sqsubseteq$-monotonic changes, aggregations, and recursions into their classical monotonic counterparts. Depending on the lattice, this transformation may be prohibitively expensive or even impossible. For example, the (integer) interval analysis in Section 4.2 would require a quadratic amount of additional interval values (one for each sub-interval); while an *infinite* amount of new tuples would have to be produced for an occurrence of Top in the singleton points-to analysis in Section 4.7. First-class support of lattice-based aggregation avoids this blowup in DRed_L.

Notes on LADDDER To efficiently support lattice-based inter-procedural analyses, we adopted DDF and designed the LADDDER algorithm in Chapter 5. As we explained already, DRed-based approaches suffer from a potential over-deletion problem, where a very large amount of tuples could be deleted only to be later re-derived. This shows especially when frequently used library functions are affected in an inter-procedural analysis. DDF does not have this specific problem, but it may also do unnecessary steps if e.g. a program change leads to a new derivation of an existing tuple at an earlier iteration round: Potentially all consequences of that tuple will have to be computed again, at a shifted timestamp. Motik et al. concludes that neither algorithm is universally superior to the other; pathological inputs can be constructed to force either solution to do significantly more work than necessary. The study by Motik et al. makes no reference to DDF, but their algorithm "Recursive Counting" is apparently an independent rediscovery of the core idea of DDF, as adapted for non-aggregating Datalog.

Regarding memory consumption, DDF clearly needs more space than DRed: DDF associates each tuple with (a sparsely stored) differential count timeline, as opposed to no provenance in DRed or support count in DRed_L. DDF would generally also require a differential existence timeline, but due to its inflationary nature, LADDDER can represent it as a single timestamp of appearance. This overhead is highly dependent on the number of different timestamps a tuple is derived at. In both cases, practical implications can only be determined empirically. Experiments in Section 5.6 reveal LADDDER to be significantly faster than DRed_L with an acceptable memory cost.

Inflationary semantics [48] is a well-known Datalog concept, but it generally lacks minimal model guarantees and is often associated with non-determinism [3, 50]. Our application of inflationary semantics in Laddder is novel in providing, under specific assumptions, important correctness guarantees, including both minimality and termination. The equivalence between Laddder and the conventional Ross and Sagiv semantics, if the latter exists, can be considered a reverse application of the concept pre-mappability [144].

Solvers There is a large number of Datalog solvers documented in the literature, plus certain solvers also see widespread use in industrial applications. Here, we review a select few of them focusing on technical details.

Shkapsky et al. propose the BigDatalog system for data analytics over Apache Spark [111]. BigDatalog uses Datalog for specification, which they argue is a good fit for this application domain due to its declarative nature and support for recursion. However, they also explain that, by default, Spark is unfit to efficiently support recursive Datalog. To this end, the BigDatalog runtime takes care of efficient data management and multi-threaded execution, plus it employs a range of optimizations to improve query plans. BigDatalog supports aggregation, but only simple ones like min, max, and sum.

μZ is a Datalog engine that is part of the Z3 SMT solver [59]. Just like BigDatalog, μZ also heavily optimizes the relational algebra tree behind a Datalog program to improve performance. A unique feature of μZ is that it also allows input relations to be represented abstractly via a first-order logic formula. This is where the integration with Z3 comes into play, as the Datalog program in this case gets evaluated symbolically with SMT solving techniques.

LogicBlox is a commercial Datalog solver targeted towards data analytics applications [7]. LogicBlox uses a range of efficient data structures to speed up query evaluation. For example, it uses persistent immutable data structures to represent relations, which allows data sharing upon modifications, plus immutability is key in enabling parallel execution. An interesting aspect of LogicBlox is that it not only allows incremental maintenance in response to changes in EDB relations, but it also allows to change the Datalog program itself while reusing as much of the previously computed results as possible.

bddbddb is a Datalog solver that represents relations with binary decision diagrams (BDDs) [140]. A BDD encodes a boolean function over a set of variables, and bddbddb uses it to represent the tuples of a relation. BDDs provide space-efficient storage, as a value domain with n values can be encoded with just $log_2 n$ bits. Whaley et al. describe how to obtain the boolean functions behind BDDs from relational algebra operations. It is important to note that the efficiency of the BDD-based encoding is highly sensitive to variable ordering. Scholz et al. benchmarked the performance of bddbddb with a context-sensitive points-to analysis, and they found that bddbddb had both excessive run time and memory consumption with the variable orderings computed by bddbddb itself [107]. They needed to do a significant amount of manual reorderings to scale bddbddb to the benchmark analysis.

We have already discussed the Soufflé framework [66] in Section 8.2 as a framework that uses partial evaluation in its compiler. We discuss two Datalog-specific details about the Soufflé solver here. Subotić et al. propose an efficient strategy for index selection in Soufflé [120]. Soufflé uses B-trees to represent relations. B-trees are indexed data structures providing efficient (range) lookups for an index key. The lookup functionality of B-trees is used to efficiently perform joins when evaluating a Datalog program in Soufflé. The key observation of Subotić et al. is that it is unnecessary to construct separate B-trees for each index key that a Datalog program requires. Instead, index keys can “cover” other index keys, meaning that under certain conditions a B-tree constructed for an index can be used to answer queries for other indices, as well. The goal of their work is to find the minimal set of indices that cover all required indices. They propose a polynomial time algorithm to do this, which is based on a maximum matching problem in a bipartite graph.

Zhao et al. add support for debugging in Soufflé [146]. Their idea is to generate proof trees for tuples, which allow developers to inspect the series of rule applications that resulted in a derivation. They extend the semi-naïve bottom-up evaluation of Datalog to augment each tuple derivation with two pieces of meta information: a reference to the rule that derives the tuple, plus the “height” of the derivation. Their notion of height directly corresponds to the existence timestamp of a tuple in LADDDER: A height of a tuple is equal to the maximum height of the tuples used in a rule body to infer the tuple plus one. Once the fixpoint computation terminates, developers can issue queries iteratively to explore the proof tree for a tuple. This boils down to reversing the lookup of tuples that could have derived another tuple at a certain height, essentially requiring to walk backwards towards EDB tuples.

In Section 5.6, we used LADDDER to incrementalize inter-procedural program analyses in IncA. LADDDER is based on DDF, but there are also other solvers that make use of DDF. DDF has a reference implementation in Rust available open source.[3] Differential Datalog (DDLog) is a framework that compiles Datalog programs to the API provided by the DDF reference implementation [102]. DDLog extends Datalog in several ways; e.g. with an expression language, rich type system, and module system. 3DF [51] follows a similar approach, as it also compiles Datalog programs to DDF. The primary application area of 3DF is to provide live updates in streaming analytics systems. A key difference between these approaches and LADDDER is in the aggregation support. First, LADDDER uses an inflationary aggregation semantics which enables looser requirements on analysis definitions (see Section 5.3.3). Second, these approaches employ naive aggregation architecture compared to the optimized sequential architecture of LADDDER (see Section 5.4). As we showed in Section 5.6, both the aggregation semantics and the optimized aggregation play a crucial role in efficiently scaling to lattice-based inter-procedural analyses.

Applications DOOP is a collection of Java program analyses implemented in Datalog, mostly known for its highly precise points-to analyses [112]. The researchers behind DOOP constantly push the boundaries in terms of scalability and precision in this domain. For example, Smaragdakis et al. evaluate a range of context-sensitive points-to analyses with different precision characteristics in DOOP. Smaragdakis and Kastrinis present a defensive points-to analysis, which provides soundness even in the presence of opaque code [113]. They define opaque code as code that cannot be analyzed normally, e.g. because of reflection or dynamic language features. A typical approach for static analyses is to simply restrict the subject language to not allow opaque code at all. However, Smaragdakis and Kastrinis argue that this approach essentially rules out all realistic subject programs. They take a different approach. Instead of aiming to provide full soundness in the presence of arbitrary opaque code, their analysis computes the claimed domain of soundness, that is, the program parts for which the analysis result is guaranteed to be sound. They show that this approach actually yields actionable feedback for the points-to analysis.

[3]`https://github.com/TimelyDataflow/differential-dataflow`

Gigahorse is a decompiler for Ethereum smart contract bytecode [43]. The decompilation logic is implemented in Datalog. The goal is to turn low-level bytecode into a 3-address code representation where control- and data-flow is explicit, unlike in the low-level representation. The authors show that Gigahorse is able to decompile almost all of the smart contracts available in the Ethereum blockchain, which significantly outperforms the decompiling capabilities of other approaches.

A related approach is MadMax, which is a Datalog-based tool for verifying the implementation of Ethereum smart contracts [42]. In the Ethereum Virtual Machine (EVM), smart contracts use gas while they run; Gas is the price of the computation. Gas must be paid upfront based on the estimated cost of the execution of the contract. If the execution uses more gas than agreed on, the EVM interrupts the contract with an out-of-gas exception. If the contract is not prepared to correctly handle the interrupt and later restart execution, attackers can cause denial-of-service attacks. MadMax helps to find these gas-related vulnerabilities in smart contracts. It identifies constructs in the decompiled 3-address code representation of smart contracts that can lead to out-of-gas exceptions.

8.4 DSLs for Program Analysis

In Chapter 2, we have presented Datalog in-depth, and we explained that it is a popular choice for implementing various kinds of static analyses. Given that we also designed the IncA$_{fun}$ language as an alternative to IncA Datalog, we look at other approaches that also come with a tailored DSL for analysis specification.

Frameworks with Datalog-based languages Semmle QL [12] is a programming language and runtime system for the implementation of static analyses. The language offers constructs similar to object-oriented languages, such as classes and methods, but they are re-interpreted in logical terms, as QL compiles to Datalog. QL comes with fact extractors for most of today's mainstream languages, plus it offers a rich standard library of reusable analyses (e.g. taint analysis for Java). In the back end, QL uses a heavily optimizing compiler to ensure good performance for batch execution on large code bases. QL is non-incremental, and it only supports simple aggregators like `sum` or `count`.

Flix [79] is a non-incremental analysis framework that uses Datalog and lattices. Analysis developers in Flix use a Scala-based functional language to define lattice operations. This language is similar to the DSL that IncA offers for the definition of lattices. Flix puts emphasis on safe and sound analysis definitions in the meta end because its compiler automatically verifies the mathematical properties of lattice operations, for example, if a `lub` operation is indeed monotone wrt. the partial order of the lattice [78]. An interesting future work for IncA would be to adopt these techniques in the meta end, as, currently, IncA offers limited automation to help developers reason about their analyses.

Like in Flix, Conway et al. also combine lattices and logic programming in Bloom^L [27]. Bloom^L offers built-in lattices (e.g. bool, map) in its standard library, and developers can also define their own. An interesting back end aspect of Bloom^L is that it is incremental, but it only supports insertions of facts. The lack of support for deletions would limit Bloom^L's applications in IDEs, but the authors actually target distributed application development. They argue that supporting insertions is sufficient for this use case.

Datafun is a functional language inspired by Datalog [8]. The design of the language is centered around monotone map functions on custom lattices with support for fixed points. The language was intentionally designed in a way that termination of a Datafun program can be guaranteed just like in Datalog. Datafun promotes to track monotonicity of a program through types, instead of e.g. verifying the lattice properties via verification as in Flix. Regarding expressive power, the support for custom lattices makes Datafun similar to IncA or Flix, but Datafun actually offers more. There is support for higher-order functions, that is, functions that can take entire relations as input. This allows for example to encode a node-reachability computation that is parametric over the edge relation and the starting node as follows:

```
reach : Set (Node × Node) → Node → Set Node
reach edge start = fix (λ R. {start} ∪ {y | x ∈ R, (x, y) ∈ edge})
```

This example is interesting because this is something that Datalog itself does not support. To support such computations, the authors of Datafun present an approach for extending the Datalog semi-naïve evaluation with support for higher-order functions.

Soufflé defines its own Datalog variant for implementing static analyses. A Soufflé program works with a relational data model, it computes sets of tuples in relations, just like IncA. However, the Soufflé language is limited to two primitive data types: symbols and numbers. Symbols represent the facts extracted from subject programs, and they are backed by string values. Numbers are computed values. Custom data types can only be defined in the form of records which are potentially recursive structures consisting of symbols and numbers. There is no support for custom lattices. Soufflé supports simple aggregates (e.g. min, max, sum, and count) on numbers.

Non-Datalog-based languages A closely related approach to IncA is the Viatra Query library (formerly known as EMF-IncQuery) [129], which we rely on in the IncA back end for incrementalization. Originally, Viatra Query is a library for developing incremental model queries, which has many applications in the domain of model-driven software development. Viatra Query comes with its own query language, and IncA Datalog was heavily inspired by that language. Both of these languages are compiled to the graph pattern matching API in Viatra Query. There are some important program analysis-related additions in IncA compared to Viatra Query though: virtual links (e.g. next, prev) to navigate in the AST and most importantly lattices and aggregation.

Söderberg et al. propose a declarative language based on attribute grammars to define intra-procedural control flow and data-flow analyses [117]. Attribute grammars are about associating attribute values to nodes of the subject program's AST. Attributes can be of several kinds. Reference attributes are used in the control flow analysis to model CFG predecessors/successors. Higher-order attributes are used to synthesize virtual CFG nodes, such as the entry and exit points of functions. Circular and collection attributes are used for fixpoint computation for data-flow analyses. A similarity with IncA is that analyses are encoded directly over the AST of the subject program.

FlowSpec is a declarative specification language tailored to intra-procedural and flow-sensitive data-flow analyses [115]. In FlowSpec, first a control flow analysis must be defined for a subject language. FlowSpec offers tailored abstractions that allow specifying which AST nodes constitute the CFG and how control flows between CFG nodes, potentially with branching and merging. FlowSpec offers a DSL for specifying lattices and their operations, which is similar to the IncA DSL for lattice definition. The final step in FlowSpec is to define transfer functions that determine how lattice values get computed/merged along CFG nodes. The authors demonstrate the use of FlowSpec for several traditional set-based data-flow analyses.

DCFlow is a declarative language and Rascal library for defining control flow analysis [58]. It comes with tailored abstractions that allow to model straight line control flow, conditional and unconditional jumps, and exception handling directly over the AST of the subject program. Similar to IncA, DCFlow supports virtual links on AST nodes such as `first`, `last`, and `next`, plus control statements like `foreach` and `if`. The authors argue that DCFlow significantly reduces the size of the analysis implementation compared to implementing the control flow rules directly in Rascal. The authors report a 10X increase in terms of lines of code with Rascal for their benchmark control flow analysis.

MPS-DF is the data-flow analysis component of the MPS language workbench [124]. MPS-DF comes with two DSLs (with Java embedding): one for specifying data flow graphs (DFGs) and the other for implementing data-flow analyses. A DFG is a control flow graph where nodes also encode data-flow-relevant information like reads and writes to variables, (conditional) jumps, and exception handling. Data flow analyses then run over the DFG of the subject program. MPS-DF uses so called DFG builders, which are specific to an AST node type and specify the sub-DFG that is contributed by an instance of the respective type. Decoupling the creation of an intermediate input representation (the DFG) and the analysis itself has important consequences. First, MPS-DF analyses becomes extensible over extensions of the subject language. Second, analyses with different precision characteristics can be obtained e.g. by defining intra-procedural and inter-procedural builders for function call AST nodes.

PAG is a framework for generating efficient data-flow analyses from a set of high-level DSLs [80]. First, there is a DSL for defining the data types and lattices that the analysis works with. Second, there is an ML-style functional language for defining transfer functions that compute and propagate data-flow values along CFG nodes. Third, there is a DSL for specifying the details of the analysis as a whole; for example by defining if it is a forward or backward analysis or the function to be used for merging data-flow values at CFG merge points. A PAG analysis is compiled into an efficient C executable.

RML is a programming language that is used to encode analyses over a relational data model [17], just like Datalog. The primary application area of RML is pattern matching over graphs, which the authors use to formulate coding standards or to implement syntax-driven analyses (in the style of FindBugs). RML is integrated into the CrocoPat library, which is a relational algebra machine that uses binary decision diagrams for efficient data storage.

Alive is a DSL for specifying verified LLVM peephole optimizations [76]. A peephole optimization considers a small set of instructions and tries to replace them with functionally equivalent but more efficient instructions. An Alive optimization has a left and right hand side: The left hand side encodes the pattern of the sought LLVM instructions (while abstracting over concrete values and types) and the right hand side specifies the output of the optimization. The semantics of Alive code is defined in terms of logical formulas. The system uses the Z3 SMT solver to verify the satisfiability of the logical conditions, or to provide a counterexample if they cannot be satisfied.

CAnDL is a declarative DSL for specifying analyses used in the LLVM compiler [41]. The input to a CAnDL program is the single static assignment form of a subject program. A CAnDL analysis can be imagined as a series of constraints that, just like graph patterns, encode a pattern for sought code fragments that serve as the basis for optimizations. CAnDL offers a range of features, e.g. atomic constraints to match against LLVM instructions, looping constructs, logical connectives, and imports for modularization. The authors demonstrate how to use CAnDL for practically relevant use cases, such as peephole optimization or the optimization of graphics shader code.

8.5 Tree Differencing Techniques

The front end of IncA is responsible for computing the AST diffs in response to changes in subject programs because the back end uses those diffs to incrementalize analyses. In Chapter 3, we used a front end that relied on projectional editing, which can deliver diffs with no extra computational overhead. Then, we designed an alternative textual front end in Chapter 7. To provide background on our design decisions, we have already considered a number of parsers and tree diffing tools in Section 7.2. In this section, we focus on the inner working of other tree diffing tools.

Several diffing tools make use of the tree differencing algorithm of Chawathe et al. [26]. We review this algorithm in greater detail here because we will refer to the algorithmic details from other pieces of related work that we discuss in this section. The initial assumption of the algorithm is that each node in the AST has a label (representing the type of the AST node) and a primitive value. The algorithm follows a two step approach:

1. Find good matching between the two ASTs.
2. Generate a minimum cost edit script to turn the old AST into the new AST.

Matching is about identifying node pairs that correspond to/similar to each other. Intuitively, these are nodes that either remain unchanged or only their value gets updated in the new AST. They call such node pairs a matching or simply say that those node pairs are matched. They call the process of finding matches between the two ASTs the “Good Matching” problem. Note that just because two nodes are in a matching, it does not mean that their respective subtrees are identical. The authors define criteria when two nodes can be matched. For leaf nodes, the comparison is simply based on the comparison of the node values. For internal nodes, the score assessing similarity is based on counting the number of similar leaf nodes and checking if the count is larger than a set threshold.

The matching process itself is about checking the equality of each pair of nodes (modulo optimizations) based on the above criteria. This means that the matching step has the complexity $\mathcal{O}(n^2)$ where n is the size of the larger tree. This is an important difference compared to *hdiff*, as *hdiff* requires several traversal of the trees, but every one of those operations has $\mathcal{O}(n)$ complexity. Finding common subtrees is really efficient in *hdiff* due to the use of cryptographic hashes.

An *edit script* is a sequence of change operations (insert, delete, move, update) that transform the old AST into the new AST. Importantly, an edit script shall not insert or delete nodes that are part of a matching. The algorithm relies on the matchings found in the previous step because matched nodes act as anchors, i.e., intuitively, they are the reference points where change operations are required to turn one tree into the other. This is where the criteria used for matching comes into play. If the matching criteria is loose in the sense that nodes in entirely different parts of the two ASTs are considered matched, then a lot of edit operations will be required to turn the old tree into the new. If the matching criteria is too strict, then no nodes will be matched, so we can end up with the deletion of the entire subtree and the insertion of the new subtree.

To assess the quality of edit scripts, they introduce a cost metric for edit scripts. Insert, delete, and move all have unit cost, but the cost of update depends on how similar the old and the new values in the nodes are. If they are very similar, the cost shall be less than 1, while if they differ significantly, the cost shall be more than 1. The intuition is that for similar nodes, it should be cheaper to use a move/update than an insert/delete pair. The cost of an edit script is the sum of the costs of the individual change operations. The goal of the edit script generation is to find an edit script with the minimum cost given a concrete matching. They present an algorithm for finding the minimum cost edit script, and they prove that the computed edit script has minimal cost.

Fluri et al. employ the Chawathe algorithm in the context of AST diffing for programming languages [38]. They find that the matching criteria used in the original paper often result in mismatches that lead to unnecessarily large edit scripts. For example, the original approach already fails to find a match if a string value gets updated (e.g. when a method was renamed). Note that the Chawathe algorithm guarantees minimality of the produced edit script only wrt. a given matching, so the goal of Fluri et al. is to improve the matching step. To this end, the authors employ more sophisticated string matching algorithms for leaf nodes based on Levenshtein distance and n-grams. For inner nodes, they employ a range of optimizations to tune the matching, e.g. so called Dice-coefficient as a measure of similarity, dynamic thresholds, and similarity weighing. Their evaluation on open source projects shows a 45% reduction in the size of the edit script with their matching techniques compared to the original criteria by Chawathe et al..

We considered GumTree as a candidate tool for tree differencing in Section 7.2. GumTree is also based on the Chawathe algorithm, but the authors of GumTree also improve on the matching of the original algorithm [37]. Technically, their approach is an involved graph traversal that identifies matches between nodes, and the process is largely governed by a number of parameters. Falleri et al. make recommendations for the values of the parameters, but the users of GumTree have the option to tune them, which may be necessary, as the quality of the found matches heavily depends on the parameter values. This is an important difference compared to *hdiff*, as there is no need to tune any parameters in that algorithm. When compared to other state-of-the-art diffing algorithms, Falleri et al. found that the matching technique used in GumTree results in shorter edit scripts than with other tools. Based on a large collection of commits in Java and JavaScript projects, they found that, on average, GumTree can compute an edit script for a commit on the ballpark of 10-100 ms, which is not what we have seen in our experiments (see Section 7.2).

Yang present an algorithm for finding the largest common subtrees between two ASTs based on dynamic programming [141]. The result of the algorithm is a matching that describes which nodes are the same and which are different. However, this work does not describe how to actually obtain an edit script that would turn the non-matched nodes in the source AST to non-matched nodes in the target AST.

Sager et al. present algorithms that can be used to find similar Java classes in a code base [104]. First, they prepare a language-independent tree representation of the sources: They use Eclipse JDT to create the AST of Java source files, and then they transform the JDT ASTs into a language-independent tree format called FAMIX. The second step is finding similarity between each pair of subtrees. They employ three kinds of algorithms already documented in the literature before [130]. Bottom-up maximum common subtree isomorphism and top-down maximum common subtree isomorphism are two algorithms that compute similarity scores between two subtrees. If this score is larger than a set parameter, the two trees are considered similar. The third algorithm is based on tree edit distance. The goal is to find the minimum set of edit operations based on a cost model that transforms one tree into the other. If the edit distance (sum of costs) is smaller than a set value, two trees are considered similar. The authors argue based on a benchmarking with Java source code that the best results are obtained with the tree edit distance approach.

Asenov et al. use an interesting approach to compute differences between ASTs using line-based diffing algorithms [10]. Their idea is to encode ASTs in a special text format where each line represents an AST node with special traceability links that help to identify the position of the node in the original AST. Given the flattened text representation of the source and target ASTs, they use a line-based diff tool to compute the differences between the two files. Then, using the annotations in the textual files, they recover the AST diff from the textual diff.

8.6 Chapter Summary

We surveyed a number of related approaches in this chapter, ranging from techniques for speeding up static analyses, through advanced Datalog techniques and other analysis DSLs, to tree diffing algorithms for the front ends of program analyses. We cannot even aim for completeness wrt. the number of discussed approaches, as static analyses attract a lot of research attention due to their importance in the software development pipeline. We conclude with the following key takeaways.

First, providing efficient incrementalization for sophisticated analyses was still a significant problem before this dissertation. Existing state-of-the-art approaches either (i) cannot deliver sufficiently fast update times for live feedback in IDEs, (ii) cannot handle realistic subject programs, (iii) specialized to specific static analyses, or (iv) cannot support one or more of the important features of static analyses in terms of recursive dependencies, custom lattices and recursive aggregation, or inter-procedurality. We demonstrated throughout this dissertation with concrete case studies that IncA delivers on all of these requirements.

Second, the design of static analyses is always a balancing act along the precision (reliability and accuracy) and performance trade-offs. Finding the right balance is often influenced by specific use cases of analyses. There is no one-size-fits-all solution, as each one of the techniques used for speeding up analyses come both with pros and cons. For example, IFDS is highly scalable due to summaries but is limited to distributive problems, incrementalization can provide millisecond update times but may incur significant memory overhead, while compositionality can achieve significant speedups only if components are small.

Third, DSLs are becoming increasingly important for the definition of complex analyses, as (i) they help to shield analysis developers from the low-level details of efficient execution, and (ii) through tailored abstractions, they help to capture the domain-specific knowledge of developers, which, in turn, enables further optimizations.

9 Conclusions and Future Work

In this chapter, we revisit our thesis and summarize if any how we proved our thesis true. Then, we provide an outlook on possible future directions with pointers on relevant literature that can help to tackle the discussed challenges.

9.1 Revisiting the Thesis of the Dissertation

We formulated the following thesis in Section 1.3: *Incrementalization can significantly improve the performance of sophisticated static analyses. We can achieve this automatically while shielding analysis developers from the technical details of incrementalization.* To prove this thesis true, we designed and implemented the IncA static analysis framework through five main contributions:

- We proposed the generic architecture of IncA with projectional editing in the front end, IncA Datalog in the meta end, and an optimizing compiler and incremental graph pattern matching in the back end.
- We designed the DRed$_\mathrm{L}$ solver algorithm to support lattice-based analyses with recursive aggregation.
- We designed the Laddder solver algorithm to support inter-procedural lattice-based analyses.
- We proposed the IncA$_{fun}$ DSL for program analyses.
- We designed a textual front end for IncA based on the *hdiff*$_S$ AST differencing algorithm.

We defined five main requirements in Section 1.4. Satisfying these requirements directly leads to proving the thesis true, as we argue in the following.

Efficiency (R2) The ability to provide continuous feedback in IDEs based on IncA analyses was a guiding principle for the entire dissertation. Efficiency is relevant for both the back end and front end of IncA. In the back end, we focused on the design and implementation of efficient solver algorithms. We started off with using an incremental graph pattern matching library, which we extended with DRed-style evaluation to *support recursive analyses*. We designed a meta analysis that partially evaluates IncA analysis code to compute hints that tell relevant and irrelevant changes apart, thereby improving incremental performance. Then, we designed and implemented two novel algorithms DRed_L and LADDDER, each significantly advancing the Datalog state of the art. DRed_L efficiently supports *lattice-based program analyses*, and LADDDER pushes this to the next level by also supporting *inter-procedural analyses*. In the front end, efficiency became an important factor when we moved from the projectional IDE to textual IDEs, as computing AST differences incurs computational overhead in the latter. To this end, we adopted the *hdiff* algorithm and implemented $\textit{hdiff}_S$, which we then integrated with the back end of IncA. In every chapter, we evaluated the performance of our new additions and found that IncA consistently provides good incremental performance for a wide range of analyses. We also saw that the price of sub-second update times is memory. The overhead can get large, but not prohibitive. We address this issue later in this chapter with a suggestion for future work.

Expressiveness (R3) Through a number of different case studies, we demonstrated the expressiveness of IncA. We started off with analyses that even standard Datalog can express: FindBugs analyses, well-formedness checks for DSLs, and control flow and points-to analyses. Then, we extended IncA with support for lattices and aggregation, which are fundamental to static analyses. We implemented strong-update points-to analysis, string analyses, and static analyses used in Rust. After employing LADDDER in the back end, we also experimented with inter-procedural points-to analyses. Finally, we used IncA_{fun} to implement overload resolution for FJ and a type checker for Rust, which was an entirely new class of analyses, as it concerns type systems. This demonstrates that IncA is applicable for a wide range of analyses.

Correctness (R1) To ensure that good incremental performance also comes with correct incremental updates, we provided correctness proofs whenever we designed a new solver algorithm in the back end. We provided a formal treatment of the theory behind both DRed_L and LADDDER. We discussed their semantics, formulated correctness properties, and proved that the algorithms satisfy those.

Genericity (R4) We designed IncA to be a generic analysis framework that works with any kind of subject language, subject program, and analyses. Regarding analyses, we already discussed that we implemented a number of different static analyses. Regarding subject languages, our experiments ranged from DSLs for embedded software development, through Featherweight Java and Jimple, to C, Java, Rust, and Python. Regarding subject programs, we employed generic editor technologies in the IncA architecture. First, with the projectional editor this posed no problem, as it could deliver AST differences for any program editable in the IDE. Then, with the textual front end, we intentionally chose *hdiff* to serve as the basis for $\textit{hdiff}_S$, as it can compute AST differences for arbitrary tree-shaped data. Finally, both DRed_L and LADDDER are generic solver algorithms that work with any Datalog program and custom lattices.

Declarativity (R5) To shield analysis developers from the incrementalization details in the back end, IncA provides two declarative DSLs for the specification of analyses. First, we designed the IncA dialect of Datalog, which extends standard Datalog with virtual EDB relations that allow to precisely capture what types and links an analysis requires from the AST of a subject program. Then, we re-designed IncA Datalog and created IncA$_{fun}$. The new DSL comes with constructs that are familiar from typical programming languages, plus it offers several language extensions that help to succinctly capture common coding patterns in analyses.

Based on these observations, we argue that we were successful in our attempt to validate our thesis statement. We managed to advance the state-of-the-art in both static analysis and in Datalog. For the static analysis community, we showed that Datalog is a good fit for implementing lattice-based inter-procedural analyses, while supporting interactive applications in IDEs. For the Datalog community, DRed$_L$ and LADDDER significantly advance the state of the art in terms of what is possible in a solver. Especially with the textual front end based on *hdiff*$_S$, we believe that our contributions can enable industrial applications, as well.

9.2 Suggestions for Future Work

We have already discussed a number of next steps centered around our textual front end and *hdiff*$_S$ in Section 7.6. However, those were more or less concrete in the sense that we mostly already know how to tackle the challenges. Here, we discuss a few possible directions that can be considered more as future research areas.

Reducing the memory overhead of IncA As benchmarking of our case studies in this dissertation demonstrated, incrementalization can yield significant performance improvements in terms of update times, but, often, the price of this is significant memory overheads due to extensive caching. A major source of memory overhead in IncA is due to the strategy that our incremental back end VIATRA QUERY uses to break down a large rule body into subqueries, each performing elementary relational algebra operation. For instance, a rule `h :-` a_1, a_2, a_3, a_4 may be automatically substituted with `h :-` b_1, b_2 and b_1 `:-` a_1, a_2 and b_2 `:-` a_3, a_4, so that the rule body is decomposed into a tree of three simple relational joins. The results of these subqueries are each stored, maintained, and indexed separately. This reduces the run time of change propagation because e.g. when a_3 changes then b_1 can be looked up from the index instead of re-computed. However, it comes with additional memory because both b_1 and b_2 must be stored and indexed. An alternative approach could be to not break down large rules (like `h` above) into subqueries, which saves memory, but increases the time required for incremental maintenance, as we need to maintain less indices.

An interesting direction for future work would be to design an approach that can automatically find the right decomposition strategy for an IncA analysis. We can draw inspiration from several systems. For example, the current approach in VIATRA QUERY is what the Rete algorithm does [39]. The TREAT system [86] follows a different approach, as it consumes less memory by using more coarse-grained operations, but it loses efficiency in maintenance run time. Gator networks [54] form a continuum between Rete and TREAT by selecting subqueries of various granularities for caching. We envision that finding the right decomposition strategy for queries could both be based on a static approach by examining the shape of the computation network built for an analysis, or a dynamic approach that could tune the caching behavior based on the sizes of relations as the analysis runs. Relevant literature here includes the approach by Sereni et al. that abstractly interprets relational algebra operations to influence magic sets rewriting [109] or resource aware ML by Hoffmann et al. which is about computing quantitative resource consumption for first-order functional programs [60].

Context-sensitive analyses with IncA In the context of inter-procedural analyses, precision-performance trade-off is largely influenced by whether the analysis is *context-insensitive* or *context-sensitive*. A context-insensitive analysis does not distinguish the different calling contexts of the same procedure and computes a result that over-approximates all possible calls of a procedure. Instead, a context-sensitive analysis uses some form of abstraction to distinguish calling contexts and collapses analysis results only if two calls have the same context according to the used abstraction. This clearly improves precision in exchange for computational cost, as the analysis must track more values and approximate less. Typical abstractions include the allocation site of the receiver object (i.e. object-sensitivity) or the invocation instruction itself (call-site sensitivity). The work by Smaragdakis et al. provides a detailed overview on a number of practically interesting abstractions [114].

An interesting direction for future work would be to experiment with context-sensitive analysis in IncA. We used the word "sophisticated" in this dissertation to refer to analyses that use (i) recursive dependencies, (ii) custom lattices and aggregation, and (iii) inter-procedurality. Context-sensitivity would push "sophistication" to the next level, as it can significantly improve the precision of an analysis. However, this is not something that is readily available, even though, IncA Datalog or IncA$_{fun}$ would already be expressive enough to implement context-sensitive analyses. We see challenges in the back end. First, incremental fixpoint algorithms (like DRed$_{\mathrm{L}}$) typically require Cost consistency (A 4.3), but context-sensitive analyses easily violate this requirement. The problem would immediately arise if contexts are represented with synthesized values and an analysis result consists of tuples that only differ in the computed contexts. LADDDER does not have this kind of limitation, as it does not require Cost consistency (A 4.3). Second, care should be taken because context-sensitivity can significantly increase the number of tuples in the analysis results (due to differentiating calling contexts), so memory optimizations may become more important than ever when tackling this direction.

Designing a more expressive DSL for analysis specification Datalog is a popular choice for implementing static analyses. We demonstrated this with a number of case studies throughout this dissertation, and we discussed numerous applications of Datalog in Section 8.3. However, as Datalog gains more traction in the static analysis community, so do different variations of the language: We designed IncA Datalog and IncA$_{fun}$ as an alternative, but other systems also frequently use custom extensions or completely new syntax instead of standard Datalog (see Section 8.4).

A future research direction for IncA could focus on the development of a more expressive analysis DSL that consolidates the design decisions from the variations of Datalog. IncA$_{fun}$ was already going in this direction, but its design was not the result of a systematic study of the different applications of Datalog, as it was more based on our personal experiences with developing static analyses. We envision a DSL that is applicable to a wide range of analyses with suitable language abstractions (potentially in the form of language extensions): type checking and inference, data-flow analyses, clone detection, or even general computations. An added difficulty in the language design is that it would constantly need to be coordinated with the capabilities of the back end, so that efficient incrementalization can be guaranteed for all analyses expressible in the new language.

Bibliography

[1] CVE-2014-1266, 2014. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995. ISBN 0-201-53771-0.

[3] S. Abiteboul, D. Deutch, and V. Vianu. Deduction with Contradictions in Datalog. In *International Conference on Database Theory*, Athens, Greece, 2014. URL https://hal.inria.fr/hal-00923265.

[4] U. A. Acar, G. Blelloch, and R. Harper. *Self-adjusting computation.* PhD thesis, Citeseer, 2005.

[5] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, June 2012. ISSN 2150-8097. doi: 10.14778/2336664.2336670. URL http://dx.doi.org/10.14778/2336664.2336670.

[6] L. O. Andersen. *Program Analysis and Specialization of the C Programming Language.* PhD thesis, University of Copenhagen, 1994.

[7] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742796. URL https://doi.org/10.1145/2723372.2742796.

[8] M. Arntzenius and N. Krishnaswami. Seminaïve Evaluation for a Higher-Order Functional Language. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. doi: 10.1145/3371090. URL https://doi.org/10.1145/3371090.

[9] S. Arzt and E. 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, pages 288–298, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568243.

[10] D. Asenov, B. Guenat, P. Müller, and M. Otth. Precise Version Control of Trees with Line-Based Version Control Systems. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, page 152–169, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662544938. doi: 10.1007/978-3-662-54494-5_9. URL https://doi.org/10.1007/978-3-662-54494-5_9.

[11] M. F. Atiyah and I. G. Macdonald. *Introduction to commutative algebra.* Westview press, 1994.

[12] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer. QL: Object-oriented Queries on Relational Data. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL `http://drops.dagstuhl.de/opus/volltexte/2016/6096`.

[13] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, S. McLaughlin, J. Reed, N. Rungta, J. Sizemore, M. A. Stalzer, P. Srinivasan, P. Subotic, C. Varming, and B. Whaley. Reachability Analysis for AWS-Based Networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 231–241, 2019. doi: 10.1007/978-3-030-25543-5_14. URL `https://doi.org/10.1007/978-3-030-25543-5_14`.

[14] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.*, 4(3):259–262, Sept. 1987. ISSN 0743-1066. doi: 10.1016/0743-1066(87)90004-5. URL `https://doi.org/10.1016/0743-1066(87)90004-5`.

[15] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, page 269–284, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912233. doi: 10.1145/28659.28689. URL `https://doi.org/10.1145/28659.28689`.

[16] L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with dynamic and static overloading. *Sci. Comput. Program.*, 74(5-6):261–278, 2009. doi: 10.1016/j.scico.2009.01.007. URL `https://doi.org/10.1016/j.scico.2009.01.007`.

[17] D. Beyer. Relational Programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 807–810, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134420. URL `https://doi.org/10.1145/1134285.1134420`.

[18] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309769. doi: 10.1145/2038916.2038923. URL `https://doi.org/10.1145/2038916.2038923`.

[19] E. Bodden. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 3–8, New York, NY, USA, 2012. Association for

Computing Machinery. ISBN 9781450314909. doi: 10.1145/2259051.2259052. URL `https://doi.org/10.1145/2259051.2259052`.

[20] E. Bodden. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-Based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, page 85–93, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359399. doi: 10.1145/3236454.3236500. URL `https://doi.org/10.1145/3236454.3236500`.

[21] S. Bosma. *Incremental Type Checking in IncA*. Master's thesis, Delft University of Technology, 2018.

[22] J. Cabot and E. Teniente. Incremental Integrity Checking of UML/OCL Conceptual Schemas. *J. Syst. Softw.*, 82(9):1459–1478, Sept. 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.03.009.

[23] F. Cacace, S. Ceri, and M. Houtsma. A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. *Distrib. Parallel Databases*, 1(4):337–382, Oct. 1993. ISSN 0926-8782. doi: 10.1007/BF01264013. URL `https://doi.org/10.1007/BF01264013`.

[24] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, 58(6), Dec. 2011. ISSN 0004-5411. doi: 10.1145/2049697.2049700. URL `https://doi.org/10.1145/2049697.2049700`.

[25] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017. doi: 10.1145/3133872. URL `https://doi.org/10.1145/3133872`.

[26] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.*, 25(2):493–504, June 1996. ISSN 0163-5808. doi: 10.1145/235968.233366. URL `https://doi.org/10.1145/235968.233366`.

[27] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391230.

[28] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 247–258, New York, NY, USA, 1984. ACM. ISBN 0-89791-139-3. doi: 10.1145/502874.502898. URL `http://doi.acm.org/10.1145/502874.502898`.

[29] G. Costantini, P. Ferrara, and A. Cortesi. Static Analysis of String Values. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*,

ICFEM'11, pages 505–521, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24558-9. URL `http://dl.acm.org/citation.cfm?id=2075089.2075132`.

[30] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Springer US, Boston, MA, 2004. ISBN 978-1-4020-8157-6. doi: 10.1007/978-1-4020-8157-6_27. URL `https://doi.org/10.1007/978-1-4020-8157-6_27`.

[31] A. Demers, T. Reps, and T. Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 105–116, New York, NY, USA, 1981. Association for Computing Machinery. ISBN 089791029X. doi: 10.1145/567532.567544. URL `https://doi.org/10.1145/567532.567544`.

[32] J. Dietrich, N. Hollingum, and B. Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. *SIGPLAN Not.*, 50(10):535–551, Oct. 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814307. URL `https://doi.org/10.1145/2858965.2814307`.

[33] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Just-in-Time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 307–317, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350761. doi: 10.1145/3092703.3092705. URL `https://doi.org/10.1145/3092703.3092705`.

[34] A. Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 381–390, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134339.

[35] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic Incrementalization of Prolog Based Static Analyses. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, pages 109–123, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-69608-3, 978-3-540-69608-7. doi: 10.1007/978-3-540-69611-7_7. URL `http://dx.doi.org/10.1007/978-3-540-69611-7_7`.

[36] S. Erdweg, O. Bracevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 880–897. ACM, 2015. doi: 10.1145/2814270.2814277. URL `http://doi.acm.org/10.1145/2814270.2814277`.

[37] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on*

Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pages 313–324, 2014. doi: 10.1145/2642937.2642982. URL `http://doi.acm.org/10.1145/2642937.2642982`.

[38] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11): 725–743, Nov. 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70731. URL `https://doi.org/10.1109/TSE.2007.70731`.

[39] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artif. Intell.*, 19(1):17–37, Sept. 1982. ISSN 0004-3702. doi: 10.1016/0004-3702(82)90020-0. URL `http://dx.doi.org/10.1016/0004-3702(82)90020-0`.

[40] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 209–220, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5699-2. doi: 10.1145/3213846.3213864. URL `http://doi.acm.org/10.1145/3213846.3213864`.

[41] P. Ginsbach, L. Crawford, and M. F. P. O'Boyle. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 151–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356442. doi: 10.1145/3178372.3179515. URL `https://doi.org/10.1145/3178372.3179515`.

[42] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018. doi: 10.1145/3276486. URL `https://doi.org/10.1145/3276486`.

[43] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1176–1186. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00120. URL `https://doi.org/10.1109/ICSE.2019.00120`.

[44] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and Recursive Query Processing. *Found. Trends databases*, 5(2):105–195, Nov. 2013. ISSN 1931-7883. doi: 10.1561/1900000017. URL `http://dx.doi.org/10.1561/1900000017`.

[45] I. Groher, A. Reder, and A. Egyed. Incremental Consistency Checking of Dynamic Constraints. In D. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12028-2. doi: 10.1007/978-3-642-12029-9_15.

[46] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995. URL `http://sites.computer.org/debull/95JUN-CD.pdf`.

[47] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. doi: 10.1145/170035.170066. URL `http://doi.acm.org/10.1145/170035.170066`.

[48] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 346–353, Oct 1985. doi: 10.1109/SFCS.1985.27.

[49] S. Z. Guyer and C. Lin. Client-Driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, page 214–236, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3540403256.

[50] A. Guzzo and D. Saccà. Semi-inflationary datalog: A declarative database language with procedural features. *AI Commun.*, 18(2):79–92, Apr. 2005. ISSN 0921-7126. URL `http://dl.acm.org/citation.cfm?id=1218852.1218854`.

[51] N. Göbel. Incremental datalog with differential dataflows, 09 2018. Retrieved 2019-10-11 from `https://www.nikolasgoebel.com/2018/09/13/incremental-datalog.html`.

[52] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-Based Language for Self-Adjusting Computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 25–37, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542480. URL `https://doi.org/10.1145/1542476.1542480`.

[53] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, Demand-Driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 156–166, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594324. URL `https://doi.org/10.1145/2594291.2594324`.

[54] E. N. Hanson and M. S. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical Report TR93-036, Univ. of Florida, 1993.

[55] D. C. Harkes, D. M. Groenewegen, and E. Visser. IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.11. URL `http://drops.dagstuhl.de/opus/volltexte/2016/6105`.

[56] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.

[57] G. Hedin. Reference Attributed Grammars. 24(3):301–317, 2000. ISSN 0868-4952.

[58] M. Hills. Streamlining Control Flow Graph Construction with DCFlow. In B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, editors, *Software Language Engineering*, pages 322–341, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11245-9.

[59] K. Hoder, N. Bjørner, and L. de Moura. μZ– An Efficient Engine for Fixed Points with Constraints. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 457–462, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[60] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, pages 781–786, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7.

[61] N. Hollingum and B. Scholz. Cauliflower: a Solver Generator for Context-Free Language Reachability. In T. Eiter and D. Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 171–180. EasyChair, 2017. doi: 10.29007/tbm7. URL `https://easychair.org/publications/paper/bnVq`.

[62] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004. doi: 10.1145/1052883.1052895.

[63] N. Immerman. *Descriptive complexity.* Springer Science & Business Media, 2012.

[64] Y. Ioannidis and R. Ramakrishnan. Efficient Transitive Closure Algorithms. pages 382–394, 01 1988.

[65] B. Johnson, Y. Song, E. Murphy-Hill, and R. 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, page 672–681. IEEE Press, 2013. ISBN 9781467330763.

[66] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing. ISBN 978-3-319-41540-6.

[67] M. Kallay. The complexity of incremental convex hull algorithms in Rd. *Information Processing Letters*, 19(4):197, 1984. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(84)90084-X.

[68] U. Khedker. *A Generalised Theory of Bit Vector Data Flow Analysis.* PhD thesis, Department of Computer Science and Engineering, IIT Bombay, 1995.

[69] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

[70] D. E. Knuth. The Genesis of Attribute Grammars. In *Proceedings of the International Conference WAGA on Attribute Grammars and Their Applications*, page 1–12, Berlin, Heidelberg, 1990. Springer-Verlag. ISBN 3540531017.

[71] H. Krasner. The Cost of Poor Quality Software in the US: A 2018 Report, 2018. `https://perma.cc/TQN7-A68B`, accessed on January 4, 2020.

[72] E. Kuci, S. Erdweg, O. Bračevac, A. Bejleri, and M. Mezini. A Co-contextual Type Checker for Featherweight Java (incl. Proofs). 05 2017.

[73] O. Lhoták and K.-C. A. Chung. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926389. URL `http://doi.acm.org/10.1145/1926385.1926389`.

[74] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. Precision-guided context sensitivity for pointer analysis. *PACMPL*, 2(OOPSLA):141:1–141:29, 2018. doi: 10.1145/3276511. URL `https://doi.org/10.1145/3276511`.

[75] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive Computation of Regions and Connectivity in Networks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1108–1119, March 2009. doi: 10.1109/ICDE.2009.36.

[76] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 22–32, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737965. URL `https://doi.org/10.1145/2737924.2737965`.

[77] Y. Lu, L. Shang, X. Xie, and J. Xue. An Incremental Points-to Analysis with CFL-Reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37050-2. doi: 10.1007/978-3-642-37051-9_4. URL `http://dx.doi.org/10.1007/978-3-642-37051-9_4`.

[78] M. Madsen and O. Lhoták. Safe and Sound Program Analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 38–48, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5699-2. doi: 10.1145/3213846.3213847. URL `http://doi.acm.org/10.1145/3213846.3213847`.

[79] M. Madsen, M.-H. Yee, and O. Lhoták. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 194–208, New

York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908096. URL `http://doi.acm.org/10.1145/2908080.2908096`.

[80] F. Martin. PAG–an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[81] M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of datalog recursion. 22, 08 2013.

[82] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *CIDR*, 2013.

[83] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel Inclusion-Based Points-to Analysis. *SIGPLAN Not.*, 45(10):428–443, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869495. URL `https://doi.org/10.1145/1932682.1869495`.

[84] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987. doi: 10.1007/3-540-48184-2_32. URL `https://doi.org/10.1007/3-540-48184-2_32`.

[85] V. C. Miraldo, P.-E. Dagand, and W. Swierstra. Type-Directed Diffing of Structured Data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, page 2–15, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351836. doi: 10.1145/3122975.3122976. URL `https://doi.org/10.1145/3122975.3122976`.

[86] D. P. Miranker. Treat: A better match algorithm for ai production systems; long version. 1987.

[87] R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi. I3QL: Language-Integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 417–432, 2014. doi: 10.1145/2660193.2660242.

[88] B. Motik, Y. Nenov, R. Piro, and I. Horrocks. Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1560–1568. AAAI Press, 2015. ISBN 0-262-51129-0. URL `http://dl.acm.org/citation.cfm?id=2886521.2886537`.

[89] B. Motik, Y. Nenov, R. Piro, and I. Horrocks. Maintenance of Datalog Materialisations Revisited. *Artificial Intelligence*, 269, 04 2019. doi: 10.1016/j.artint.2018.12.004.

[90] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 2015.

[91] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 579–598, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660202. URL `https://doi.org/10.1145/2660193.2660202`.

[92] J. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. *SIGPLAN Not.*, 29(10):324–340, Oct. 1994. ISSN 0362-1340. doi: 10.1145/191081.191130. URL `https://doi.org/10.1145/191081.191130`.

[93] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, Dec. 1989. ISSN 0098-5589. doi: 10.1109/32.58766.

[94] S. Putta and R. Nasre. Parallel Replication-Based Points-To Analysis. In M. O'Boyle, editor, *Compiler Construction*, pages 61–80, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28652-0.

[95] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158710.

[96] A. Rensink. Representing First-Order Logic Using Graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-30203-2_23.

[97] T. Reps. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, page 5–19, Cambridge, MA, USA, 1997. MIT Press. ISBN 0262631806.

[98] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL `https://doi.org/10.1145/199448.199462`.

[99] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[100] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 114–126, New York, NY, USA, 1992. ACM. ISBN 0-89791-519-4. doi: 10.1145/137097.137852. URL `http://doi.acm.org/10.1145/137097.137852`.

[101] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. ISBN 98-102288-48.

[102] L. Ryzhyk and M. Budiu. Differential datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019.*, pages 56–67, 2019. URL `http://ceur-ws.org/Vol-2368/paper6.pdf`.

[103] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 598–608, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL `http://dl.acm.org/citation.cfm?id=2818754.2818828`.

[104] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting Similar Java Classes Using Tree Algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 65–71, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933972. doi: 10.1145/1137983.1138000. URL `https://doi.org/10.1145/1137983.1138000`.

[105] M. Sagiv, T. Reps, and S. Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, page 131–170, NLD, 1996. Elsevier Science Publishers B. V.

[106] D. Saha and C. R. Ramakrishnan. Incremental and Demand-driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6. doi: 10.1145/1069774.1069785. URL `http://doi.acm.org/10.1145/1069774.1069785`.

[107] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892226. URL `https://doi.org/10.1145/2892208.2892226`.

[108] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley Professional, 4th edition, 2011. ISBN 032157351X, 9780321573513.

[109] D. Sereni, P. Avgustinov, and O. de Moor. Adding magic to an optimising datalog compiler. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 553–566, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376673. URL `https://doi.org/10.1145/1376616.1376673`.

[110] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In *2015 IEEE 31st International Conference on Data Engineering*, pages 867–878, April 2015. doi: 10.1109/ICDE.2015.7113340.

[111] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915229. URL `https://doi.org/10.1145/2882903.2915229`.

[112] Y. Smaragdakis and M. Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 245–251, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24205-2. doi: 10.1007/978-3-642-24206-9_14. URL `http://dx.doi.org/10.1007/978-3-642-24206-9_14`.

[113] Y. Smaragdakis and G. Kastrinis. Defensive Points-To Analysis: Effective Soundness via Laziness. In T. Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:28, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-079-8. doi: 10.4230/LIPIcs.ECOOP.2018.23. URL `http://drops.dagstuhl.de/opus/volltexte/2018/9228`.

[114] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. *SIGPLAN Not.*, 46(1):17–30, Jan. 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926390. URL `https://doi.org/10.1145/1925844.1926390`.

[115] J. Smits, G. Wachsmuth, and E. Visser. FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis. *Journal of Computer Languages*, page 100924, 2019. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2019.100924. URL `http://www.sciencedirect.com/science/article/pii/S2590118419300474`.

[116] E. Söderberg and G. Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*, volume 98 of *LU-CS-TR:2012-249*. Department of Computer Science, Lund University, 2012.

[117] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level. *Sci. Comput. Program.*, 78(10): 1809–1827, Oct. 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2012.02.002. URL `https://doi.org/10.1016/j.scico.2012.02.002`.

[118] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.22.

[119] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1134027. URL `https://doi.org/10.1145/1133255.1134027`.

[120] P. Subotić, H. Jordan, L. Chang, A. Fekete, and B. Scholz. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.*, 12(2):141–153, Oct. 2018. ISSN 2150-8097. doi: 10.14778/3282495.3282500. URL `https://doi.org/10.14778/3282495.3282500`.

[121] Y. Sui and J. Xue. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 460–473, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950296. URL `https://doi.org/10.1145/2950290.2950296`.

[122] O. Sumer, U. Acar, A. T. Ihler, and R. R. Mettu. Efficient Bayesian Inference for Dynamically Changing Graphs. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1441–1448. Curran Associates, Inc., 2008.

[123] C. Sung, S. K. Lahiri, C. Enea, and C. Wang. Datalog-based scalable semantic diffing of concurrent programs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 656–666, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238211. URL `http://doi.acm.org/10.1145/3238147.3238211`.

[124] T. Szabó, S. Alperovich, M. Voelter, and S. Erdweg. An Extensible Framework for Variable-Precision Data-Flow Analyses in MPS. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 870–875, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970296. URL `https://doi.org/10.1145/2970276.2970296`.

[125] T. Szabó, S. Erdweg, and M. Voelter. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 320–331, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970298. URL `http://doi.acm.org/10.1145/2970276.2970298`.

[126] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter. Incrementalizing Lattice-based Program Analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, Oct. 2018. ISSN 2475-1421. doi: 10.1145/3276509. URL `http://doi.acm.org/10.1145/3276509`.

[127] T. Szabó, E. Kuci, M. Bijman, M. Mezini, and S. Erdweg. Incremental overload resolution in object-oriented programming languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 27–33, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5939-9. doi: 10.1145/3236454.3236485. URL `http://doi.acm.org/10.1145/3236454.3236485`.

[128] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345, Nov 2010. doi: 10.1109/APSEC.2010.46.

[129] Z. Ujhelyi, G. Bergmann, Ábel Hegedüs, Ákos Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1(0):80 – 99, 2015. doi: http://dx.doi.org/10.1016/j.scico.2014.01.004. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[130] G. Valiente. *Algorithms on Trees and Graphs.* Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540435506.

[131] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[132] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, 23(4):733–742, Oct. 1976. ISSN 0004-5411. doi: 10.1145/321978.321991. URL `https://doi.org/10.1145/321978.321991`.

[133] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *J. ACM*, 38(3):619–649, July 1991. ISSN 0004-5411. doi: 10.1145/116825.116838. URL `https://doi.org/10.1145/116825.116838`.

[134] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Proc. First Intl. Conf. on Expert Database Systems, Charleston, 1986*, pages 179–193, 1986.

[135] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20 (3):339–390, 2013.

[136] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In B. Combemale, D. Pearce, O. Barais, and J. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer International Publishing, 2014. doi: 10.1007/978-3-319-11245-9_3.

[137] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using C Language Extensions for Developing Embedded Software: A Case Study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 655–674, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814276.

[138] G. Wachsmuth, G. Konat, V. Vergu, D. Groenewegen, and E. Visser. A Language Independent Task Engine for Incremental Name and Type Analysis. In M. Erwig, R. Paige, and E. Van Wyk, editors, *Software Language Engineering*, volume 8225 of

Lecture Notes in Computer Science, pages 260–280. Springer International Publishing, 2013. doi: 10.1007/978-3-319-02654-1_15.

[139] T. A. Wagner. *Practical Algorithms for Incremental Software Development Environments.* PhD thesis, EECS Department, University of California, Berkeley, Mar 1998. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html`.

[140] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In K. Yi, editor, *Programming Languages and Systems*, pages 97–118, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32247-4.

[141] W. Yang. Identifying Syntactic Differences between Two Programs. *Softw. Pract. Exper.*, 21(7):739–755, June 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210706. URL `https://doi.org/10.1002/spe.4380210706`.

[142] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, page 230–242, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913523. doi: 10.1145/298514.298576. URL `https://doi.org/10.1145/298514.298576`.

[143] J.-s. Yur, B. G. Ryder, and W. A. Landi. An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302676.

[144] C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory and Practice of Logic Programming*, 17(5-6):1048–1065, 2017. doi: 10.1017/S1471068417000436.

[145] S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with combined analysis for pointer aliasing. *SIGPLAN Not.*, 33(7):11–18, July 1998. ISSN 0362-1340. doi: 10.1145/277633.277635. URL `https://doi.org/10.1145/277633.277635`.

[146] D. Zhao, P. Subotic, and B. Scholz. Provenance for Large-scale Datalog, 2019.

Curriculum Vitæ

Tamás Szabó

26/03/1989	Date of birth in Debrecen, Hungary
09/2007 - 01/2011	B.Sc. in Computer Science Budapest University of Technology and Economics, Hungary
02/2011 - 02/2013	M.Sc. in Computer Science Budapest University of Technology and Economics, Hungary
05/2015 - 01/2021	Ph.D. in Computer Science
05/2015 - 04/2016	Technical University of Darmstadt, Germany
05/2016 - 04/2019	Delft University of Technology, Netherlands
05/2019 - 01/2021	Johannes Gutenberg University of Mainz, Germany
03/2013 - 09/2020	Software Engineer itemis AG, Stuttgart, Germany
10/2020 - present	Software Engineer Workday GmbH, Stuttgart, Germany