TECHNISCHE
UNIVERSITÄT
DARMSTADT

# Modular Collaborative Program Analysis

*vom Fachbereich Informatik*
*der Technischen Universität Darmstadt*
*genehmigte*

Dissertation

*zur Erlangung des akademischen Grades eines*

Doktor-Ingenieurs (Dr.-Ing.)

*vorgelegt von*

Dominik Helm

*geboren in Aschaffenburg*

Gutachter

Prof. Dr.-Ing. Mira Mezini
Technische Universität Darmstadt

Prof. Dr. Karim Ali
University of Alberta

Darmstadt 2023

Dominik Helm: Modular Collaborative Program Analysis
Darmstadt, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 09.02.2023
Jahr der Veröffentlichung der Dissertation auf TUprints: 2023
URN: urn:nbn:de:tuda-tuprints-232206

# ABSTRACT

With our world increasingly relying on computers, it is important to ensure the quality, correctness, security, and performance of software systems. Static analysis that computes properties of computer programs without executing them has been an important method to achieve this for decades. However, static analysis faces major challenges in increasingly complex programming languages and software systems and increasing and sometimes conflicting demands for soundness, precision, and scalability. In order to cope with these challenges, it is necessary to build static analyses for complex problems from small, independent, yet collaborating modules that can be developed in isolation and combined in a plug-and-play manner.

So far, no generic architecture to implement and combine a broad range of dissimilar static analyses exists. The goal of this thesis is thus to design such an architecture and implement it as a generic framework for developing modular, collaborative static analyses. We use several, diverse case-study analyses from which we systematically derive requirements to guide the design of the framework. Based on this, we propose the use of a blackboard-architecture style collaboration of analyses that we implement in the OPAL framework. We also develop a formal model of our architecture's core concepts and show how it enables freely composing analyses while retaining their soundness guarantees.

We showcase and evaluate our architecture using the case-study analyses, each of which shows how important and complex problems of static analysis can be addressed using a modular, collaborative implementation style. In particular, we show how a modular architecture for the construction of call graphs ensures consistent soundness of different algorithms. We show how modular analyses for different aspects of immutability mutually benefit each other. Finally, we show how the analysis of method purity can benefit from the use of other complex analyses in a collaborative manner and from exchanging different analysis implementations that exhibit different characteristics. Each of these case studies improves over the respective state of the art in terms of soundness, precision, and/or scalability and shows how our architecture enables experimenting with and fine-tuning trade-offs between these qualities.

# ZUSAMMENFASSUNG

Unsere Welt hängt zunehmend von Computern ab. Daher ist es wichtig, die Qualität, Korrektheit, Sicherheit und Leistung von Softwaresystemen sicherzustellen. Statische Analyse, die Eigenschaften von Computerprogrammen berechnet, ohne sie auszuführen, ist seit Jahrzehnten eine wichtige Methode, um dies zu erreichen. Jedoch steht statische Analyse vor großen Herausforderungen aufgrund zunehmend komplexer Programmiersprachen und Softwaresysteme und zunehmenden und teils einander widersprechender Anforderungen an Vollständigkeit, Präzision und Skalierbarkeit. Um mit diesen Herausforderungen umzugehen, ist es nötig, statische Analysen für komplexe Probleme aus kleinen, unabhängigen, aber miteinander kollaborierenden Modulen aufzubauen, die getrennt voneinander entwickelt und anschließend flexibel kombiniert werden können.

Bisher existiert keine generische Architektur, um ein breites Spektrum an unterschiedlichen statischen Analysen zu entwickeln und zu kombinieren. Das Ziel dieser Arbeit ist daher, eine solche Architektur zu entwerfen und als ein generisches Framework für die Entwicklung modularer, kollaborativer statischer Analysen zu implementieren. Wir nutzen mehrere verschiedenartige Fallstudienanalysen von denen ausgehend wir systematisch Anforderungen ableiten, um die Gestaltung des Frameworks zu leiten. Basierend darauf schlagen wir vor, Analysen ähnlich einer Blackboard-Architektur kollaborieren zu lassen. Diesen Ansatz verwirklichen wir im OPAL Framework. Wir entwickeln außerdem ein formales Modell der Kernkonzepte unserer Architektur und zeigen damit, wie Analysen frei miteinander kombiniert und dabei ihre Korrektheitsgarantien erhalten werden können.

Wir präsentieren und evaluieren unsere Architektur anhand der Fallstudienanalysen, von denen jede zeigt, wie wichtige und komplexe statische Analysen modulare und kollaborative umgesetzt werden können. Konkret zeigen wir, wie eine modulare Architektur für die Berechnung von Methodenaufrufgraphen eine konsistente Vollständigkeit verschiedener Algorithmen sicherstellt. Wir zeigen, wie modulare Analysen für verschiedene Ausprägungen von Unveränderbarkeit gegenseitig voneinander profitieren. Schließlich zeigen wir, wie die Analyse von Seiteneffektfreiheit von Methoden davon profitieren kann, Ergebnisse anderer komplexer Analysen kollaborativ zu nutzen, sowie davon, verschiedene Varianten der Analyse, die unterschiedliche Charakteristiken aufweisen, gegeneinander austauschen zu können. Jede der Fallstudien stellt eine Verbesserung gegenüber dem Stand der Technik in Bezug auf Vollständigkeit, Präzision und/oder Skalierbarkeit dar und zeigt, wie unsere Architektur es ermöglicht, Zielkonflikte zwischen diesen Eigenschaften zu studieren und feinabzustimmen.

# ACKNOWLEDGMENTS

Submitting a dissertation marks the end of a journey. As with all journeys, it is not so much the destination that matters or even the route you took, but the people that went the way with you. With this, I want to thank them, because I would not be here today without each one of them.

First of all, I want to thank my advisor Mira Mezini. Mira, it was an honor and a pleasure to work with you. You always gave me the room to pursue my research interests and your feedback and contributions to my publications were invaluable. Now I am one of your long list of graduates, and I can only hope to be as successful as many of them.

I also want to thank my second examiner Karim Ali. Karim, you were very supportive of my work, putting my mind at ease for the defense. I hope you enjoyed reading this thesis. Thanks also to Reiner Hähnle, Kristian Kersting, and Christian Reuter for serving on my committee.

Thanks to Gudrun Harris, who was tough on the outside, for making everything possible in administration, but always had a warm heart for all of us, and to Claudia Roßmann who too always has our backs.

Thanks to Michael Eichberg whose supervision of my master's thesis and continued support made me realize that I want to pursue a PhD and Ben Hermann, who first got me into static analysis when supervising my bachelor's thesis. Special thanks to Martin Kühnert, the high school teacher who played a key role in my decision to study computer science. Martin, you passed away way too early, but you will always be remembered.

Thanks, of course, to my other co-authors, Philipp Haller, Sven Keidel, Jan Thomas Kölzer, Florian Kübler, Michael Reif, Tobias Roth, and Guido Salvaneschi. Without you, this thesis would never have been written. Thanks to the rest of our language security group, Lars Baumgärtner, Leonid Glanz, Patrick Müller, Krishna Narasimhan, and Anna-Katharina Wickert. You have become not just colleagues, but friends. Thanks also to everyone else who helped proofreading this thesis: my colleagues Matthias Eichholz, Ragnar Mogk, and Daniel Sokolowski, and my friends Lilith Babilon, Frederik Fleißner, David Reis, and Lukas Sommer. I am grateful for your help and for all of the discussions and the feedback you gave me during these years. I cannot list everyone else I met over these years at the Software Technology Group, but to all of my present and former colleagues and to the students I had the pleasure to supervise: I am glad to have met you and to have worked with you.

Thanks to the *Studienstiftung des deutschen Volkes*. I am thankful for the financial support, but what mattered much more were the numerous summer schools, workshops, and other activities: the things I learned there and the wonderful people I met.

Thanks to the people of the *DPSG Stockstadt Stamm Mauritius*, you were home to me for more than two decades, and scouting was my counterbalance to academia. I have learned so much more from and with you than universities could teach and more than I could ever give back.

Last but not least, I want to thank my family. My grandparents Heinz, Helga, Maria, and Stani, who sparked my interests and always supported me in everything I did. In particular, I got into programming with the books and Commodore 64 of my grandfather. My final thanks belong to my parents: my father Stefan, who always got my back, and my mother Christine, probably the only person to have read every single word of this thesis. I am grateful for your support and for every minute with you.

Submitting a dissertation marks the end of a journey, but it also marks the beginning of a new one. I know I will meet more wonderful people, but I also know I can count on your continued support whatever I do. Thank you all for this, it was your journey just as well as mine, and it would not have been successful without you.

# CONTENTS

II Case Studies

III Evaluation

IV Conclusion

V Appendix

# 1 INTRODUCTION

The world around us is driven by software: the most critical infrastructures that we rely on, including law enforcement, financial systems, and public and personal health, all depend on the correct operation of complex software systems. From data centers to the supercomputers in our pockets that smartphones are to household appliances and implanted devices, software has become ubiquitous in our lives. This makes our societies vulnerable both to the ever-growing threat of malware crafted to intentionally harm us and to malfunctions of benign software.

Static analysis is an important method that allows us to find and prevent errors in software, mitigating the risks of malware and malfunctions alike [10]: it automatically computes properties of software without executing it. As such, it is used for a wide range of applications: In integrated development environments (IDEs), static analyses provide information to developers to understand their programs and mitigate problems early during software development [70]. Such information may include, e.g., the callers or callees of methods [83] or potential bug,s such as unclosed resources or uninitialized values [19]. In program verification, static analysis extracts the information necessary to prove a program's adherence to a given specification [143, 235]. It also plays a major role in dissecting software to find security risks, e.g., by identifying data that can be controlled by an attacker in such a way as to exploit vulnerabilities [39, 55], or privacy concerns, e.g., by tracking whether private data can be sent via network connections [147]. Static analysis can identify malware [177], in particular, if malicious application behavior is obfuscated and thus hidden [205]. Finally, static analysis can help to improve software performance: in optimizing compilers, it discovers potential for optimizations, such as dead code that can be removed [63], expressions that can be simplified [3, 11], or virtual method calls that can be replaced with static ones [113].

Modern static analyses face many challenges that can be addressed by a modular, collaborative approach where decouple sub-analyses interact to solve complex problems. The goal of this thesis is to develop a framework for modular, collaborative static analyses in order to aid their soundness, precision, and scalability and foster fine-tuning the trade-offs between these qualities.

## 1.1 CHALLENGES FOR STATIC ANALYSES

Three sources of complexity particularly impact static analyses: complex programming language features, complex analysis problems, and complex trade-offs.

As a running example to illustrate all of them, consider the most recent example of the global extent of the threats posed by complex software systems, the *log4shell* vulnerability [220]. In 2021, it was estimated to have affected hundreds of millions of devices [211] including 93% of cloud environments [152] using the popular *log4j* logging library. Importantly, the issue was not intentional malevolence. Instead, it was the result of a complex interaction of subsystems that all individually worked as designed: Vulnerable versions of *log4j* allowed for log message strings to contain placeholders that could be dynamically substituted. One supported method of substitution was the use of the *Java Naming and Directory Interface (JNDI)* that in turn allowed the use of the *Lightweight Directory Access Protocol (LDAP)*, which could then be used to load arbitrary code from remote sources and subsequently execute it in the context of the original process [80].

COMPLEX PROGRAMMING LANGUAGE FEATURES Modern programming languages include a multitude of features that extend beyond the language's core semantics. These additional features add complexity that is often difficult to analyze for static analyses [236]. Also, language features are not set in stone but evolve with time. This requires static analyses to be kept up to date continuously with programming language evolution.

Programming language features that impact static analyses include, among others, reflection [27, 136], (de-)serialization [203], concurrency [191], and implicit control flow [16]. Obfuscation, a multitude of techniques explicitly designed to complicate manual and automated static analysis [184, 212], is also often used [71, 241]. These difficulties posed by these features are exacerbated by the fact that they interact with each other. This forces static analysis developers to not only deal with each feature individually but to consider their possible interactions. As an example, deserializing program state can lead to dynamically loading classes on which then code is executed reflectively which has to be taken into account when statically analyzing what code is executed.

Consider reflection as an example of a programming language feature that is particularly problematic for static analyses [136]: reflection allows programs to introspect and change their own state and behavior. This is usually done using strings to access programming language constructs such as classes, methods, or fields. As these strings can come from arbitrary sources and can be created at runtime, it is difficult for static analyses to determine which constructs are accessed.

Reflection can also dynamically load and execute additional code unknown to static analyses. This leaves static analysis developers with only few choices that each have severe drawbacks: They can ignore the effects of reflection, leaving their analyses unsound, i.e., unable to reason about all possible executions of the program, and thus incapable of providing trustworthy information about the program. Alternatively, they can over-approximate reflective code, rendering the analysis imprecise, i.e., taking into account many program executions that are not actually possible. Imprecise analyses generate false positive results that not only are not helpful to the analysis' user but can actually be detrimental as they divert attention away from actual issues. Finally, analysis developers can try to identify the reflective constructs as precisely as possible, but this is a complex endeavor and, in the case of strings that are fully created at runtime, not always possible. To make matters worse, reflection often allows circumventing programming language mechanisms that restrict access to particular constructs, such as visibility (e.g., `private`) or immutability (e.g., `final`) modifiers. This additionally complicates static analyses as they cannot trust these modifiers at all if they can be circumvented by code that could be anywhere in a program. That is, the mere presence of reflection anywhere in the program can invalidate assumptions in other, unrelated parts of the program.

The *log4shell* example above shows how these complexities affect static analyses: The messages logged by *log4j* are strings created at runtime that can potentially be controlled by an attacker. Their content is then used to alter the program's behavior, loading code through a network connection. Finally, this code, which is not available to static analysis, is executed reflectively in the original process. Thus, finding this vulnerability using static analysis is hard, as assumptions about the potential content of strings at runtime have to be made as well as assumptions about the code loaded from the network and executed reflectively.

To further highlight how much complex language features affect static analyses, consider Table 1.1 (taken from Chapter 8). It shows the results of a benchmark test suite that assesses whether call-graph algorithms (i.e., static analyses that compute method-calling relationships) such as Class-Hierarchy Analysis (CHA) or Rapid Type Analysis (RTA) in the state-of-the-art Java static analysis frameworks *WALA* [111] and *Soot* [238] are sound in handling different language features. The table gives the number of test cases passed for each feature by each algorithm, showing sound handling of a particular use of a language feature. While some language features like standard virtual calls are handled soundly in all frameworks, more complex features like reflection show large differences between analyses and other features like dynamic class loading are not handled soundly at all. Features recently introduced into the Java Virtual Machine (JVM) such as method han-

Table 1.1: Soundness of Call Graphs for Different Java Features

| | **WALA** | | | **Soot** | | |
|---|---|---|---|---|---|---|
| **Feature** | **CHA** | **RTA** | **0-CFA** | **CHA** | **RTA** | **SPARK** |
| Non-virtual Calls | ● 6/6 | ● 6/6 | ● 6/6 | ◐ 5/6 | ◐ 5/6 | ◐ 5/6 |
| Virtual Calls | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Types | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Static Initializer | ◐ 4/8 | ◐ 7/8 | ◐ 6/8 | ● 8/8 | ● 8/8 | ● 8/8 |
| Java 8 Interfaces | ● 7/7 | ● 7/7 | ● 7/7 | ◐ 3/7 | ◐ 3/7 | ◐ 3/7 |
| Unsafe | ● 7/7 | ● 7/7 | ○ 0/7 | ● 7/7 | ● 7/7 | ○ 0/7 |
| Invokedynamic | ○ 0/16 | ◐ 10/16 | ◐ 10/16 | ◐ 10/16 | ◐ 10/16 | ◐ 10/16 |
| Class.forName | ◐ 2/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Reflection | ◐ 2/16 | ◐ 3/16 | ◐ 6/16 | ◐ 12/16 | ◐ 11/16 | ◐ 10/16 |
| MethodHandle | ◐ 2/9 | ◐ 2/9 | ○ 0/9 | ◐ 3/9 | ◐ 3/9 | ◐ 1/9 |
| Class Loading | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 |
| DynamicProxy | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 |
| JVM Calls | ◐ 2/5 | ◐ 3/5 | ◐ 3/5 | ◐ 4/5 | ◐ 4/5 | ◐ 3/5 |
| Serialization | ◐ 3/14 | ◐ 1/14 | ◐ 1/14 | ◐ 5/14 | ◐ 5/14 | ◐ 1/14 |
| Library Analysis | ◐ 2/5 | ◐ 2/5 | ◐ 1/5 | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 |
| Sign. Polymorph. | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 |
| Java 9+ | ● 2/2 | ◐ 1/2 | ◐ 1/2 | ◐ 1/2 | ◐ 1/2 | ◐ 1/2 |
| Non-Java | ● 2/2 | ● 2/2 | ● 2/2 | ● 2/2 | ○ 0/2 | ○ 0/2 |
| **Sum** (out of 123) | 51 (41%) | 65 (53%) | 57 (46%) | 76 (62%) | 75 (61%) | 60 (49%) |

Soundness: all ●, some ◐, or no ○ test cases passed soundly

dles and signature polymorphic methods are also barely supported even though they are posed to replace tradition reflection [42], highlighting the need for continuous evolution of static analyses. Thus, results from these analyses are of unknown trustworthiness for programs using reflection and cannot generally be trusted at all for programs using class loading. This is concerning given the large number of programs that make use of reflection: Landman et al. [136] found that 96% of the Java projects they studied made some use of it and 78% made use of parts of the Java reflection API that are particularly hard to analyze, and Dong et al. [71] found reflection in about half of the studied Android applications.

COMPLEX ANALYSIS PROBLEMS Not only the programs that are analyzed are complex, but the problems to be solved and questions to be answered by static analyses today are complex themselves. They often require solving a multitude of different sub-problems that all have to be reasoned about by the static analysis (e.g., [33]), increasing its complexity. Dependencies between sub-problems can be complex as well; in particular, they must be solved simultaneously if they cyclically

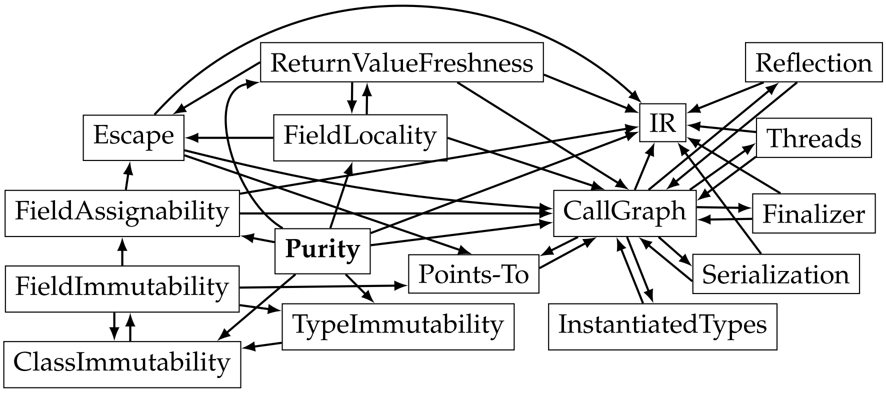


Figure 1.1: Dependencies Between Sub-problems for a Purity Analysis

depend on each other. This requires different parts of the static analysis dealing with different sub-problems to exchange intermediate results and to improve upon these until a final solution is found. Analysis developers have to keep these interactions in mind at all times.

For example, many of today's static analysis problems are interprocedural, i.e., they require not only local information from one method but to consider multiple methods and the call relationships between them. Thus, to solve the actual problem, these static analyses first need to construct a call graph that captures these method-call relationships. However, constructing call graphs faces different sub-problems in itself due to complex language features depending cyclically on each other, as described above.

As an illustration of a complex analysis problem, consider once again the running example of *log4shell*. Identifying the vulnerability might have to include solving multiple distinct analysis problems: constructing a call graph is needed to discover the connections between the different methods from the logging method to the substitution process to the protocol handlers. A dataflow analysis is necessary to find that data in the log message can influence the executed protocol handlers and that code loaded from the network might be used reflectively. Analysis of reflective code might be required to see that the untrusted code can actually be executed. These analysis problems are all different from each other and may require different approaches to solve them, yet an analysis to identify the vulnerability needs to incorporate them all at the same time and also consider their interactions.

Figure 1.1 shows another example of interdependent sub-problems of an analysis: for the seemingly simple task of a purity analysis, i.e., an analysis that checks whether methods have side effects and behave deterministically, several sub-problems arise. We built such a purity analysis as a case study in Chapter 10. As for all interprocedural analyses, a call graph is required. Thus, the sub-problems mentioned above for call-graph construction must also be solved. In addition to that, the purity analysis relies on information from several immutability [180] and escape [40] analyses with complex, sometimes cyclic, interdependencies. These in turn use the results of a points-to analysis [78] that has a mutual dependency with the call graph.

COMPLEX TRADE-OFFS Finally, static analysis developers face the problem of balancing between a static analysis' three most important qualities—soundness, precision, and scalability [114]:

- *Soundness* means that the analysis reasons about all potential dynamic program executions. This is important for analysis results to be dependable, which is necessary, e.g., for compiler optimizations that must not alter the behavior of the program, but also for the analysis of software to be used in critical systems. An analysis is said to be more (*empirically* [214]) sound if it reasons about more potential dynamic executions.

- *Precision* means that the analysis reasons only about program executions that can actually occur dynamically. This is important for analysis results to be useful in directing attention to actual issues. Programmers have been found to not engage with static analyses if they produce too many false alarms [19, 198].

- *Scalability*, lastly, is about the practicability of using static analyses at a large scale. Considering that, e.g., the Google Play Store as the main source for software on Android devices has around 2.7 million apps available [7] and the Maven Central repository has almost 30 million software artifacts [158], even small improvements in analysis scalability can incur large savings in the time, cost, and resources required to conduct comprehensive analyses.

Often, these qualities are in conflict with each other, resulting in complex trade-offs, e.g., an analysis can be made more sound by over-approximating more coarsely, but that can reduce precision by considering more program states that cannot be reached in actual program execution. Improved soundness can also hamper scalability if additional program states have to be reasoned about [214, 236]. Full soundness is often hard to achieve, at least while maintaining reasonable levels of precision and scalability. Thus, analysis developers may have to make choices to unsoundly handle programming language features like reflection that pose a challenge for static analysis. Analyses that are unsound deliberately and in a well-documented way are called *soundy* [149]. Similar arguments can be made about precision that can hurt soundness through under-approximation as well as scalability through more complex analyses but can also benefit scalability if it leads to fewer states to be analyzed [26]. These trade-offs get more important yet even more complex if analyses themselves get more complex as, e.g., changing the precision with which one sub-task is handled can affect the soundness, precision, or scalability of other sub-tasks in unexpected ways.

Considering our running example of *log4shell*, these trade-offs become apparent: Given the impact of the vulnerability, it is important to quickly find affected software among the millions of libraries and applications that could potentially use *log4j*, demanding high scalability of any analysis. On the other hand, with the widespread use of *log4j*, it is also important to identify vulnerable software as precisely as possible to make sure efforts are spent where they are needed, not on software that is not actually vulnerable. Yet, this should not compromise soundness, as any software erroneously deemed not vulnerable would remain a significant security risk.

## 1.2 NEED FOR MODULAR, COLLABORATIVE STATIC ANALYSES

The complexities discussed above affect all static analysis implementations. However, how difficult dealing with these complexities is depends on the style in which the static analyses are implemented. We contrast monolithic implementations with modular architectures to show how they can cope with the complexity differently.

MONOLITHIC IMPLEMENTATIONS Traditionally, static analyses have been developed individually in a monolithic fashion, with a single analysis or a small number of tightly coupled analyses to handle the full extent of what is to be analyzed (e.g., [228]). This model of implementation can not scale to the complex challenges that modern static analyses face: all complexities described above and their interactions always have to be kept in mind and reasoned about as a whole. Thus, development of such analyses requires experts that deeply understand both all aspects of the problem and of the design of static analyses. Even for such experts, developing monolithic analyses is a difficult feat in light of the complexities mentioned above:

A monolithic static analysis for a modern programming language has to handle all complex language features at the same time. This requires the analysis developer to have expertise in every single of these programming language features. All possible interactions between the language features also have to be considered in the design of a monolithic static analysis, further complicating implementation and maintenance of the analysis. This is particularly problematic in the face of new and changed features as programming languages evolve.

Complex analysis problems are hard to handle monolithically as well: the sub-problems all have to be addressed at the same time using the same implementation. As they require different algorithms and data structures to be solved efficiently, this complicates the analysis' implementation.

Finally, monolithic analysis implementations make balancing complex trade-offs hard. The implementations solving the different subproblems are coupled tightly. Thus, it is not possible to experiment with different implementations to explore and balance the respective trade-offs.

MODULAR ARCHITECTURES Different from monolithic analyses, modular architectures for static analysis implementation compose complex analyses from multiple interacting sub-analyses. Each subanalysis can then focus on a limited problem and contribute to a complex overall analysis by communicating with other sub-analyses [208]. This allows sub-analyses to be developed and reasoned about in isolation by developers that are knowledgeable only of the respective sub-problem.

A modular architecture allows static analyses to handle each complex programming language feature in isolation. Modules handling each feature can be implemented by respective experts. Later, these modules can be composed with other modules that handle the actual analysis problem. As programming languages evolve, only the affected modules need to be kept up to date and new language features can be supported by adding additional modules.

The individual sub-problems of a complex analysis problem can also be mapped to individual modules. This allows them to be tailored to the respective sub-problem, using the most natural and efficient algorithms and data structures. The modules can be implemented by experts in the respective kinds of static analyses and then combined to solve the larger overall problem.

Exploring and fine-tuning the trade-offs between soundness, precision, and scalability is enabled by modularization: individual modules can be combined and exchanged in a plug-and-play manner in order to reach different points in the design space for the solution to the overall analysis problem.

CHALLENGES FOR MODULAR STATIC ANALYSIS ARCHITECTURES Modular static analyses have the potential to alleviate monolithic analyses' problem of having to reason about the full extent of the problem and analysis all at once. However, implementing them can still be challenging. The need for communication between different modules often imposes significant restrictions on how analyses can exchange results and how they can be implemented, limiting the possible interactions as well as the kinds of analyses that can collaborate.

We say that a static analysis implementation is *modular* if a complex analysis can be composed from individual sub-analyses (*modules*) that use each other's results. However, modularity alone does not imply the way the sub-analyses interact with each other. In particular, it is necessary to compute the fixed point of different sub-analyses simultaneously: Consider, for example, an analysis that propagates the values of constant variables and an analysis that identifies expressions that can be evaluated at compile time as they use only constants. Executing the constant propagation first means that the results of constant expressions cannot be propagated as they have not yet been identified. Instead identifying constant expressions first, on the other hand, will miss expressions that use constant variables instead of constant literal values as the constant variable values have not yet been propagated. This can be solved by re-executing analyses several times, but re-analysis comes at the price of scalability and the question of which analyses to execute in which order and how often is complex and may be dependent on the individual programs analyzed. Entire books revolve around recommendations for execution orders of static analyses in optimizing compilers [165].

We thus define *collaborative* static analyses as analyses where submodules can interact in an interleaved way, benefiting from each other during a single, combined execution. Such collaboration poses new questions on the interaction of sub-analyses. They could be restricted to using specific data representations that are shared between all sub-analyses, but this makes it difficult to naturally express analyses in terms of the data they actually work upon. It is also a threat to analysis scalability as it precludes the use of data structures optimized specifically for particular analyses. In order to ensure termination, it may also be necessary to restrict the queries that sub-analyses can pose to each other to be strictly reducing in size or complexity according to a suitable metric [116]. However, this means that cyclic interdependencies cannot be resolved. Finally, it may be necessary to define an explicit super-analysis for each required composition of sub-analyses (e.g., [21]). This then limits experimentation with different compositions to explore different trade-offs between soundness, precision, and scalability.

## 1.3 PROBLEM STATEMENT

Software has become ubiquitous and so have the needs to ensure its correctness and security. Static analyses can be used for this purpose, but the complexity of modern programming languages, software systems and the problems to be solved by static analyses increase the burden of developing them. Traditional monolithic analyses cannot scale to cope with these complexities. This is even more true when considering the demands to trade-offs between the soundness, precision, and scalability of static analyses.

Modular, collaborative static analyses are thus needed where subanalyses for isolated sub-problems, implemented by domain experts, can be composed in a plug-and-play manner to solve complex analysis problems. Defining a generic architecture for such modular, collaborative analysis, however, is challenging in itself. Restrictions on the interaction between sub-analyses can limit the applicability and usefulness of the analyses. We thus conclude as this thesis' main goal that:

> A general framework for modular, collaborative program analysis should allow for complex systems of analyses that offer good soundness, precision, and scalability, including exploring the trade-offs between these qualities.

## 1.4 CONTRIBUTIONS OF THIS THESIS

In order to achieve this goal, this thesis makes the following contributions:

**A FRAMEWORK FOR MODULAR COLLABORATIVE ANALYSES** Our main contribution is the development of a novel, flexible framework for modular, collaborative program analysis, the core component of the *OPAL* static analysis framework[1]. This framework is reminiscent of a blackboard architecture [170] and aims to deal with the complexities discussed above. In our framework, sub-analyses can be implemented and reasoned about in isolation and then composed into complex systems of analyses to analyze complex software properties. Because they are ignorant of each other, sub-analyses can also be added, removed and exchanged easily in a plug-and-play manner to explore different trade-offs. As the blackboard architecture makes no restrictions on the data representation, sub-analyses can be implemented in a way that most naturally and efficiently solves the respective problem, ensuring applicability of the framework for a wide range of diverse analyses. The execution order of sub-analyses is also not fixed, allowing for arbitrary interactions and chances for improving scalability through parallelization and intelligent scheduling.

The development of this framework includes multiple individual contributions:

- We systematically derive the requirements for a modular, collaborative static analysis framework from a diverse range of case studies, each of which is a complex analysis (Chapter 2).
- We then use these requirements to develop a static analysis framework based on the blackboard architecture (Chapter 3).
- We provide *RA2*[2], an alternative implementation of our approach that uses a reactive-programming system as its basis (Chapter 4).

[1] Available at `https://www.opal-project.de`; BSD 2-clause open source license
[2] Available at `https://github.com/phaller/reactive-async`; BSD 2-clause license

- We also develop a formal model of the core of our blackboard analysis architecture and show its usefulness in making modular, reusable proofs of analyses' soundness possible (Chapter 5).
- We perform a thorough survey of the related work (Chapter 6).
- We show that the framework can be used to implement the diverse case-study analyses, meeting all the requirements laid out before (Chapter 11).
- We evaluate how our case-study analyses improve over respective state-of-the-art analyses in terms of soundness and precision, and we show how the flexible composition of sub-analyses can be used to explore trade-offs between these properties (Chapter 12).
- We consider how our framework aids the scalability of the analyses, in particular because it is directly amenable to automatic parallelization to harness the resources of multicore processors and can use different strategies for scheduling individual analysis tasks to further improve scalability (Chapter 13).

Additionally, several of our case studies are individual contributions that each improve over the respective state of the art[3]:

A COLLABORATIVE CALL-GRAPH CONSTRUCTION FRAMEWORK *Unimocg* is a framework for composing call-graph construction algorithms from collaborating modules (Chapter 8). *Unimocg* makes it easy to support different call-graph algorithms and complex language features by encapsulating them in decoupled modules. This allows *Unimocg* to achieve consistently cover more languages features soundly across a wide range of call-graph algorithms than other state-of-the-art call-graph construction frameworks. At the same time, *Unimocg* does not sacrifice precision or scalability, being on par with or superior to the state of the art.

MODULAR IMMUTABILITY ANALYSES We developed *CiFi*, a system of modular, collaborating analyses for field-, class-, and type immutability as another case study (Chapter 9). It uses our modular approach to provide precise, modular definitions of field assignability and field-, class-, and type immutability which did not have such proper definitions before. We implemented *CiFi*'s sub-analyses directly following this modular definition and composed them into a system of immutability analyses. This allows for simpler analyses that are more sound and often more precise than the state of the art. We use *CiFi* to explore the prevalence of immutability in real-world software libraries and evaluate the impact of applying a closed- or open-world

[3] The analyses are all available as part of *OPAL*

assumption, i.e., considering the analyzed code to be either completely available or extendable by yet unknown code, on both the precision and scalability of the analysis. Additionally, we provide *CiFi-Bench*, a manually annotated benchmark to test the soundness and precision of immutability analyses.

MODULAR PURITY ANALYSES Our final case study is *OPIUM*, a model and framework for modular analysis of method purity, i.e., deterministic behavior and absence of side effects in methods (Chapter 10). *OPIUM* defines a fine-grained model of purity that extends the state-of-the-art and unifies terminology that has been used inconsistently. *OPIUM* provides purity analyses with different trade-offs between precision and scalability. The analyses make use of and collaborate with immutability analyses such as *CiFi* as well as escape analyses to provide more precise and more fine-grained results than the state of the art. It outperforms the state of the art with regard to execution time as well, benefiting from automatic parallelization enabled by our blackboard analysis architecture.

## 1.5 STRUCTURE OF THIS THESIS

The remainder of this thesis is structured in four parts:

Part I start by introducing the necessary background and terminology and by deriving requirements posed to a modular, collaborative static analysis framework (Chapter 2). It then describes in detail our approach and its implementation in the *OPAL* static analysis framework (Chapter 3). In addition, we introduce *RA2*, an alternative implementation of our approach using a reactive-programming system (Chapter 4). We then give a formalization of the core concepts of our approach and show that our approach allows for modular, reusable soundness proof (Chapter 5). Finally, we survey work related to blackboard architectures and different approaches to modular, collaborative, or parallelized static analyses (Chapter 6).

Part II is dedicated to the individual case studies. We first describe *TACAI*, an intermediate representation based on abstract interpretation that forms the basis for the other case-study analyses (Chapter 7). Next, we introduce *Unimocg*, our modular call-graph architecture that decouples the computation of type information from the resolution of calls to achieve consistently sound handling of language features (Chapter 8). We then discuss *CiFi*, our system of analyses for field-, class-, and type immutability (Chapter 9). As our final case study, we detail *OPIUM*, our unified model and implementation for the analysis of method purity (Chapter 10). The case-study chapters include a validation of the proposed models where applicable.

Part III evaluates our approach, discussing three main research questions. First, we consider the applicability of our approach to a variety of dissimilar analyses, also discussing the effect of our approaches plug-and-play-like exchangeability of analyses (Chapter 11). Second, we compare the soundness and precision of our case-study analyses to the respective state of the art, demonstrating how they outperform it in both regards (Chapter 12). Third, we evaluate the scalability of our implementation, discussing different aspects of performance optimizations and showing that our case-study analyses are on par with or outperform the respective state of the art (Chapter 13).

Part IV concludes this thesis. We start with a summary of the thesis' results (Chapter 14). We then present ideas and directions for future research to expand the applicability of modular, collaborative static analysis and to further improve its scalability (Chapter 15). Finally, we give a closing discussion on the challenges and possible solutions for static analyses going forward (Chapter 16).

## 1.6 PUBLICATIONS

Many contributions of this thesis have been published before at software engineering venues, others are yet under review. Here, we give an overview of these publications and how they have been incorporated into this thesis.

MODULAR COLLABORATIVE PROGRAM ANALYSIS IN OPAL [103] This paper describes our approach of a modular, collaborative static-analysis framework based on the ideas of the blackboard architecture. It includes the definition of our requirements, a short overview of our case studies and presents the approach and its implementation in *OPAL*. In particular, it explains how our approach combines imperative and declarative aspects to achieve exchangeability of analyses and pluggable analysis extensions for improving soundness, precision and, scalability and their trade-offs. It highlights the applicability of the approach and evaluates the effects of exchangeability and the use of data structures that have been optimized for specific analyses. It analyzes the scalability of our proof-of-concept parallelization and shows that our approach outperforms the *Doop* framework [34] in an analysis that *Doop* is highly specialized for. The contents of this paper form the basis of Chapter 2 and Chapter 3. The paper's related work section has been merged into Chapter 6, its evaluation has been integrated into Chapter 11 and Chapter 13.

Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. “Modular Collaborative Program Analysis in OPAL.” In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE'20. Virtual Event, USA: ACM, 2020, pp. 184–196

A PROGRAMMING MODEL FOR SEMI-IMPLICIT PARALLELIZATION OF STATIC ANALYSES [102] In this paper, we present *RA2*, an approach for the semi-implicit parallelization of static analysis that builds on the ideas of the blackboard architecture but uses the *Reactive Async* library. Thus, *RA2* is an alternative implementation of our approach besides the implementation in *OPAL*. In particular, the paper discusses the importance of supporting stateful computations and how semi-implicit parallelization relieves developers from thinking about concurrency issues. The paper's contents are presented in Chapter 4, its related work was integrated into Chapter 6, and its evaluation forms part of Chapter 11 and Chapter 13.

Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. "A Programming Model for Semi-implicit Parallelization of Static Analyses." In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'20. Virtual Event, USA: ACM, 2020, pp. 428–439

A MODULAR SOUNDNESS THEORY FOR THE BLACKBOARD ANALYSIS ARCHITECTURE This paper, which is currently under review, develops a formal model of the core of our blackboard analysis architecture. It then uses this formal model to show that soundness proofs of analyses in the blackboard analysis architecture are modular and soundness of an analysis composed from multiple sub-analyses follows directly from the soundness of the individual sub-analyses. This significantly eases the effort to prove the soundness of complex analyses, as they can be composed freely from any modules already proven sound. We evaluate the applicability on four case-study analyses that are inspired by the case studies also presented in this thesis. Chapter 5 gives the paper's contents, its related work is part of Chapter 6.

Sven Keidel, Dominik Helm, Tobias Roth, and Mira Mezini. "A Modular Soundness Theory for the Blackboard Analysis Architecture." Currently under review

TACAI: AN INTERMEDIATE REPRESENTATION BASED ON ABSTRACT INTERPRETATION [188] In this paper, we introduce *TACAI*, the immediate representation used by all of our case-study analyses. *TACAI* itself is a case study of this thesis as it uses the blackboard architecture to be extensible by modules that improve the precision of type information. We evaluate the effect of exchanging the abstract interpretation domain and find that compared to the Soot framework's [238] *Shimple* intermediate representation, *TACAI* is more efficient to compute and provides slightly more precise type information. This is presented in Chapter 7, while the evaluation is part of Chapter 12 and Chapter 13.

Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. "TACAI: An Intermediate Representation Based on Abstract Interpretation." In: Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. SOAP'20. London, UK: ACM, 2020, pp. 2–7

unimocg: modular call-graph algorithms for consistent handling of language features This paper, which is currently under review, presents *Unimocg*, our modular architecture for call-graph construction. The architecture decouples the computation of type information from the resolution of individual call edges in order to make the modules reusable across all call-graph algorithms. This allows consistently sound handling of language features and also allows modules to all operate on the same, user-chosen, precision of type information instead of relying on ad-hoc solutions. The papers shows that this indeed leads to more consistently sound handling of language features compared to the state of the art without sacrificing precision or scalability, and we also show how further analyses can benefit from the computed type information at the example of an immutability analysis. We discuss *Unimocg* in Chapter 8 and evaluate it in Chapter 11, Chapter 12, and Chapter 13.

Dominik Helm, Tobias Roth, Sven Keidel, Michael Reif, and Mira Mezini. "Unimocg: Modular Call-Graph Algorithms for Consistent Handling of Language Features." Currently under review

cifi: versatile analysis of class and field immutability [195] In this paper, we present *CiFi*, our model and implementation of analyses for the immutability of fields, classes, and types. The model unifies inconsistently used concepts and terminology found in the literature. The implementation consists of four analyses for field assignability and for field-, class-, and type immutability that are combined in a modular way and are supported by other analyses such as *TACAI* and *Unimocg* to achieve better soundness and precision than the state of the art in immutability inference and enforcement. To show the latter, we also propose *CiFi-Bench*, a benchmark of manually annotated tests for immutability analyses that allows their comparison for soundness and precision. The contents of the paper are given in Chapter 9, the evaluation forms part of Chapter 12.

Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. "CiFi: Versatile Analysis of Class and Field Immutability." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering*. ASE'21. Virtual Event, Australia: IEEE, 2021, pp. 979–990

A UNIFIED LATTICE MODEL AND FRAMEWORK FOR PURITY ANALYSES [101] This paper introduces *OPIUM*, our final case study. It unifies inconsistently used terminology found in the literature into a lattice-based model of method purity—lack of side effects and deterministic behavior— and introduces novel levels of purity to uncover more restricted side effects than previously possible. The paper discusses our most precise purity analysis while in this thesis, we also refer to two further implementations that are less precise but more scalable. The analysis described is more precise and more scalable than the previous state of the art and it is also more fine-grained, computing several distinct levels of purity instead of only a single one tailored to a specific purpose. The paper's contents are presented in Chapter 10, its evaluation in Chapter 12 and Chapter 13.

Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. "A Unified Lattice Model and Framework for Purity Analyses." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE'18. Montpellier, France: ACM, 2018, pp. 340–350

### 1.6.1 *Further Publications*

In addition to the publications above, I co-authored further papers that were not directly included in this dissertation. These papers are:

LATTICE BASED MODULARIZATION OF STATIC ANALYSES [74] This paper presented early ideas on a modular architecture for static analysis frameworks. In particular, we discussed the importance of using reified lattices as the sole means of communication between analyses to enable their flexible composition. These ideas were later extended into the blackboard analysis architecture described in [103].

Michael Eichberg, Florian Kübler, Dominik Helm, Michael Reif, Guido Salvaneschi, and Mira Mezini. "Lattice Based Modularization of Static Analyses." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. SOAP'18. Amsterdam, The Netherlands: ACM, 2018, pp. 113–118

JUDGE: IDENTIFYING, UNDERSTANDING, AND EVALUATING SOURCES OF UNSOUNDNESS IN CALL GRAPHS [186] In this paper, we identified why call graphs are often unsound, in particular, which language features are the reason for perceived unsoundness in call graphs. We showed that language features that impact call-graph soundness are frequently used across different analyzed corpora. We studied the soundness profiles of different call-graph algorithms across popular static analysis frameworks and showed that they differ significantly and unexpectedly even across algorithms in a single framework, rendering the comparison of different resulting call graphs bogus. Finally, we proposed to use limited manual effort guided by automated analysis to improve call-graph soundness to a reasonable level. The results of this paper prompted us to develop a modular call-graph architecture where programming language features are handled consistently sound across dissimilar call-graph algorithms. We present this architecture, *Unimocg*, in Chapter 8.

Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. "Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'19. Beijing, China: ACM, 2019, pp. 251–261

## 1.7 MY CONTRIBUTIONS

Much like the static analyses discussed in this thesis, research is a collaborative effort. All of the publications that have been incorporated into this thesis are the work of myself and several co-authors, and I am grateful for their contributions to my research. Pointing out individual contributions is not easy, as we worked closely as a team, discussing ideas together, collaboratively working on building and extending the *OPAL* framework, implementing the case-study analyses, performing evaluations, and writing the publications. Work was often driven by shared, overlapping research interests, where a single analysis and publication furthered the research of several contributors in different ways, mutually benefiting each other. Content from these joint publications has been incorporated into this thesis verbatim, and it is often not possible to distinguish the individual author as the texts have been refined by each of us several times over. In general, for all of the mentioned publications, I was heavily involved in both developing the initial ideas and concepts, and I spent significant work on drafting, revising, and polishing the publications' writing, often contributing significant parts of the final texts. The same is to be said for my advisor Mira Mezini. In the following, I try to point out my own specific contributions as much as possible.

Chapter 1 has been written solely by myself explicitly for this thesis.

Chapter 2 and Chapter 3 are based on [103]. The concepts of *OPAL*'s collaborative approach are the result of intense discussions between Michael Eichberg, Florian Kübler, and myself. I contributed to significant parts of the implementation. In particular, I created the current implementation of the parallel blackboard solver. The evaluation that is part of Chapter 11 and Chapter 13 was jointly performed by Michael Reif, Florian Kübler, and me.

Chapter 4 builds on [102]. I contributed the code for the IFDS solver and analysis and helped to integrate it into the *RA2* framework.

I also aided Jan Thomas Kölzer in debugging and improving the implementation of *RA2*. The evaluation as it is included in Chapter 11 and Chapter 13 was performed by Florian Kübler and me in a joint effort, building on initial work by Jan Thomas Kölzer.

Chapter 5 is taken from a yet unpublished paper that is currently under review as stated above. The concepts have been developed from intense discussions between Sven Keidel and me. I ensured that the model matches the core ideas of our blackboard analysis architecture as closely as possible and helped in designing the case studies to be similar to the ones presented in Part II. I particularly contributed to the design of the main definitions and to the idea of using interfaces to abstract over different kinds of values for improved extensibility.

Chapter 6 merges parts of [103], [102], and the yet unpublished paper that Chapter 5 builds upon. I extended further upon these parts, adding additional related work for the purpose of this thesis.

Chapter 7 is based on [188]. My main contribution is the idea of supporting pessimistic analyses dual to the optimistic analyses we were exclusively developing at the time. I also contributed to the evaluation that is part of Chapter 12 and Chapter 13.

Chapter 8 is from a yet unpublished paper currently under review. The initial idea was sparked from our previous research in [186] and developed in discussion between Michael Reif, Florian Kübler, and me. The actual design and implementation is my own work, parts of the implementation of some modules were contributed by Florian Kübler, Michael Reif, and Andreas Bauer. The evaluation performed by Tobias Roth and me is part of Chapter 11, Chapter 12, and Chapter 13.

Chapter 9 builds on [195]. Ideas and concepts were developed between Tobias Roth, Michael Reif, and me. I contributed to the implementation and debugging of the analyses and parts of the implementation build on immutability analyses that I developed or extended from previous work to support my analyses for *OPIUM*. The evaluation that is part of Chapter 13 was performed by Tobias Roth and me.

Chapter 10 is based on [101]. All of the publication is my original work, Florian Kübler contributed the escape analyses used. I also performed the evaluation incorporated into Chapter 12 and Chapter 13.

Chapter 11, Chapter 12, and Chapter 13 have been drawn from the individual publications as mentioned above.

Chapter 14, Chapter 15, and Chapter 16 finally have again been written solely by myself, using excerpts from the individual publications.

As for the publications not part of this thesis, I contributed significantly to the ideas and text of [74] and provided *OPIUM* for the evaluation. For [186], I revised, streamlined, and extended the benchmark from [187] and contributed to the paper's text, evaluation, and presentation of results.

# Part I

# APPROACH

The first part of this thesis presents *OPAL*, our main contribution. *OPAL* is a general framework for modular, collaborative program analysis, the need for which we motivated in Chapter 1. *OPAL* is based on the *blackboard architecture* [170], a software architecture that allows decoupled modules to interact and communicate via a central data storage called the *blackboard*. We first analyze the requirements that have driven the design of *OPAL*. Then we present the details of *OPAL*'s architecture and implementation. We formalize the core concepts of our approach to allow proving the soundness of analyses implemented in *OPAL*. Finally, we discuss research related to our approach.

The chapters of this part are as follows:

BACKGROUND AND REQUIREMENTS In Chapter 2, we introduce the necessary background and terminology. We then present a series of case studies from which we derive the requirements that a static analysis framework has to fulfill to support modular, collaborative analyses. We will discuss these case studies in more depth in Part II as they are contributions of their own.

ARCHITECTURE We present our architecture and its implementation in *OPAL* in Chapter 3. We show how it is reminiscent of the blackboard architecture and explain in detail how the individual requirements from Chapter 2 are supported by it. This includes how program properties are represented, how analyses are structured and declared, and how results are expressed. We explain constraints on analyses and the fixed-point computation and how the computation can benefit from scheduling and parallelization.

ALTERNATIVE IMPLEMENTATION IN REACTIVE ASYNC We discuss *RA2*, a second implementation of our approach, in Chapter 4. It is based on the *Reactive Async* library [97] for lattice-based reactive computation. It shows how the blackboard analysis architecture can be implemented using a generic library for asynchronous computation and shows semi-implicit parallelization as well as different scheduling strategies.

FORMALIZATION In Chapter 5, we present a formal model for the core concepts of our approach. We show how this formal model can be used to prove the soundness of analyses implemented in the blackboard analysis architecture. These soundness proofs are *compositional*, i.e., the soundness of a complex analysis follows directly from the individual soundness proofs of the sub-analyses. We showcase these proofs based on simple analyses inspired by our case studies from Chapter 2.

RELATED WORK In Chapter 6, we give an overview of the history of research on blackboard systems, the architecture paradigm that is the basis of *OPAL*. We then survey different approaches for static program analysis, in particular for achieving modularity, collaboration, and parallelization of analyses. In addition, we discuss other general frameworks for static analysis.

# 2 TERMINOLOGY AND REQUIREMENTS

In this chapter, we first introduce blackboard systems and present further terminology used throughout this thesis. We then present three case studies, each a complex system of multiple analyses. From these case studies, we derive requirements that a general framework for modular, collaborative program analysis should fulfill.

## 2.1 TERMINOLOGY

The *Blackboard Architecture* [48] is a software architecture to solve problems using independent modules. The blackboard metaphor describes problem-solving systems as a set of independent experts that communicate via a central data storage, the *blackboard*. Experts find data on the blackboard that helps them make progress toward an overall goal, adding their respective results back to the blackboard for other experts to pick up. The blackboard allows coordinating the collaborative work of otherwise decoupled expert modules, known as *knowledge sources*. The latter contribute information to the blackboard toward collaboratively reaching an overall goal. This information may be partial and can be improved and extended later on. The blackboard notifies knowledge sources about when new information they might require is available. This happens through a control mechanism that decides which knowledge sources should be executed in what order. Information in the blackboard can then be queried by the knowledge sources, which produce further information. Each execution of a knowledge source is called an *activation*.

*Entity:* The term is used to characterize anything one can compute some information for. Entities can be concrete code elements, e.g., classes, methods, or allocation sites, or abstract concepts such as all subtypes of a class. The set of entities is not necessarily defined a priori and can be created on-the-fly while analyses execute.
*Property Kind:* The term characterizes a specific kind of information that can be computed for an entity, e.g., the mutability of classes, the purity of methods, or the callees of a specific method. Each property kind represents one lattice of possible values.
*Property:* The term characterizes a specific value in the lattice of some property kind that is attached to some entity, e.g., a class can be mutable or immutable, a method can be pure or impure, and a particular method may invoke a specific set of methods. Per entity, at most one property of a specific kind can be computed.

*Analysis:* The term characterizes an algorithm that, given an entity, computes a certain kind of property for that entity. Analyses are knowledge sources in the sense of the blackboard architecture; the properties they compute constitute the information that they contribute to and/or query from the blackboard. We say that *an analysis computes a property kind* as shorthand for "an analysis computes properties of that property kind for a given kind of entity".

## 2.2 REQUIREMENTS

In Chapter 1, we motivated the need for a general framework for modular, collaborative static analyses. However, it is not clear what is required from such a framework in order to support a broad range of analyses. It is imperative to first systematically gather a requirements profile before designing any such framework.

In this section, we thus discuss three case studies of different analyses composed of interrelated sub-analyses in order to distill a list of requirements. An overview of these requirements will be given at the end of this section in Table 2.1.

During the discussion, we *emphasize* concepts whenever they occur. The case studies represent very dissimilar kinds of analyses. In particular, they require different kinds of lattices, including set-based lattices (e.g., in 2.2.2) and singleton-value lattices (e.g., in 2.2.3). This motivates the first requirement: Static analysis frameworks must support varied domain lattices (**R1**).

### 2.2.1 *Three-Address Code*

The first case study is an analysis to produce a three-address code representation[1] (TAC) of Java Virtual Machine bytecode, presented in more detail in Chapter 7. In its basic version, TAC uses def/use, type, and value information (including constant propagation) provided by an abstract-interpretation-based analysis (AI). To increase precision, AI may be enhanced with analyses that refine type and the value information for method return values and fields. However, such additional analyses may negatively affect the runtime. Hence, systematic investigation of the trade-off between precision and scalability is needed to decide whether to use such additional analyses on a case-by-case basis. To this end, a separation into sub-analyses that are pluggable, i.e., that can be enabled or disabled easily by the end user, is beneficial. In general, we derive the following requirements regarding support for modular pluggable analyses.

For systematically studying precision/soundness/scalability tradeoffs, static analysis frameworks should support enabling and disabling

[1] A three-address code is an intermediate representation where, except for method calls, operations have at most two source operands and one target.

interdependent analyses (**R2**). To maximize pluggability, analyses should be defined in decoupled sub-analyses, and yet be able to collaboratively compute properties (*collaborative analyses*). As individual analyses can be disabled, it should be possible to specify soundly over-approximated *fallback values*[2] for the properties they compute, to be used by dependent analyses in lack of actual results (**R3**). For example, if not executing an analysis to refine the type information for fields, the fields' static types should be used by AI without AI having to know that the refinement analysis was not executed.

Moreover, an approach for modular collaborative analyses should support their *interleaved execution* without them knowing about each other's existence (**R4**). Two analyses are executed interleaved if they can interchange *intermediate results*. This is important for optimal precision [52]: knowledge gained during the execution of analysis $A_1$ may be used by the execution of another analysis $A_2$ on-the-fly to refine its result and, in turn, this may enable further refinement for $A_1$. The precision of field- and return-value refinement analyses profits from interleaved executions, as they depend on each other cyclically. If a method `m` returns the value of a field `f`, then `m`'s return value depends on `f`'s value. If the value returned by `m` is written into `f`, then `f`'s value also depends on `m`'s return value.

However, interleaved execution must in specific cases be suppressed to ensure correctness. This is the case for the composition of *pessimistic* and *optimistic* analyses. Pessimistic analyses start with a sound but potentially imprecise assumption and eventually refine it. Optimistic analyses start with an unsound but (over)precise assumption and progress by reducing (over)precision toward a sound result. Field- and return-value refinement analyses are pessimistic—the declared return type of method $m$, say `List`, is a sound but eventually imprecise initial value for the return-value analysis; during the execution, the analysis may find out that $m$ actually returns the more precise result, say `ArrayList`. AI is an optimistic analysis—it starts with the unsound assumption that all code is dead and refines it by adding statements found to be alive towards a sound but potentially less precise result. Optimistic and pessimistic analyses are *incompatible* for interleaved execution because they refine the respective lattices in opposite directions. As a result, exchanging intermediate results may cause inconsistencies, thereby violating monotonicity. Thus, the analysis framework must enforce that only *final results* of pessimistic analyses are passed to dependent optimistic analyses (and vice-versa), avoiding interleaving and *suppressing* non-final updates (**R5**).

For illustration, consider the example of some piece of code $c$, that contains a call to a method $m_1$, which is mutually recursive with a

[2] To minimize the effect of fallback values on precision, it makes sense to compute the fallback by using locally available information, e.g., using declared type information, instead of always returning the same over-approximated value.

method $m_2$, conditioned on a field $f$ being an instance of `LinkedList`. To analyze $c$, we combine a field-value analysis $FA$, an $AI$ analysis, and a call-graph-construction algorithm $CG$. Assume that $FA$, a pessimistic analysis, initially reports the type of the field $f$ to be `List`. Given this information, AI would optimistically consider $c$ to be live and $CG$, hence, will consider both $m_1$ and $m_2$ to be reachable. Because of the mutual recursion (and also because of the monotonicity requirement), this result cannot be changed later if $FA$ finds out that $f$ can only contain `ArrayLists`. If, however, the latter information was present when AI analyzed the code, $c$ would have been marked as dead because it is conditioned on requiring a `LinkedList` instead. As a result, $CG$ would have marked $m_1$ and $m_2$ as unreachable. Thus, the result of this combination of analyses is non-deterministic and possibly incorrect (imprecise, if $m_1$ and $m_2$ are falsely reported to be reachable).

### 2.2.2 *Modular Call Graph Construction*

Interprocedural analyses presume a call graph: Given method `m`, a call graph provides information about (a) methods that may be invoked at a call site in `m` (callees) and (b) call sites from which `m` may be invoked (callers). We use call-graph analyses, further detailed in Chapter 8, to motivate the need for supporting further kinds of execution interleaving (beyond **R4**) as well as further requirements. The previous case study illustrated the need for interleaved execution of interdependent analyses that calculate different properties and operate on different entities (composition of analyses for refining field and return values with TAC). The call-graph use case illustrates two further kinds of interleaved execution.

First, we need interleaved execution of multiple instances of the same analysis operating on different code entities to collaboratively compute a single property, whereby each instance contributes partial results (**R6**). For example, different executions of a call-graph analysis for different callers of a method $m$ need to contribute their *partial results* to collaboratively derive all of $m$'s callers (computing callers of a method is inherently non-local).

Second, we also need to support interleaving of independent analyses that collaboratively compute a single property (**R7**). Consider, e.g., the computation of the callees of `m`. A call-graph analysis can in principle consider $m$ in isolation. A monolithic analysis for callees is nonetheless not suitable. It makes sense to distinguish between one sub-analysis that handles standard invocation instructions (e.g., Class-Hierarchy Analysis (CHA) [62], Rapid Type Analysis (RTA) [12], or points-to-based [34] call-graph analysis) and sub-analyses dedicated to non-standard ways of method invocation through specific language features, e.g., reflection, native methods, or functionality related to threads, serialization, etc. Non-standard invocation requires specific handling (e.g., one may deliberately not want to perform reflection resolution, or perform it based on dynamic execution traces). By offering such specialized analyses as decoupled modules, they become highly reusable and can be combined with different call-graph analyses for standard invocation instructions. This makes the call-graph construction highly configurable for fine-tuning its scalability and sound(i)ness. Hence, not only a method's callers but also its callees need to be computed collaboratively. Here, different sub-analyses targeting different language features, rather than different executions of the same call-graph sub-analysis, contribute to the same property.

Handling special language features may even rely on integrating results of external tools or precomputed values (**R8**). For instance, one may choose to use the results of the dynamic TamiFlex [27] analysis for reflective calls, or the results of external tools for analyzing native methods.

The call-graph case study also motivates support for specifying precise *default values* (**R9**) (in addition to sound fallback values). Consider the case of an unreachable method $m$. The call-graph analysis will never compute callees or caller information for $m$. However, this lack of results is an inherent property of the entity and not the result of a missing or disabled analysis. A sound fallback value for $m$ to compensate for the deactivation of the call-graph sub-analysis may have to include all methods and hence be too imprecise. Instead, analyses depending on the call graph should get the information that $m$ is unreachable—the precise default value. The analysis developer knows such information and should be enabled to tell the framework.

Finally, consider that call-graph construction unfolds along the transitive closure of methods reachable from some entry points. Hence, it does not make sense to execute the decoupled sub-analyses collaboratively constructing the call graph—each handling a particular language feature—globally on all methods of a program. Instead, they should be *triggered* only when the overall analysis progress discovers a newly reachable method. Hence, the framework must support triggering analyses once the first (intermediate) result for a property is recorded (**R10**).

Our previous work [186] provides empirical evidence that encoding an RTA sub-analysis and sub-analyses for language-specific features as collaborative, interleaved modules results in more sound call graphs and better performance compared to call graph analyses of the *Soot* [238], *WALA* [111], and *Doop* [34] frameworks.

### 2.2.3 *Immutability, Purity, and Escape Analysis*

Our final case study involves analyses for class and field immutability (cf. Chapter 9), method purity (cf. Chapter 10), and escape information [40, 131]. The latter includes aggregated information on field locality and return-value freshness (cf. Section 10.3.4). The analyses interact tightly and compute properties that may be relevant for both end users (e.g., method purity) and further analyses (e.g., escape information). Complex dependencies exist between all these analyses. To fine-tune the precision/scalability trade-off, several analyses for these property kinds with different precision can be exchanged as needed; all are *optimistic* and use TAC and/or the CG information.

The analyses in this subsection illustrate the need for further kinds of activation modes in addition to triggered analyses, illustrated in the previous subsection: (a) *eager analyses*, which refers to computing an analysis for all entities in the analyzed program, and (b) *lazy analyses*, i.e., executing an analysis $A_1$ only for the entities for which the property that $A_1$ computes is queried by some (potentially the same) analysis $A_2$. A further requirement shown by analyses in this subsection is that the framework should allow analyses to enforce an execution order that overrides the one determined by the solver.

Since the results of analyses in this case study may be of interest to the end user, it is useful to compute them for all possible entities eagerly (**R11**), i.e., computing the immutability of all fields in the program. However, when the field immutability is only used to support, e.g., the purity analysis, it may be beneficial for scalability to compute it lazily (**R12**), i.e., only for the fields for which immutability is queried by the purity analysis. This illustrates that we need both eager and lazy execution modes. Eager and lazy versions of one analysis can typically share the code and only be registered with the framework differently. The class immutability analysis also illustrates the need to configure the framework with analysis-specific execution orders (**R13**): For performance reasons, it makes sense to analyze classes in a top-down order starting with parent classes before their children.

Our evaluation in Part III provides empirical evidence for the requirements stated in this section. Implementations of the immutability and purity sub-analyses of this case study (and transitively the escape sub-analyses) as collaborative analyses with interleaved execution showed higher precision, more fine-granular results, and similar performance characteristics compared to respective state-of-the-art tools.

### 2.2.4 *Summary*

Table 2.1 summarizes the requirements along the case studies motivating them, grouping them by whether they relate to representing analysis results using lattices and values, to the composition of analyses, or to how computations are initiated. Existing frameworks do not satisfy all of these requirements. Imperative frameworks lack support for modularity, especially **R5**, **R6**, and **R7**. Declarative approaches, e.g., *Doop* [34], have other limitations: Being bound to relations for modeling properties, they cannot express the range of different analyses represented by our case studies (**R1**). They also fail to support sound interactions between incompatible analyses (**R5**). By giving the solver full control, they do not support different analysis-specific activation modes (**R10**-**R13**).

Table 2.1: Summary of Requirements

| | |
|---|---|
| *Lattices and values* | |
| **R1** | Support for different kinds of lattices (2.2.1, 2.2.2, 2.2.3) |
| **R3** | Fallbacks of properties when no analysis is scheduled (2.2.1, 2.2.3) |
| **R9** | Default values for entities not reached by an analysis (2.2.2) |
| *Composability* | |
| **R2** | Support for enabling/disabling individual analyses (2.2.1, 2.2.2, 2.2.3) |
| **R4** | Interleaved execution with circular dependencies (2.2.1, 2.2.2, 2.2.3) |
| **R5** | Combination of optimistic and pessimistic analyses (2.2.1) |
| **R6** | Different activations contributing to a single property (2.2.2) |
| **R7** | Independent analyses contributing to a single property (2.2.2) |
| *Initiation of property computations* | |
| **R8** | Precomputed property values (2.2.2, 2.2.3) |
| **R10** | Start computation once an analysis reaches an entity (2.2.2) |
| **R11** | Start computation eagerly for a predefined set of entities (2.2.3) |
| **R12** | Start computation lazily for entities requested (2.2.1, 2.2.3) |
| **R13** | Start computation as guided by an analysis (2.2.3) |

# 3 ARCHITECTURE

*OPAL* is the first static analysis framework to build upon the concept of blackboard systems (cf. Chapter 2): Static analysis modules correspond to knowledge sources; the store that manages the computed properties corresponds to the blackboard. *OPAL* combines imperative and declarative programming styles for analyses. Whereas declarative approaches facilitate modularity and automated, analysis-independent optimizations, imperative approaches foster manual, analysis-specific optimizations.

In this chapter, we first describe the approach taken by *OPAL*. We explain in detail what an analysis developer has to do in order to implement modular, collaborative analyses in *OPAL*. Additionally, we describe the design space of analyses in *OPAL* and how *OPAL* automatically coordinates the execution of analyses.

## 3.1 OVERVIEW

In *OPAL*, the developer of an analysis `A`: (a) implements the lattice representation of the property values computed by `A` (3.2), (b) implements two *imperative* functions the so-called *initial analysis function* (IAF) and *continuation function* (CF) (3.3), (c) declares the property kinds computed by `A` and properties `A` depends on (3.4), and (d) defines how `A`'s results are reported to the blackboard (3.5). Guided by the declared dependencies, the blackboard and its fixed-point solver coordinate the execution of the analyses, thereby (e) ensuring all execution constraints (3.6), (f) performing fixed-point computations, whenever circular dependencies are involved (3.7), and (g) automatically scheduling and parallelizing the execution of analyses (3.8).

## 3.2 REPRESENTING PROPERTIES

Values of a property kind constitute a lattice structure. *OPAL* supports singleton value-based, interval, or set-based lattices (**R1**). A lattice's bottom value models the best possible value (e.g., pure for method purity); its top value the sound over-approximation (e.g., impure). Lattices must satisfy the ascending (descending) chain condition to ensure termination of optimistic (pessimistic) analyses. When defining a property kind, developers can choose the most suitable data structures for efficiency.

Developers can also specify *fallback* and *default* values. The blackboard will return the *fallback value* for some requested property, `p` of kind `k`, if no analysis is available for `k` (**R3**). As it is a sound over-approximation, the lattice's top value is a good choice—however, the fallback value can also be provided by a "proxy" analysis function that does not query the blackboard, avoiding cyclic dependencies. The blackboard will return a *default value* for `p`, if an analysis is available but did not produce any result for some entity (**R9**). For instance, call graph analyses only examine methods reachable from entry points - for any non-reachable method, `m`, a default value can be used to state that `m` is dead and has no (relevant) callees. A sound fallback value would include all possible methods as callees of `m`; thus, in this case, the default value provides more information than a fallback value. If no default value is declared, the fallback value is returned.

```
1 sealed trait ClassImmutability extends PropertyKind {
2   def fallback(Type theClass) = MutableClass
3 }
4 case object ImmutableClass extends ClassImmutability
5 case object MutableClass extends ClassImmutability
```

Listing 3.1: Class Immutability Lattice

Developers implement property kinds by specifying an interface, which can be used to access and manipulate the property values. When the `PropertyKind` trait is extended, the framework assigns an identifier that analyses use to query the blackboard for properties of that kind. Listing 3.1 shows exemplary Scala code of a simple class immutability property kind. Lines 1 to 3 define the base trait for the property kind and give a sound fallback value in Line 2. The two possible property values are defined in Lines 4 and 5.

## 3.3 ANALYSIS STRUCTURE

An overview of *OPAL*'s analysis structure is shown in Figure 3.1. As mentioned in Section 3.1, the analyses are structured in two parts: An *initial analysis function (IAF)* and one or more *continuation functions (CFs)*. These functions can be implemented in any way, as long as they provide their results as defined by the property kind.

For each entity `e` to be analyzed by `A`, `A`'s IAF is executed. The IAF collects information directly from `e`'s bytecode in order to compute its result. If it needs additional information pertaining to some other entity `e2` or from another analysis that computes a property kind `k`, the IAF queries the blackboard for these dependencies, using the identifiers of `e2` and `k` to find the relevant information (arrow 1. in Figure 3.1). The blackboard will return the currently available value (2.). This value may, however, not be available, or not be final, either because the respective analysis was not yet executed or because it has dependencies that yet need to be satisfied. Once the IAF completes

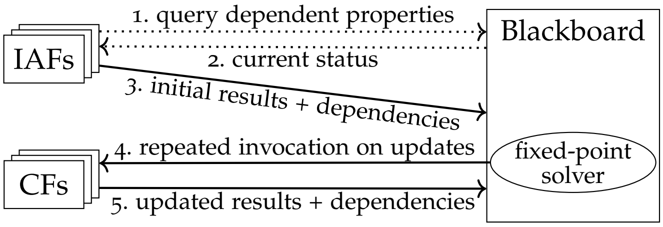

IAFs
CFs
Blackboard
fixed-point solver
1. query dependent properties
2. current status
3. initial results + dependencies
4. repeated invocation on updates
5. updated results + dependencies

Figure 3.1: Overview

analyzing the entity, it returns to the blackboard (a) a result computed based on the currently available information and (b) any remaining dependencies, along with a continuation function (CF) (3.). Similar to the solver of *declarative* frameworks, the blackboard resolves dependencies and automatically invokes the CFs whenever updates to these dependencies become available (4.). On completion, CFs also return their updated results to the blackboard (5.), potentially triggering the execution of other CFs. While the IAF is written imperatively (dotted queries in Figure 3.1), the subsequent execution is performed similarly to declarative frameworks (straight lines): results declare their dependencies and the solver is responsible for satisfying them. Executions of the IAFs and CFs are called *analysis activations*. To ensure determinism, *OPAL* executes the activations for a single property sequentially, while IAFs and CFs for other properties can execute concurrently.

As analyses get notified about dependency updates through the invocation of the CF, it is not necessary that dependencies are computed before or when they are queried. Instead, they can be computed asynchronously and lazily, i.e., on-demand (**R12**). This also allows *OPAL* to handle cyclic dependencies (**R4**).

Apart from adhering to this basic structure, developers may use any suitable strategy to implement an analysis `A`. Analysis `A` may, e.g., focus on specific statements instead of traversing the entire code of a method (*OPAL* provides pre-analyses to query specific parts of the code, e.g., all statements that access a specific field). Also, analyses can internally use any data structure suitable to achieve good scalability. For illustration, Listing 3.2 shows an excerpt from a simple class immutability analysis' initial analysis function. The IAF is given the entity to analyze (Line 1). Lines 3 to 7 show how to retrieve and handle properties required to compute the IAF's result: The required property (the mutability of an instance field of the analyzed class) is queried from the blackboard (Line 3) and based on the returned value, the IAF may compute its result (as in Line 4) or keep the dependency in a list of dependees (Line 6) to return it alongside an intermediate result later (Line 9). Line 9 also specifies the continuation function to be invoked when any of the properties in `dependees` is updated. We do not show the code for that CF here, as its implementation is very similar to

```
 1 def analyze(theClass: Type) = {
 2   [...]
 3   Blackboard.get(field, FieldImmutability) match {
 4     case _: MutableField => return Result(theClass, MutableClass)
 5     case dependee: ImmutableField =>
 6       if (!dependee.isFinal) dependees += (field -> dependee)
 7   }
 8   [...]
 9   Result(theClass, ImmutableClass, dependees, continuation)
10 }
```

Listing 3.2: Class Immutability Analysis

Lines 4 to 9, i.e., based on the updated value, the (intermediate) result of the CF is determined.

The implementations of the analyses must satisfy two constraints:

First, they must ensure *monotonicity of result updates* according to the used lattice. Analyses that optimistically start at a lattice's bottom value may only refine approximations upwards; pessimistic analyses only downwards. *OPAL* can automatically check the monotonicity of updates. Monotonicity allows analyses to know which refinements of intermediate results are still possible.

Second, analyses must be *scheduling independent*: Whenever they receive the value of some other property they depend on, they must use the information provided by that value to compute the result of the current activation, i.e., they may not defer the incorporation of the newly gained information to a later activation of a continuation function. Immediately using the provided value ensures that all available information is used regardless of whether the continuation is later scheduled for execution. This is important, because no further activation may ever occur in case there are cyclic dependencies. For example, once the immutability analysis of a class `C` knows that `C`'s instance field `f` is mutable, it may no longer report that `C` could be immutable. The developer of an analysis `A` is responsible for ensuring that `A` is scheduling independent. However, scheduling independence can usually be achieved easily by following a simple pattern to immediately evaluate the effect of updated dependencies.

## 3.4 declarative specifications

On top of the IAF and CF, the developer of an analysis `A` specifies (a) the property kinds computed by `A`, (b) its dependencies, (c) on which entities `A` will be executed and (d) when the blackboard should start `A's` execution. These specifications are evaluated when the analysis is registered with the blackboard, before the latter takes over control of

```
1 override def derivesLazily = Optimistic(ClassImmutability)
2 override def uses = Set(Optimistic(FieldImmutability))
3 override def register() = {
4   Blackboard.set(Type.Object, ImmutableClass)
5   val analysis = new ClassMutabilityAnalysis
6   Blackboard.registerLazyAnalysis(this, analysis.analyze)
7 }
```

Listing 3.3: Registration of Class Immutability Analysis

analysis activation. When registering analyses, developers may also report precomputed values to the blackboard (**R8**).

The specification of the computed property kinds also states whether intermediate results are optimistic or pessimistic and whether the analysis contributes to a collaborative computation or intends to be the only analysis computing the specified property kinds. Dependency specifications state other property kinds on which `A` depends (which `A` queries) and whether `A` can process optimistic/pessimistic intermediate values or final values only.

Analyses can eagerly select a set of entities (e.g., all methods of the analyzed program) if it is likely necessary to perform the analysis for all of these entities (**R11**). This is, e.g., useful for analyses that are of interest to the end user, e.g., if the user is interested in the purity of all methods. Alternatively, analyses can be registered to be invoked lazily [26, 115]. Lazy analyses only compute a property if that property is queried (**R12**) by another analysis or by the end user. Finally, an analysis can specify a property kind *k* such that it is started for every entity for which *k* has been computed (**R10**).

Some analyses benefit from enforcing a specific order for computing the properties for different entities (**R13**). For instance, the class mutability analysis benefits from traversing the class hierarchy downwards, such that results for a parent class are available before any subclass is analyzed. In *OPAL*, this is supported by enabling the developer of an analysis `A` to declare a number of computations to be scheduled whenever `A` returns a result to the blackboard.

For illustration, Listing 3.3 shows the registration code for a class immutability analysis. Line 1 declares that the analysis optimistically and lazily derives class mutability. Line 2 declares that in performing its computation, it may require field mutability and that it can handle intermediate results for this property if they were computed optimistically. This declaration is complete: No property kinds other than field mutability (and class mutability) may be queried by this analysis. Line 4 registers a predefined value (**R8**) stating that the base class `Object` is immutable. The IAF `analyze` is registered as a lazy analysis in Line 6, i.e., the mutability of a certain class will only be computed on demand, e.g., when a purity analysis queries it.

## 3.5 REPORTING RESULTS

As already mentioned, analyses write intermediate and final results to the blackboard. They can report results for each single entity individually or for multiple entities at the same time. A result consists of a single lattice value representing the new value for the property or of an update function for updating the property's current value (as recorded in the blackboard) to incorporate the new result.

An update function is used for properties whose computation is not localized to a specific part of the program, e.g., the callers of a method. For such properties, constraint-based analyses [4, 169] have been used in the past; declarative analyses also provide such updates, called deltas, that only specify the change to the property value instead of the full new property value. The update function merges the results of one activation with the current state of the property (e.g., adding a new caller to an existing set of callers). This way, activations of one or of different analyses can collaboratively contribute to a property (**R6**, **R7**).

## 3.6 EXECUTION CONSTRAINTS

Once the end user chooses a set of analyses to be executed (**R2**), *OPAL* uses the declarative specifications (Section 3.4) to check and automatically enforce restrictions on analyses that can be executed together. First, it ensures that any property kind is computed either by at most one analysis or collaboratively by multiple analyses; this is to avoid that conflicting results are reported to the blackboard. Second, if several analyses derive a property kind collaboratively, *OPAL* ensures that they are all either optimistic or pessimistic. Finally, *OPAL* ensures that all property kinds required by any analysis are derived by another analysis or there is a fallback value provided; this is to ensure that dependencies can be satisfied.

*OPAL*'s blackboard may run optimistic and pessimistic analyses simultaneously. But, when doing so, it ensures that no intermediate results are propagated between them (**R5**). Given property kind $p$ that is computed optimistically and pessimistic analysis $A$ depending on $p$, *OPAL* does not forward any intermediate values of $p$ to $A$'s CF. The latter is triggered only when a value of $p$ is submitted as a final result. We say that the dependency of $A$ on $p$ is *suppressed*. There are subtle interactions between dependency suppression and cyclic and collaborative computations, which we explain next.

First, there can be no cyclic dependencies between pessimistic and optimistic analyses. The correctness of cyclic dependency resolution relies on the assumption that all intermediate approximations have been processed and no further updates to any property involved in the cycle may happen (cf. Section 3.7). This obviously is not the case when updates are suppressed.

The interaction between dependency suppression and collaboratively computed properties is more involved. Assume a collaboratively computed property $p_1$ that (transitively) depends on another collaboratively computed property $p_2$ and consider the case when one or more of the transitive dependencies between them is suppressed[1]. In this case, *OPAL* must ensure that $p_2$'s values are committed as final before $p_1$'s values can be committed as final, too. This ensures that final values have been propagated along the suppressed dependencies. To this end, *OPAL* derives a *commit order* when checking the execution constraints before executing analyses. The commit order is a partial order between collaboratively computed property kinds: $p_1$ must be finalized later than any other collaboratively computed property kind $p_2$ on which $p_1$ depends when there is suppression between them. This commit order is computed when checking the execution constraints before executing analyses.

Suppression of intermediate updates can also be used to improve scalability: Consider the relation between TAC and AI (cf. Section 2.2.1). Both are optimistic and TAC could use intermediate AI results. But these results are typically not useful. Hence, it can be beneficial to use suppression to avoid costly computation of these intermediate results and instead compute the TAC only once on the final AI result.

## 3.7 FIXED-POINT COMPUTATION

Computation is started for the entities selected by eager analyses (**R11**) (cf. Section 3.4). Whenever intermediate values for properties are submitted, the blackboard schedules activations of continuation functions, distributing updated results to analyses that depend on them. Additionally, the blackboard starts new computations by invoking the initial analysis function for properties that are requested lazily (**R12**), are triggered by some analyses reaching a certain entity (**R10**), or whenever it is guided to do so by running analyses (**R13**). This process of scheduling IAF and CF activations is performed until no further updates are generated—the blackboard has reached a *quiescent* state. At this point, however, the properties' values may not necessarily be final, as there still may be unresolved dependencies. There are three cases to be considered.

First, an analysis was scheduled for some property kind $p$, but it did not analyze some entity $e$, for which $p$ was requested, e.g., because $e$ was not reachable in the call graph. In this case, the *default value* (**R9**)

[1] On a chain of dependencies, more than one may be suppressed. Also, if $p_1$ depends on $p_3$ and $p_4$ and each of those depends on $p_2$, there is more than one path between $p_1$ and $p_2$, on which dependencies may get suppressed.

is inserted, which may trigger further computations, until quiescence is reached again.

Second, properties that cyclically depend on each other are not finalized yet. If such properties form a *closed strongly connected component*, i.e., they do not have any dependees outside of the cycle (but other properties may still depend on them), they are now finalized to their current value. By requiring analyses to report their results in a monotonous and scheduling-independent way (cf. Section 3.3), *OPAL* guarantees that the cycle resolution is deterministic and sound. Again, further computations may arise from resolving cyclic dependencies (including supplying more default values and resolving further cycles) until quiescence is reached again.

Finally, the blackboard finalizes values for collaboratively computed properties. It respects the *commit order* computed previously (cf. Section 3.6): After finalizing a set of collaboratively computed properties, computation is resumed again. Only once quiescence is reached again, the next property kinds, as given by the commit order, are finalized. This is repeated until all collaboratively computed properties have been finalized.

## 3.8 scheduling and parallelization

Blackboard systems require a control component that, upon updates of the blackboard, decides which knowledge sources to activate next. In our case, this control component determines the order in which activations of dependent analyses are executed and is called *scheduler*. The order in which dependent analyses are activated can have significant effects on scalability [192].

*OPAL* allows for the scheduler to be easily exchanged in order to select the best-performing one for any chosen set of analyses. Apart from general strategies such as first-in-first-out, more specific algorithms may use the dependency structure or the values of intermediate approximations to decide the scheduling order. This is similar to the control component of blackboard systems asking knowledge sources for an estimated information gain (cf. [48]).

Blackboard systems lend themselves well to parallelization. The individual knowledge sources, i.e., analyses in our case, are decoupled and their activations (both the initial analysis and the continuations) can be executed in parallel on multiple threads. Updates to the blackboard, on the other hand, can be synchronized on a special thread or, if that becomes a bottleneck, distributed to several threads based on the property kind and/or entity. A simple implementation may consist of several threads that use a shared data structure holding the property data and use locks or other mechanisms to synchronize accesses to this shared storage.

## 3.9 SUMMARY

Our approach fosters strong decoupling of reified lattices (choice of data structures), analyses (choice of algorithm), and the solver infrastructure (the concrete fixed-point solving implementation). This enables exchanging and optimizing these parts independently. As reified lattices are the basis for all communication between analyses, different versions of analyses can be implemented at different trade-offs. The fixed-point solver manages the execution of analyses, tracks dependencies and propagates updates, performs monotonicity checks, and computes the fixed-point solution.

# 4 ALTERNATIVE IMPLEMENTATION IN REACTIVE ASYNC

The approach presented in this thesis is general and not tied to *OPAL* alone. As we show in this chapter, implementations in other frameworks that allow for decoupled computation are possible: We present *RA2*, an alternative implementation of our approach built using the *Reactive Async* library for asynchronous computations [97]. This implementation allows for analysis-independent and semi-implicit parallelization of static analyses and for the use of scheduling strategies.

In particular, to support static analyses, we extended *Reactive Async* to allow stateful computations. We also added support for exchangeable strategies for task scheduling, which is important for scalability.

We start with a high-level overview of the programming model that the approach imposes (Section 4.1) followed by advanced concepts to ensure correctness in the presence of concurrent updates and shared mutable state (Section 4.2). Next, we introduce *RA2*'s solver, which performs the parallelization and resolves (cyclic) dependencies (Section 4.3). We present pluggable scheduling strategies, which enable analysis-specific tuning and aggregation of updates with the goal to improve on scalability (Section 4.4) and finally give an in-depth example of a simple purity analysis implemented in *RA2* (Section 4.5).

## 4.1 PROGRAMMING MODEL BASICS

To illustrate how analyses are implemented in *RA2*, assume we want to develop an analysis for determining whether methods are deterministic and free of side effects (*pure*) or not (*impure*).

The goal of any static analysis is to compute a specific property for a given entity. In the example of the purity analysis, methods are the entities and the property is the purity, which can have one of two values: pure or impure. In *RA2*, each property must form a lattice (or at least a partial order with a bottom element), a common data structure for properties calculated by static analyses. The property of an entity may depend on some other properties of related entities. In a purity analysis, if `foo` calls `bar`, the computation of `foo`'s purity requires the purity of `bar`. We say `foo` *depends on/is a depender of* `bar`, or `bar` *is a dependee of* `foo`.

Analysis results and dependencies are maintained in *cells*. When implementing a static analysis, we create one cell per pair of analyzed entity and property. Cells are shared by the different concurrent tasks. Every cell is explicitly associated with the *lattice* of the property values that are managed by the cell: a cell that was created to store purity information thus cannot be used—at some later point in time—to store data-flow information.

As described in Section 3.3, analyses in *RA2* are implemented as two functions—an *initial analysis function* to analyze the code and a *continuation function* to process updates of dependees' properties. Both functions are invoked by the reactive framework underlying *RA2*.

Given an entity, the initial analysis function computes the initial property of that entity based on the information already available, i.e., local information, such as the source code, and the current property values of dependees. The initial analysis function also queries and collects the dependencies of an entity. The dependencies may have a non-final property value (or none at all) at the time the initial function is executed. The list of dependencies is used to get notified of potential future changes of their property values. These changes may in turn lead to updates of the entity's property value. The initial analysis function returns the initial property and registers the dependencies along with a continuation function.

A continuation function of an entity *e* is invoked by *RA2* whenever dependencies of *e* (e.g., the purity information for a called method) change. Its result defines how the property value of *e* is to be updated, whereby the updated value must be a monotonic refinement according to the property's lattice. Unless the continuation function declares its result as final, it will be invoked again for further updates of dependees. Both the initial analysis and invocations of the continuation functions are *tasks* that *RA2* executes concurrently.

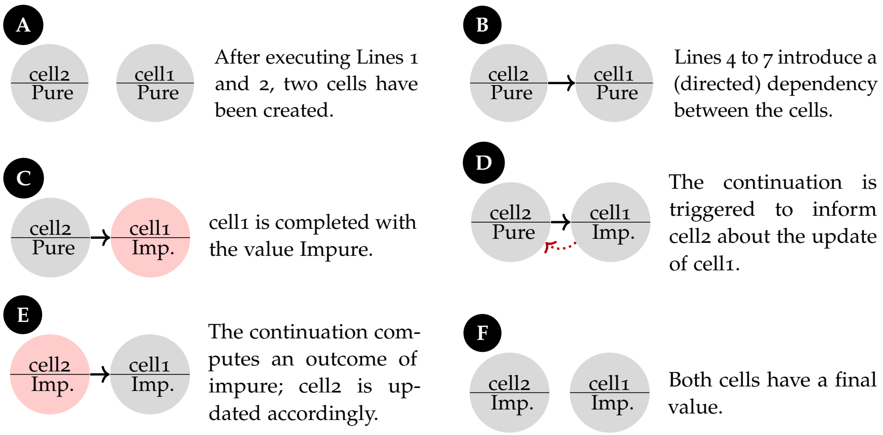
Dependencies are created by connecting two cells using the continuation function—to respond to updated cells, continuation functions are used as callbacks. To ensure that a cell is notified about the update of its dependees, its dependencies are explicitly declared and registered using the `when` method. For instance, `cell2.when(cell1)(cont)` registers `cell1` as a dependee of `cell2`: the function `cont` is called *when* the value of `cell1` changes.

The continuation function processes the changed dependee's (`cell1` in our case) new value and returns an `Outcome` object, which decides whether and how the dependent cell (`cell2` in our case) should be updated. *RA2* provides three types of `Outcome` objects: A `NextOutcome(v)` result means that the dependent cell should be updated with value `v` according to the specified updater (cf. Section 4.2). `FinalOutcome(v)` states that, additionally, this update is final. If `NoOutcome` is returned, the value of `cell2` is not changed at all.

To illustrate the use of cells and continuations, consider Figure 4.1 which graphically depicts the cells and dependency from Listing 4.1 and the propagation of an update for this dependency. Assume that the two cells, `cell1` and `cell2`, have been initialized with `Pure` (A) through the use of `NextOutcome`, i.e., their values can still change. In

```
1 val cell1: Cell[Purity] = ...
2 val cell2: Cell[Purity] = ...
3
4 cell2.when(cell1) { update =>
5   if (update.head.get.value == Impure) FinalOutcome(Impure)
6   else NoOutcome
7 }
```

Listing 4.1: Example of Dependencies and Continuations



A: After executing Lines 1 and 2, two cells have been created.

B: Lines 4 to 7 introduce a (directed) dependency between the cells.

C: cell1 is completed with the value Impure.

D: The continuation is triggered to inform cell2 about the update of cell1.

E: The continuation computes an outcome of impure; cell2 is updated accordingly.

F: Both cells have a final value.

Figure 4.1: Execution of Listing 4.1

Lines 4 to 7, a dependency (straight arrow) between `cell1` and `cell2` is introduced (B). Recall that for a purity analysis, this is necessary if the method represented by `cell2` invokes the one represented by `cell1`. Whenever `cell1` is updated (using `FinalOutcome` or `NextOutcome`), the continuation is executed and the returned `Outcome` is used to update `cell2`. Consider the case when a final update for `cell1`—with the value `Impure`—occurs (C). This will cause the continuation to be invoked (dotted arrow) with `Impure` as an argument (the new value of `cell1`) (D). Since a method that calls an impure method is also impure, the continuation returns a `FinalOutcome` with value `Impure`. Thus, `cell2` is completed with value `Impure` (E), and the dependency is removed (F). If the continuation returned a `NextOutcome`, the update of `cell2` would be an intermediate one.

## 4.2 ADVANCED CONSTRUCTS FOR CORRECTNESS

To ensure correctness and termination, cell updates must have a well-defined semantics. *RA2* executes updates concurrently, and thus without a guaranteed order. Yet, monotonic updates ensure determinism regardless of the order. Mutable state shared between the continuations of a cell could otherwise lead to non-determinism when the updates are executed concurrently due to data races.

MONOTONIC UPDATES To guarantee determinism of cell updates, they must be monotonically ordered according to the underlying lattice. *RA2*, therefore, encapsulates cell updates in so-called *updater* objects that determine how updates for the cell are processed.

Monotonicity can be automatically guaranteed by using the *join* operator (i.e., the least upper bound of a set of lattice values) of a cell's lattice to aggregate new values with the previous value of the cell. The outcome returned by each continuation is joined with the cell's previous value to compute the least upper bound, which then becomes the cell's new value. *RA2* provides the updater type `AggregationUpdater` that offers this semantics. However, there are several good reasons for supporting other updater semantics. In general, performing joins can be expensive when dealing with lattices that are not based on singleton values, e.g., sets. Complex analyses may already have to perform the *join* explicitly as part of the continuation function, thereby guaranteeing the monotonicity by design and making another implicit join during the cell update obsolete. To cover such needs, *RA2* provides the `MonotonicUpdater` as a drop-in replacement for the `AggregrationUpdater`.

The `MonotonicUpdater` does not perform a *join* operation but only checks whether the given update fulfills monotonicity. This check is defined by the lattice. For expensive checks, it can be used only during development and simplified or disabled in production when the analysis is known to guarantee monotonicity. The `MonotonicUpdater` also allows for using partial orders instead of lattices. The partial orders, however, still require a *bottom element* and must fulfill the ascending chain condition, i.e., monotonically increasing operations must converge eventually—we use this kind of updater, e.g., for our IFDS solver (see Section 11.1). As the IFDS solver's computations are required to be performed sequentially and only introduce additional flow edges, updates are guaranteed to be monotonic and no other (potentially expensive) join operation is required.

Which updater should be used for a specific cell can be defined when creating the cell by the `HandlerPool`—the interface of the analysis implementations to the underlying reactive system of *RA2*, which is presented in Section 4.3.

SEQUENTIAL UPDATES Advanced analyses require maintaining mutable state between cell updates. For example, an IFDS analysis keeps track of already computed path edges in order to extend them once updates on the analyzed method's callees become available. Mutable state can also be used to keep track of the set of dependencies explicitly. Shared mutable state affected by updates needs to be thread-safe; thus, updates that affect mutable state need to be sequential. Otherwise, continuations for several incoming updates could be ex-

```
val seqCell1 = pool.mkSequentialCell(...)
val seqCell2 = pool.mkSequentialCell(...)

seqCell1.when(dependee1)(continuation1)
seqCell1.when(dependee2)(continuation2)
seqCell2.when(dependee1)(continuation3)
```

Listing 4.2: Example of Sequential Updates

ecuted concurrently, leading to non-determinism in the presence of mutable state due to race conditions.

To enable the thread-safe use of mutable state, *RA2* provides cells with *sequential updates*, whose continuations are ensured to run sequentially. Cells with sequential updates free developers from the need to use locks or other concurrency mechanisms to protect shared mutable state. The example in Listing 4.2 illustrates cells with sequential updates. Those cells are created using the `mkSequentialCell` method of the `HandlerPool` (cf. Section 4.3). While `dependee1` and `dependee2` may be updated concurrently, *RA2* ensures that all callbacks targeting `seqCell1`—`continuation1` and `continuation2`—are invoked sequentially (though with no guarantee on the relative order). Hence, both callbacks may safely access shared state—as long as that state is only shared among callbacks targeting `seqCell1`. To still exploit the benefits of parallel computations, `continuation3` is run concurrently, even with `continuation1` being triggered by the same dependee, `dependee1`. Hence, `continuation1` and `continuation2` must not share state with `continuation3`, as the latter targets a different cell.

## 4.3 HANDLER POOL

A central unit of *RA2* is the `HandlerPool`. It creates cells and manages the execution of initial functions and the propagation of updates along cell dependency chains by executing respective continuations. It implements the parallelization which analyses can use out-of-the-box and do not need to implement on their own. For each initial analysis function and each triggered continuation, the runtime system creates a task that is eventually executed by an idle thread. The `HandlerPool` keeps track of all active threads and all tasks being registered for execution and schedules their execution.

The `HandlerPool` is *RA2*'s fixed-point solver; as such, it also resolves situations where the execution gets stuck: As it keeps track of running and pending tasks, the pool is able to detect *quiescence*. The system is quiescent when there are no unfinished submitted tasks currently queued or running. However, even when quiescence is reached, not all cells are guaranteed to contain final results. This is true in the following cases:

1. A chain of dependencies such that each cell's result depends on another cell's result where these dependencies form a cycle is called *cyclic dependency*. A simple example are two cells $A$ and $B$, where $A$ depends on $B$ and $B$ depends on $A$. Such a cyclic dependency where each cell in the cycle solely depends on cells of that cycle is called a closed strongly connected component (cSCC).

2. Cells that do not depend on any other cells are called *independent*. Independent cells that have not been completed are referred to as *independent unresolvable cells* (IUC). An IUC arises when all dependees of a cell have been completed (and the corresponding dependencies have therefore been dropped), but the cell itself was not completed yet. This happens if on the last invocation of the continuation, that continuation—without tracking dependencies explicitly—cannot recognize that there will not be any future invocations. In the example of Figure 4.1, `cell2` becomes independent after the final update of `cell1` as the dependency between `cell2` and `cell1` is removed by the system (F). In that case, `cell2` was completed. If, however, `cell1` had instead been completed with a result property value *pure*, the continuation would have returned `NoOutcome` and, therefore, `cell2` would not be completed. As the dependency would still be removed, `cell2` would become an IUC.

Once cycles and independent cells are detected, two methods, that are implemented by the analysis designer when specifying the lattice, are used to resolve them: `resolve` and `fallback`. The `resolve` method takes a list of all cells in one cSCC and returns for each cell the final value it should be resolved to. Resolving one cSCC does not trigger callbacks of cells in that cSCC but does so for their dependers outside of it. The `fallback` method works similarly but is given a list of IUCs. Different from the cells of a cSCC, the IUCs must be resolved independently of each other. Each cell is then completed with the returned associated value, which is typically the cell's current value.

## 4.4 SCHEDULING

As outlined in Chapter 1, scalability of static analyses is necessary to deal with the ever-growing amount of software. The order in which individual tasks are scheduled is one particularly important factor for runtime performance and in order to provide effective parallelization for a specific (analysis) domain. For example, scheduling strategies that prioritize values that might have a bigger impact were found to be beneficial in the *IFDS-A* framework [192]. To support different analysis domains, different scheduling strategies might be needed.

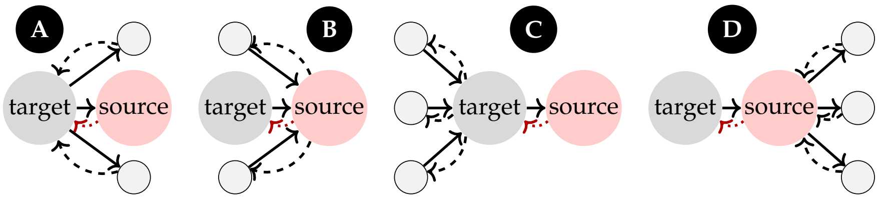


Figure 4.2: Scheduling Strategies

The HandlerPool is parametric in the scheduling strategy to enable an analysis designer to plug in a scheduling strategy to influence the order in which tasks ready for execution are picked up. Whenever a task is submitted to the pool for execution, the scheduling strategy is invoked to calculate a priority for the dependency in question. This priority is used in a priority queue that tracks all tasks that may be executed concurrently.

Dependency continuations that target sequential cells must not run concurrently. To ensure this, each sequential cell keeps track of all tasks updating it. Again, this is done via a priority queue that uses the developer-supplied scheduling strategy. Tasks can be dequeued from this queue in the order of respective priority, hence respecting the scheduling strategy.

*RA2* provides a default last-in-first-out scheduling strategy. This includes first-in-first-out work stealing, i.e., threads with an empty working queue can execute tasks from other threads' queues in order to improve processor utilization. Additionally, *RA2* includes several other general-purpose scheduling strategies out of the box. These are applicable to any kind of analysis as they only take into account how many dependees a cell has and how many cells (directly) depend on the value that is being updated via the continuation. These strategies are illustrated in Figure 4.2 (A–D). As in Figure 4.1, straight arrows represent dependencies, while discontinuous arrows represent potential update messages. The strategies determine the priority of the message from source to target (dotted red message in Figure 4.2).

(A) **TargetsWithManySourcesFirst/Last.** The higher the number of cells (in addition to source) that target depends on, the higher/lower the priority.

(B) **SourcesWithManyTargetsFirst/Last.** The higher the number of cells depending on source (in addition to target), the higher/lower the priority.

(C) **TargetsWithManyTargetsFirst/Last.** The higher the number of cells depending on target, the higher/lower the priority.

(D) **SourcesWithManySourcesFirst/Last.** The higher the number of cells that source depends on, the higher/lower the priority.

```
 1 case class LatticeValueStrategy[V](prioritizedValue: V)
 2   extends SchedulingStrategy[V] {
 3   override def calcPriority(
 4     tgt: Cell[V], src: Cell[V], v: Outcome[V]
 5   ): Int = calcPriority(tgt, v)
 6
 7   override def calcPriority(
 8     tgt: Cell[V], v: Outcome[V]
 9   ): Int = v match {
10     case FinalOutcome(prioritizedValue) => −1
11     case _ => 1
12   }
13 }
```

Listing 4.3: Scheduling Strategy for Lattice Values

The generic-purpose strategies are simple and only take the (local) shape of the dependency graph into account. Nonetheless, they can influence the analysis' behavior significantly. Some strategies (the `xFirst` ones) try to prioritize updates that may have a greater impact on the result by influencing many cells. This may lead to faster stabilization of the result. Other strategies (the `xLast` ones) delay such influential updates, providing more opportunities to aggregate them (cf. below), potentially reducing the overall number of updates required. Our evaluation (cf. Section 13.3) shows that this has a significant impact on scalability and performance.

Scheduling strategies also allow encoding and taking into consideration domain-specific knowledge about the relevance of different property values. For our purity analysis, propagating `Impure` may be more relevant to target cells than propagating `Pure`. This is because a call to an impure method is impure, thus a single impure dependee will result in the cell being completed. In contrast, a call to a pure method does not change the outcome as long as there are other dependees that might be impure. To accelerate the propagation of specific lattice values, we can use a strategy that returns a higher priority for the respective continuations.

The implementation of such a strategy prioritizing a given lattice element is shown in Listing 4.3. *RA2* provides two functions to calculate the priority for scheduling: the first one (Lines 3 to 5) is used in scheduling continuations, while the second one (Lines 7 to 12) is used for resolving cycles and IUCs where no source cell exists. Note that, based on Java's priority queues, a low value will prioritize the task to be scheduled early.

As the invocation of continuations may be delayed according to prioritization, the cell that triggered a continuation may be updated again before the continuation is actually invoked. This enables an important optimization: instead of invoking the continuation with the first updated value and later invoking the continuation again, the continuation will be invoked only once using the most recent value.

*RA2* can also aggregate updates from multiple sources that are passed to the continuation as a list of updates. Both kinds of aggregations can reduce the overhead of continuation invocations. This also has a transitive effect: fewer invocations of continuations produce fewer intermediate results that in turn trigger fewer continuations.

## 4.5 RA2 AT WORK

In the following, we demonstrate how to apply our programming model to implement a very simple purity analysis.

We show the complete analysis, including the lattice definition, the initial analysis function, its continuation, and how to bootstrap the analysis, leaving out just minor details, e.g., error handling.

```
sealed trait Purity
case object Pure extends Purity
case object Impure extends Purity

object Purity {
  implicit object PurityLattice extends Lattice[Purity] {
    override def join(v1: Purity, v2: Purity): Purity = {
      if (v1 == Impure) Impure else v2
    }

    override val bottom: Purity = Pure
  }
}
```

Listing 4.4: Simple Lattice for Purity Information

Listing 4.4 shows the specification of the lattice, including the bottom element and the *join* function. The lattice has two elements, namely `Pure` (its bottom element) and `Impure`.

```
object PurityKey extends Key[Purity] {
  def resolve(cs: Iterable[Cell[Purity]]): Iterable[(Cell[Purity], Purity)] =
    cs.map(cell => (cell, Pure))

  def fallback(cs: Iterable[Cell[Purity]]): Iterable[(Cell[Purity], Purity)] =
    cs.map(cell => (cell, Pure))
}
```

Listing 4.5: Resolving Cycles and IUCs

Listing 4.5 shows the fallback and cycle-resolution strategies as explained in Section 4.4. For the purity analysis, all impure values will be marked as final immediately. Therefore, unresolved cells must be `Pure` for both, cycles and IUCs.

```
 1 def analyze(method: Method): Outcome[Purity] = {
 2   val cell = methodToCell(method)
 3
 4   if (method.isNative || method.hasReferenceTypeParameter())
 5     return FinalOutcome(Impure)
 6
 7   val dependencies = mutable.Set.empty[Cell[Purity]]
 8   for (instruction <- method.instructions) {
 9     instruction match {
10       case gs: GETSTATIC =>
11         resolveFieldReference(gs) match {
12           case Some(field) if field.isFinal => // Constant
13           case _ => return FinalOutcome(Impure)
14         }
15       case INVOKESPECIAL | INVOKESTATIC => // Monomorphic
16         resolveNonVirtualCall(instruction) match {
17           case Some(callee) =>
18             if (callee != method) // Not self-recursive
19               dependencies.add(methodToCell(callee))
20           case _ => return FinalOutcome(Impure) // Unknown callee
21         }
22       case PUTSTATIC | ... => return FinalOutcome(Impure)
23       case _ => // All other instructions are pure
24     }
25   }
26
27   if (dependencies.isEmpty) return FinalOutcome(Pure)
28
29   cell.when(dependencies)(continuation)
30   NextOutcome(Pure)
31 }
```

Listing 4.6: Determining the Purity of a Method

The `analysis` function shown in Listing 4.6 computes the purity of a given method by checking its instructions. If there is an instruction affecting the purity, e.g., a static field write, the method is immediately considered impure (Line 22). For simplicity, we also consider native methods, methods with reference type parameters, or non-monomorphic (i.e., virtual and interface) method calls as impure. For method calls that are not self-recursive, the callee's cell is added as a dependency (Line 19). If such a callee was impure, the analyzed method itself would be impure. With a call graph available, polymorphic calls could simply be handled by adding a dependency for each possible callee. If no impure instruction was found after checking all instructions, the method is initially found to be pure (Line 30). It can then only be refined to impure by an update of a dependency (cf. Line 27). To cover the latter case, we use `when` to register the continuation function to react upon updates for these dependencies (Line 29).

```
def continuation(
  v: Iterable[(Cell[Purity], Outcome[Purity])]
): Outcome[Purity] = {
  if (v.exists(_._2 == FinalOutcome(Impure))) FinalOutcome(Impure)
  else NoOutcome
}
```

Listing 4.7: Continue with Updates for Callees

The continuation function that handles updates of dependencies is shown in Listing 4.7. As stated above, impure callees lead to an impure method. We take advantage of *RA2*'s aggregation of updates: The continuation function may receive updates for multiple dependees at once, and if a single one is impure, the cell is completed.

```
1  def main(project: Project): Unit = {
2    // 1. Initialize HandlerPool and Cells
3    val pool: HandlerPool[Purity] = new HandlerPool(
4      key = PurityKey,
5      parallelism = 10,
6      schedulingStrategy = LatticeValueStrategy(Impure)
7    )
8    var methodToCell = Map.empty[Method, Cell[Purity]]
9    for (method <- project.allMethods) {
10     val cellCreator = pool.mkCell(_ => analyze(method))
11     val cell = cellCreator(AggregationUpdater)
12     methodToCell += method -> cell
13   }
14
15   // 2. Start analyses
16   for (method <- project.allMethods) methodToCell(method).trigger()
17
18   // 3. Wait for completion
19   val future = pool.quiescentResolveCell()
20   Await.ready(future, 30.minutes)
21   pool.shutdown()
22
23   // 4. Retrieve results
24   val pureMethods = methodToCell.filter(_._2.getResult() == Pure).keys
25 }
```

Listing 4.8: Setting Up and Starting the Analysis

Listing 4.8 shows how to (1) initialize (Lines 3 to 13) the analysis and (2) start it (Line 16). The code then (3) awaits the analysis' completion (Lines 19 to 21) and (4) retrieves the results (Line 24). Parallelization is done by the `HandlerPool`—the number of threads is set explicitly in this example (Line 5). We also specify the scheduling strategy here, using the lattice-based strategy from Listing 4.3 to prioritize updates of impure methods (Line 6). `Cells` are created through the `HandlerPool` (Line 10) and their initial analysis is then triggered (Line 16). In particular, note how we specify the `AggregationUpdater` for each cell (Line 11). This explicit specification is for demonstration only, as aggregating updates is the default in *RA2*. Triggering the cells starts the concurrent computation of the initial analysis functions. The `HandlerPool` provides a future to await quiescence (Line 19). After quiescence has been reached, the cells contain the final results (Line 24).

## 4.6 SUMMARY

In this chapter, we presented *RA2*, an alternative implementation of our approach to modular, collaborative analysis. *RA2* builds on the *Reactive Async* library to provide semi-implicit parallelism for analyses without developers having to worry about concurrency issues. In order to create *RA2*, we extended *Reactive Async* with support for stateful computations that are key for static analyses. Furthermore, we used *RA2* to discuss and present exchangeable scheduling strategies that can further enhance analysis scalability by reducing the overhead in processing property updates. *RA2* shows that our approach is not limited to *OPAL* but can be adapted to any framework that allows for tasks to be scheduled independently while tracking dependencies between these tasks.

# 5 FORMALIZATION

A significant part of the complexity of developing static analyses pertains to ensuring that they are *sound*, i.e., over-approximate the runtime behavior of analyzed programs. Unfortunately, even well-established static analyses are shown to be unsound, e.g., since 2010, more than 80 soundness bugs have been found in different analyses used in the *LLVM* compiler [232]. Testing helps to find soundness bugs but cannot prove their absence, thus leaving the trustworthiness of these analyses in question.

Mathematical soundness proofs, which ensure the absence of soundness bugs, are difficult for two reasons. First, such proofs relate two program semantics, the static semantics describing an analysis that should cover all possible program executions and the dynamic semantics describing each particular actual program execution [52]. Each of these semantics is complex on its own. Especially modern programming language features such as reflection [136], concurrency [127], or native code [1] are notoriously difficult to analyze and hard to reason about in soundness proofs. Second, the style of static and dynamic semantics can differ significantly, e.g., the static semantics of *Doop* [34], which is described in Datalog, is different from dynamic semantics typically described with small-step rules [29]. This impedance mismatch makes it difficult to determine which parts of the static semantics relate to which parts of the dynamic semantics, requiring soundness proofs to reason about both semantics as a whole. These problems make soundness proofs intricate, such that only leading experts with multiple years of experience can conduct them [53, 120].

Modularization has been used before to manage different aspects of the complexity of soundness proofs. One line of work has used modularization to manage the complexity of dynamic and static semantics [34, 116, 148, 164, 194], without, however, providing a theory for modular and compositional soundness proofs. Another line of work proposes soundness theories [28, 125] that address the impedance mismatch between dynamic and static semantics by deriving both of them from the same artifact, often called *generic interpreter*. The latter describes the operational semantics of a language, without referring to details of dynamic or static semantics, and provides a common structure between dynamic and static semantics along which a soundness proof can be *composed*. Soundness proof composition alleviates the need to reason about the generic interpreter, which reduces proof complexity and effort. However, existing soundness theories are limited by generic interpreters restricting what type of analyses can be derived. In particular, generic interpreters are suitable for deriving analyses that follow the program execution order, specifically, forward whole-program abstract interpreters. But it is unclear how analyses can be derived that do not follow the program execution order, such as backward, demand-driven/lazy, or summary-based analyses.

In this chapter, we lift this restriction by developing a soundness theory based on *OPAL*'s blackboard architecture. As explained in Chapter 3, *complex analyses* are modularly composed from smaller, simpler *analysis modules* that handle individual language features, e.g., reflection, or program properties, e.g., immutability. These modules are decoupled and are not allowed to call each other directly. Instead, they communicate with each other by exchanging information via a central data store called *blackboard*, orchestrated by a fixed-point solver.

The theory introduced in this chapter uses the blackboard architecture to describe both the static and dynamic semantics in a modular way, thus addressing the impedance mismatch problem, and comes with a proof that soundness of complex analyses can be composed from independent soundness proofs of their modules. As a result, there is no need to reason about a complex analysis as a whole, which reduces the complexity of soundness proofs. We also extend the theory to make soundness proofs of existing analysis modules reusable across different analyses. In particular, we prove that the soundness proof of an analysis module remains valid, even if (a) the compound analysis processes source code elements that are unknown to the module and (b) the store contains types of analysis information that are unknown to the module. Furthermore, proofs are polymorphic in the lattices on which analysis modules operate, i.e., lattices can be changed without affecting soundness. For instance, we can reuse a pointer-analysis module, which typically depends on an allocation-site lattice, in a reflection analysis to propagate string information by extending this lattice without invalidating the pointer-analysis modules' soundness proof.

We demonstrate the applicability of our theory by implementing four different analyses and their dynamic semantics in the blackboard analysis architecture: (1) a pointer and call-graph analysis, (2) an analysis for reflection, (3) an immutability analysis, and (4) a demand-driven reaching-definitions analysis. These analyses are inspired by the case-study analyses outlined in Chapter 2. We use our theory to prove each analysis sound, where each analysis exercises a different aspect of our theory: (1) analysis modules can be proven sound independently despite mutually depending on each other, (2) soundness of modules remains valid even though the lattice changes, (3) soundness of a module remains valid even though different source code elements are analyzed, and (4) our theory applies to analyses that do not follow the program execution order.

## 5.1 formal definition

In this section, we define *OPAL*'s blackboard analysis architecture formally.

### 5.1.1 *Static Semantics*

Analyses in the blackboard architecture consist of multiple *analysis modules* communicating by information exchange via a *blackboard* [170], i.e., a central data store. That approach avoids coupling between modules as they are not allowed to call each other directly. This allows to replace analysis modules with more precise or more scalable versions without changing other modules.

**Definition 5.1** (Blackboard Analysis Architecture)**.** *We define basic notions and data types of the blackboard analysis architecture:*

1. *Entities ($\widehat{e} \in \widehat{\mathsf{Entity}}$)*[1] *are parts of programs an analysis can compute information for. For example, entities could be classes, methods, statements, fields, variables, or allocation sites of objects. Entities are ordered discretely: $\widehat{e}_1 \sqsubseteq \widehat{e}_2$ iff $\widehat{e}_1 = \widehat{e}_2$.*

2. *Kinds ($\kappa \in \mathsf{Kind}$) identify analysis information that can be computed for an entity. For example, a class entity could have kinds for its immutability or thread safety, whereas a variable entity could have kinds for its definition site or approximations of its value. Kinds are also ordered discretely.*

3. *Properties ($\widehat{p} \in \widehat{\mathsf{Property}}[\kappa]$ where $\widehat{\mathsf{Property}} : \mathsf{Kind} \rightarrow \mathsf{Lattice}$) denote analysis information which is identified by a kind $\kappa$. For instance, a class entity could have an immutability property "$\mathsf{mutable}$" or "$\mathsf{immutable}$". Properties of a kind are partially ordered and form a lattice.*

4. *A central store ($\widehat{\sigma} \in \widehat{\mathsf{Store}} \subseteq \widehat{\mathsf{Entity}} \times (\kappa : \mathsf{Kind}) \rightharpoonup \widehat{\mathsf{Property}}[\kappa]$)*[2] *contains all properties for each entity and kind. We use the notation $\widehat{\sigma}(\widehat{e}, \kappa)$ for a store lookup of an entity $\widehat{e}$ and kind $\kappa$, which results in the bottom element $\bot$ in case the property is not present. Furthermore, we use the notation $\widehat{\sigma} \sqcup [\widehat{e}, \kappa \mapsto \widehat{p}]$ for writing a new property $\widehat{p}$ to the store. If a property for the entity $\widehat{e}$ and kind $\kappa$ already exists in the store, then the old property is joined with the new property. Stores are ordered point-wise:*

$$\begin{aligned}\widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 \text{ iff } & dom(\widehat{\sigma}_1) \subseteq dom(\widehat{\sigma}_2) \wedge \\ & \forall \widehat{e}, \kappa \in dom(\widehat{\sigma}_1), \widehat{\sigma}_1(\widehat{e}, \kappa) \sqsubseteq \widehat{\sigma}_2(\widehat{e}, \kappa).\end{aligned}$$

[1] We use a hat symbol ^ to disambiguate abstract definitions from concrete definitions with the same name but without hat.

[2] The syntax $A \rightharpoonup B$ denotes a partial function from $A$ to $B$. Furthermore, $dom(f)$ is the set of all inputs for which a partial function $f$ is defined.

5. *Analysis modules* ($\widehat{f} \in \widehat{\mathsf{Module}} = \widehat{\mathsf{Entity}} \times \widehat{\mathsf{Store}} \rightarrow \widehat{\mathsf{Store}}$) *are monotone functions that compute properties of a given entity. The store allows multiple analysis modules to communicate and exchange information without having to call each other directly. Each analysis module has access to the entire store and can contribute to one or more properties.*

6. *The fixed-point algorithm* ($\mathsf{fix} : \mathcal{P}(\widehat{\mathsf{Module}}) \times \widehat{\mathsf{Store}} \rightarrow \widehat{\mathsf{Store}}$) *computes a fixed point of a compound analysis* $\widehat{F} \in \mathcal{P}(\widehat{\mathsf{Module}})$ *for an initial store* $\widehat{\sigma}_1$*.*

*The types* $\widehat{\mathsf{Entity}}$, $\mathsf{Kind}$, *and* $\widehat{\mathsf{Property}}$ *are defined by analysis developers, whereas the other types and functions are fixed by this definition.* □

```
$\widehat{\mathsf{Entity}} = \mathsf{Stmt}$
$\widehat{\mathsf{Property}}[\kappa_{\texttt{ControlFlowPred}}] = \mathcal{P}(\mathsf{Stmt})$
$\widehat{\mathsf{Property}}[\kappa_{\texttt{ReachingDefs}}] = \mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign})$
$\widehat{\mathsf{Store}} = [\mathsf{Stmt} \times \kappa_{\texttt{ControlFlowPred}} \rightharpoonup \mathcal{P}(\mathsf{Stmt})]$
  $\cup\, [\mathsf{Stmt} \times \kappa_{\texttt{ReachingDefs}} \rightharpoonup (\mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign}))]$

$\widehat{\mathsf{reachingDefs}}$(stmt: $\widehat{\mathsf{Entity}}$, $\widehat{\sigma}$: $\widehat{\mathsf{Store}}$): $\widehat{\mathsf{Store}}$ =
  predecessors = $\widehat{\sigma}$(stmt, $\kappa_{\texttt{ControlFlowPred}}$)
  $\widehat{in}$ = $\bigsqcup_{\texttt{p}\in\texttt{predecessors}} \widehat{\sigma}(\texttt{p}, \kappa_{\texttt{ReachingDefs}})$
  $\widehat{out}$ = stmt **match**
    **case** Assign(x,_,_) => $\widehat{in}[\texttt{x} \mapsto \{\texttt{stmt}\}]$
    **case** _ => $\widehat{in}$
  $\widehat{\sigma} \sqcup [\texttt{stmt}, \kappa_{\texttt{ReachingDefs}} \mapsto \widehat{out}]$
```

Figure 5.1: Analysis Module of a Reaching-Definitions Analysis

Figure 5.1 and Figure 5.2 show the analysis and the dynamic module of the reaching-definitions analysis. While both modules use the same Kind, i.e., $\mathsf{Kind} = \kappa_{\texttt{ReachingDefs}} \mid \kappa_{\texttt{ControlFlowPred}}$, Figure 5.1 shows the type of entities, properties, and code of the analysis module. The reaching-definitions analysis computes for every statement of the program which variable definitions may reach it. Therefore, the entities of the analysis are statements and its property is a mapping from variables to assignments that may have defined the variable. The analysis depends on information about potential control-flow predecessors of a statement. Both properties can be accessed via the kinds $\kappa_{\texttt{ReachingDefs}}$ and $\kappa_{\texttt{ControlFlowPred}}$ respectively.

We illustrate Definition 5.1 by the example of a text-book reaching-definitions analysis [169] for an imperative language with labeled assignments and expressions:

$$\mathsf{Stmt} ::= \mathsf{Assign}(\mathsf{Var}, \mathsf{Expr}, \mathsf{Label}) \mid \ldots$$

The code of the analysis module is described with Scala-like pseudo code. Module $\widehat{\mathsf{reachingDefs}}$ joins the reaching definitions of all control-flow predecessors and then updates them on variable assignments.

```
Entity = Stmt
Property[$\kappa_{\texttt{ControlFlowPred}}$] = Stmt
Property[$\kappa_{\texttt{ReachingDefs}}$] = Var $\rightharpoonup$ Assign
Store = [Stmt $\times$ $\kappa_{\texttt{ControlFlowPred}}$ $\rightharpoonup$ Stmt]
  $\cup$ [Stmt $\times$ $\kappa_{\texttt{ReachingDefs}}$ $\rightharpoonup$ (Var $\rightharpoonup$ Assign)]

reachingDefs(stmt: Entity, $\sigma$: Store): Store =
  predecessor = $\sigma$(stmt, $\kappa_{\texttt{ControlFlowPred}}$)
  in = $\sigma$(predecessor, $\kappa_{\texttt{ReachingDefs}}$)
  out = stmt match
    case Assign(x,_,_) => in[x $\mapsto$ stmt]
    case _ => in
  $\sigma$[stmt, $\kappa_{\texttt{ReachingDefs}}$ $\mapsto$ out]
```

Figure 5.2: Dynamic Module of a Reaching-Definitions Analysis

Note that module $\widehat{\text{reachingDefs}}$ neither computes the control-flow predecessors directly nor does it call another module that computes this information. Instead, it retrieves this information from the store $\widehat{\sigma}$. This decoupling avoids dependencies between analysis modules and enables compositional soundness proofs.

### 5.1.2 *Dynamic Semantics*

Analyses in the blackboard analysis architecture are proven sound with respect to a dynamic semantics in the same style, which we define formally in this subsection:

**Definition 5.2.** *We define the dynamic semantics used to prove soundness of analyses in the blackboard analysis architecture:*

1. *The dynamic semantics depends on concrete versions of* entities *(*$e \in \mathsf{Entity}$*),* properties *(*$p \in \mathsf{Property}[\kappa]$ *where* $\mathsf{Property} : \mathsf{Kind} \to \mathsf{Set}$*) and* stores *(*$\sigma \in \mathsf{Store} \subseteq \mathsf{Entity} \times (\kappa : \mathsf{Kind}) \to \mathsf{Property}[\kappa]$*). The kinds are the same as for static modules.*

2. *Dynamic modules (*$f \in \mathsf{Module} = \mathsf{Entity} \times \mathsf{Store} \rightharpoonup \mathsf{Store}$*) are partial functions that may only be defined for a subset of entities. Furthermore, the partial function is undefined in case it tries to look up an element from the store which is not present.*

3. *Static analyses are proven sound with respect to a dynamic reachability semantics. The reachability semantics (*$\mathsf{reachable} : \mathcal{P}(\mathsf{Module}) \times \mathsf{Store} \to \mathcal{P}(\mathsf{Store})$*) returns the set of all reachable stores by iteratively applying a set of dynamic modules. More specifically, the set* $\mathsf{reachable}(F, \sigma_1)$ *contains store* $\sigma_1$ *and for all* $f \in F$*, reachable stores* $\sigma$*, and for entities* $e \in dom(\sigma)$*, the set contains* $f(e, \sigma)$*, if it is defined.* □

We illustrate these definitions again at the example of the reaching-definitions analysis which we introduced in the previous subsection. Figure 5.2 shows the dynamic semantics of the analysis. The dynamic module $\mathsf{reachingDefs}$ is analogous to its static counterpart $\widehat{\mathsf{reachingDefs}}$ but computes the *most recent* definition of a variable instead of all possible definitions. The dynamic module depends on the control-flow predecessor, which is the most recently executed statement. While in this example, the analysis module and corresponding dynamic module are very similar, they do not necessarily have to be (e.g., module $\widehat{\mathsf{method}}$ in Section 5.4.1.1). However, the less similar they are, the more work is needed in a soundness proof to bridge the differences.

Similar to the static semantics, the blackboard analysis architecture also modularizes the dynamic semantics, which is crucial for enabling compositional and reusable soundness proofs. In particular, each analysis module is proven sound with respect to exactly one dynamic module, which limits the proof scope and guarantees proof independence. Furthermore, for analyses that approximate non-standard dynamic semantics, the standard dynamic semantics can be extended in a modular way with additional modules (e.g., Section 5.4.1.3).

To summarize, in this section we formally defined the blackboard analysis architecture, which allows implementing static analyses in a modular way. In addition, we defined a dynamic semantics in the same style against which analyses modules are proven sound.

## 5.2 COMPOSITIONAL SOUNDNESS PROOFS

Next, we develop a theory of compositional soundness proofs for analyses in the blackboard style. In particular, soundness of a compound analysis follows directly from soundness of the individual analysis modules. We start the section by defining soundness of analysis modules and then work up to soundness of whole analyses. The definitions of soundness are standard and build upon the theory of *abstract interpretation* [52]:

**Definition 5.3** (Soundness of Analysis Modules)**.** *An analysis module $\widehat{f} \in \widehat{\mathsf{Module}}$ is sound if it over-approximates its dynamic counterpart $f \in \mathsf{Module}$:*

$$\begin{aligned}&\mathsf{sound}(f,\widehat{f}) \text{ iff } \forall \widehat{e} \in \widehat{\mathsf{Entity}}, \widehat{\sigma} \in \widehat{\mathsf{Store}}, e \in \gamma_{\mathsf{Entity}}(\widehat{e}), \sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma}).\\ &\qquad f(e,\sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e},\widehat{\sigma}))\end{aligned} \qquad \square$$

The expression $x \in \gamma(\widehat{y})$ reads as "element $\widehat{y}$ soundly over-approximates the concrete element $x$." Function $\gamma : \widehat{L} \to \mathcal{P}(L)$ is a monotone function from an abstract domain $\widehat{L}$ to a powerset of a concrete domain $L$ and is called *concretization function*. We do not require that an *abstraction function* $\alpha : \mathcal{P}(L) \to \widehat{L}$ in the opposite direction exists nor that $\gamma$ and $\alpha$ form a Galois connection, both of which are not necessary for soundness proofs. We illustrate this soundness definition later with Lemma 5.7.

The soundness definition above requires that analysis developers define concretizations for entities ($\gamma_{\mathsf{Entity}} : \widehat{\mathsf{Entity}} \to \mathcal{P}(\mathsf{Entity})$) and properties ($\gamma_{\mathsf{Property}} : \widehat{\mathsf{Property}}[\kappa] \to \mathcal{P}(\mathsf{Property}[\kappa])$). Often the abstract and concrete entities are of the same type ($\widehat{\mathsf{Entity}} = \mathsf{Entity}$). In this case, the concretization functions map to singleton sets ($\gamma_{\mathsf{Entity}}(e) = \{e\}$).

Based on concretization functions for entities, kinds, and properties, we define a point-wise concretization for stores:

**Definition 5.4** (Concretization for Stores)**.**

$$\begin{aligned}
&\gamma_{\mathsf{Store}} : \widehat{\mathsf{Store}} \to \mathcal{P}(\mathsf{Store})\\
&\gamma_{\mathsf{Store}}(\widehat{\sigma}) = \{\sigma \mid dom(\sigma) = \gamma_{\mathsf{Entity}\times\mathsf{Kind}}(dom(\widehat{\sigma})) \land\\
&\qquad\qquad\qquad \forall(\widehat{e},\kappa) \in dom(\widehat{\sigma}), e \in \gamma_{\mathsf{Entity}}(\widehat{e}).\\
&\qquad\qquad\qquad\quad \sigma(\widehat{e},\kappa) \in \gamma_{\mathsf{Property}}(\widehat{\sigma}(\widehat{e},\kappa))\}
\end{aligned} \qquad \square$$

We illustrate this concretization function at the example of a store that contains control-flow information for statements $s_1$, $s_2$, and $s_3$:

$$\begin{aligned}
&\gamma_{\mathsf{Store}}([s_1, \kappa_{\mathtt{ControlFlowPred}} \mapsto \{s_2, s_3\}]) =\\
&\qquad\qquad \{[s_1, \kappa_{\mathtt{ControlFlowPred}} \mapsto s_2], [s_1, \kappa_{\mathtt{ControlFlowPred}} \mapsto s_3]\}
\end{aligned}$$

The concretization function returns a set of concrete stores, where each mapping $e, \kappa \mapsto p$ in a concrete store is a concretization of a mapping $\widehat{e}, \kappa \mapsto \widehat{p}$ in the abstract store, i.e., $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$ and $p \in \gamma_{\mathsf{Property}}(\widehat{p})$.

In the remainder of this section, we define soundness of compound analyses. Afterward, we prove that soundness of a compound analysis follows from soundness of each module.

**Definition 5.5** (Soundness of a Compound Analysis)**.** *Let* $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ *be a set of analysis modules paired with corresponding dynamic modules. A compound analysis is sound if the fixed point of all of its analysis modules over-approximates the reachability semantics of the corresponding dynamic modules:*

$$\begin{aligned}
&\mathsf{sound}(\Phi) \;\; \textit{iff} \;\; \forall\widehat{\sigma} \in \widehat{\mathsf{Store}}.\; \mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma})) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}))\\
&\qquad \text{where } F = \{f \mid (f, \_) \in \Phi\} \text{ and } \widehat{F} = \{\widehat{f} \mid (\_, \widehat{f}) \in \Phi\}.
\end{aligned} \qquad \square$$

The reachability semantics $\mathsf{reachable}(F, S)$ is defined inductively as the set of initial stores $S$ and the set of stores $f(e, \sigma)$ for all $f \in F, \sigma \in \mathsf{reachable}(F, S)$, and $(e, \_) \in dom(\sigma)$.

**Theorem 5.6** (Soundness Composition)**.** *Let* $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ *be a set of analysis modules paired with corresponding dynamic modules. Soundness of a compound analysis follows from soundness of all of its analysis modules:*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ for all } (f, \widehat{f}) \in \Phi \text{ then } \mathsf{sound}(\Phi). \qquad \square$$

*Proof.* By fixed-point induction on the set of reachable stores, we show that $\mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma}_1)) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$. Let $\widehat{\sigma}_n = \mathsf{fix}(\widehat{F}, \widehat{\sigma}_1)$ and $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma}_n)$. By the assumption $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma}_n)$, we get $dom(\sigma) = \gamma_{\mathsf{Entity}\times\mathsf{Kind}}(dom(\widehat{\sigma}_n))$ and $\sigma(e,\sigma) \in \gamma_{\mathsf{Property}}(\widehat{\sigma}_n(\widehat{e},\kappa))$ for all $\forall(\widehat{e},\kappa) \in dom(\widehat{\sigma}_n)$ and $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. Furthermore, since $\widehat{\sigma}_n$ is a fixed point, we get $\widehat{f}(\widehat{e},\widehat{\sigma}_n) \sqsubseteq \widehat{\sigma}_n$ for all $\widehat{f} \in \widehat{F}$ and $\widehat{e} \in dom(\widehat{\sigma}_n)$. From $\mathsf{sound}(f,\widehat{f})$ we conclude $f(e,\sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e},\widehat{\sigma}_n)) = \gamma_{\mathsf{Store}}(\widehat{\sigma}_n) = \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F},\widehat{\sigma}_1))$ for all $(f,\widehat{f}) \in \Phi$, $(e,\_) \in dom(\sigma)$, and $(\widehat{e},\_) \in dom(\widehat{\sigma}_n)$ with $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. □

We illustrate this soundness-composition theorem at the example of the reaching-definitions analysis from Section 5.1.1. More specifically, the analysis consists of modules $\widehat{\mathsf{reachingDefs}}$ (defined in Figure 5.1) and $\widehat{\mathsf{controlFlowPred}}$, the latter of which computes control-flow predecessors (not shown). Theorem 5.6 guarantees that

$$\begin{aligned}&\mathsf{sound}(\{(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}}),\\ &\qquad(\mathsf{controlFlowPred}, \widehat{\mathsf{controlFlowPred}})\})\end{aligned}$$

follows from

$$\mathsf{sound}(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}})$$

and

$$\mathsf{sound}(\mathsf{controlFlowPred}, \widehat{\mathsf{controlFlowPred}}).$$

Next, we prove module $\widehat{\mathsf{reachingDefs}}$ sound to demonstrate that its proof is independent of module $\widehat{\mathsf{controlFlowPred}}$. The independence of soundness proofs of analysis modules is important because it allows us to compose them into a soundness proof for the whole analysis.

**Lemma 5.7.** *Module $\widehat{reachingDefs}$ is sound with respect to its dynamic counterpart $reachingDefs$.*

*Proof.* To show: $\forall \widehat{\sigma} \in \widehat{\mathsf{Store}}, \widehat{e} \in \widehat{\mathsf{Entity}}, e \in \gamma_{\mathsf{Entity}}(\widehat{e}), \sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$, $reachingDefs(e,\sigma) \in \gamma_{\mathsf{Store}}(\widehat{reachingDefs}(\widehat{e},\widehat{\sigma}))$. Given an abstract entitity $\widehat{e}$, abstract store $\widehat{\sigma}$, and a concretization $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$, it follows that the set of control-flow predecessors is sound $\sigma(\widehat{e}, \kappa_{\texttt{ControlFlowPred}}) \in \gamma_{\mathsf{Property}}(\widehat{\sigma}(\widehat{e}, \kappa_{\texttt{ControlFlowPred}}))$ by definition of $\gamma_{\mathsf{Store}}$ and $\gamma_{\mathsf{Entity}}(\widehat{e}) = \{e\}$.

Next, we prove that the set of reachable definitions of all predecessors $\widehat{in}$ is sound:

$$\begin{aligned}\mathtt{in} &= \sigma(\sigma(\widehat{e}, \kappa_{\texttt{ControlFlowPred}}), \kappa_{\texttt{ReachingDefs}}) && \text{(by definition)}\\ &\in \gamma_{\mathsf{Property}}\Big(\bigsqcup_{p \in \widehat{\sigma}(\widehat{e}, \kappa_{\texttt{ControlFlowPred}})} \widehat{\sigma}(p, \kappa_{\texttt{ReachingDefs}})\Big) && \text{(by soundness of cflow. pred. and def. of } \gamma_{\mathsf{Store}}\text{)}\\ &= \gamma_{\mathsf{Property}}(\widehat{in}) && \text{(by definition)}\end{aligned}$$

We now prove that the set of reachable definitions for the current statement $\widehat{out}$ is sound by case distinction on $\widehat{e}$:

- If $\widehat{e} = \mathsf{Assign}(x, \_, l)$ then $\mathtt{out} = \mathtt{in}[x \mapsto l] \in \gamma_{\mathsf{Property}}(\widehat{\mathtt{in}}[x \mapsto \{l\}]) = \gamma_{\mathsf{Property}}(\widehat{out})$,
- Otherwise $\mathtt{out} = \mathtt{in} \in \gamma_{\mathsf{Property}}(\widehat{\mathtt{in}}) = \gamma_{\mathsf{Property}}(\widehat{out})$.

Finally, we conclude

$$\begin{aligned}\mathsf{reachingDefs}(\widehat{e}, \sigma) &= \sigma[\widehat{e}, \kappa_{\mathtt{ReachingDefs}} \mapsto \mathtt{out}] \\ &\in \gamma_{\mathsf{Property}}(\widehat{\sigma} \sqcup [\widehat{e}, \kappa_{\mathtt{ReachingDefs}} \mapsto \widehat{out}]) \\ &= \gamma_{\mathsf{Property}}(\widehat{\mathsf{reachingDefs}}(\widehat{e}, \widehat{\sigma})). \qquad \square\end{aligned}$$

Note that Lemma 5.7 does not depend on the soundness proof of $\widehat{\mathsf{controlFlowPred}}$. This is possible because neither module $\widehat{\mathsf{reachingDefs}}$ nor $\mathsf{reachingDefs}$ call the control-flow modules directly. Instead, both the static and dynamic module read the control-flow information from the store, which is guaranteed to be a sound over-approximation initially (assumption $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$). Furthermore, only properties that the reaching-definitions modules themselves wrote to the store need to be sound over-approximations. Properties that other modules wrote to the store are not subject of the soundness proof of the reaching-definitions modules.

To summarize, in this section we developed a theory of compositional soundness proofs for analyses described in the blackboard architectural style. Each analysis module can be proven sound independently of other modules. Furthermore, soundness of a whole analysis follows directly from soundness of each module. In particular, no reasoning about the analysis as a whole is required.

## 5.3 REUSABLE SOUNDNESS PROOFS

As of now, analysis modules refer to a specific type of entities, kinds, properties, and stores. However, adding new modules to an analysis may require extending these types. This invalidates the soundness proofs of existing modules and they need to be re-established. In this section, we extend our theory to make analysis modules and their soundness proofs reusable.

### 5.3.1 *Extending the Types of Entities and Kinds*

We start by explaining how entities and kinds can be extended without invalidating existing soundness proofs.

For example, if we were to add a taint-analysis module to an existing analysis over types $\widehat{\mathsf{Entity}}$, $\mathsf{Kind}$, and $\widehat{\mathsf{Store}}$, we needed to extend these types to hold the new analysis information:

$$\begin{aligned}&\widehat{\mathsf{Entity}}' = \widehat{\mathsf{Entity}} \mid \mathsf{Var} \\ &\mathsf{Kind}' = \mathsf{Kind} \mid \kappa_{\mathtt{Taint}} \\ &\widehat{\mathsf{Store}}' = \widehat{\mathsf{Store}} \cup [\mathsf{Var} \times \kappa_{\mathtt{Taint}} \rightharpoonup \widehat{\mathsf{Taint}}]\end{aligned}$$

However, this invalidates the proofs of existing modules that depend on the subsets $\widehat{\mathsf{Entity}}$ and $\mathsf{Kind}$. To solve this problem, we first parameterize the type of modules to make explicit what types of entities and kinds they depend on:

**Definition 5.8** (Parameterized Modules (Preliminary))**.** *We define a type of module parameterized by the types of entities $E$, kinds $K$, and store $S$:*

$$f \in \mathsf{Module}[E, K] = \forall S : \mathsf{Store}[E, K].\ E \times S \rightarrow S \qquad \square$$

Interface $\mathsf{Store}[E, K]$ defines read and write operations for an abstract store type $S$, that restricts access to entities of type $E$ and kinds of type $K$. The store interface allows us to call parameterized modules with stores containing supersets of the type of entities and kinds.

For these parameterized modules, we define a sound *lifting* to supersets of entities and kinds:

```
lift : $\forall E', K', E \subseteq E', K \subseteq K', \mathsf{Module}[E, K] \rightarrow \mathsf{Module}[E', K']$
lift$(f)(e', \sigma)$ = $e'$ **match**
  **case** $e : E$ => $f(e, \sigma)$
  **case** _ => $\sigma$
```

The lifting calls module $f$ on all entities of type $E$ on which $f$ is defined and simply ignores all other entities, returning the store unchanged. For example, the lifted reaching-definitions module $\mathsf{lift}[\mathsf{Stmt} \mid \mathsf{Var}, \kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}} \mid \kappa_{\mathtt{Taint}}](\widehat{\mathsf{reachingDefs}})$ operates on entities $\mathsf{Stmt}$ and kinds $\kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}}$ but ignores entities $\mathsf{Var}$ and kinds $\kappa_{\mathtt{Taint}}$.

The lifting preserves soundness of the lifted modules for disjoint extensions of entities:

**Definition 5.9** (Disjoint Extension)**.** *Entities $\widehat{E}' \supseteq \widehat{E}$ and $E' \supseteq E$ are a disjoint extension iff $\gamma_{\mathsf{Entity}}(\widehat{E}) \subseteq E$ and $\gamma_{\mathsf{Entity}}(\widehat{E}' \setminus \widehat{E}) \subseteq E' \setminus E$.*

In other words, the concretization function $\gamma_{\mathsf{Entity}}$ does not mix up entities in $\widehat{E}$ and $\widehat{E}' \setminus \widehat{E}$.

**Lemma 5.10** (Lifting preserves Soundness)**.** *Let $\widehat{f} \in \mathsf{Module}[\widehat{E}, K]$ and $f \in \mathsf{Module}[E, K]$ be a parameterized analysis module and dynamic module, $\widehat{E}' \supseteq \widehat{E}$ and $E' \supseteq E$ be a disjoint extension of entities, and $K' \supseteq K$ a superset of kinds.*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ then } \mathsf{sound}(\mathsf{lift}[E', K'](f), \mathsf{lift}[\widehat{E}', K'](\widehat{f})). \qquad \square$$

*Proof.* Let $\widehat{f} : \mathsf{Module}[\widehat{E}, K]$ and $f : \mathsf{Module}[E, K]$ be an analysis and dynamic module. Let $\widehat{e} : \widehat{E}'$ and $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$ be an entity and $\widehat{\sigma} : \mathsf{Store}[\widehat{E}', K']$ and $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$ be an abstract and concrete store.

- In case $\widehat{e} \in \widehat{E}$, then also $e \in E$. Hence, $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(\widehat{e}, \widehat{\sigma})$. Soundness follows by $\mathsf{sound}(f, \widehat{f})$.

- In case $\widehat{e} \in \widehat{E}' \setminus \widehat{E}$, then also $e \in E' \setminus E$ for all $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. Hence $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(\widehat{e}, \widehat{\sigma})$. □

As a result of this lemma, we can prove the soundness of analysis modules once for specific types of entities and kinds. Later, we can reuse the modules in a compound analysis with extended entities and kinds without having to prove soundness again.

### 5.3.2 *Changing the Type of Properties*

Next, we extend our theory to allow changing the type of properties without invalidating the soundness proofs of existing modules that use them.

For example, consider a pointer-analysis module that propagates object allocation information $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}}$. If we were to add a string-analysis module to the analysis, we could reuse the same pointer-analysis module to propagate string information $\widehat{\mathsf{Str}}$ by changing its lattice to $\widehat{\mathsf{Property}}'[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}} \times \widehat{\mathsf{Str}}$. However, this invalidates the soundness proof of the pointer-analysis module as it depends on type $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}]$.

To solve this problem, we generalize the type of analysis modules again to be polymorphic over the type $\widehat{\mathsf{Property}}$:

**Definition 5.11** (Parameterized Modules (Final))**.** *We define a type of module that is parameterized by the type of entities $E$, kinds $K$, properties $P$, and stores $S$:*

$$f \in \mathsf{Module}[E, K, I] = \forall P : I, S : \mathsf{Store}[E, K, P], E \times S \to S \qquad \square$$

Interface $\mathsf{Store}[E, K, P]$ restricts access to entities of type $E$ and type $K$ and contains properties of type $P$, while interface $I$ defines operations on properties $P$. For example, a pointer-analysis module may depend on the Scala-like interface Objects in Listing 5.1. Interface Objects depends on a type variable `Value`, which refers to possible values of variables. Function newObj creates a new object of a certain class and context. Function forObj iterates over all such objects applying continuation `f`. Continuation `f` takes a class name, context, and store and returns a modified store. Interface Objects can be instantiated to support different value abstractions. For example, instance $\widehat{\mathsf{AllocationSite}}$ implements the interface with an allocation-site abstraction $\widehat{\mathsf{Obj}} = \widehat{\mathsf{Obj}}(\mathcal{P}(\mathsf{Class} \times \mathsf{Context}))$ which abstracts object allocations by their class names and a call string to their allocation site. Instance $\widehat{\mathsf{AllocationSiteAndStrings}}$ implements a reduced product [49] of objects $\widehat{\mathsf{Obj}}$ and strings $\widehat{\mathsf{Str}} = \mathtt{Constant[String]}$, which abstracts the value of strings with a constant abstraction. This allows us to reuse the same pointer-analysis module to propagate string information. Finally, dynamic modules require a concrete instance of the same interface Objects. For example, instance ConcreteValues implements the interface with all concrete value types. The concrete value types are singleton versions of their static counterparts.

```
trait Objects[Value]:
  newObj(class: Class, ctx: Context): Value
  forObj[$S$](Value, $S$)(f: (class: Class, ctx: Context, $\sigma$: $S$) => $S$): $S$

object $\widehat{\text{AllocationSite}}$ extends Objects[$\widehat{\text{Obj}}$]:
  newObj(class, ctx) = {(class, ctx)}
  forObj[$S$]($\widehat{\text{Obj}}$(objs), $\widehat{\sigma}$)(f) = $\bigsqcup_{\texttt{(class,ctx)}\in\texttt{objs}}$ f(class, ctx, $\widehat{\sigma}$)

object $\widehat{\text{AllocationSiteAndStrings}}$ extends Objects[$\widehat{\text{Obj}} \times \widehat{\text{Str}}$]:
  newObj(class, ctx) = ({(class, ctx)}, $\bot$)
  forObj[$S$](value, $\widehat{\sigma}$)(f) = value match
    case (objs,_) => $\bigsqcup_{\texttt{(class,ctx)}\in\texttt{objs}}$ f(class, ctx, $\widehat{\sigma}$)

object ConcreteValues extends Objects[Obj | Str | ...]:
  newObj(class, ctx) = Obj(class, ctx)
  forObj[$S$](value, $\sigma$)(f) = value match
    case Obj(class,ctx) => f(class, ctx, $\sigma$)
    case Str(_, obj) => forObj(obj, $\widehat{\sigma}$)(f)
    ...
```

Listing 5.1: Interface and Instances for Different Object Abstractions

### 5.3.3 *Soundness of Parameterized Analysis Modules*

In this subsection, we define soundness of parameterized analysis modules and prove a generalized soundness-composition theorem.

**Definition 5.12** (Soundness of Parameterized Analysis Modules)**.** *A parameterized analysis module $\widehat{f} : \widehat{\mathsf{Module}}[\widehat{E}, K, I]$ is sound w.r.t. a parameterized dynamic module $f : \mathsf{Module}[E, K, I]$ iff all their instances are sound:*

$$\begin{aligned}\mathsf{sound}(f, \widehat{f}) \text{ iff } \forall P : I, \widehat{P} : I, S : \mathsf{Store}[E, K, P], \widehat{S} : \mathsf{Store}[\widehat{E}, K, \widehat{P}].\\ \mathsf{sound}(P, \widehat{P}) \implies \mathsf{sound}(f[P, S], \widehat{f}[\widehat{P}, \widehat{S}]). \qquad \square\end{aligned}$$

Parameterized analysis modules are proven sound for all sound instances of property interface $I$. A static instance $\widehat{P} : I$ is sound w.r.t. to a dynamic instance $P : I$, if all of its operations are sound. Soundness for dynamic and static instances of interface Objects in Listing 5.1 is defined as follows:

$$\begin{aligned}&\mathsf{sound}(\mathsf{newObj}, \widehat{\mathsf{newObj}}) \ \text{ iff}\\ &\qquad \forall c, \widehat{h}, h \in \gamma(\widehat{h}), \mathsf{newObj}(c, h) \in \gamma_{\mathsf{Obj}}(\widehat{\mathsf{newObj}}(c, \widehat{h}))\\ &\mathsf{sound}(\mathsf{forObj}, \widehat{\mathsf{forObj}}) \ \text{ iff}\\ &\qquad \forall f, \widehat{f}, \mathsf{sound}(f, \widehat{f}) \implies \mathsf{sound}(\mathsf{forObj}(f), \widehat{\mathsf{forObj}}(\widehat{f}))\end{aligned}$$

Soundness of first-order operations like $\widehat{\mathsf{newObj}}$ is similar to that of analysis modules (Definition 5.3). Soundness of higher-order operations like $\widehat{\mathsf{forObj}}$ is proven with respect to all sound functions $\widehat{f}$.

Finally, we generalize the soundness-composition Theorem 5.6 to parameterized analysis modules. In particular, an analysis composed of parameterized analysis modules is sound if all of its modules are sound and the instance of its property interface is sound.

**Theorem 5.13** (Soundness Composition for Parameterized Analysis Modules)**.** *Let $\Phi$ be parameterized analysis modules paired with corresponding dynamic modules over families of entities $\widehat{E}' = \bigcup_i \widehat{E}_i$, $E' = \bigcup_i E_i$, kinds $K' = \bigcup_i K_i$, properties $\widehat{P}$, $P$.*

$$\begin{aligned}&\text{If } \mathsf{sound}(f, \widehat{f}) \text{ for all } (f, \widehat{f}) \in \Phi \text{ and } \mathsf{sound}(P, \widehat{P}) \text{ then } \mathsf{sound}(\Phi'),\\ &\quad \text{where } \Phi' = \{(\mathsf{lift}[E', K'](f)[P, S], \mathsf{lift}[\widehat{E}', K'](\widehat{f})[\widehat{P}, \widehat{S}])\\ &\qquad\qquad\qquad \mid (f, \widehat{f}) \in \Phi\}.\end{aligned}$$ □

*Proof.* We instantiate the polymorphic modules $f, \widehat{f}$ with the compound types to obtain $\mathsf{sound}[E', K'](\mathsf{lift}(f), \mathsf{lift}[E', K'](\widehat{f}))$. Then the soundness-composition Theorem 5.6 for monomorphic modules applies. □

To summarize, in this section we explained how the type of entities, kinds, and properties can be changed without invalidating the soundness proofs of existing modules. To this end, we generalized the type of modules to be parametric over the type of entities, kinds, and properties. The parameterized modules access properties via an interface. The instances of this interface are specific to certain types of properties and require a soundness proof.

## 5.4 APPLICABILITY OF THE THEORY

In this section, we demonstrate the applicability of our theory by first developing four analyses in the blackboard architecture and then proving them sound compositionally.

### 5.4.1 *Case Studies*

We developed four different analyses in the blackboard architecture (Section 5.1) together with their dynamic semantics (Section 5.1.2). We proved each analysis sound (Part V) and a discussion of those proofs is presented later in Section 5.4.2. Each analysis exercises a specific part of our soundness theory:

- A pointer analysis, which mutually depends on a call-graph analysis (exercises the part of our theory presented in Section 5.2).
- A reflection analysis, which reuses the pointer analysis to propagate string information (exercises the part of our theory presented in Section 5.3.2).

- A field and object immutability analysis, which depends on all of the analyses above (exercises the part of our theory presented in Section 5.3.1).
- A demand-driven reaching-definitions analysis, which demonstrates that our theory applies to this type of analyses.

Our choice of analyses is inspired by the case studies outlined in Chapter 2 and detailed in Part II but simplified to focus on the core aspects of the proofs. The analyses operate on a simpler object-oriented language with the following abstract syntax:

$$\begin{aligned}
&\mathsf{Class} = \mathsf{Class}(\mathsf{ClassName}, \mathsf{ClassName}, \mathsf{Field}^*, \mathsf{Method}^*)\\
&\mathsf{Method} = \mathsf{SourceMethod}(\mathsf{MethodName}, \mathsf{Var}^*, \mathsf{Stmt}^*)\\
&\qquad\qquad \mid \mathsf{NativeMethod}(\mathsf{MethodName})\\
&\mathsf{Stmt} = \mathsf{Assign}(\mathsf{Ref}, \mathsf{Expr}) \mid \mathsf{Return}(\mathsf{Method}, \mathsf{Expr}) \mid \mathsf{If}(\mathsf{Expr}, \mathsf{Stmt}^*, \mathsf{Stmt}^*)\\
&\qquad\quad \mid \mathsf{While}(\mathsf{Expr}, \mathsf{Stmt}^*)\\
&\mathsf{Expr} = \mathsf{Ref} \mid \mathsf{New}(\mathsf{ClassName}, (\mathsf{Field} \times \mathsf{Expr})^*)\\
&\qquad\quad \mid \mathsf{Call}(\mathsf{Expr}, \mathsf{MethodName}, \mathsf{Expr}^*) \mid \mathsf{StringLit}(\mathsf{String})\\
&\qquad\quad \mid \mathsf{Concat}(\mathsf{Expr}, \mathsf{Expr}) \mid \mathsf{BoolLit}(\mathsf{Bool}) \mid \mathsf{Equals}(\mathsf{Expr}, \mathsf{Expr})\\
&\mathsf{Ref} = \mathsf{VarRef}(\mathsf{Var}) \mid \mathsf{FieldRef}(\mathsf{Ref}, \mathsf{Field})
\end{aligned}$$

The language features inheritance, mutable memory, class fields, virtual method calls, and Java-like reflection [150]. Reflection is modeled as virtual calls to native methods. Furthermore, we deliberately added language features such as control-flow constructs and boolean operations. While these features are ignored by the analyses, they do need to be modeled by dynamic semantics which complicates the soundness proof of the analyses.

We implemented and tested each analysis in Scala to ensure they are executable. Furthermore, we implemented and tested the corresponding dynamic semantics to ensure they are sensible[3].

In the following, we discuss the implementation of each analysis in more detail.

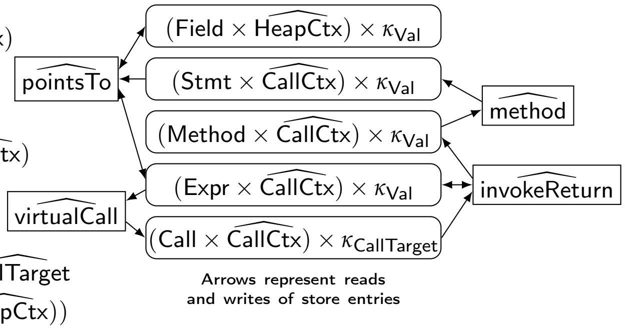
#### 5.4.1.1 *Pointer and Call-Graph Analysis*

A pointer analysis for an object-oriented language computes which objects a variable or field may point to. A call-graph analysis determines which methods may be called at specific call sites. Pointer and call-graph analyses are the foundation which many other analyses build upon. Our analyses are $k$-call-site sensitive, $l$-heap sensitive, and flow-insensitive [94].

The analyses are composed from four analysis modules, whose dependencies are shown in Figure 5.3. An arrow from a store entry to a module represents a read, an arrow in the other direction represents a write. Even though all modules implicitly depend on each other,

[3] Implementation and tests available at `https://doi.org/10.5281/zenodo.7311078`

$$
\begin{aligned}
\widehat{\mathsf{Entity}} = {} & (\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \\
& \mid (\mathsf{Stmt} \times \widehat{\mathsf{CallCtx}}) \\
& \mid (\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \\
& \mid (\mathsf{Method} \times \widehat{\mathsf{CallCtx}}) \\
& \mid (\mathsf{Call} \times \widehat{\mathsf{CallCtx}}) \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{Val}}] = {} & \bot \mid \widehat{\mathsf{Obj}} \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{CallTarget}}] = {} & \widehat{\mathsf{CallTarget}} \\
\widehat{\mathsf{Obj}} = {} & \widehat{\mathsf{Obj}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}})) \\
\widehat{\mathsf{CallTarget}} = {} & \widehat{\mathsf{CallTarget}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}} \times \mathsf{Method} \times \mathsf{Expr}^*))
\end{aligned}
$$



Arrows represent reads and writes of store entries

Figure 5.3: Points-To and Call-Graph Analysis

they can be proven sound independently of each other (Section 5.2). This is possible because none of these modules call other modules directly; instead, all communication takes place via the store.

Module $\widehat{\mathsf{method}}$ registers each statement of a method in the store to trigger other analysis modules. The module disregards the control flow because the analysis is flow-insensitive and hence also registers statements that can never be executed. Such a flow-insensitive analysis can be more performant than a flow-sensitive one, but traditional approaches that use generic abstract interpreters do not allow for flow-insentitive analyses. Module $\widehat{\mathsf{pointsTo}}$ analyzes `New` expressions and assignments of variable and field references. Module $\widehat{\mathsf{virtualCall}}$ resolves the target method of virtual method calls based on the receiver object. Once a call is resolved, module $\widehat{\mathsf{invokeReturn}}$ extends the call context and assigns the method parameters and return value. Furthermore, module $\widehat{\mathsf{invokeReturn}}$ registers the called method as an entity in the store, which in turn triggers module $\widehat{\mathsf{method}}$.

The entities of the analyses are fields, statements, expressions, methods, and calls (Figure 5.3). Each entity is paired with a call context or heap context, which allows tuning the precision of the analysis. The analysis modules communicate via two kinds: Kind $\kappa_{\mathsf{Val}}$ refers to possible values of expressions and fields and the return value of methods. Values are abstract objects containing information about where objects were allocated. Kind $\kappa_{\mathsf{CallTarget}}$ refers to possible targets of method calls. Call targets are sets of receiver objects paired with the target method and their arguments.

To illustrate the analysis, Listing 5.2 shows the code of modules $\widehat{\mathsf{virtualCall}}$ and $\widehat{\mathsf{invokeReturn}}$. Both modules implicitly communicate with each other via the store but do not call each other directly. Module $\widehat{\mathsf{virtualCall}}$ resolves virtual method calls by first fetching the points-to set of the receiver reference from the store. Afterward, it iterates over all possible receivers and fetches possible target methods from the class table. Finally, it writes the new call target to the store. Storing the receiver object and argument expressions as part of the call target allows reusing module $\widehat{\mathsf{invokeReturn}}$ for different types of calls. If the

```
$\widehat{\mathsf{virtualCall}}(e, \widehat{\sigma})$ = $e$ match
  case (call@Call(receiver, methodName, args), callCtx) =>
    receiverVal = $\widehat{\sigma}$((receiver, callCtx), $\kappa_{\mathsf{Val}}$)
    $\widehat{\mathsf{forObj}}$(receiverVal, $\widehat{\sigma}$) { (class, heapCtx, $\widehat{\sigma}'$) =>
      method = classTable(class, methodName)
      $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}),$
            $\kappa_{\mathsf{CallTarget}} \mapsto \widehat{\mathsf{newCallTarget}}(\mathtt{class}, \mathtt{heapCtx}, \mathtt{method}, \mathtt{args})]$
    }
  case _ => $\widehat{\sigma}$

$\widehat{\mathsf{invokeReturn}}(e, \widehat{\sigma})$ = $e$ match
  case (call@Call(_,_,_), callCtx) =>
    targets = $\widehat{\sigma}((\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{CallTarget}})$
    $\widehat{\mathsf{forCallTarget}}$(targets, $\widehat{\sigma}$) { (class, heapCtx, method, args, $\widehat{\sigma}'$) =>
      method match
      case SourceMethod(_,params,_) =>
      newCallCtx = $\widehat{\mathit{extendContext}}$(call.label, heapCtx, callCtx)
      $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
       $\sqcup [(\mathtt{p}, \mathtt{newCallCtx}),$
          $\kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{a}, \mathtt{callCtx}), \kappa_{\mathsf{Val}}) \mid (\mathtt{p}, \mathtt{a}) \in \mathsf{zip}(\mathtt{params}, \mathtt{args})]$
       $\sqcup [(\mathsf{VarRef}(\text{"}\mathtt{this}\text{"}), \mathtt{newCallCtx}),$
          $\kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{newObj}}(\mathtt{class}, \mathtt{heapCtx})]$
       $\sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{nullPointer}}()]$
       $\sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
    case NativeMethod(_,_,_) => $\widehat{\sigma}'$
  }
  case Return(method,expr) =>
      $\widehat{\sigma} \sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}(\mathtt{expr}, \mathtt{callCtx}, \kappa_{\mathsf{Val}})]$
  case _ => $\widehat{\sigma}$
```

Listing 5.2: Analysis modules for Invoking Calls and Resolving the Receiver of Virtual Calls

entity is a `Call` expression, module $\widehat{\mathsf{invokeReturn}}$ first fetches the targets of the call from the store. Afterward, it iterates over all targets, extends the call context with function $\widehat{\mathsf{extendContext}}$, binds the parameters to the values of the arguments and variable `this` to the receiver object. Furthermore, it registers the called method as an entity in the store, triggering module $\widehat{\mathit{method}}$ to process the statements of the called method. Finally, module $\widehat{\mathsf{invokeReturn}}$ writes the return value of a method to the method entity in the store and copies it to call entities of this method.

The modules depend on interface Objects shown in Listing 5.1 and an analogous interface for call targets. Operations $\widehat{\mathsf{newObj}}$ and $\widehat{\mathsf{newCallTarget}}$ create new abstract objects and call targets. Operations $\widehat{\mathsf{forObj}}$ and $\widehat{\mathsf{forCallTarget}}$ iterate over all objects and call targets. Additionally, interface Objects includes an operation $\widehat{\mathsf{nullPointer}}$ not shown in the listing, which returns an empty set of object allocation-sites

($\widehat{\mathsf{Obj}}(\varnothing)$). The dynamic instances are analogous except that they operate on singleton types.

The dynamic modules compute the *heap* of a program and describe how it changes during the execution. The dynamic modules are analogous to their static counterparts except that they operate on singleton types $\mathsf{Obj}(\mathsf{Class} \times \mathsf{HeapCtx})$ and $\mathsf{CallTarget}(\mathsf{Class} \times \mathsf{HeapCtx} \times \mathsf{Method} \times \mathsf{Expr}^*)$.

All dynamic modules combined do not cover the entire language. In particular, there are no dynamic modules that cover reflective calls. This means, as of now, the dynamic semantics of reflection is undefined, and the soundness proof only covers programs without reflective calls. We address this point with the following case study.

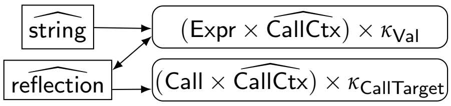
#### 5.4.1.2 *Reflection Analysis*

Reflection is a language feature that allows querying information about classes and methods at runtime [150]. Our language supports three reflective methods: Methods Class.forName and Class.getMethod retrieve classes and methods by a string, respectively. Method Method.invoke invokes a method, where the target method is determined at runtime. Reflection is notoriously difficult to be statically analyzed in a sound and precise way [136]. In particular, analyses need to approximate the content of the string passed into a reflective call. If the analysis cannot determine the string content precisely, it needs to over-approximate or risk unsoundness. In this case study, we choose the former to be able to prove the analysis sound.

This case study demonstrates two important features of our formalization: First, the reflection analysis reuses all modules of the pointer and call-graph analysis of the previous section ($\widehat{pointsTo}$, $\widehat{method}$, $\widehat{virtualCall}$, and $\widehat{invokeReturn}$). It extends the value lattice to propagate new types of analysis information about strings. However, even though the pointer analysis propagates new information, its modules do not require any changes and their soundness proof remains valid (Section 5.3.2). Second, the reflection analysis *cooperates* with the call-graph analysis module $\widehat{virtualCall}$. In particular, reflective calls are regular virtual calls. For example, a call m.invoke(...) where variable m is of type Method is first resolved by virtual call resolution and its target Method.invoke is then resolved by reflective call resolution. This means both analyses add elements to the same set of call targets, yet, the analyses can be proven sound independently of each other (Section 5.2).

The reflection analysis extends the $\widehat{Obj}$ values of the pointer analysis with three new types of values—$\widehat{\mathsf{Str}}$, $\widehat{\mathsf{Class}}$, and $\widehat{\mathsf{Method}}$—as a reduced product [49]. String values are approximated with a constant lattice. Class and method values are approximated with a finite set of classes or methods or $\top$. We reuse the modules of this pointer and call-graph analysis by implementing a new instance of interface

$$\begin{aligned}
&\widehat{\mathsf{Property}}[\kappa_{\mathsf{Val}}] = \bot \mid (\widehat{\mathsf{Obj}} \times \widehat{\mathsf{Str}} \times \widehat{\mathsf{Class}} \times \widehat{\mathsf{Method}})\\
&\widehat{\mathsf{Str}} = \bot \mid \mathsf{String} \mid \top\\
&\widehat{\mathsf{Class}} = \mathcal{P}(\mathsf{Class}) \mid \top\\
&\widehat{\mathsf{Method}} = \mathcal{P}(\mathsf{Method}) \mid \top
\end{aligned}$$


$\widehat{\mathsf{string}}$
$\widehat{\mathsf{reflection}}$
$(\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$(\mathsf{Call} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{CallTarget}}$

Arrows represent reads
and writes of store entries

Figure 5.4: Reflection Analysis

Objects in Listing 5.1 for the new values. The new instance is similar to $\widehat{\mathsf{AllocationSiteAndStrings}}$ and iterates over all allocation-site information in strings, class and method values, as well as other objects.

The reflection analysis adds two new modules to the existing analysis in Figure 5.3. The new modules and their dependencies are visualized in Figure 5.4. Module $\widehat{\mathsf{reflection}}$ analyzes reflective calls to Class.forName, Class.getMethod, and Method.invoke. Module $\widehat{\mathsf{string}}$ analyzes string literals and concatenation. Listing 5.3 shows an excerpt of module $\widehat{\mathsf{reflection}}$ for Method.invoke. Module $\widehat{\mathsf{reflection}}$ first fetches the targets of a call resolved by module $\widehat{\mathsf{virtualCall}}$. If the call target is the native method `invoke`, module $\widehat{\mathsf{reflection}}$ matches on the arguments of the virtual call to extract the receiver and arguments of the reflective call target. Finally, it calls operation $\widehat{\mathsf{methodInvoke}}$ which returns the set of call targets.

Operation $\widehat{\mathsf{methodInvoke}}$ is part of an interface for reflective calls. The interface contains two other operations for retrieving class names and methods. Operation $\widehat{\mathsf{methodInvoke}}$ matches on the call receiver and the method value. If the method value contains a finite set of methods, the operation checks if the receiver class has these methods and adds them as call targets. If the method value contains $\top$, the operation adds all methods of the receiver class to the set of call targets. This over-approximates the dynamic module reflection where only one method is added as a call target.

The dynamic reflection modules are analogous except that different types of values are alternatives. In contrast to Section 5.4.1.1, the dynamic pointer and call-graph modules combined with the reflection and string modules now cover the entire language. This means the analysis is sound for all programs, even those with reflective calls.

#### 5.4.1.3 *Field and Object Immutability Analysis*

The analysis presented in this case study computes the immutability of objects and their fields inspired by a class and field immutability analysis detailed in Chapter 9. This information is useful for assessing the thread safety of programs, where multiple threads have access to the same objects.

This case study highlights two important features of our formalization. First, the core dynamic semantics of our language does not

```
$\widehat{\mathsf{reflection}}(e, \widehat{\sigma})$ = $e$ match
  case (call@Call(receiver, method, args), callCtx) =>
    target = $\widehat{\sigma}$((call, callContext), $\kappa_{\mathsf{CallTarget}}$)
    $\widehat{\mathsf{forCallTarget}}$(target, $\widehat{\sigma}$) { (_, heapCtx, method, arguments, $\widehat{\sigma}'$) =>
        method match
      case NativeMethod("invoke") =>
        arguments match
          case (invokeReceiver :: invokeArgs) =>
          invokeRecVal = $\widehat{\sigma}'$((invokeReceiver, heapCtx), $\kappa_{\mathsf{Val}}$)
          methodVal = $\widehat{\sigma}'$((receiver, callContext), $\kappa_{\mathsf{Val}}$)
          reflectiveTarget = $\widehat{\mathsf{methodInvoke}}$(invokeRecVal, methodVal,
              invokeArgs)
          $\widehat{\sigma}' \sqcup [(\texttt{call}, \texttt{callCtx}), \kappa_{\mathsf{CallTarget}} \mapsto \texttt{reflectiveTarget}]$
      ...
      }
  case _ => $\widehat{\sigma}$

$\widehat{\mathsf{methodInvoke}}$(receiver: $\widehat{Value}$, methodVal: $\widehat{Value}$, invokeArgs: Expr$^*$) =
    methodVal match
  case (_,_,_,methods) =>
    $\widehat{\mathsf{CallTarget}}$({ (class, heapCtx, method, invokeArgs) |
      (class,heapCtx) $\in$ receiver, method $\in$ methods, method $\in$
          classTable(class) })
  case (_,_,_,$\top$) =>
    $\widehat{\mathsf{CallTarget}}$({ (class, heapCtx, method, invokeArgs) |
      (class,heapCtx) $\in$ receiver, method $\in$ classTable(class) })
  case $\bot$ => $\bot$
```

Listing 5.3: Analysis Modules and Operations for Reflective Calls

describe the immutability property. Therefore, we need to prove the static immutability analysis sound with respect to a dynamic immutability analysis. The case study demonstrates that the immutability concern can be encapsulated with analysis and dynamic modules, added modularly to the existing analysis and dynamic semantics, and reasoned about independently (Section 5.2). It is unclear how this can be achieved with a non-modular, monolithic analysis implementation. Second, the immutability analysis adds new types of entities and kinds to the store and reuses all modules of the pointer, call-graph, and reflection analysis. Even though the reused modules can be called with the new entities and have access to new kinds in the store, their soundness proofs remain valid (Section 5.3.1).

The immutability analysis adds objects (Class × HeapCtx) to the existing types of entities and adds two new kinds $\kappa_{\mathsf{Mut}}$ and $\kappa_{\mathsf{Assign}}$ for their immutability and the assignability of their fields. $\kappa_{\mathsf{Mut}}$ uses a lattice with three elements shown in Figure 5.5. The greatest element $\widehat{\mathsf{Mutable}}$ describes objects the fields of which have been reassigned. The middle element $\widehat{\mathsf{NonTransitivelyImmutable}}$ describes objects the fields of which

$$\begin{aligned}\widehat{\mathsf{Entity}}' &= \widehat{\mathsf{Entity}} \mid (\mathsf{Class}\times\widehat{\mathsf{HeapCtx}})\\ \widehat{\mathsf{Property}}[\kappa_{\mathsf{Mut}}] &= \widehat{\mathsf{TransitivelyImmutable}}\\ &\mid \widehat{\mathsf{NonTransitivelyImmutable}}\\ &\mid \widehat{\mathsf{Mutable}}\\ \widehat{\mathsf{Property}}[\kappa_{\mathsf{Assign}}] &= \widehat{\mathsf{Assignable}}\\ &\mid \widehat{\mathsf{NonAssignable}}\end{aligned}$$

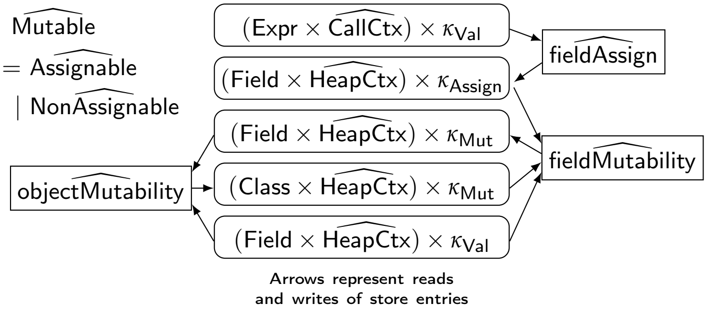


Figure 5.5: Immutability Analysis

have not been reassigned, but some objects transitively reachable via fields have been mutated. The least element $\widehat{\mathsf{TransitivelyImmutable}}$ describes objects the fields of which have not been reassigned and no transitively reachable objects have been mutated. $\kappa_{\mathsf{Assign}}$ uses only two elements for fields reassigned and fields that are not reassigned.

The immutability analysis is implemented with three modules as shown in Figure 5.5. Module $\widehat{\mathsf{fieldAssign}}$ sets fields `f` of objects `o` to $\widehat{\mathsf{Assignable}}$ for every assignment of the form `x.f = e`, where `x` may point to `o`. Module $\widehat{\mathsf{fieldMutability}}$ sets a field to $\widehat{\mathsf{Mutable}}$ if the field is assignable, to $\widehat{\mathsf{NonTransitivelyImmutable}}$ if the field is non-assignable but one of the objects the field points to is mutable, and to $\widehat{\mathsf{TransitivelyImmutable}}$ otherwise. Lastly, module $\widehat{\mathsf{objectMutability}}$ sets the immutability of an object to the least upper bound of the immutability of all of its fields.

The dynamic modules are analogous except that they operate on concrete objects instead of abstract objects.

### 5.4.1.4 *Demand-Driven Reaching-Definitions Analysis*

As a final case study, we developed a demand-driven intraprocedural reaching-definitions analysis for our object-oriented language. This case study demonstrates that our theory lifts a restriction of existing soundness theories for generic interpreters. In particular, our theory applies to analyses, which do not follow the program execution order.

The analysis computes which definitions of variables and fields reach a statement without being overwritten. The analysis is demand-driven, as it performs only the minimum amount of work to compute the reaching definitions of a query statement. More specifically, the analysis only computes the reaching definitions of the query statement and all of its predecessors. Additionally, the analysis does not compute the entire control-flow graph but only the predecessors of the query statement.

The reaching-definitions analysis consists of two modules, namely $\widehat{\mathsf{reachingDefs}}$ and $\widehat{\mathsf{controlFlowPred}}$, that are analogous to the modules in Figure 5.1 and Figure 5.2. We already discussed module $\widehat{\mathsf{reachingDefs}}$ and its dynamic counterpart in depth in Section 5.1. $\widehat{\mathsf{controlFlowPred}}$ calculates the set of control-flow predecessors of a given statement by computing the set of control-flow exits of the preceding statement within the abstract syntax tree. For example, the control-flow exits of an `if` statement are the exits of the last statements of both branches. The dynamic module $\mathsf{controlFlowPred}$ computes the predecessor immediately executed before the given statement. To this end, the module remembers the most recently executed statement in a mutable variable and only updates it if the given statement is the control-flow successor.

### 5.4.2 *Soundness Proofs of the Case Studies*

We apply our theory to compositionally prove the analyses from the previous section sound. The proofs can be found in Part V. They are pen-and-paper proofs and do not make use of mechanization—but due to modularization, they are short and easy to verify.

Proving each analysis sound includes (a) proving each of its modules sound (Definition 5.12), (b) proving the instances of the property interface sound, and (c) verifying that Theorem 5.13 applies. To ensure the latter, we checked that there are no dependencies between modules and that all communication between them happens via the store (Definition 5.1). This can be easily checked by inspecting the code of the modules. Furthermore, we verified that modules do not make any assumption about abstract domains and are polymorphic in the store (Definition 5.11), which can be done by simply inspecting the polymorphic type of the modules.

To prove the individual modules of an analysis sound, step (a) in the overall soundness proof, we use two proof techniques. The first technique uses the observation that analysis modules and their corresponding dynamic modules are often very similar, except for differences in the type of entities and properties. We can abstract over these differences with a generic module, from which we derive both a dynamic and an analysis module. In this case, soundness follows immediately as a free theorem from parametricity [125]. In cases where abstracting with a generic module is not possible or desirable, we resort to a manual proof. We were able to use the first proof technique for all analysis modules, except for $\widehat{\mathsf{method}}$, $\widehat{\mathsf{reachingDefs}}$, and $\widehat{\mathsf{controlFlowPred}}$. For illustrating cases where we need manual proofs, consider the flow-insensitive analysis module $\widehat{\mathsf{method}}$ of the pointer analysis and its corresponding dynamic module $\mathsf{method}$. While we could potentially derive them from the same generic module, the derived analysis module would be less performant. This is because it would trigger the analysis of parts of the code, e.g., `if` conditions, which our current flow-insensitive module does not. This is an example where our approach leads to more freedom in the design of static analyses than the existing approach based on a generic interpreter (Section 6.5).

The soundness proofs of the analysis modules are reusable across different analyses because the modules can be soundly lifted to supersets of entities and kinds (Lemma 5.10). For example, the immutability analysis adds class entities, requiring to lift the modules of the pointer and reflection analysis. Furthermore, the soundness proofs of analysis modules can be reused because the proofs are independent of the lattices used (Definition 5.12). For example, the reflection analysis reuses all modules of the pointer analysis, extending the value lattice with string, class, and method information. The soundness proofs of the pointer analysis modules remain valid because they do not depend on a specific value lattice. Instead, the proofs of the pointer modules depend on soundness lemmas of the $\mathsf{newObj}$ and $\mathsf{forObj}$ operations of $\mathsf{Objects}$ interface.

Finally, we consider step (b) in the overall process of proving an analysis sound—the soundness proof of the instances of the property interface. These instances still need to be proven sound manually, because the proof cannot be decomposed any further. To prove the instances of the property interface sound, we proved each of the interface's operations sound. More specifically, for the pointer analysis we had to prove 7 operations sound, for the reflection analysis 6 operations, for the immutability analysis 6 operations, and for the reaching-definitions analysis 0 operations. Of these 19 operations, 13 operations could be proven sound trivially, requiring only a single proof step after unfolding the definitions. The remaining 6 operations required more elaborate proofs with multiple steps and case distinctions. These include $\widehat{\mathsf{forObj}}$ used in the pointer analysis, $\widehat{\mathsf{classForName}}$, $\widehat{\mathsf{getMethod}}$, and $\widehat{\mathsf{methodInvoke}}$ used in the reflection analysis, and $\widehat{\mathsf{getFieldMutability}}$ and $\widehat{\mathsf{joinMutability}}$ used in the immutability analysis.

### 5.4.3 *Case-Study Summary*

To showcase the applicability of our theory, we developed four case-study analyses and their dynamic semantics in the blackboard analysis architecture. Each case study was chosen to exercise a different aspect of our soundness theory: The pointer analysis demonstrates proof independence despite mutual dependencies between analysis modules. The reflection analysis demonstrates proof reuse despite changes to the value lattice. The immutability analysis demonstrates proof reuse despite an extended entity type. Lastly, the demand-driven reaching-definitions analysis demonstrates that our theory applies to a wider range of analyses than existing soundness theories.

We implemented and tested the analyses and their dynamic semantics in Scala to show they are executable. Language features such as reflection and properties such as immutability can be encapsulated with modules and added to existing analyses in a modular way. Furthermore, analysis modules are loosely coupled by communicating via the store, even though they implicitly depend on each other and collaborate with each other. We proved each analysis of our case studies sound. The proofs can be found in Part V. The soundness of each analysis follows directly from independent soundness proofs of each module. This means that no reasoning about analyses as a whole is necessary.

## 5.5 SUMMARY

In this chapter, we developed a theory for compositional and reusable soundness proofs for static analyses in our blackboard analysis architecture. We proved that soundness of an analysis follows directly from independent soundness proofs of each module. Furthermore, we extended our theory to enable the reuse of soundness proofs of existing modules across different analyses. We evaluated our approach by implementing four analyses and proving them sound: A pointer, a call-graph, a reflection, an immutability analysis, and a demand-driven reaching definitions analysis.

This showcases how analyses in our blackboard analysis architecture can be proven sound. Work remains to be done to prove full analyses sound that use all features of *OPAL* instead of just the core concepts formalized here, but these extensions are rather straightforward and would only hamper comprehensibility of the proofs and not add deeper insights into the approach.

This is an effort to narrow the gap between impractical academic analyses that can be proven sound and useful applied analyses that are too complex to be proven sound. We believe that some complexity of applied analyses is incidental and can be better managed by modularizing their implementation, which makes a compositional soundness proof more feasible.

# 6 RELATED WORK

This chapter presents work related to our approach. We start by giving an overview of the history of blackboard systems, then discuss in detail different approaches to implement static analyses. In particular, we survey abstract interpretation, declarative approaches, attribute grammars, and imperative approaches.

## 6.1 BLACKBOARD SYSTEMS

The blackboard metaphor described in Chapter 2 was introduced by Newell [168]. Blackboard systems were, e.g., used for speech- [79] and image recognition [146], vessel identification [171], and industrial process control [68]. For these domains, no efficient, deterministic algorithm is known, leading to several problems mentioned by Buschmann et al. [37]: nondeterminism making testing difficult, no guarantee for good solutions, scalability suffering from wrong hypotheses, and high development effort due to ill-defined domains.

As static analyses have a well-defined domain and deterministic algorithms, these do not apply to our approach. Using a blackboard architecture allows *OPAL*'s analyses to be modularized strictly independent of each other, communicating solely via the blackboard. The blackboard architecture inherently allows for easy parallelization due to the independence of individual analyses, alleviating the need for analysis-specific parallelization schemes.

The structure of blackboard systems is described, e.g., by Nii [170], Craig [56], and Corkill [48]. Corkill also discusses concurrent execution of knowledge sources and the control component [47], similar to *OPAL*. *OPAL* resembles a more modern interpretation of blackboard systems [57]: its blackboard is not hierarchical and analyses may keep state between activations. Information is, however, never erased, and all communication is done via the blackboard.

Brogi and Ciancarini used the blackboard approach to provide concurrency for their *Shared Prolog* language [35]. Like static analyses, this domain is well-defined. Their knowledge sources are restricted to be Prolog logical programs, while *OPAL*'s analyses can be implemented in a way best suited to the analysis needs.

Decker et al. [64] discuss the importance of heuristics for scheduling concurrent knowledge source activations. Focusing on static analyses and well-defined dependency relations, *OPAL*'s scheduling strategies provide good general heuristics, agnostic to individual analyses, and we evaluate an analysis-specific scheduling strategy in Section 13.3.

## 6.2 general purpose analysis frameworks

*OPAL* is a general-purpose framework for static analysis. As such, it is similar to some state-of-the-art frameworks that we discuss in this section. In particular, the *WALA* and *Soot* frameworks have similar goals and target audiences.

wala The T.J. Watson Libraries for Analysis (*WALA*) [111] are a static analysis framework for Java Bytecode similar to *OPAL*. Initiated by IBM, *WALA* is open source like *OPAL* and comes with many similar components and analyses. These include an intermediate representation for easing the implementation of static analyses, points-to and call-graph analyses, and dataflow analyses. We particularly compared our modular architecture for call-graph construction, *Unimocg* (cf. Chapter 8) to state-of-the-art call-graph implementations from *WALA*. We showed that *Unimocg* is on par with them with respect to precision and scalability while providing a more consistently high support for language features that impact call-graph construction. An overview of *WALA*'s call-graph architecture will be given in Section 8.2. Other than *OPAL*, *WALA* also has some support for static analysis of JavaScript. However, with the modular, collaborative architecture of *OPAL*'s analyses, we plan to support not only analyses for languages that are not based on Java Bytecode but also cross-language analyses targeting hybrid applications in the near future (cf. Chapter 15).

soot The *Soot* framework [176] originally started out as a framework for optimizing Java Bytecode [238]. Today, it is a general-purpose static analysis framework that can handle Java Bytecode as well as Android Bytecode. Like *WALA* and *OPAL*, *Soot* provides intermediate representations (*Jimple* [239] and *Shimple* in particular), points-to and call-graph analyses as well as dataflow analyses. Interprocedural dataflow analysis using the IFDS [190] and IDE [199] algorithms is supported via the Heros tool [25]. Taint analysis is available using the *FlowDroid* [8] or *IDEaL* [218] tools. *Boomerang* [219] is a points-to and alias analysis based on *Soot*. We discuss the *Jimple* and *Shimple* intermediate representations in Chapter 7 and compare *TACAI*, our intermediate representation based on abstract interpretation, against *Shimple* in Section 12.1. In Section 8.2, we give an overview of *Soot*'s call-graph architecture, which we compare our *Unimocg* architecture (cf. Chapter 8) against.

phasar The *PhASAR* framework [206] provides tools for interprocedural static dataflow analysis of C and C++ [207]. It enables analysis developers to specify IFDS and IDE analyses as well as analyses in the monotone dataflow analysis framework. *PhASAR* provides supporting analyses including call graphs and points-to analyses ready to use for analysis developers. Analyses in *PhASAR* are modular, allowing analysis developers to use only those components that they require. Building on experiences with *Soot*, *PhASAR* makes analysis results available to other analyses using a mediator pattern, which is similar to our blackboard architecture style [208]. However, while supporting analyses can execute in an interleaved manner, dataflow analyses are executed sequentially after the supporting analyses and cannot be interleaved to allow them to collaboratively compute properties or depend on each other cyclically. Also, *PhASAR* is not built to support analyses other than dataflow analyses and a small set of supporting analyses.

## 6.3 DECLARATIVE ANALYSES USING DATALOG

Datalog is often used to implement static analyses in a strictly declarative fashion [73, 96, 135, 189, 242–244]. Properties are represented as relations and analyses given in terms of Datalog rules specify how to compute them. This enables modularization, as rules can be easily exchanged and/or added (e.g. for new language features).

The *Doop* framework [34] by Bravenboer and Smaragdakis, has shown that the rule-based approach enables precise and scalable points-to analyses. For this reason, *Doop* became the state of the art for such analyses [121, 214, 216, 230, 231]. Taint analysis is possible with *Doop* using Grech and Smaragdakis' *P/Taint* analysis [93]. *Doop* describes analyses with relations in Datalog. Each relation is defined as a set of rules. These rules can be modularly added or replaced, without requiring changes to other rules. Input facts, i.e., information extracted from the program to be analyzed, are generated using *Soot* to bootstrap the Datalog computation. The latter is performed using Jordan et al.'s highly optimized Datalog solver *Soufflé* [118].

*Soufflé* is a parallel Datalog solver specifically developed for static analyses [204]. It takes the analysis specification as an input and compiles it into a C++ program. *Soufflé* makes use of OpenMP, e.g., to parallelize nested join loops. Using 4 cores, a speed-up of 2.1x compared to the sequential version was achieved. Similar speed-ups of over 2x have been reported for *Doop* when using *Soufflé* as its underlying solver but using 4 to 8 threads on 24 cores [6].

Datalog-based frameworks, however, are limited in their expressiveness by using relations, i.e., set-based abstractions, to represent all analysis results. *OPAL*'s approach that combines imperative and declarative features provides similar benefits as Datalog-based approaches while allowing for more expressive ways to represent data and to implement analyses. We compare the scalability of our points-to-based call-graph algorithms against *Doop*'s in Chapter 13, showing that *OPAL* outperforms *Doop* even though it is more general and less optimized for points-to analyses than *Doop*.

Individual analyses in *Doop* have been proven sound before [217], but the proofs are not compositional or reusable. In particular, if one rule changes, the proof becomes invalid and needs to be re-established. This is because the proof reasons about the soundness of all rules at once instead of individual rules or relations. As we showed in Chapter 5, this is not the case with *OPAL*.

Datalog's limitation to relations has also been pointed out by Madsen et al. [154]. They propose *Flix* to overcome this shortcoming using a language inspired by Datalog and Scala to specify declarative pluggable analyses using arbitrary lattices as in *OPAL* as well as functions. *Flix* proves individual functions sound with an automated theorem prover [153]. While an automated theorem prover reduces the proof effort and increases proof trustworthiness, there is no guarantee that the automated theorem prover is able to conduct a proof. Furthermore, the automated theorem prover does not establish a soundness proof of Datalog relations. However, *Flix* focuses on verifying soundness and safety properties of static analyses and not on scalability. For instance, *Flix* does not allow optimized data structures or scheduling strategies. We wanted to compare our approach against *Flix* and contacted the authors, but they answered that their IFDS implementation is dysfunctional now and suggested comparing against *Doop* with the Soufflé engine, which we do in Section 13.4.2.1.

Szabó et al. [229] also extend Datalog to allow relations over arbitrary lattices for static analysis. Their solver *IncA* focuses on incrementalization. *OPAL* allows optimizations, e.g., of the data structures or scheduling strategies used. Furthermore, the coarser granularity of *OPAL*'s analyses compared to individual rules reduces overhead in parallelization. Also, no soundness theory exists for *IncA*'s analyses.

## 6.4 ATTRIBUTE GRAMMARS

Attribute grammars [130] used in compilers such as *JastAdd* [76] enable modular inference of program properties by adding computation rules to the nodes of a program's abstract syntax tree (AST). In traditional attribute grammars, attributes may only depend on parent, sibling, and child nodes. Circular reference attribute grammars [82, 99, 117, 155] enable attributes to depend on arbitrary AST nodes and allow circular dependencies. Still, analyses are tightly bound to the AST, impeding natural expression of analyses based on different structures, such as a control-flow graph. Similar to *OPAL*, *JastAdd* enables pluggability for new language features. However, *JastAdd* requires at least one attribute in a cyclic dependency to be marked explicitly, while *OPAL* handles this transparently.

Öqvist and Hedin [172] proposed concurrent evaluation of low complexity attributes in circular reference attribute grammars. *OPAL*, on the other hand, supports arbitrary granularity of concurrent computation. *OPAL*'s explicit dependency management enables analyses to drop dependencies and commit final results early for improved scalability. Finally, as memorization of properties is done in *OPAL*'s blackboard, temporary values are garbage collected automatically, whereas JastAdd requires explicit removal.

## 6.5 ABSTRACT INTERPRETATION

Modular static analysis has been an important target for abstract interpretation. Abstract interpretation is a theory for proving soundness of static analyses, first conceived by Cousot and Cousot [52] but since then has found widespread adoption in academia and industry [53, 89, 107, 119, 142, 221].

Previous research has proven that multiple (possibly computationally inexpensive) abstract domains (i.e., analyses) can be combined using the reduced product to increase overall precision [52]. Examples of combined domains include different orthogonal domains for the same value, such as signs and parity [52] domains, and combination of value information and relations, e.g., intervals and linear equations [81]. In implementations such as *Astrée* [54] or *Clousot* [81], communication between domains is performed using a hierarchy of domains: Each domain has access to preconditions for the current statement from all domains but only to postconditions from previously computed domains. This is either realized via an IO channel [54] or via *pushing* and *pulling* the results from/to other domains [81]. Thus, the same program statement must be analyzed multiple times. *OPAL*, by contrast, makes dependency handling explicit, reducing the communication overhead. Typical analyses in *OPAL* only need to analyze each statement once, keeping only the state required to handle updates of dependencies later.

Abstract interpretation typically aims to compute abstract *approximations* [51] of concrete values, such as an integer variable's value, while it is not clear how to deal with interactions of analyses of different entities on different levels of granularity. *OPAL* further allows natural expression of analyses on all granularity levels.

Jourdan et al.'s *Verasco* [120] is a modular analysis for *C#minor* [141], an intermediate language used by the *CompCert* C compiler [142]. *Verasco* is proven sound with the *Coq* proof assistant [18]. The soundness proof of the abstract *C#minor* semantics is independent of the abstract domain, which makes the proof reusable for other abstract domains. However, the abstract semantics is proven sound with respect to the *standard* concrete semantics. This means the proof cannot be reused for abstract semantics which approximate *non-standard* concrete semantics, such as information flow analyses [9] or liveness analyses [50].

### 6.5.1 *Generic Abstract Interpreters*

Keidel et al. [125] developed a theory for modular *big-step abstract interpreters*, deriving both the static and dynamic semantics from a generic big-step interpreter. This simplifies soundness proofs, as the shared generic interpreter is sound by definition and only the differences in instantiating it for the static and dynamic semantics need to be proven sound w.r.t. each other. The generic interpreter is composed from primitive operations that are proven sound independently, similar to different domain component traits in the abstract interpreter for our *TACAI* intermediate representation. The theory enables soundness composition [125, Theorem 4 and 5] under the assumption that the generic big-step interpreter is implemented with *arrows* [110] or in a meta-language that enjoys *parametricity*. However, there is no theory on how parts of soundness proofs can be reused between different analyses. They implement their theory in *Sturdy* [122], a Haskell library for implementing sound static analyses, such as type checkers, bug finders, taint analyses, and analyses for compiler optimizations in a modular fashion. Keidel and Erdweg [123] later refined the theory by introducing reusable *analysis components* that capture different aspects of the language, such as values, mutable state, or exceptions, and are described with *arrow transformers* [110]. These analysis components can then be combined to model different semantics. They also used *Sturdy* to implement sound abstract interpreters for the *Stratego* program transformation language [124].

While components can be proven sound independently of each other, their *composition* requires glue code, which needs to be proven sound. Furthermore, the composition creates large arrow transformer stacks—unless optimized away by the compiler, this may lead to inefficient analysis code. For example, a taint analysis for WebAssembly developed by using the approach depends on a stack of 18 arrow transformers.[1] Eliminating the overhead of an arrow transformer stack of this size requires aggressive inlining and optimizations causing binary bloat and excessive compile times.

Analyses in *OPAL* are not required to be implemented in a particular style and need no specialized glue code. Finally, Keidel also modularizes fixed-point algorithms as reusable *fixpoint combinators* in *Sturdy*, enabling their composition and exchangeability. *OPAL*'s fixed-point solver is also decoupled from individual analyses and can thus be exchanged. While *OPAL* currently features only two such solvers, a sequential and a parallel one, we described exchangeable scheduling strategies in Chapter 4 to control the order in which tasks are processed.

[1] `https://gitlab.rlp.net/plmz/sturdy/-/blob/wasm/wasm/src/TaintAnalysis.hs#L96-113`

Bodin et al. [28] developed a theory of compositional soundness proofs for a style of semantics called *skeletal semantics*, which consists of hooks (recursive calls to the interpreter), filters (tests if variables satisfy a condition), and branches. Both the dynamic and static semantics are derived from the same *skeleton*. Furthermore, soundness of the instantiated skeleton follows from soundness of the dynamic and static instance [28, Lemma 3.4 and 3.5]. However, their work does not describe how soundness proofs can be reused across different analyses.

To recap, in all theories above both the static and dynamic semantics *must* be derived from the same generic interpreter. This restricts what types of analyses can be derived. In particular, the static analysis is forced to closely follow the program execution order dictated by the generic interpreter, and it is unclear how static analyses can be derived that do not closely follow the program execution order. For example, backward analyses process programs in reverse order, flow-insensitive analyses may process statements in any order, and summary-based analyses construct summaries in bottom-up order. Our work lifts the restriction that both static and dynamic semantics must be derived from the same artifact. In particular, analyses must follow the blackboard architecture style, but no further restrictions apply to their implementation. This gives greater freedom as to which types of analyses can be implemented. For example, our blackboard analysis architecture has been used to develop backward analyses [91], on-demand/lazy analyses (cf. Chapter 9 and Chapter 10), and summary-based analyses like IFDS (cf. Chapter 11). Furthermore, we have demonstrated in Section 5.4.1.4 that our theory applies to a demand-driven reaching definitions analysis. It is unclear how such an analysis can be derived from a generic interpreter.

## 6.6 imperative approaches

Beyer et al. introduced *Configurable Program Analysis* (CPA) [20], a modular analysis architecture that describes analyses with transfer relations between control-flow nodes. Multiple CPAs can be systematically composed with reduced products. Furthermore, soundness of a component-wise transfer relation follows immediately from soundness of its constituents. However, it is unclear how soundness proofs of primitive CPAs can be composed or how proof parts can be reused across analyses. CPA analyses must specify several functions, including a transfer relation, merge operation, and termination check function, conforming to the requirements of CPA. *CPAchecker* [20] uses CPA for configurable software verification and analysis. For any combination of analyses, *CPAchecker* requires defining a compound analysis to integrate results of individual analyses and manage their interaction. For *CPA+* [21], combined analyses must work with the same domain and provide an explicit measure of result precision and a precision adjustment function. In turn, the precision of analyses can be adjusted and analyses enabled and disabled to fine-tune the trade-off between precision and scalability. By contrast, *OPAL* allows more freedom in the design of analyses, posing minimal restrictions for the initial analysis and continuation functions. In particular, analyses in *OPAL* are not required to work with control-flow automata as CPA analyses are, allowing them to easily analyze properties that are not coupled to the control flow of the analyzed program. *OPAL* enables tight interaction and interleaved execution of independently-developed analyses without requiring a compound analysis or explicit measure of precision. Fine-tuning trade-offs regarding soundness is also possible in *OPAL*.

Lerner et al. [140] proposed modularly composed dataflow analyses that communicate implicitly through optimizations of the analyzed code or explicitly through *snooping*. A fixed-point algorithm repeatedly reanalyzes the code, while *OPAL*'s explicit dependencies avoid reanalysis. The system only supports dataflow analyses, while *OPAL* enables a wide range of analyses including but not limited to dataflow analyses.

Johnson et al. [116] present a framework for collaborative alias analysis. Clients ask queries that are processed by a sequence of analyses. Each analysis can either answer the query or forward it to the next one. Analyses can also generate additional (premise) queries. [116, 140, 208] To ensure termination, a complexity metric must be defined and premises must be simpler than the queries they originate from. Therefore, cyclic dependencies, required for optimal precision, and results combined from different analyses are not supported.

In a similar approach, Christakis et al. [41] make assumptions of static checkers explicit. These assumptions are then verified by further checkers or—if no checker can prove or disprove them—used for automated test generation. Their approach is restricted to verification and, again, cyclic dependencies are not supported.

Early ideas that were later incorporated into *OPAL*'s blackboard architecture were developed by Eichberg et al. for *Magellan* [75]. Like *OPAL*, *Magellan* supports combining dissimilar analyses to solve an overall goal with analysis results communicated via a central store, the *whole-program database*. In *Magellan*, analyses are accompanied by a declarative specification of their dependencies. A constraint solver is then used to compute a schedule that allows analyses to be executed such that results from analyses are computed before they are required by other analyses while at the same time allowing analyses to be executed in parallel whenever possible. In contrast to *Magellan*, analyses in the blackboard architecture can also communicate intermediate results, allowing them to cyclically depend on each other to improve their respective precision and/or soundness. No constraint solver is required as analyses can be scheduled with minimal constraints.

None of these analysis architectures have formal theories for soundness. In contrast, we present a formalization that captures the core of our blackboard analysis architecture as implemented in the *OPAL* framework, while deliberately ignoring implementation details. For example, our formalization does not describe the fixed-point algorithm and the order in which it executes analysis modules to resolve their dependencies. Proving the fixed-point algorithm correct is a separate concern compared to proving analyses sound, which is the focus of our formalization. That said, our formalization covers a variety of *OPAL*'s features described in Chapter 3. For example, *OPAL* supports *default* and *fallback* properties for missing properties in the blackboard. Fallback properties can be described by our formalization by adding them to the initial store passed to the fixed-point algorithm. We deliberately leave out default properties, which are an edge case in *OPAL* to mark properties not computed, e.g., because of dead code. They could be added to our formalization by extending analyses with a second set of analysis modules to be executed after the fixed point is reached. Furthermore, *OPAL* supports *optimistic* analyses which ascend the lattice and *pessimistic* analyses which descend the lattice during fixed-point iteration. Both of these are covered by our formalization, which describes analyses as monotone functions that ascend or descend the lattice. However, we deliberately do not cover *OPAL*'s mechanisms for allowing interaction between optimistic and pessimistic analyses, another edge case.

## 6.7 REACTIVE FRAMEWORKS FOR STATIC ANALYSES

Reactive programming provides abstractions for event streams and time-changing values (signals) [77, 156, 159, 162, 200, 201], which are well-suited for smart dependency management for static analyses. However, general-purpose reactive programming approaches cited above organize computations in an acyclic graph—by being general purpose, they have no means to resolve cycles out of the box and hence the requirement that the graph is acyclic. However, cyclic data dependencies are essential for the target domain of static analysis. To address this need, the reactive programming framework underlying *RA2*, our implementation presented in Chapter 4, is more specialized. It is a reactive framework for time-efficient, concurrent, fixed-point computation on lattices Using lattices, *RA2* can resolve cycles in a generic way. *RA2* builds on Haller et al.'s *Reactive Async* [97]. *Reactive Async* is a programming system for deterministic concurrency in Scala that extends lattice-based shared variables, LVars [134], with cyclic data dependencies, which are resolved after the computation has reached a quiescent state and shared variables are no longer updated. The system concurrently executes tasks based on lattices and the authors apply this to static analysis. *RA2* significantly extends upon *Reactive Async*.

First, *RA2* allows shared variables (cells) to use a sequential update strategy, enabling continuations to safely access shared mutable state by executing them sequentially. Supporting shared mutable state is essential for implementing a state-of-the-art IFDS solver, which is the basis for our experimental evaluation (see Section 11.4). Second, *RA2* allows custom cell updaters such that monotonicity violations are detected dynamically, while expensive additional joins can be omitted. Third, *RA2* supports pluggable scheduling strategies which, as we show in Section 13.3, have a significant impact on scalability and allow analysis-specific tuning of the parallelization. Finally, aggregation of updates, both for a single dependee and across multiple dependees, reduces the number of continuation invocations for further scalability improvements. Both *Reactive Async* and *RA2* require dependencies to be managed by the client, while our *OPAL* implementation manages them automatically based on declarative specifications.

## 6.8 PARALLEL STATIC ANALYSES

There exist several previous efforts to parallelize the solution of static analysis problems.

*Heros* [25, 104] is a parallel, state-of-the-art IFDS solver [190]; it is one of the benchmark implementations in our experimental evaluation (cf. Section 13.4.4). Later approaches, e.g., *Boomerang* [219] that built upon *Heros*, were not parallelized at all, however.

Méndez-Lojo et al. [161] parallelized a points-to-analysis algorithm using the *Galois* system [132, 133]—a programming system for thread-safe parallel iteration over unordered sets. Like *RA2*, their approach relies on an underlying programming framework to provide thread safety out of the box to the analyses. However, unlike *RA2*'s, *Galois* is, a generic framework not specifically tailored to static analysis. For example, it does not provide support to automatically find fixed points. As a result, the approach by Méndez-Lojo et al. is not directly applicable to the parallel execution of static analyses like *RA2*.

Rodriguez and Lhotak present *IFDS-A* [192], an algorithm for solving IFDS dataflow analysis problems using the actor model [2, 105] of concurrency. In order to apply IFDS-specific scheduling strategies, the authors were required to completely exchange the Scala Actors scheduler [98] with their own implementation. Combined with their custom strategy, this was necessary for significant scalability improvements. In contrast to *IFDS-A*, our approach is not limited to parallelizing IFDS; in fact, all of our case-study analyses from Part II can be executed concurrently automatically without explicitly being designed for that. While being more general and using a scheduling strategy not specific to IFDS, our approach achieves similar speed-ups. Finally, our pluggable scheduling strategies also enable significant scalability improvements (cf. Section 13.3).

## 6.9 summary

In this chapter, we saw that there is a multitude of approaches for the modularization and parallelization of static analyses. However, none of these approaches fulfill all of the requirements that we identified for a generic framework for modular, collaborative static analysis (Chapter 2). Most approaches were developed with a particular kind of static analyses in mind, e.g., dataflow analyses or call graphs based on points-to analysis. They also often enforce a particular implementation style, such as a generic abstract interpreter or the use of Datalog rules. Collaboration between sub-analyses is usually not supported as well, i.e., analyses can not share intermediate results in a way that allows them to solve problems with cyclical dependencies in a sound and precise manner.

This is in contrast to *OPAL*, which was developed based on these requirements and thus supports a broad range of dissimilar static analyses in a modular, collaborative way. *OPAL* is built to enable fine-tuning and experimenting with trade-offs between soundness, precision, and scalability, regardless of a particular kind or implementation style of static analyses. Chapter 5 also provides a formalization and shows the compositionality of soundness proofs for the core concepts of our approach.

Part II

# CASE STUDIES

In this part, we discuss four families of analyses that we built using *OPAL*: a three-address-code intermediate representation, call-graph construction, immutability analyses, and purity analysis.

These case studies fulfill three purposes: First, they showcase how to use *OPAL* and the concepts we developed in Chapter 3 to implement complex, modular analyses. This includes relating the analyses to the requirements from Chapter 2 that we list again for reference in Table II-1. Second, as each case study depends upon all of the case-study analyses described before it, they show how complex systems of analyses are combined from individual, isolated building blocks in a modular way. This happens in a plug-and-play manner, enabling to study and fine-tune trade-offs between soundness, precision, and scalability. Finally, each case study is a contribution of its own, advancing the state of the art in the respective sub-field of research in static analysis.

The chapters of this part each discuss one individual case study. They motivate the presented work both from the perspective of showcasing the features of *OPAL* and from the perspective of advancing the specific research area. In Part III, we will use the case studies to evaluate our approach. In particular, we use them to show *OPAL*'s broad applicability, the case studies' modularity, precision, and soundness as well as their scalability.

We start with a short overview of each of the case studies, the requirements they particularly exercise, and their relations to each other.

Table II-1: Summary of Requirements

| | |
|---|---|
| *Lattices and values* | |
| **R1** | Support for different kinds of lattices (7, 8, 9, 10) |
| **R3** | Fallbacks of properties when no analysis is scheduled (7, 9, 10) |
| **R9** | Default values for entities not reached by an analysis (8) |
| *Composability* | |
| **R2** | Support for enabling/disabling individual analyses (7, 8, 9, 10) |
| **R4** | Interleaved execution with circular dependencies (7, 8, 9) |
| **R5** | Combination of optimistic and pessimistic analyses (7) |
| **R6** | Different activations contributing to a single property (8) |
| **R7** | Independent analyses contributing to a single property (8) |
| *Initiation of property computations* | |
| **R8** | Precomputed property values (8, 10) |
| **R10** | Start computation once an analysis reaches an entity (8) |
| **R11** | Start computation eagerly for a predefined set of entities (9, 10) |
| **R12** | Start computation lazily for entities requested (7, 9, 10) |
| **R13** | Start computation as guided by an analysis (9) |

**INTERMEDIATE REPRESENTATION BASED ON ABSTRACT INTERPRETATION** Chapter 7 presents *TACAI*, an analysis for constructing a three-address-code intermediate representation (IR) based on abstract interpretation. *TACAI* uses abstract interpretation, a major static analysis technique of its own, implemented as a modular analysis in our framework, to compute an intermediate representation. This IR provides additional information that is not directly present in Java Virtual Machine bytecode. This includes information on the definitions and uses of local variables or whether values can never be `null`. This is beneficial for all subsequent analyses, in particular the call-graph analyses of Chapter 8. *TACAI* also provides more precise type information than the static types declared for fields and method return values.

For this purpose, we supplement *TACAI*'s main analysis with two optional sub-analyses, exercising our support for combinations of optimistic and pessimistic analyses (**R5**). These analyses refine the declared types of method return values and fields, respectively. While the main analysis is optimistic (e.g., it considers code dead until it is proven to be reachable), these optional analyses are pessimistic (they refine sound, imprecise declared types to more precise types). If these sub-analyses are not executed (**R2**), the declared types are used as fallback values (**R3**). As the two additional sub-analyses cyclically depend on each other, requirement **R4** is also exercised. This is depicted in Figure II-1 with pessimistic analyses in italics (optimistic analyses are in straight font):

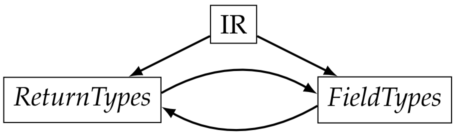


Figure II-1: Sub-analyses of the TACAI Intermediate Representation

**MODULAR ARCHITECTURE FOR CALL-GRAPH CONSTRUCTION** In Chapter 8, we present *Unimocg*, a modular architecture for call-graph construction. As call graphs provide information on which methods call each other, they are the foundation for all interprocedural analyses, such as the ones presented in Chapter 9 and Chapter 10. *Unimocg* provides different call-graph algorithms that can be used interchangeably and thus allows fine-tuning trade-offs between precision and scalability. Moreover, support for different programming language features is achieved by individual sub-analyses. Adding, removing, or exchanging these sub-analyses further allows making fine-tuned trade-offs between soundness and scalability.

Figure II-2 shows the modules contributing to call-graph construction, including their dependency on the intermediate representation of the previous case study (IR): *Type producers* are modules that compute information about the possible runtime types of local variables. This is used by *type iterators* to provide type information to further modules in ways that correspond to traditional call-graph algorithms such as *Class-Hierarchy Analysis (CHA)*, *Rapid Type Analysis (RTA)*, or others. *Call resolvers* then construct the actual call graph by resolving individual calls. Further analyses, such as the immutability analyses of Chapter 9, can also benefit from type information and are thus called *type consumers*.

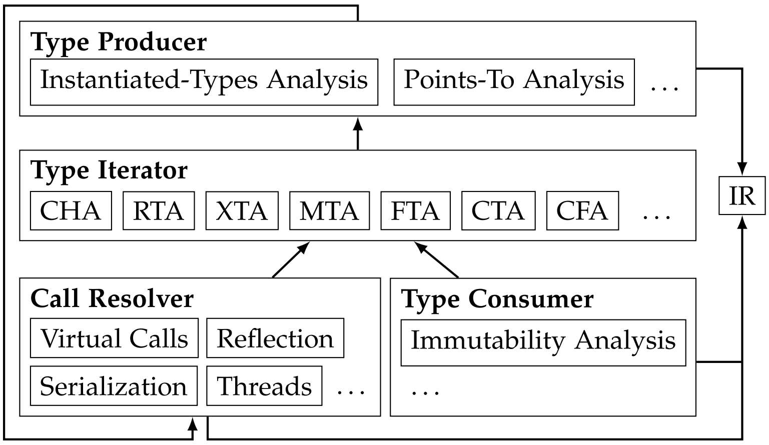


Figure II-2: Sub-analyses of the Unimocg Call-Graph Architecture

*Unimocg* demonstrates many requirements of our framework: it uses set-based lattices to represent sets of types, points-to sets, and sets of callers and callees (**R1**) and default values for unreachable, i.e., dead, methods (**R9**). Dependencies between modules that compute types (type producers) and modules that resolve calls (call resolvers) are cyclic (**R4**). Constructing call graphs requires different activations of the same (**R6**) as well as different (**R7**) analyses to contribute to the same properties such as the callers and callees of a method. Finally, *Unimocg* also schedules the analysis of methods once they are found reachable (**R10**).

MODULAR IMMUTABILITY ANALYSES Chapter 9 presents *CiFi*, a system of immutability analyses. It consists of four interdependent analyses for field assignability and field-, class-, and type immutability. These analyses determine whether fields can be reassigned and whether the values stored in fields and classes can be mutated after their creation. This information is important for reasoning about the correctness and security of programs and is also useful for further analyses such as the purity analysis of Chapter 10.

Compared to the call-graph architecture, results are represented in singleton lattices (**R1**). Dependencies between the analyses are inherently cyclic (**R4**). The analyses can be executed eagerly for all fields, classes, or types (**R11**) as well as lazily only for those required (**R12**) or guided by the class hierarchy (**R13**).

The sub-analyses of *CiFi* depend on the intermediate representation from *TACAI* and the call graphs from the previous case studies. They also depend on an escape analysis. The escape analysis is not detailed here as it is not a contribution of this dissertation. Figure II-3 shows the dependencies, with the new analyses of this case study depicted in bold font:

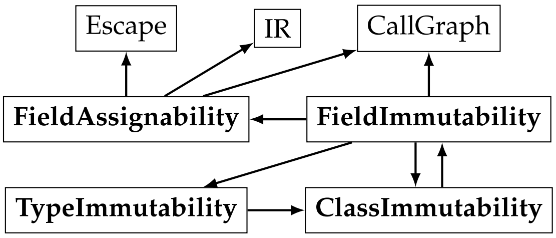


Figure II-3: Dependencies between CiFi Sub-analyses

MODULAR PURITY ANALYSIS As our final case study, Chapter 10 presents *OPIUM*, a family of three analyses for method purity with different precision/scalability trade-offs. The purity property tells whether methods have side effects or behave non-deterministically. This is useful for optimizing code, verifying code correctness in particular in case of concurrency, and for finding bugs and code smells.

As shown in Figure II-4, the analyses of *OPIUM* depend on analyses from all of the previous case studies and two further modules related to escape analysis. Each of *OPIUM*'s analyses has different dependencies. Thus, when selecting a different analysis from *OPIUM*, we want to be able to easily enable, disable, and exchange other analyses as well in a plug-and-play fashion (**R2**). *OPIUM* also makes use of fallback values when some analyses that it depends on are not executed (**R3**) and makes heavy use of precomputed values for native and other methods that are hard to analyze (**R8**).

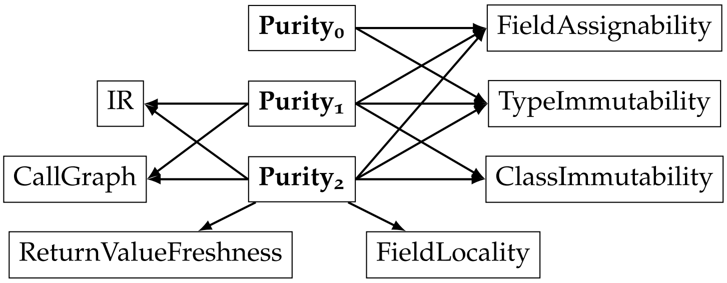


Figure II-4: Dependencies of the OPIUM Purity Analyses

# 7 INTERMEDIATE REPRESENTATION BASED ON ABSTRACT INTERPRETATION

To ease the implementation of static analyses, common static analysis frameworks for Java Bytecode like WALA [111] or Soot [238] transform the stack-based bytecode into a three-address-code (TAC) intermediate representation (IR). TAC representations have a much smaller instruction set than the original bytecode and are designed to be more amenable to static analysis. Transforming Java Bytecode into a TAC representation does not only remove the bytecode's operand stack—which complicates static analysis—but also enables the immediate application of optimizations [65, 239], e.g., constant propagation or dead path removal. As Bodden [26] observed, efficient static analyses with higher precision also yield better scalability in subsequent analyses. This emphasizes the importance of improved precision of precursory analyses. However, the effect that different precision optimizations have on subsequent analyses is still not well understood; in particular for optimizations concerning the IR on which subsequent analyses are built.

In this chapter, we describe *TACAI*, an analysis that uses abstract interpretation to construct a TAC IR that provides additional information, e.g., more precise type information than is available in the Java Bytecode. The precision of this IR can be further increased by employing additional sub-analyses (cf. **R2**). This enables rapid experimentation with different trade-offs between the scalability of the IR generation, the IR's precision, and the impact of this precision on subsequent analyses. In particular, we implemented two sub-analyses that can provide more precise type information for method return values and fields, respectively. These sub-analyses are pessimistic, starting at the sound but possibly imprecise statically declared types and refining them to be more precise (cf. **R5**). Methods can return values loaded from fields, and fields can store values returned from methods. Thus, the additional sub-analyses cyclically depend on each other (**R4**); declared types are used as fallback values (**R3**) if the sub-analyses are not executed.

## 7.1 STATE OF THE ART

Static analysis tools often work on an intermediate representation of bytecode. For instance, *Soot* [238] provides several IRs to operate on: *Baf*, *Jimple*, *Grimple*, and *Shimple*. However, *Jimple* and *Shimple* are the only TAC-based representations.

*Jimple* is generated in 5 steps [239]. At first, a naïve, verbose, and typeless TAC is generated. Step 2 takes the generated TAC and applies several code optimizations, such as constant propagation and dead code elimination. Step 3 splits, step 4 types, and step 5 packs local variables so that they are reused as often as possible. *Shimple* is produced by converting *Jimple* into SSA form.

In contrast to *Jimple* and *Shimple*, *TACAI* performs all optimizations in one step. This simplifies the process and improves scalability as we show in Section 13.4.1. Also, while *Jimple* and *Shimple* always provide a single type bound, *TACAI* can derive more precise union and intersection types and provides information on whether a specific type is an upper type bound or a concrete type.

*TACAI*'s domains for abstract interpretation are configurable. With the most basic domain $TACAI^{L0}$, we get an IR with precision comparable to *Shimple*, but which can be computed faster. Using more advanced domains results in a more precise IR that contains additional information, such as def-use information, or a variable's nullness. In particular, our modular design allows adding more sub-analyses to further improve precision, such as for the types of method return values or fields.

We observe that *Jimple*, the *WALA* framework's IR, and the *TACAI* IR differ w.r.t. the precision of the available type information. We use Listing 7.1 to explain the differences between *Jimple*, *WALA* IR, and *TACAI*. For the method call at Line 3, the three IRs provide type information with different precision: Whereas *WALA* IR only provides `c`'s declared type `Collection`, *Jimple* encodes the upper-type bound `List`, i.e., the common supertype of `ArrayList` and `Vector`. *TACAI* provides a union type of `ArrayList` and `Vector`—the most precise type information if `cond` is unknown. This has repercussions on the analyses using the different IRs. For example, the different type information in Listing 7.1 will lead to different precision call graphs constructed with the simple class hierarchy analysis algorithm [62].

```
1 Collection c;
2 if(cond){ c = new ArrayList(); } else { c = new Vector(); }
3 c.add(null); // Call site
```

Listing 7.1: Precision Example

## 7.2 APPROACH

For *TACAI*, we employ an abstract interpreter as an analysis in *OPAL*. This analysis computes the IR from each method's bytecode and can use further information from the blackboard. This approach has three main properties: first, it enables the derivation of an IR at different precision levels by exchanging the domains underlying the abstract interpretation. Second, all information is computed at the same time in one step. This improves scalability compared to classical compiler frameworks which compute comparable information in a step-wise manner [165]. While performing the abstract interpretation, *OPAL* always computes the method's control-flow graph (CFG) and def-use/use-def information on the fly. Therefore, the CFG and def-use information immediately benefit from better domains. The CFG and def-use information are also accessible through the *TACAI* IR for subsequent analyses to use. Finally, additional sub-analyses can be employed in a plug-and-play manner to improve the precision of the resulting IR.

The abstract interpretation domains used by *TACAI* are defined in *OPAL*. The most basic domains *OPAL* offers are those which operate at the type level and which will lead to an IR that has roughly the same precision as offered by *Soot*'s *Shimple* representation. However, *OPAL* also provides domains that enable constant propagation and constant folding for primitive types. For reference values, domains are available which, for instance, precisely track the nullness, provide must-alias information, compute intersection and union types, or can resolve local `Class.forName` calls. Using more advanced domains enables the computation of an IR that is more precise when compared to typical IRs offered by the other frameworks. Furthermore, it is possible to tailor the precision at a very fine-grained level to a client's needs.

To configure the abstract interpretation, *OPAL* relies on Scala's mixin-composition mechanism. For example, the default configuration that performs all operations at the level of static types is shown in Listing 7.2.

```
trait TypeLevelDomain extends Domain
    with DefaultReferenceValuesBinding
    with DefaultTypeLevelIntegerValues
    with DefaultTypeLevelLongValues
    with TypeLevelLongValuesShiftOperators
    with TypeLevelPrimitiveValuesConversions
    with DefaultTypeLevelFloatValues
    with DefaultTypeLevelDoubleValues
    with TypeLevelFieldAccessInstructions
    with TypeLevelInvokeInstructions
```

Listing 7.2: Example TypeLevelDomain Configuration

The semantics for each set of closely related instructions is implemented by one specialized trait. *OPAL* provides traits for different primitive values, method invocations, field accesses, and reference-value-based operations. The latter trait handles, e.g., `instanceof` checks, casts, and tests against `null`.

The hierarchy of traits defines query methods that can be used by other traits. For example, every implementation that handles reference values has to implement a method to test if a value is `null`, where the result is either `Yes`, `No`, or `Unknown`. The nullness information is used by the domain that handles method calls. That domain checks for each method invocation if the method's receiver object is `null`. If the receiver is known to be `null`, the target method is not invoked, but a `NullPointerException` will be thrown instead. If the receiver cannot be `null`, only the invocation is necessary and no `NullPointerException` can be raised. Finally, if the answer is `Unknown`, different domains either include both the invocation and the possible exception or ignore the exception.

Besides the default configuration (referred to as $TACAI^{L0}$ in the following), two further configurations for a more precise TAC are preconfigured, $TACAI^{L1}$ and $TACAI^{L2}$:

In $TACAI^{L1}$, the `DefaultReferenceValuesBinding` (Line 2) is exchanged for an implementation that computes intersection and union types as well as must-alias information for reference values. Furthermore, special support for calls of the native method `System.arraycopy` is provided which checks for the non-nullness of the arrays and also validates the range that is to be copied. If the validation fails, appropriate exceptions are thrown.[1] The `DefaultTypeLevelIntegerValues` (Line 3) domain is exchanged to perform constant folding and propagation for integer values. The latter is in particular required to identify `if` statements where the conditions evaluate to constant values and are therefore useless.

$TACAI^{L2}$ is the most precise configuration. It builds on top of $TACAI^{L1}$ and additionally performs method inlining for monomorphic calls. This is, e.g., useful for builders (e.g. StringBuilder/StringBuffer) which provide a fluent interface to enable the chaining of calls by always returning the current instance (`return this;`). In such cases, it is then possible to determine that all calls actually happen on the same instance. For that, Scala's stackable trait pattern is used to adapt the handling of method invocations, i.e., an additional trait is configured.

Table 7.1 shows the respective TAC for method `m` (cf. Listing 7.3) for all three levels. $TACAI^{L0}$ basically reflects the source code: The type of the variable `p1` (Line 2) is considered to be `Cloneable` after the cast operation. The code also contains the (useless) reference comparison (Line 7), which compares the reference of the newly created `StringBuffer` (Line 4) with the reference returned by the `append` call (Line 6).

$TACAI^{L1}$ correctly identifies that `p1`'s type is `Serializable with Cloneable`. This intersection type significantly restricts the set of subtypes when compared to the previous version. Additionally, both `p1` and `lv4` are found not to be `null`: `p1` because of the explicit nullness

[1] Special handling is provided for System.arraycopy because it is by far the most widely used native method in the JDK.

Table 7.1: TACAI Representation of Listing 7.3 using Different Domains

| **TACAI$^{L0}$** | **TACAI$^{L1}$** | **TACAI$^{L2}$** |
|---|---|---|
| void m(Serializable) { | void m(Serializable) { | void m(Serializable) { |
| 0: if(p1 ! = null) goto 2 | 0: if(p1 ! = null) goto 2 | 0: if(p1 ! = null) goto 2 |
| 1: return | 1: return | 1: return |
| 2: (Cloneable) p1 | 2: (Cloneable) p1 | 2: (Cloneable) p1 |
| *p1 <: Cloneable* | *p1 <: Serializable* | *p1 <: Serializable* |
| | *& Cloneable* | *& Cloneable* |
| | *p1 not null* | *p1 not null* |
| 3: lv3 = p1.toString() | 3: lv3 = p1.toString() | 3: lv3 = p1.toString() |
| 4: lv4 = new StringBuffer | 4: lv4 = new StringBuffer | 4: lv4 = new StringBuffer |
| | *lv4 not null* | *lv4 not null* |
| 5: lv4.<init>() | 5: lv4.<init>() | 5: lv4.<init>() |
| 6: lv6 = lv4.append(lv3) | 6: lv6 = lv4.append(lv3) | 6: lv4.append(lv3) |
| | | /* value ignored */ |
| 7: if(lv4==lv6) goto 10 | 7: if(lv4==lv6) goto 10 | 7: ; /* NOP */ |
| 8: lv8 = po.e() | 8: lv8 = po.e() | — |
| 9: throw lv8 | 9: throw lv8 | — |
| 10: lva = lv4.toString() | 10: lva = lv4.toString() | 8: lv8 = lv4.toString() |
| 11: po.p(lva) | 11: po.p(lva) | 9: po.p(lv8) |
| 12: return | 12: return | 10: return |
| } | } | } |

check (Line 0), `lv4` because it is freshly allocated (Line 4). That the variables cannot be `null` guarantees that the invocations on `p1` (Line 3) and `lv4` (Lines 6 and 10) will not cause `NullPointerExceptions`.

*TACAI*$^{L1}$ computes must-alias information, i.e., which local variables must point to the same object, intraprocedurally. This is not enough to remove the useless reference comparison in Line 7. To identify that `lv4` and `lv6` must point to the same object requires knowing that the value returned by `append` is the self-reference `this`. By performing inlining in *TACAI*$^{L2}$, this information becomes available and, therefore, the useless comparison can be removed and subsequently, the `if` statement is removed as well as the `throw` statement. A `NOP` statement (*TACAI*$^{L2}$ Line 7) is added because the CFG is not rewritten during the initial transformation, which requires that every basic block contains at least one instruction. It would be straightforward to remove NOPs and update the CFG in a second step if required.

*TACAI* can be extended in a modular way by adding further analysis modules. We developed two such modular sub-analyses to improve the precision of type information for method return values and for fields, respectively. These analyses identify the values that are returned from methods or stored in fields. Consider a method that, according to its signature, has return type `List`, but only ever returns `LinkedLists`.

```
RuntimeException e() { return new RuntimeException(); }
void p(String s) { System.out.println(s); }

void m(Serializable serializable) {
  if(serializable == null) return ;
  Object o = (Cloneable) serializable;
  String s = o.toString();
  StringBuffer sb0 = new StringBuffer();
  StringBuffer sb1 = sb0.append(s);
  if(sb0 != sb1)
    throw e();
  p(sb0.toString());
}
```

Listing 7.3: Java Code Used to Generate *TACAI* IR for Table 7.1

The return type analysis recognizes this and provides the more precise type `LinkedList`.

The two sub-analyses circularly depend on each other (cf. **R4**) as values loaded from a field can be returned from a method, and values returned from a method can be stored in a field. The sub-analyses provide additional information for the main IR generation analysis as shown in Figure 7.1. We show them in italics to indicate they are pessimistic analyses, starting at the sound but possibly imprecise statically declared types and refining them to be more precise (cf. **R5**), while the main IR generation analysis is optimistic. If these analyses are not executed, the declared types are used as fallback values (**R3**).

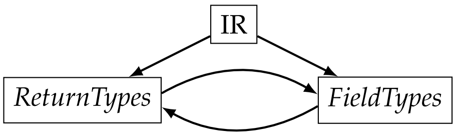


Figure 7.1: Dependencies Between Sub-modules of TACAI IR Generation

## 7.3 Summary

In this chapter, we presented *TACAI*, an analysis for constructing a three-address code intermediate representation. *TACAI* is based on abstract interpretation with configurable abstract domains. It comes with three preconfigured abstract domains which—when used—result in three-address codes with different levels of precision regarding nullness or available type information.

*TACAI* uses *OPAL*'s support for combining optimistic and pessimistic analyses (**R5**) that can be used in a plug-and-play manner (**R2**). Sub-analyses have cyclic dependencies (**R4**). If the sub-analyses are not executed, fallback values can be used (**R3**).

# 8 COLLABORATIVE CALL-GRAPH CONSTRUCTION

Our second case study is concerned with defining an architecture in which individual modules collaborate to compute call graphs. In particular, it uses *OPAL*'s blackboard architecture in order to decouple modules computing type information about local variables from modules that compute the actual call graph edges. The blackboard serves as an intermediary between them.

Sound and precise call graphs are a prerequisite for interprocedural static analysis. Over the past decades, dozens of call-graph algorithms for object-oriented programming languages have been proposed [12, 62, 94, 210, 234]. However, their implementations have inconsistent support for crucial language features, e.g., reflection, serialization, or threads—they often support these features unsoundly, or not at all [186, 187, 225].

Table 8.1 shows the state of affairs for the *WALA* [111] and *Soot* [238] analysis frameworks. We generated the table by reproducing the study of the soundness[1] of call-graph algorithms for JVM-based languages from our previous work [186]. We used the current version of the benchmark[2]—a suite of manually annotated tests—and current versions of *WALA* (1.5.7) and *Soot* (4.3.0) for the algorithms *Class-Hierarchy Analysis* (CHA) [62], *Rapid Type Analysis* (RTA) [12], *Control-Flow Analysis* (CFA) [210], and *Soot*'s default configuration of *SPARK* [144].

While the exact numbers have changed, the overall picture stays the same as reported previously [186]: Language feature support varies significantly not only across frameworks but even across algorithms within the same framework. In general, more precise call-graph algorithms become less sound. The most precise call-graph algorithms (*WALA*'s 0-CFA and *Soot*'s *SPARK*) fail to soundly analyze about half of the test cases. For some features, call-graph algorithms even fail all test cases, an indication that they may not support the feature explicitly. In some cases, less precise call-graph algorithms like CHA and RTA pass tests but do so due to excessive imprecision rather than to actual feature support. Surprisingly, *WALA*'s CHA is not only less precise but apparently also less sound than *WALA*'s RTA. The inconsistent support of language features makes it difficult for users to systematically choose an appropriate call-graph algorithm. One has to know in detail which language features each algorithm supports soundly.

[1] None of these call-graph algorithms are sound in the mathematical sense. In this paper, we refer to the term *soundness* as a lower number of missing edges in call graphs. This is in line with related work [186, 225].

[2] https://github.com/opalj/JCG

Table 8.1: Soundness of Call Graphs for Different JVM Features

| | **WALA** | | | **Soot** | | |
|---|---|---|---|---|---|---|
| **Feature** | **CHA** | **RTA** | **0-CFA** | **CHA** | **RTA** | **SPARK** |
| Non-virtual Calls | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Virtual Calls | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Types | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Static Initializer | ◐ 4/8 | ◐ 7/8 | ◐ 6/8 | ● 8/8 | ● 8/8 | ● 8/8 |
| Java 8 Interfaces | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 | ◐ 6/7 | ● 7/7 |
| Unsafe | ● 7/7 | ● 7/7 | ○ 0/7 | ● 7/7 | ● 7/7 | ○ 0/7 |
| Invokedynamic | ○ 0/16 | ◐ 10/16 | ◐ 10/16 | ◐ 11/16 | ◐ 11/16 | ◐ 11/16 |
| Class.forName | ◐ 2/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Reflection | ◐ 2/16 | ◐ 3/16 | ◐ 6/16 | ◐ 12/16 | ◐ 12/16 | ◐ 10/16 |
| MethodHandle | ◐ 2/9 | ◐ 2/9 | ○ 0/9 | ◐ 3/9 | ◐ 3/9 | ◐ 1/9 |
| Class Loading | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 |
| DynamicProxy | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 |
| JVM Calls | ◐ 2/5 | ◐ 3/5 | ◐ 3/5 | ◐ 4/5 | ◐ 4/5 | ◐ 3/5 |
| Serialization | ◐ 3/14 | ◐ 1/14 | ◐ 1/14 | ◐ 5/14 | ◐ 1/14 | ◐ 1/14 |
| Library Analysis | ◐ 2/5 | ◐ 2/5 | ◐ 1/5 | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 |
| Sign. Polymorph. | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 |
| Java 9+ | ● 2/2 | ◐ 1/2 | ◐ 1/2 | ● 2/2 | ● 2/2 | ● 2/2 |
| Non-Java | ● 2/2 | ● 2/2 | ● 2/2 | ○ 0/2 | ○ 0/2 | ○ 0/2 |
| **Sum** (out of 123) | 51 (41%) | 65 (53%) | 57 (46%) | 81 (66%) | 76 (62%) | 65 (53%) |

Algorithms within each framework are ordered by increasing precision
Soundness: all ●, some ◐, or no ○ test cases passed soundly

In this chapter, we analyze reasons for this observed inconsistency and propose a solution to the problem. In short, the problem is that different call-graph algorithms handle language features in specific ways by making specific use of different kinds of information they have access to. For example, a CFA algorithm can soundly handle more reflection calls because it has access to pointer information. CHA and RTA handle reflection differently because they do not have access to pointer information. As a result, code that handles call resolution for individual language features is coupled to specific call-graph algorithms, which makes it difficult to reuse that code across different call-graph algorithms. Thus, the available resources and priorities of developers of a certain framework determine which features are supported by which algorithm; maintenance is also complicated as features evolve.

We demonstrate that it is possible to implement a variety of call-graph algorithms with consistent handling of language features. Specifically, we introduce *Unimocg* (*UNIfied MOdular Call Graphs*), a novel architecture for modular implementation of call-graph algorithms that decouples the implementation of the following concerns: (1) compu-

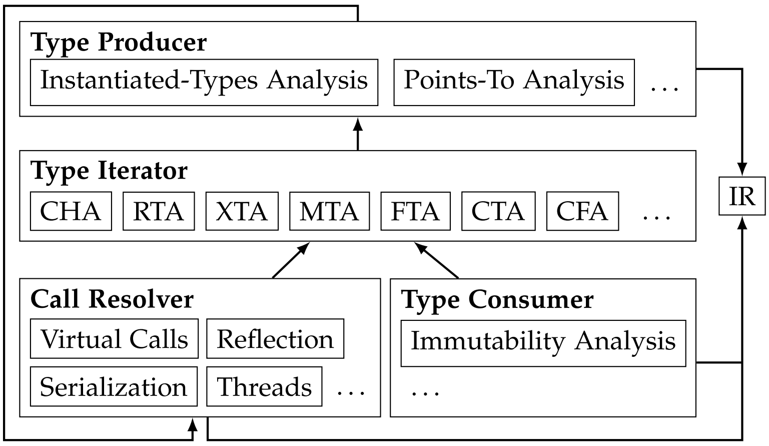


Figure 8.1: Unimocg's Modular Call-Graph Architecture

tation of type information, (2) interpretation of type information, (3) resolution of calls, and (4) analyses that depend on type information. Figure 8.1 overviews the components that handle these concerns in *Unimocg* and their relations.

*Type producers* compute information about the runtime types of variables and fields. *Type iterators* interpret this type information and make it available to call resolvers and type consumers—keeping them decoupled from type producers. *Call resolvers* resolve method calls or calls of language features, such as reflection, serialization, or threads. They query a type iterator for type information about call receivers or arguments of reflective calls; in turn, the information about resolved method calls is used by type producers. *Type consumers* are static analyses that depend on type information but do not contribute to call-graph construction. Without type iterators, type consumers would have to rely on imprecise type information, e.g., provided by static types, to avoid dependence on a specific call-graph algorithm. For instance, our immutability analysis in Chapter 9 originally used imprecise type information from static types and the class hierarchy.

The modular architecture of *Unimocg* enables deriving call graphs with consistent coverage of language features and hence soundness by reusing and combining type producers, type iterator, and call resolvers in a plug-and-play manner. We show that *Unimocg* enables a wide range of call-graph algorithms to share the same support for language features, such as reflection and serialization, thus ensuring consistent soundness, by implementing ten algorithms from different families: CHA [62], RTA [12], the XTA family with MTA, FTA, and CTA [234], as well as *k*-*l*-CFA-based algorithms 0-CFA, 0-1-CFA, 1-0-CFA, and 1-1-CFA [94].

Type consumers also benefit from the modular architecture by reusing precise type information computed for call-graph construction. To showcase this, we implemented an alternative version of the immutability analysis from Chapter 9 as a type consumer in *Unimocg*.

*Unimocg*'s separation of call-graph construction from the computation of type information helps to ensure consistent and improved precision of the immutability analysis when used with more precise call graphs.

*Unimocg* benefits both users and developers of static analyses (call-graph and other analyses). Users can rely on consistent soundness and can systematically choose appropriate algorithms for their respective applications considering only their intuition about the relative precision and scalability of different algorithms. Analysis developers, on the other hand, can easily extend *Unimocg* to support new language features across all available algorithms or add new call-graph and/or other analysis algorithms while retaining all available feature support.

## 8.1 PROBLEM STATEMENT

We analyze two problems with existing call-graph algorithms responsible for the observed soundness inconsistency (Table 8.1).

**PROBLEM 1: COUPLING OF CALL RESOLUTION OF LANGUAGE FEATURES TO BASE CALL-GRAPH ALGORITHMS** Modern programming languages have many features, which are difficult to analyze. Three such Java features are reflection, (de)serialization, and threads. *Reflection* [136] dynamically instantiates classes and calls methods based on runtime strings and types. To resolve reflective calls, a call-graph analysis needs to statically determine strings for the class and method names. It also has to determine receiver and class objects as well as argument types. *(De)serialization* [203] writes or reads Java objects from a stream of bytes, e.g., a file. For (de)serialization, the JVM invokes special methods, e.g., `readResolve`, that must be handled by the call-graph analysis. To resolve a call on a deserialized object, a call-graph analysis also needs to determine the types of objects in the byte stream. *Threads* are started by calling the built-in `Thread.start` method, which leads to the JVM invoking `Thread.run`; hence, it requires specific handling by call-graph algorithms. What complicates the problem further is that these features can be used in combination, e.g., threads may start `Runnable` objects loaded via reflection.

Advanced features induce unique challenges for different call-graph algorithms—hence, each call-graph analysis typically treats them specifically. For example, reflection is easier to handle by call-graph algorithms that have allocation information and deserialization is easier for imprecise algorithms that over-approximate the possible classes deserialized.

Handling language features differently creates coupling between call resolution of language features and the base call-graph algorithms and violates separation of concerns. For example, *WALA*'s RTA algorithm handles static initializers differently from *WALA*'s CHA—with RTA, they should be deemed reachable only for classes actually instantiated.

PROBLEM 2: DIFFERENT TYPE INFORMATION Different call-graph algorithms require different type information to resolve virtual calls. For example, CHA requires only the declared types, RTA additionally requires information about the classes that are instantiated anywhere in the program, and CFA requires the precise information produced by a pointer analysis.

These different representations typically require calls to be resolved specifically for each call-graph algorithm. An implementation for call resolution for CHA is not compatible with RTA and vice-versa because the code to retrieve subtypes from the class hierarchy is different from code to retrieve suitable types from a global type set. This is especially true if the global type set for RTA is constructed on the fly during call-graph construction, i.e., it constantly changes, while type-hierarchy information is constant. The same problem holds for CHA and CFA or RTA and CFA—in fact, for any two call-graph algorithms.

A potential solution is to adopt a single representation for type information for all base algorithms—typically, a points-to representation. For instance, *Soot* and *WALA* do not directly implement RTA, but emulate it by means of a points-to analysis. This strategy is, however, inefficient for algorithms that do not require such an intricate representation. Our evaluation (Section 13.4.2) of RTA in both *Soot* and *WALA* confirms this.

SUMMARY OF PROBLEMS Together, the outlined problems make it difficult to support language features across multiple call-graph algorithms, thus complicating call-graph implementation. Complex code for resolving language features needs to be re-implemented over and over for different base algorithms, and different kinds of type information need to be handled in implementing resolution code. Ultimately, this leads to soundness inconsistencies (Table 8.1). For instance, soundness with regard to static initializers differs between *WALA*'s CHA and RTA in an unexpected way. *WALA*'s RTA algorithm is in many cases more sound than *WALA*'s CHA. This is surprising because the less precise CHA should theoretically be more sound.

OUR SOLUTION IN A NUTSHELL To address the problems, we decouple the call resolution of special language features from the base call-graph algorithm and capture them in independent *call-resolver* modules. To enable this decoupling, we introduce the *type iterator*, an abstraction layer that retrieves and interprets the different type representations produced by different base algorithms (*type producers*). This way, call resolvers for individual language features can be implemented once by using the type iterator as a unified interface to access type information. As a result, individual call resolvers, are decoupled from each other and agnostic of the base call-graph algorithm.

Our approach addresses problem 1 by having the decoupled call resolvers collaborate to resolve calls for different language features. It addresses problem 2 by having type information be kept in the most efficient representation for each individual base algorithm. We show that this approach leads to more consistent soundness. Furthermore, it improves the maintainability of call-graph algorithms, as one can easily add, reuse, or exchange call resolvers to tune precision and performance.

## 8.2 STATE OF THE ART

In this section, we discuss the state of the art regarding call-graph construction. We discuss general-purpose analysis frameworks, like *Soot* or *WALA*, and families of call-graph algorithms, such as XTA and *k*-*l*-CFA. Finally, we discuss the state of the art on supporting complex language features in call-graph algorithms and measuring soundness w.r.t. such features.

### 8.2.1 *Analysis Frameworks*

*Soot* [238] supports different call-graph algorithms, including CHA, RTA, and VTA (Variable Type Analysis [227]). While CHA is implemented directly, other call-graph algorithms like RTA and VTA are emulated in the points-to framework *SPARK* [144]. This allows them to reuse call-resolution code across different algorithms. However, the emulation of less precise algorithms such as RTA comes at the cost of scalability, as we will show in our evaluation (Section 13.4.2).

The *Watson Libraries for Analysis (WALA)* [111] also support different call-graph algorithms like CHA, RTA, and CFA. *WALA* decouples the creation of call graphs from the call resolution for language features with the Java interfaces called *call-graph builder* and *context interpreter*. In particular, a call-graph builder computes a call graph with RTA or CFA precision, whereas a context interpreter resolves calls of built-in language features such as reflection. Crucially, unlike *Unimocg*, *WALA* does not decouple the analysis of type information. For example, the RTA call-graph builder is closely coupled to a points-to analysis to determine the instantiated classes. Also, the RTA call-graph builder implements special handling of the `clone` method and, in doing so, is strongly coupled to an interpreter for that feature. As *context interpreters* are only invoked on explicit call instructions, features such as static initialization that happen regardless of explicit calls cannot be handled. *WALA*'s CHA implementation does not use the call-graph-builder facilities and has its own redundant implementation of some features such as the invocation of static class initializers. WALA's architecture for call-graph construction is not extensively documented and has not been discussed in a scientific publication so far.

*OPAL* previously supported RTA with a high level of soundness [186]. While language features were supported by individual modules, they were not built to interpret different kinds of type information. Thus, they had a fixed level of precision and could not be reused for consistent soundness with other call-graph algorithms, such as CFA. They also used ad-hoc methods of computing local allocation information to improve soundness and precision. A first implementation of our immutability analysis (cf. Chapter 9) also relied on such ad-hoc methods that are not needed with *Unimocg*.

The *Doop* framework [34] similarly supports a family of points-to based call-graph algorithms. Based on Datalog, it includes rule sets for additional language features like reflection. These rule sets are modularly shared between call graphs of different context sensitivity. However, it is unclear whether popular algorithms such as CHA and RTA would be feasible with *Doop*: they would have to be emulated by a points-to analysis, which, as evidenced by *Soot* and *WALA*, is inefficient.

### 8.2.2 *Families of Call-Graph Algorithms*

Class-Hierarchy Analysis [62] (CHA) is the simplest type-based call-graph algorithm, as its call resolution depends solely on the statically declared type of the call's receiver. Bacon and Sweeney's Rapid Type Analysis [12] (RTA) improves over CHA by only considering subtypes that are instantiated by the analyzed program. However, these algorithms solely describe the resolution of standard virtual calls, neglecting other aspects, such as language features like reflection, which additionally affect call-graph construction.

Tip and Palsberg [234] propose a propagation-based call-graph framework, introducing four call-graph algorithms: CTA, FTA, MTA, and XTA. In general, they attribute a call graph's precision to the number of sets used to approximate run-time values of expressions. CTA uses distinct sets for classes, MTA uses distinct sets for classes and fields, FTA uses distinct sets for classes and methods, and XTA uses distinct sets for fields and methods. Therefore, the framework allows instantiating various context-insensitive call-graph algorithms. However, the authors only discuss and evaluate standard virtual-call resolution. It remains unclear whether sharing additional modules to support other language features is generically possible.

Grove and Chambers [94] give a visualization of the relative precision and computation cost of the previously discussed and further call-graph algorithms. Moreover, they present a framework for call-graph algorithms that is parametric in the choice of context sensitivity. They distinguish three *contour selection functions* to allow varying levels of context sensitivity. Here, a contour denotes each context-sensitive version of a procedure. These functions enabled them to extend Shivers' *k*-CFA [209] to the more precise *k*-*l*-CFA algorithm. Thus, the framework allows for a single implementation for a range of points-to-based call-graph algorithms. However, their framework is not applicable to commonly used highly scalable algorithms such as CHA and RTA. Furthermore, forms of context sensitivity are restricted by the signatures of the four *contour key selection functions* for procedures, instance variables, classes, and the environment (the latter of which is only necessary for nested closures). Finally, their framework again only considers standard virtual-call resolution but not how to combine this with additional modules that can support various language features and are necessary for sound call graphs.

### 8.2.3 *Feature Support And Soundness*

In addition to the resolution of virtual method calls, a call graph highly depends on how other aspects, like language features or APIs, are taken into account during call-graph construction. In recent years, researchers have proposed approaches to specifically support language features and APIs such as reflection [150], dynamic proxies [86], serialization [202], or new language instructions. For example, Fourtounis et al. [86] discussed how to add support for a new Java Bytecode instruction which provides a new call instruction with user-defined semantics [193].[3] Unfortunately, all of them are presented and evaluated in the context of specific call-graph algorithms, lacking comprehensive discussion on how to generalize the concepts to other call-graph algorithms.

While researching these individual concepts is crucial to obtain sound and precise call graphs, it does not imply that they are implemented in most commonly used call-graph frameworks. Sui et al. [225] compared call graphs generated by *Soot*, *WALA*, and *Doop* and measured their differences in soundness. Finding unsoundness, they investigated its root causes in follow-up work [226]. Comparing the differences between the statically generated call graphs and dynamically recorded context call trees, they find that advanced language features, such as reflection, serialization, or native methods, are significant reasons for unsoundness.

Reif et al. [186, 187] investigated the feature support of various call-graph algorithms from the *Soot*, *WALA*, *Doop*, and *OPAL* frameworks using a hand-crafted test suite. Their test suite consists of handcrafted test cases, each testing whether a particular call-graph algorithm supports a specific Java language feature or API. As a result, they found that even call graphs from the same framework support different feature sets.

[3] The *invokedynamic* bytecode instruction introduced in Java 7 is highly-relevant to call-graph construction.

These studies show the need for modular call-graph construction that supports not only implementing call-graph analyses with different precision and scalability trade-offs but also to implement generic feature support among different families of call-graph algorithms as we do with *Unimocg*.

## 8.3 UNIMOCG MODULAR ARCHITECTURE

We start with an overview of *Unimocg*'s components. We then describe individual components in detail and discuss how they collaborate despite being decoupled.

### 8.3.1 *Architectural Overview*

*Unimocg* consists of four types of components (Figure 8.1): *Type producers* analyze the code to compute the possible runtime types of local variables and fields. A *type iterator* provides a unified view on this information for other components to use. *Call resolvers* use type information through the type iterator to resolve method calls that result from different language features. Finally, *type consumers* are further analyses that use type information but do not resolve calls.

Components are decoupled from each other using interfaces and communicate indirectly via the blackboard; the fixed-point solver integrated therein serves as an intermediary. To bootstrap the process, the blackboard is initialized with a set of entry-point contexts[4], e.g., the analyzed program's main method(s).

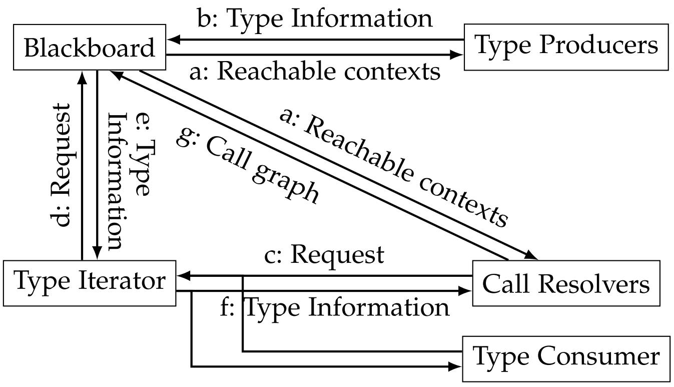


Figure 8.2: Interaction of Components

Figure 8.2 depicts how components interact: Whenever a new reachable context is discovered, the fixed-point solver triggers type producers and call resolvers (a) (cf. **R10**). Type producers process the new reachable context and return new type information to the blackboard (b). Call resolvers analyze the new reachable context and request data

[4] A context is the (context-sensitive) abstraction of a method invocation (cf. [145])

```
interface CallGraphAnalysisModule:
  blackboard: Blackboard := [...]
  typeIterator: TypeIterator := [...]
  fun analyze(ctx: Context)
```

Listing 8.1: Interface for Type Producers & Call Resolvers

from the type iterator (c); the latter forwards the request to the blackboard (d), which returns to the type iterator whatever type information is currently available (e). The type iterator interprets the information and forwards the result to call resolvers (f). Also, if step (b) found additional information, the blackboard notifies the type iterator to forward this information to call resolvers and type consumers that requested it earlier (e & f). Finally, call resolvers add new edges to the call graph (g), which may reveal more reachable contexts and the cycle repeats. When no new edges or type information are found anymore, the analysis reached a fixed point and terminates. Note that edges are never changed or removed from the blackboard, thus termination is guaranteed. Methods not found to be reachable get assigned a default value that signifies their unreachability (cf. **R9**).

### 8.3.2 *Type Producers*

*Type producers* compute type information that is required by other components. For instance, a type producer for an RTA call graph calculates which classes the program instantiates, while a type producer for a CFA call graph computes points-to information of local variables.

A call graph may use multiple type producers. For example, we may split the points-to type producer for CFA into multiple modules that handle different language features, e.g., `java.lang.System.arraycopy`. On the other hand, a call graph may also get along without any type producer: for instance, a CHA algorithm can compute type information directly from the class hierarchy without a dedicated type producer.

Type producers represent type information in an algorithm-specific way. For example, an RTA type producer represents its type information as a global type set, while different CFA type producers for different language features represent their information as points-to sets and set union is used to combine their results (cf. **R1**, **R6**). Some algorithms like $k$-$l$-CFA with $l \geq 1$ additionally provide allocation data, while other type producers cannot provide such information. Allocation data may be needed by specific call resolvers and type consumers, e.g., resolving a reflective call of `Method.invoke` requires knowledge about the particular `Method` object involved.

```
fun analyze(ctx: Context):
  for statement in method.statements:
    if statement is Assigment(local, call: Call):
      callTargets := blackboard.get((ctx, call), CallTargets)
      for target in callTargets:
        if target is constructor:
          newObject := PointsTo(ctx, call.programCounter, target.class)
          blackboard.add((ctx, local), PointsTo, newObject)
    [...]
```

Listing 8.2: Points-To Type Producer (Excerpt Showing Points-To Data Creation on Constructor Invocations)

Despite employing algorithm-specific representations, type producers implement a common interface (which they also share with call resolvers). Listing 8.1 shows pseudocode for this interface. Global singletons are used to retrieve `blackboard` and `typeIterator`, the actual analysis is defined in method `analyze`. Different type producers implement `analyze` in specific ways. It is executed once for each context `ctx` that is found reachable and computes the respective type information.

Type producers are agnostic of how calls are resolved. They are triggered by the blackboard for all reachable contexts, regardless of how the latter are computed. We illustrate this in Listing 8.2: It shows an excerpt of the points-to type producer's `analyze` method where points-to objects are created whenever the analysis of the current context method finds a call whose target is a constructor. The type producer uses only the information in the `callTargets` that are retrieved from the blackboard; the implementation is agnostic of the call resolver that found the target. In particular, the constructor invocation could be the result of reflection (e.g., `Class.newInstance`) or of deserialization instead of a direct call.

### 8.3.3 *Type Iterator*

Type iterators implement the iterator pattern [87] to allow retrieving and iterating over information on the possible runtime types of a local variable or a field from the blackboard in a uniform way. Type iterators and type producers go hand in hand: for example, the RTA type iterator requires an RTA instantiated-types analysis to be executed). For CHA, however, no type producer is needed and type iteration happens on the fly.

Despite their close relation, we separate type producers from type iterators for two reasons: First, one can have different type iterators provide different views on the type information of a single producer. For example, we can have different iterators for a single points-to type producer, each for a different context sensitivity. Second, a single type

```
interface TypeIterator[Context]:
  fun foreachType(var: Local, ctx: Context, handleType: Type -> ())
  fun foreachAlloc(var: Local, ctx: Context,
      handleAlloc: (Type, Context, ProgramCounter) -> ())

  fun newContext(method): Context
  fun expandContext(old: Context, callee: Method): Context
  [...]
```

Listing 8.3: Type Iterator Interface

iterator can provide an aggregated view on the information produced by multiple type producers. For example, a single type iterator for some given context sensitivity can aggregate the information of several points-to type producers—a basic one for local variables, fields, etc. and additional ones for advanced language features, e.g., native methods.

We abstract over specific type iterators with a unified interface `TypeIterator`. Listing 8.3 shows the methods that operate on local variables; analogous methods that operate on fields are omitted for brevity. The generic type `Context` specifies the type of context used, e.g., call strings. Methods `foreachType` and `foreachAlloc` iterate over types and allocations for a local variable `var` in a certain context `ctx`. The interface also defines two methods for iterating on incremental updates of the type data, which we omit for brevity. `newContext` returns a new context based on a method and `expandContext` extends an existing context `old` as necessary for context-sensitive analyses like *k*-*l*-CFA. This enables type producers to support different types of context. Call resolvers, on the other hand, are oblivious to the type of context and may only treat it as a method.

*Unimocg* includes type iterators for CHA, RTA, XTA, MTA, FTA, CTA, 0-CFA, 1-0-CFA, 0-1-CFA, and 1-1-CFA. In the following, we discuss three exemplary iterator instances, shown in Listing 8.4, namely for CHA, RTA, and CFA, to show how they retrieve type information from the blackboard and how they make it available to call resolvers and type consumers. We do not discuss the remaining iterators, but *Unimocg* is available under an open-source BSD 2-clause license as part of *OPAL*.

The CHA type iterator does not need data from the blackboard since the class hierarchy is computed a priori. Hence, we see that method `foreachType` simply iterates over all subtypes of the variable's declared type. The RTA iterator resolves the variable's potential types based on which types may be instantiated. It retrieves the global set of instantiated types from the blackboard, then filters this set to only the subtypes of the variable's declared type; finally, it iterates over these types. For the *k*-CFA type iterator, we see that it resolves the

```
 1 class CHATypeIterator extends TypeIterator[MethodContext]:
 2   fun foreachType(local, context, handleType):
 3     for t in var.declaredType.subtypes:
 4       handleType(t)
 5   [...]
 6
 7 class RTATypeIterator extends TypeIterator[MethodContext]:
 8   fun foreachType(local, context, handleType):
 9     for t in blackboard.get(InstantiatedTypes):
10       if t is subtype of local.declaredType:
11         handleType(t)
12   [...]
13
14 class CFATypeIterator(k: Int) extends TypeIterator[Callstring]:
15   fun foreachType(local, context, handleType):
16     allocations := blackboard.get((ctx, local), PointsTo)
17     for a in allocations:
18       handleType(a.type)
19
20   fun foreachAlloc(local, context, handleAlloc):
21     allocations := blackboard.get((ctx, local), PointsTo)
22     for a in allocations:
23       handleAlloc(a.type, a.context, a.programCounter)
24
25   fun newContext(method):
26     List(method)
27
28   fun expandContext(old, callee):
29     old.take(k).prepend(callee)
30   [...]
```

Listing 8.4: Type Iterators

variable's types based on contextual points-to information using the *k*-truncated call context [209] (Line 29). Given such a call context, the methods `foreachType` and `foreachAlloc` retrieve the set of allocation sites from a context-sensitive points-to analysis from the blackboard and iterate over only the respective types, respectively all allocation sites.

### 8.3.4 *Call Resolvers*

*Call resolvers* use information obtained from type iterators to resolve call sites to possible target contexts. Like *type producers*, they implement the interface in Listing 8.1. For illustration, we discuss two call resolvers of different complexity.

Listing 8.5 shows how the call resolver for regular calls uses the type iterator to resolve virtual method calls: After the `analyze` method finds

```
1  fun analyze(ctx: Context):
2    [...]
3    if instruction is virtual call with receiver variable r:
4      typeIterator.foreachType(r, ctx, receiverType -> {
5        callee := resolveCall(call, receiverType)
6        target := typeIterator.expandContext(ctx, callee)
7        callEdge := CallEdge(ctx, call.programCounter, target)
8        dataStore.add((ctx, call), CallTargets, callEdge)
9      })
10   [...]
```

Listing 8.5: Basic Call Resolver (Excerpt Showing Resolution of Virtual Calls)

```
1  fun analyze(ctx: Context):
2    [...]
3    targets := dataStore.get((ctx, call), CallTargets)
4    if ∃t ∈ targets : t.class=Method ∧ t.name="invoke":
5      receivers := getObjects(t.params.first)
6      params := t.params.tail.map(getObjects)
7      method := t.receiver
8      typeIterator.foreachAlloc(method, context, alloc -> {
9        newTargets := findTargets(alloc, receivers, params)
10       for target in newTargets:
11         callEdge := CallEdge(ctx, call.programCounter, target,
               receivers, params)
12         dataStore.add((ctx, call), CallTargets, callEdge)
13     })
14   [...]
```

Listing 8.6: Reflection Call Resolver (Excerpt showing resolution of `Method.invoke`)

a virtual method call `call`, it iterates over all possible runtime types of the receiver object (Line 4). Once the types are known, resolving the call to a callee method (Line 5), creating a target context for the resolved callee (Line 6) and the call edge (Line 7), and adding it to the call graph (Line 8) are standard steps in all call-resolution code.

In Listing 8.6, we show an excerpt of the reflection call resolver's `analyze` method. It is more complex but also directly uses the type iterator: When the reflection resolver finds a call, it checks whether this is a `Method.invoke` call (Lines 3-4). The reflection resolver takes this information from the blackboard, no matter which call resolver found that call edge. For calls to `Method.invoke`, the resolver gathers information about the receiver and parameters of the reflectively invoked method (Lines 5-6); this step (method `getObjects`, which is not shown here) uses the type iterator's `foreachAlloc`. The reflection resolver then iterates over the possible `Method` objects, as they encode which method can be invoked (Line 8). As in regular virtual-call resolution, the final steps are finding the possible target methods (Line 9) and adding a corresponding call edge to the blackboard (Line 12).[5] As Line 8 shows, allocation data is used for resolving reflection. Where type producers do not provide such data, like for CHA or RTA, the type iterator instead iterates over intraprocedural allocation sites from def-use information and signifies if this is incomplete.

Call resolvers are decoupled from each other (and from type producers) via the blackboard. Yet, they collaboratively compute the call graph. The information contained in the call edge in Line 11 on the receiver and the parameters is made available to other call resolvers and type producers through the blackboard. E.g., in Lines 5-7, we get this data from the individual `target` call edge, not from the `call` in the analyzed code. This is important to allow for the resolution of chained indirect invocations. For instance, if the `Method` object represented `Method.invoke` and `receivers` contained further `Method` objects, this chained invocation still can be resolved by the code in Listing 8.6.

Multiple call resolvers that cover different language features, e.g., virtual calls, reflection, threads, or serialization, collaboratively construct the call graph (cf. **R7**). By combining a set of call resolvers, one can configure the soundness of a call-graph algorithm. Individual resolvers are reusable across different algorithms because they only depend on the common interface of type iterators. As a result, it is easy to ensure consistent feature handling across different algorithms.

While we only showed adding call edges in the forward direction here (i.e., from the caller to the callee), our implementation also adds the reverse edges (from the callee to the caller) to support interprocedural backwards analyses. This requires different executions of a call resolver to collaboratively contribute to the set of callers for each method (**R6**).

### 8.3.5 *Type Consumers*

Type information is useful for a range of analyses beyond those concerned with call-graph construction. For instance, to determine the immutability of some field `f`, an immutability analysis may use the types of objects that `f` potentially refers to (cf. Chapter 9). We model such analyses as so-called *type consumers*. They access type information through the type-iterator interface, which decouples them from the call-graph algorithm that produces this information. This allows easily changing the call-graph algorithm without modifying the type-consumer analyses; by doing so, we can fine-tune the precision and scalability of type consumers. As such, they are conceptually the same

[5] Finding target methods uses a simple analysis of constant strings aided by Unimocg providing access to allocation sites. A more sophisticated string analysis can be implemented as a type consumer and used instead for improved soundness and precision.

as call resolvers, but they do not (directly) participate in call-graph construction, so we discuss them separately. As type consumers depend on the *single* type iterator, they consider type information with exactly the precision of the chosen iterator. This ensures that all type consumers operate on a consistent level of soundness, precision, and scalability. In contrast, different parts of a monolithic analysis may use varying levels of hard-coded precision, which hinders systematically exploring precision, soundness, and scalability trade-offs.

## 8.4 summary

We have shown that modular call-graph construction that decouples the computation of types of local variables from the resolution of call targets is sorely needed. This decoupling enables modular composition of different analyses that contribute to both type computation and call resolution, making it possible to model different language features and APIs in individual modules. With individual modules, feature support can be implemented and reasoned about in isolation. This is necessary to facilitate support for a multitude of such features that are relevant to call-graph construction. As a result, users of call graphs can rely on consistent feature support and analysis developers can easily add new algorithms or language features while reusing existing components.

In this chapter, we presented our modular architecture, *Unimocg*, that achieves this decoupling through a unified interface, the type iterator, that can be queried by call-resolver modules to get type information from type-producer modules. This allows all call resolvers to collaborate despite being fully independent of each other. Further analyses that need type information, such as immutability analyses, can benefit from this unified interface as well. With its modular architecture, *Unimocg* already supports ten different call-graph algorithms from vastly different families of algorithms: CHA, RTA, the XTA family including MTA, FTA, and CTA as well as several *k*-*l*-CFA-based algorithms.

While we discussed call-graph construction for JVM-based languages here, similar issues apply to other programming languages as well and *Unimocg*'s architecture is not specific to the JVM but can be used for call graphs in any language.

*Unimocg*'s modular architecture is enabled by *OPAL*'s blackboard architecture, allowing the modules to collaboratively compute the call graph. In particular, *Unimocg* uses *OPAL*'s support for collaboratively adding partial results to individual properties, both from the same (6) and different (7) modules. The modules are triggered for every reachable method (**R10**) and *OPAL* allows using default values for unreachable methods (**R9**).

# 9 MODULAR IMMUTABILITY ANALYSES

Our next case study defines a modular model and static analysis for immutability properties. Immutability is the property of a program element stating that it is unchangeable or not changed after its creation [180]. Whether program elements are immutable is important for program correctness and security. For example, immutable data structures are not prone to race conditions in multi-threaded applications [92, 95, 100]. Immutable values are less prone to security issues, hence recommended by the *Secure Coding Guidelines for Java SE* [174]. Some APIs, like Java's `Map` interface, assume objects, used as keys, are not mutated[1] [173]. Finally, immutability is also a prerequisite for precisely deriving other properties, e.g., method purity (cf. Chapter 10, [109]).

In this chapter, we focus on the immutability of classes and fields. Previous research on immutability has often focused on individual objects and references [31, 109, 183, 237, 247]. However, it has been argued [44, 45] that focusing on classes and fields simplifies the implementation of systems that enforce immutability restrictions[2] and their usage by developers.

We address the following limitations of the state of the art in checking and enforcing class and field immutability.

First, existing approaches address only individual, specific levels of immutability. For instance, with their `final`, resp. `val` annotations, the Java and Scala programming languages support a weak level of immutability called *non-assignability* [109, 180]. Coblenz et al. [44] and Porat et al. [179] deal only with *transitive immutability*, where every value referred to directly or transitively by a *transitively immutable* class or field is immutable. Nelson et al. [167], on the other hand, deal with *non-transitive* immutability of fields, where *non-transitive immutability* only guarantees that the respective field is non-assignable. However, none of the approaches handles both *transitive* and *non-transitive immutability*.

Second, existing approaches do not properly cover common programming patterns, as we elaborate in Section 9.1. Examples of programming patterns that are not properly handled are lazy initialization and generic type parameters often found in collections and collection-like classes (e.g., `java.util.Optional`). With lazy initialization (cf. Listing 9.1), a field is assigned only when it is accessed for the first time. Here, the field cannot just be restricted to assignments

[1] Mutations that do not affect `equals()` comparisons are allowed.

[2] Immutability restrictions can be, e.g., in the form of annotations.

in the class' constructor. In turn, care has to be taken to ensure that really only a single initialization can be performed. Also, it has to be ensured that the field cannot be observed before its initialization, as observing different values before and after initialization contradicts the guarantees that immutability aims to provide.

Third, generic classes require special treatment, too, as their immutability can depend on the immutability of their type parameters. In Listing 9.2, the immutability of class `Generic` depends on the type parameter `T` used for the final field `t`.

```
class C {
  private Object object;
  public synchronized Object getObject() {
    if (object==null)
      object = new Object();
    return object;
} }
```

Listing 9.1: Thread-safe Lazy Initialization Example

```
class Generic<T> {
  private final T t;
  public Gen(T t){ this.t = t; }
}
```

Listing 9.2: Dependently Immutable Class Example

Fourth and finally, there is a lack of a common model that provides unified terminology for different levels of class and field immutability. For instance, *deep* respectively *shallow* immutability are used [180], or just immutability [128] to refer to the same concepts as *(non)-transitive immutability*. Hence, we need a unified model that not only considers trivial cases, like final fields with immutable types but also common programming patterns such as lazy initialization and generic classes.

Our model and analyses address the above limitations. First, we define a model for class and field immutability that incorporates all relevant levels of immutability and precisely defines their meaning and relations, thus establishing a consistent terminology. Second, based on the model, we define *CiFi*, a set of modular, independent, collaborating static analyses to infer the different levels of immutability for classes and fields, including entire class hierarchies. Developers can directly use *CiFi* to reason about the immutability guarantees of their code or employ further analyses that make use of *CiFi*'s results, such as the purity analyses in Chapter 10.

Additionally, we create *CiFi-Bench*, a set of handcrafted test cases annotated with immutability properties. To the best of our knowledge, such a benchmark did not exist yet—*CiFi-Bench* can be used to guide and test other analyses of class and field immutability.

## 9.1 STATE OF THE ART

We survey prior work on different levels of immutability. In lack of an existing consistent terminology, we use the original names for the considered levels. We start with a discussion of class and field immutability, the focus of this chapter, and then give an overview of other kinds of immutability, such as object and reference immutability.

### 9.1.1 *Class and Field Immutability*

Weak levels of immutability enforcement have been part of programming language design for decades [45]. In Scala, fields can be declared with the keyword `val` which corresponds to Java's `final` modifier. These constructs prevent the field from being reassigned but give no guarantee that the object referenced by the field is immutable. With *case classes* in Scala and *Records* [22] introduced in Java 16, these languages also offer classes that store data in fields that, implicitly, cannot be reassigned. However, mutable objects can be assigned to them. To sum up, while the above language features underline the importance of immutability, they enforce only weak guarantees that other authors call *non-assignability* [109, 180].

Potanin et al. [180] introduce the terms *shallow* and *deep immutability* to distinguish between non-assignable fields referring to mutable objects or arrays (*shallow*) and non-assignable fields transitively referring to objects or arrays that cannot be mutated either (*deep*). Listing 9.3 illustrates both cases. The final and thus non-assignable field `s` refers to a `java.util.String` (known to be deeply immutable); thus, `s` is *deeply immutable*. In contrast, the field `iArr` refers to a mutable array that can be mutated outside the class; thus, `iArr` is *shallowly immutable*.

```
public final String s = "string"; // deeply immutable
public final int[] iArr = {42}; // shallowly immutable
```

Listing 9.3: Deep/Shallow Immutability Example

Coblenz et al. [45] use different terms for the same immutability concepts, namely *non-transitive* for *shallow* and *transitive* for *deep*. Their *Glacier* [44] system uses annotations for Java classes and fields, with `@Immutable` enforcing *transitive immutability* and `@MaybeMutable` stating that a field or class is not guaranteed to be transitively immutable. Gordon et al. [92] call transitively immutable fields just *immutable*, whereas Nelson et al. [167] use the term *immutable* for `final` fields, i.e., fields only guaranteed to be non-transitively immutable.

*Glacier* has no direct support for non-assignability or non-transitive immutability, arguing that non-transitive immutability provides only weak guarantees [45]. In order for a class `C` to be `@Immutable` in *Glacier*, (a) all fields of `C` must be transitively immutable and may only be assigned in the class' constructors, and (b) `C` must have only `@Immutable` subclasses. Because of (a), *Glacier* cannot handle cases where fields are assigned outside a constructor, e.g., in lazy initialization. For generic classes annotated as `@Immutable`, *Glacier* enforces that type parameters are instantiated with `@Immutable` types. This is overly conservative, as type parameters do not necessarily influence a class' immutability. Also, it prevents generic immutable classes, such as immutable collections, from being annotated `@Immutable` if they are used to store mutable or non-transitively immutable data.

Porat et al. [179] propose an interprocedural data-flow analysis to detect transitively immutable classes and fields in Java. According to their definition, a field is immutable if its value or referee is not mutated after being assigned in the static initializer or constructor. Like *Glacier*, this restrictive immutability definition cannot handle lazy initialization. A class is said to be *immutable*, if all of its non-static fields are immutable. The approach was implemented and evaluated on the Java Development Kit (JDK) 1.2 (released in 1998); thus, it lacks support for newer features of Java, e.g., generics.

Kjolstad et al. [128] use the term *immutable* for classes that have only *transitively immutable* instance fields. Their refactoring tool *Immutator* transforms mutable classes into *immutable* ones in order to benefit from the guarantees provided by transitive immutability. To ensure that all fields are initialized in the constructor, *Immutator* adds two new constructors: A public one without parameters initializes all fields with a default value and a private one taking an initialization parameter for each field. Immutator then rewrites all methods mutating the transitive state into factory methods. Finally, all client methods are transformed such that they use the factory methods and the newly created immutable objects returned by them. *Immutator* makes transformed classes `final` to prohibit mutable subclasses. Thus, the refactoring is limited to classes without subclasses. With fields made `final`, lazy initialization is not possible. Also, Immutator does not handle generic classes.

**Observation 9.1**

The survey of the state of the art in analyzing class and field immutability reveals that we lack a consistent terminology for class and field immutability. While some authors use *deep* and *shallow* immutability, others use *transitive* and *non-transitive* immutability. Still others use *immutable* with different meanings.

**Observation 9.2**

None of the existing approaches can simultaneously handle both non-transitive and transitive immutability. Also, none of the presented approaches can recognize lazy initialization *and* properly handle immutability of generic classes.

**Observation 9.3**

Each approach focuses on a fixed composition of immutability flavors, e.g., class and field immutability, and a single level—most often transitive immutability—and it is not possible for client analyses to get information for other immutability flavors or levels.

### 9.1.2 *Object and Reference Immutability*

Haack et al. [95] distinguish *observational* and *state-based* immutability. Observational immutability describes that an observer is not able to see any difference in an object at any two points in time after its initialization. State-based immutability describes that the internal state of an object does not change at all. Like in our model, for state-based immutability, the distinction is made between transitive and non-transitive immutability. Haack et al. express their belief that observational immutability is more intuitive, while state-based immutability is better-suited for static analysis. This is in line with our approach, which also considers state-based immutability.

Potanin et al. [180] distinguish between *abstractly immutable* objects that may change their internal representation while preserving their semantics as visible to their clients and *representationally immutable* objects that never change their internal representation. This corresponds to observational and state-based immutability as used by Haack et al.

Zibin et al. [247] enforce transitive immutability of fields that belong to an object's abstract state with their language extension *Immutability Generic Java (IGJ)* that uses Java generics to describe the immutability of a class through an additional type parameter (*Mutable*, *Immutable*, or *ReadOnly*).

*Ownership Immutability Generic Java (OIGJ)* [248] by Zibin et al. uses ownership to enforce object immutability. As only an object's owner can mutate it, it is easy to check for mutations if the owner is known. Leino et al. [138] also use ownership to *freeze* any object at any time during program execution. When an object is frozen, its owner is changed to be the *freezer object*. As that object is not exposed to the rest of the program, and as changing fields requires ownership, the frozen object becomes immutable and cannot be unfrozen again. This applies to objects owned transitively by the frozen object as well.

For references, the *readonly* property has been studied extensively [23, 31, 66, 109, 129, 183, 237, 247]. Tschantz and Ernst use it in the *Javari* type system [237]. Through a *readonly* reference, the referenced object and all transitively referenced objects belonging to the abstract state of the referenced object cannot be mutated, while they may still be mutated through other references. Thus, *readonly* is different from the *transitive immutability* property—the latter requires the referenced object, including all transitively referenced objects, to be immutable through any reference. Additionally, a *romaybe* modifier expresses polymorphic immutability of references, i.e., whether the reference returned by a method is mutable or not depends on the context in which the method is called and whether the object referred to by this reference, also transitively, is mutated or not. That is, a method may return a potentially mutable but not yet escaped (i.e., not yet accessible by other code than the method itself) object as *romaybe*, allowing the caller to treat it as immutable or mutate it. To support lazy initialization, it is possible to manually exclude lazy-initialized fields from the abstract state in *Javari* (cf. [109]). Gordon et al. [92] describe a similar concept to *readonly* but use the term *readable* instead.

Huang et al. use *Javari* as a basis for their type system *ReIm* and their immutability and purity analysis *ReImInfer* [109]. However, they use *polyread* instead of *romaybe*. Additionally, while *Javari*'s `readonly` modifier refers to the abstract state, here `readonly` applies to the concrete state of the referenced object, i.e., it includes all fields and referenced objects.

Milanova and Dong [163] build upon *ReIm* to infer and check object immutability by combining a reference immutability analysis with escape analysis. They consider transitive immutability, enforcing that no transitively referenced values, objects, or arrays of an `immutable` object are mutated. They also address delayed object initialization with their `endorse` modifier for statements. This results in the analysis ignoring the statement's effects on immutability, which is, e.g., necessary to support circular initialization. With the *unstrict* block, Gordon et al. [92] present a similar approach.

Quinonez et al. [183] find it "tedious and error-prone" to manually add modifiers like `readonly` to existing code bases. They propose to infer them automatically with *Javarifier*, which can also infer *Javari*'s modifiers for arrays and their values as well as for the type parameters of generic classes.

Boyland [32] cautioned against adding *readonly* to the Java language because its transitive rule would be too restrictive, while it cannot prevent harmful observational exposure, i.e., the state of a mutable field can be seen via a readonly reference while it can be modified through another reference. This leads to problems, e.g., in multithreading contexts or when a client expects a non-mutable object. Our proposed model is in line with Boyland and considers the immutability of an entire class rather than the immutability through a given reference. This avoids harmful observational exposure because a transitively immutable class has only transitively immutable instances.

## 9.2 MODEL

We present our unified model of immutability properties for fields, classes, and types along with their order and relations structured in lattices as required by our framework (cf. Chapter 3). The properties depend on each other in a modular way, i.e., the definition of one property uses the definition of other properties, but it is not necessary to know the actual analysis implementations that compute the used properties. We exemplify the properties with Java code snippets, but the model can be used for any object-oriented language.

### 9.2.1 *Field Assignability*

Potanin et al. [180] define assignability to indicate whether a static or instance field is or can be reassigned after it is initialized. We extend on that, defining several levels of assignability which we elaborate on below. Their order is illustrated in the singleton lattice in Figure 9.1 (cf. **R1**, support for different types of lattices).

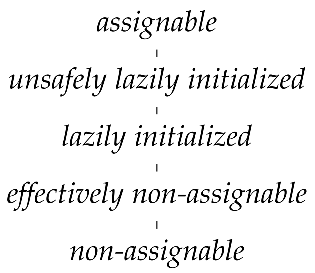


Figure 9.1: Field-Assignability Lattice

#### 9.2.1.1 *(Effectively) Non-Assignable Fields*

Fields can explicitly be enforced to be *non-assignable*, e.g., using Java's `final` keyword, or can be *effectively non-assignable* because there is no reassignment present and none can be added through other code that is not analyzed.

**Definition 9.1.** *A field is* non-assignable *if it is only assigned once and cannot be reassigned.*

**Definition 9.2.** *A field is* effectively non-assignable *if it cannot be observed with different values.*

This distinction allows finding fields that are not yet enforced to be non-assignable but could be made so. Examples for both cases are given in Listing 9.4. The field `imm` (Line 2) is `final` and, thus, it is only assigned once (during the execution of the implicit constructor). As a result, it cannot be reassigned. Similarly, the field `effImm` (Line 3) is initialized only once and is never reassigned again. As `effImm` is declared `private`, no code outside of class `C` can assign to it[3], thus rendering it *effectively non-assignable*.

```
1 class C {
2   private final int imm = 42;
3   private int effImm = 42;
4 }
```

Listing 9.4: (Effectively) Non-Assignable Fields

### 9.2.1.2 *Lazily Initialized Fields*

Lazy initialization is a common pattern used to avoid the cost of computing or storing a value if it is never accessed while performing the computation only once if it is accessed repeatedly. It is often implemented by a field accessible only through a single method that only computes and stores the value if the field still has its default value.

An example of a lazily initialized field was given in Listing 9.1. As the field `object` is `private`, no other code can access the field except through the method `getObject`. This method will initialize the field `object` only if its value is still `null`. As the method is `synchronized`, it is guaranteed that the field is only initialized once, even in the presence of multi-threaded execution. Without the `synchronized` annotation, the field `object` could be assigned to more than once. This happens if concurrent threads each see `object` at its default state (`null`) in Line 4 before any of them performs the assignment in Line 5. In this case, each thread may assign a different instance to `object`, with only the last assignment being persistent. Yet, for programs known to be single-threaded, one can still provide a valuable guarantee. For this reason, we define two properties related to lazy initialization: *lazily initialized* (Definition 9.3) and *unsafely lazily initialized* (Definition 9.4):

**Definition 9.3.** *A field is* lazily initialized *if its lifetime can be divided into two distinct phases: During the first phase, no accesses to the referenced value are made except to check whether the field must be transferred to the second phase. During the second phase, the field is effectively non-assignable.*

**Definition 9.4.** *A field is* unsafely lazily initialized *if, as long as only one thread accesses it, its lifetime can be divided into two distinct phases: During the first phase, no accesses to the referenced value are made except to check whether the field must be transferred to the second phase. During the second phase, the field is effectively non-assignable.*

[3] Except when using reflection, which also applies to `final` fields and is typically ignored in static immutability analysis, see, e.g., [179]

#### 9.2.1.3 *Assignable Fields*

Assignable is the top (least precise) value of the field-assignability lattice:

**Definition 9.5.** *A field is* assignable *if none of the previous definitions apply.*

Figure 9.1 gives the lattice order of the previously defined levels of assignability based on the provided guarantees: while non-assignable fields cannot be assigned outside the constructor, effectively non-assignable fields could be reassigned but provably are not. In turn, (unsafely) lazily initialized fields are reassigned once. However, they can be observed before they are initialized only by the check for the default value.

### 9.2.2 *Field Immutability*

*Field immutability* combines the *assignability* of a given static or instance field `f` with the immutability of `f`'s value. Our lattice for field immutability is shown in Figure 9.2. The top (least precise) value of the field-immutability lattice is *mutable*:

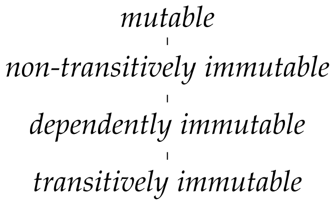


Figure 9.2: Immutability Lattice

**Definition 9.6.** *A field is* mutable *if and only if it is assignable.*

For the purpose of this definition, we treat an *unsafely lazily initialized* field as assignable if it is unknown whether multiple threads might access the field. Fields that are *lazily initialized* in a thread-safe manner are treated as non-assignable.

If a field `f` is not assignable, its immutability depends on the immutability of the values `f` can potentially refer to. Primitive values are always immutable. Array values are mutable (Java has no concept of immutable arrays); hence, the immutability of a field `f` that refers to an array `arr` depends on whether `arr` or any of its elements is actually mutated or could be mutated by unknown code. The same applies to object values, too. However, unlike arrays, for some objects it is possible to infer whether such mutation is actually possible either by inspecting the static type of `f` or by analyzing the potential runtime types of objects that `f` may refer to. Line 2 in Listing 9.5

illustrates a non-assignable field that refers to an array that is not and cannot be mutated. Line 3 illustrates a non-assignable field of type `java.lang.String`—known to be immutable.

```
1 class C {
2   private final int[] iArr = new int[]{ 1, 2, 3, 4 };
3   private final String finalString = "final string";
4 }
```

Listing 9.5: Field Immutability Example

Our immutability lattice distinguishes between *transitively immutable* and *non-transitively immutable* fields:

**Definition 9.7.** *A field is* transitively immutable *if it is not assignable, and no object (or array) that can transitively be reached through the field can ever be mutated.*

**Definition 9.8.** *A field is* non-transitively immutable *if it is not assignable, but objects (or arrays) transitively reachable through the field might be mutated.*

Finally, we define the level *dependently immutable*, which models the effect of generic types on immutability. A field with a generic type `T` (i.e., `private final T t;`) that is not assignable (including *unsafely lazily initialized* only if it is known that only one thread accesses the field) can either be *transitively* or *non-transitively immutable* depending on the concrete runtime type of `T`. Thus, we say that such a field is *dependently immutable*. The property *dependently immutable* is—as generic types are—parameterized over all types that influence the reference's immutability, e.g., above generically typed field `t` is said to be *dependently immutable for* `T`.

**Definition 9.9.** *A field is* dependently immutable *if it is not assignable, and the (transitive) immutability of the referenced object depends on at least one generic type parameter.*

### 9.2.3 *Class and Type Immutability*

To determine whether a field is transitively immutable or not, we need information about class and type immutability. Class immutability takes the same values as field immutability, i.e., the ones given in Figure 9.2 and is defined through field immutability as follows:

**Definition 9.10.** *The immutability of a class is the least upper bound (join) of the immutability of all of its instance fields, respecting specialization of generic types for dependently immutable fields.*

As a corollary, class immutability is the least upper bound (join) of the immutability of all possible instances of that class (because the instance fields' immutability is determined by the immutability of the fields' values, which make up the state of the class' instances). Not all instances of a class necessarily have the same immutability property. The following factors can lead to a more precise immutability of a particular instance in comparison to the immutability of its class:

First, while some instance field `f` of a class `C` may, in general, not be effectively non-assignable, it may provably not be assigned to for a particular instance `o`. This is, e.g., the case, if no method that assigns to `f` ever gets invoked on `o`. Second, during the creation of a particular instance `o` of a generic class, type parameters can be substituted by concrete types. This determines whether dependently immutable fields of `o` are actually transitively or non-transitively immutable. Finally, while the declared type of a field `f` might not be transitively immutable, the concrete object assigned to `f` can be, in which case `f` becomes transitively immutable after assignment. Thus, an instance of a class with fields that are not transitively immutable can still be transitively immutable depending on how the instance is created. This is illustrated in Listing 9.6. Depending on the constructor used, the field `nonTransitive` in Line 2 can be assigned either a `MutableClass` or an `ImmutableClass` instance. While an instance of `C` created with the first constructor is *non-transitively immutable*, one created with the second constructor is *transitively immutable*.

```
1 class C {
2   private final Object nonTransitive;
3   public C(MutableClass mc) { nonTransitive = mc; }
4   public C(ImmutableClass ic) { nonTransitive = ic; }
5 }
```

Listing 9.6: Immutability Dependent on Constructor

It is often useful to determine the immutability at the level of types, e.g., to quickly determine whether a field of a given static type can be transitively immutable. In object-oriented languages, a type is either populated by one class (of the same name as the type) and all of its (potential) subclasses or by an interface (of the same name as the type) and all of its implementing classes. Type immutability is defined through class immutability and also uses the lattice from Figure 9.2.

**Definition 9.11.** *The immutability of a type is the least upper bound of the immutability of all classes populating that type.*

As a corollary, the type of a `final` class has the same immutability as the class. Depending on the analysis scenario, the set of potential subclasses of a non-final class may, on the other hand, not be known completely, e.g., when analyzing an extensible library; in such an *open-world* scenario, the type must conservatively be considered to be mutable [179, 185].

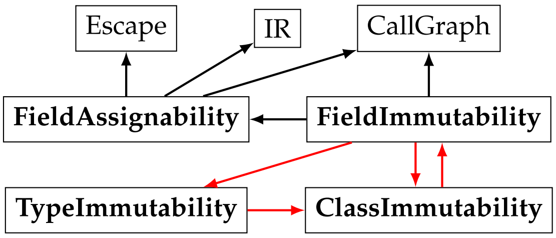


Figure 9.3: Dependencies Between CiFi Sub-analyses

## 9.3 CIFI: ANALYSIS IMPLEMENTATION

*CiFi* implements the presented model as a set of collaborating modular analyses for field assignability and for field-, class-, and type immutability on top of our modular, collaborative framework from Part I. The results of *CiFi* can be used to derive immutability guarantees as introduced in the beginning of this chapter or to reveal their possible absence.

### 9.3.1 *Overall Architecture of CiFi*

Figure 9.3 shows the dependencies between *CiFi*'s analyses (in bold font) and other analyses (in normal font). Field immutability depends on field assignability as well as class and type immutability. The latter depends on class immutability, which, in turn, depends on field immutability. Both the field assignability and field immutability analyses depend on the call graph (cf. Chapter 8). The field-assignability analysis also depends on the intermediate representation (cf. Chapter 7) and on an escape analysis for determining effective non-assignability of fields. These additional analyses are also implemented in *OPAL* and used modularly by *CiFi*. Their interdependencies are not shown here for the sake of simplicity.

As indicated by the red arrows in Figure 9.3, there are circular dependencies between the analyses. Thanks to our blackboard architecture and fixed-point solver, analyses, including cyclically dependent ones, execute in an interleaved way, even if otherwise autonomous (cf. **R4**). Thus, despite the cycle, our analyses can profit from the intermediate results of each other. This simplifies the implementation of *CiFi*'s analyses and enables to easily exchange their implementation or add further analyses for trading off precision, soundiness, and scalability.

### 9.3.2 *Field-Assignability Analysis*

The field-assignability analysis is based on the respective lattice (cf. Figure 9.1) and is a prerequisite for the field-immutability analysis. We omit a discussion of more trivial aspects and focus on handling assignments outside of constructors. Simplified pseudocode for handling lazy initialization is shown in Listing 9.7. The analysis checks whether an initialization is only performed after a default-value check (e.g., `null` in case of objects) has succeeded (Line 2). To determine thread safety, the analysis checks whether the initialization is performed in a synchronized method or a block synchronized on the object holding the field (Line 3). Furthermore, the analysis ensures that even if exceptions are thrown within the lazy initialization method, either the field is guaranteed to be written before its value is returned, or its value is not returned at all (Line 4).

```
1 fun isFieldLazilyInitialized(field):
2   if(initializationNotWithinDefaultValueCheck(field) ||
3     initializationNotSynchronized(field) ||
4     exceptionsLeakUninitializedField(field)) false
5   else true
```

Listing 9.7: Lazy Initialization Recognition (Pseudocode)

Additionally, *CiFi* is able to recognize fields that are assigned only on freshly created instances before they can be accessed elsewhere. For this purpose, *CiFi* checks that the instance does not escape before it is returned. This pattern, illustrated in Listing 9.8, is often used to implement the `clone` method.

```
class C {
  private int i;
  public C clone(){
    C c = new C();
    c.i = i;
    return c;
  }
}
```

Listing 9.8: Clone Pattern

Here, field `i` is not trivially immutable, as it is assigned in method `clone`, i.e., outside a constructor. However, the effect of this assignment is equivalent to one in a constructor, as no outside code can observe the value of `i` before this assignment. This is because the object `c` was freshly created and is only made available to other code at the end of the `clone` method, after the assignment to `i` has already occurred.

### 9.3.3 *Field-Immutability Analysis*

The field-immutability analysis combines results from analyses for field assignability and class and type immutability. Its logic is sketched in Listing 9.9. It always considers assignable fields mutable (Line 2). For all other fields, it checks whether all objects assigned to the field can be identified (Line 3). If this is the case, the join of the respective class immutability properties is computed and used (Line 4), otherwise the immutability of the field's static type is checked (Line 5).

```
 1  def getFieldImmutability(field):
 2    if (isFieldAssignable(field)) Mutable
 3    else if (canAllAssignedObjectsBeIdentified(field))
 4      join(getAssignedObjects(field).map(_.getClassImmutability))
 5    else if (getTypeImmutability(field)==TransitivelyImmutable)
 6      TransitivelyImmutable
 7    else if (hasGenericType(field)) // Dependent Immutablity
 8      if (onlyTransitivelyImmutableTypeParams(field))
 9        TransitivelyImmutable
10      else if (hasANotTransitivelyImmutableTypeParam(field))
11        NonTransitivelyImmutable
12      else DependentlyImmutable
13    else NonTransitivelyImmutable
```

Listing 9.9: Field Immutability Analysis (Pseudocode)

The analysis recognizes dependently immutable fields using information from the field's *Signature* attribute in the Java Bytecode. If the *Signature* attribute contains generic type parameters, the field might be *dependently immutable* (Line 7). In this case, it is checked whether all generic type parameters are instantiated with transitively immutable types (Line 8); if this is the case, the field is transitively immutable. It is next checked whether at least one generic type parameter was instantiated with a type that is non-transitively immutable or mutable (Line 10). In this case, the field is non-transitively immutable. If neither case applies, the field is dependently immutable (Line 12).

This handling of fields with generic types is shown in Listing 9.10. Class `GC` is *dependently immutable for T* because of its generically typed field `genericField` (Line 2). For field `gcTransitive` (Line 6), its generic type parameter is instantiated with the transitively immutable type `java.lang.Integer`. Thus, `gcTransitive` is also transitively immutable. The generic type parameter of field `gcMutable` (Line 7) is instantiated with the (presumably mutable) type `MutableClass`. Thus, `gcMutable` is only non-transitively immutable. Finally, field `gcGeneric` (Line 8) includes another generic type parameter, `T`. Thus, `gcGeneric` is dependently immutable.

```
 1  final class GC<T> {
 2    private final T genericField;
 3    public GC(T value){ this.genericField = value; }
 4  }
 5  class C<T> {
 6    private final GC<Integer> gcTransitive;
 7    private final GC<MutableClass> gcMutable;
 8    private final GC<T> gcGeneric;
 9    [...]
10  }
```

Listing 9.10: Dependent Immutability

### 9.3.4 *Class-Immutability Analysis*

The class-immutability analysis of a class `C` joins the immutability of `C`'s superclass and the immutability of the instance fields declared in `C` (cf. Definition 9.10). Simplified pseudocode of its logic is shown in Listing 9.11. Note that interfaces implemented by `C` do not have to be considered as they cannot contain instance fields. As analyzing `java.lang.String`'s immutability is difficult (e.g., two `Strings` may share the same underlying `char` array and lazily-initialized `hashCode` field), *CiFi* is configured to treat it as transitively immutable. This is in line with other immutability analyses (e.g., [179]) that are configured similarly. Also, we do not consider specialization of generic type parameters.

```
fun getClassImmutability(class):
  classImm = getClassImmutability(getSuperClass(class))
  for field in class.instanceFields
    fieldImm = getFieldImmutability(field)
    if (fieldImm > classImm) classImm = fieldImm
  classImm
```

Listing 9.11: Class Immutability Analysis (Pseudocode)

All analyses of *CiFi* can be executed either eagerly for all fields, classes, or types (cf. **R11**), or lazily only for those that are actually queried (cf. **R12**). The execution of the eager class immutability analysis can be further optimized to follow the structure of the analyzed program's class hierarchy (cf. **R13**): Computing the immutability for a class `C` requires the immutability value of its superclass. Thus, computation is started at the type hierarchy's root (i.e., `java.lang.Object`). Only once the immutability of a class `C` has been analyzed, the computations for its direct subclasses are started.

### 9.3.5 *Type-Immutability Analysis*

The type-immutability analysis' logic is sketched in Listing 9.12. It follows the definition of *type immutability* in Definition 9.11, joining the individual classes' immutability properties while taking into consideration whether the analysis is performed in a closed- or open-world scenario (Line 2).

```
1 fun getTypeImmutability(class):
2   if (isExtensible(class)) return Mutable
3   typeImm = getClassImmutability(class)
4   for subclass in class.allSubclasses
5     classImm = getClassImmutability(subclass)
6     if (classImm > typeImm) typeImm = classImm
7   typeImm
```

Listing 9.12: Type Immutability Analysis (Pseudocode)

While other tools usually support only one (cf. [45]), *CiFi* lets users configure either an open- or closed-world assumption. An open-world assumption means that the analysis assumes classes can be added to all packages except for subpackages of the JDK's `java` package[4] and that all non-final classes can be extended. This is suitable for analyzing libraries. Under a closed-world assumption, the analysis assumes that no classes can be added to existing packages and that existing classes cannot be extended. However, `public` fields and methods are assumed to be accessible. This is a common assumption for applications.

### 9.3.6 *Threats To Soundness*

*CiFi* does not consider any field access by means of reflection, the `sun.misc.Unsafe` class, or native methods calls. Such accesses, potentially anywhere in the program, cannot reliably be linked to specific fields. Consciously omitting such features in order to improve precision is called *soundiness* by Livshits et al. [149]. Doing so is in line with other state-of-the-art static immutability analyses; e.g., Porat et al. [179] do not consider native code and "dynamic effects resulting from reflection" in their class- and field-immutability analyses.

## 9.4 CIFI-BENCH

A ground truth is needed to validate the precision and recall of *CiFi* and other analyses w.r.t. our model. To the best of our knowledge, no benchmark for class and field immutability exists that could be annotated with our model's properties. Thus, we created *CiFi-Bench*, a set of manually annotated test cases for immutability analyses.

*CiFi-Bench*[5] includes a total of more than 470 test cases for all immutability levels defined in our model, organized into the following 13 categories:

- **Assignability:** different (effectively) (non-)assignable fields including clone pattern (counter)examples.
- **General:** simple cases, e.g., static fields, interfaces, trivially transitively immutable and mutable classes.
- **Known Types**
  - **Single:** cases where a single concrete object is assigned to a field, yielding stronger immutability guarantees than is possible to infer from the field's static type.
  - **Multiple:** cases where different objects can be assigned to a field and stronger immutability guarantees can be inferred than possible from the field's static type, including cases where concrete objects or only their types are known.

[4] The classloader usually prohibits adding new classes to these packages.

[5] https://github.com/opalj/CiFi-Benchmark

- **Generic**
  - **Simple:** cases of immutability in combination with generic types, i.e., dependent immutability.
  - **Extended:** advanced uses of generics, such as multiple nested generic types and generic types with bounds.
- **Arrays**
  - **Non-Transitive:** cases with mutable arrays resulting in non-transitively immutable fields.
  - **Transitive:** cases with arrays that cannot be or are not mutated, resulting in transitively immutable fields.
- **Lazy Initialization**
  - **Arrays:** cases of lazy initialization of array-typed fields.
  - **Objects:** cases of thread-safely as well as unsafely lazily initialized fields with object types.
  - **Primitive Types:** lazy initialization without synchronization which can be thread-safe for primitive types.
  - **Scala Lazy val:** an example modeled after Scala 2.12's implementation of `lazy val`[182].
- **String:** a class modeled to resemble some complexities of the class `java.lang.String`, in particular a shared `char` array and a shared, lazily-initialized `hashCode` field.

We annotated fields, classes, and types with the respective assignability and immutability properties as expected with an open-world assumption.

## 9.5 VALIDATION

In this section, we validate our model for immutability analyses by first characterizing its expressiveness and then executing our analyses on real-world applications to verify that the immutability levels we defined are actually applicable.

### 9.5.1 *Characterization of the Model*

To characterize *CiFi*, we use the system that Coblenz et al. [45] proposed for classifying mutability restrictions along several dimensions.

TYPE OF RESTRICTION Our model considers the immutability of fields, classes, and types, not just read-only restrictions on individual references. This provides stronger guarantees for developers [45]. We also consider assignability for fields. We do not consider ownership of objects, which we discuss in Section 9.1.2 together with read-only restrictions.

SCOPE Our model focuses on class immutability, which Coblenz et al. [45] point out to be frequently needed.

TRANSITIVITY We consider both transitive and non-transitive immutability. This enables a more fine-grained view compared to systems surveyed by Coblenz et al.

INITIALIZATION We do not support explicit relaxation of restrictions during initialization. However, our definition of lazy initialization also encompasses delayed initialization, if fields are assigned only once, also enabling cyclic data structures.

ABSTRACT VS. CONCRETE STATE We consider the set of all instance fields of an object, i.e., its *concrete state*. Immutability can also be defined on *abstract state* [237, 247]—excluding non-essential state, e.g., fields used for caching—specified by annotations. Assuming such annotations are available, our model can be applied to abstract state, too.

BACKWARD COMPATIBILITY Our approach performs static analysis to infer immutability; it does not require developers to use specific language features or annotations. Thus, it is soundy regardless of potentially unknown code interfacing with the analyzed software when used with an open-world assumption. If the analyzed program cannot be extended, a closed-world assumption can be used to uncover more immutability.

ENFORCEMENT We infer immutability instead of enforcing it but do provide static guarantees on immutability. Static enforcement may burden developers if they have to annotate all relevant program constructs [45]. This concern does not apply to our automated inference.

POLYMORPHISM Handling mutable and immutable parameters of functions is not applicable to our approach that infers actual immutability instead of enforcing restrictions.

**Observation 9.4**

Our model is more expressive than approaches surveyed by Coblenz et al. [45] without completely covering the described design space. To balance expressiveness with usability [44], we focus on fields, classes, and types, which improves usability [44, 45]. Yet, the model can be easily extended, e.g., with object or reference immutability. Such extensions are well-supported by *CiFi*'s inference approach (no annotations) and its modular architecture, enabling to plug and play analyses depending on what results are considered relevant.

### 9.5.2 *Immutability Prevalence*

To validate that *CiFi* applies to real-world software, we analyzed the following libraries: OpenJDK 1.8.0_292, Google Guava 30.1.1, Eclipse Collections 10.4, Apache Commons Collections 4.4.4., and Scala 2.12.10. We performed the evaluation on a server with two AMD(R) EPYC(R) 7542 CPUs (32 cores / 64 threads each) @ 2.90 GHz and 512 GB RAM. For runtimes, we report the median of 15 executions as runtimes of *OPAL* vary significantly. *CiFi* was run using OpenJDK 1.8.0_292, Scala 2.12.13, and the Scala build tool sbt 1.4.6 with 32 GB of heap memory. In this experiment, we applied an open-world assumption. To ease analysis, *OPAL* replaces `invokedynamic` bytecode instructions with synthetic fields and classes that are also included in the result figures. The number of fields having the respective levels of assignability and the total number of analyzed fields are shown in Table 9.1. Results for the field-, class-, and type-immutability analyses are given in Table 9.2, listing the number of entities with respective levels of immutability, their total count, and the execution time for all analyses combined. Total runtime including preparatory steps, e.g., loading the libraries' files, is given in parentheses. While the numbers for types include interfaces, those for classes do not (interfaces do not contain potentially mutable state).

The results provide empirical evidence that most of the immutability properties defined in Section 9.2 are prevalent in real-world libraries. Even if absolute numbers for dependent immutability appear to be low, one has to consider that generic classes are often widely used collections and thus can have a significant impact. We found several hundreds of safely and unsafely lazily initialized fields in the JDK and some in Guava and Apache Commons Collections but none in Eclipse Collections. We studied the latter library's source code, and indeed Eclipse Collections seems not to use any lazy initialization at all. *CiFi* does not (yet) handle Scala's `lazy val`, but lazy initialization is a prominent feature of the Scala language, too.

We can also see that all libraries have significant quantities of *(effectively) non-assignable* fields; OpenJDK has about 46% of transitively immutable fields, while the other libraries have mostly non-transitively immutable fields. All libraries also have significant shares of (non-)transitively and dependently immutable classes, ranging from 43% to 88%.

To recap, the results presented so far signify the relevance of our immutability model in practice. The properties of the model are all found in real-world code despite the fact that *CiFi* over-approximates the model in some cases and that it was executed with a conservative open-world assumption.

To investigate the effect of the latter, we re-executed *CiFi* on the same libraries with the same setup but with a closed-world assumption.

Table 9.1: Library Results Assignability (Open World)

| Library | ass. | unsafe l. i. | l. i. | eff. non ass. | non ass. | ∑ |
|---|---|---|---|---|---|---|
| OpenJDK | 26 684 | 351 | 189 | 8 615 | 58 115 | 93 954 |
| Eclipse | 2 380 | 0 | 0 | 30 | 11 478 | 13 888 |
| Guava | 656 | 12 | 0 | 4 | 3 215 | 3 887 |
| Apache | 275 | 18 | 0 | 4 | 652 | 949 |
| Scala | 1 249 | 0 | 0 | 4 | 5 373 | 6 626 |

ass. = assignable, l. i. = lazily initialized, eff. = effectively

Table 9.2: Library Results Immutability (Open World)

| Library | Analysis | mutable | non-tra. | dep. | tra. | ∑ | time (s) |
|---|---|---|---|---|---|---|---|
| OpenJDK | Field | 27 035 | 23 004 | 78 | 43 837 | 93 954 | 5.47<br>(6.17) |
| | Class | 12 398 | 4 259 | 27 | 5 393 | 22 077 | |
| | Type | 20 155 | 1 475 | 6 | 3 203 | 24 839 | |
| Eclipse | Field | 2 380 | 7 620 | 142 | 3 746 | 13 888 | 1.56<br>(2.72) |
| | Class | 883 | 4 410 | 61 | 2 247 | 7 601 | |
| | Type | 6 186 | 364 | 41 | 1 057 | 7 648 | |
| Guava | Field | 668 | 1 995 | 35 | 1 189 | 3 887 | 1.06<br>(1.79) |
| | Class | 636 | 785 | 17 | 721 | 2 159 | |
| | Type | 1 697 | 195 | 9 | 391 | 2 292 | |
| Apache | Fields | 293 | 360 | 18 | 278 | 949 | 0.81<br>(1.66) |
| | Class | 262 | 147 | 7 | 69 | 485 | |
| | Type | 424 | 49 | 1 | 50 | 524 | |
| Scala | Field | 1 249 | 3 433 | 344 | 1 600 | 6 626 | 1.57<br>(5.50) |
| | Class | 490 | 2 109 | 74 | 1 150 | 3 823 | |
| | Type | 3 331 | 661 | 60 | 430 | 4 482 | |

dep. = dependently immutable, tra. = transitively immutable

Results for field assignability are given in Table 9.3 and for the other analyses in Table 9.4. Comparing this to the open-world scenario (cf. Tables 9.1 and 9.2), we make the following observations.

First, the number of types with stronger immutability guarantees increases significantly. This is to be expected, as no subclasses can be added in the closed-world scenario. Second, the impact on the number of fields and classes found to exhibit different levels of assignability and immutability is minimal. Differences are most significant for OpenJDK, where 14.2% of formerly assignable and 13.7% of for-

Table 9.3: Library Results Assignability (Closed World)

| Library | ass. | unsafe l. i. | l. i. | eff. non ass. | non ass. | ∑ |
|---|---|---|---|---|---|---|
| OpenJDK | 22 885 | 435 | 198 | 12 321 | 58 115 | 93 954 |
| Eclipse | 2 380 | 0 | 0 | 30 | 11 478 | 13 888 |
| Guava | 598 | 30 | 0 | 44 | 3 215 | 3 887 |
| Apache | 269 | 21 | 0 | 7 | 652 | 949 |
| Scala | 1 249 | 0 | 0 | 4 | 5 373 | 6 626 |

ass. = assignable, l. i. = lazily initialized, eff. = effectively

Table 9.4: Library Results Immutability (Closed World)

| Library | Analysis | mutable | non-tra. | dep. | tra. | ∑ | time (s) |
|---|---|---|---|---|---|---|---|
| OpenJDK | Field | 23 320 | 24 269 | 80 | 46 285 | 93 954 | 7.61<br>(8.58) |
| | Class | 11 573 | 4 741 | 31 | 5 732 | 22 077 | |
| | Type | 13 378 | 5 225 | 35 | 6 201 | 24 839 | |
| Eclipse | Field | 2 380 | 7 595 | 142 | 3 771 | 13 888 | 1.92<br>(3.27) |
| | Class | 883 | 4 397 | 61 | 2 260 | 7 601 | |
| | Type | 950 | 4 552 | 60 | 2 086 | 7 648 | |
| Guava | Field | 628 | 1 931 | 36 | 1 292 | 3 887 | 1.58<br>(2.35) |
| | Class | 633 | 773 | 18 | 735 | 2 159 | |
| | Type | 715 | 848 | 20 | 709 | 2 292 | |
| Apache | Fields | 290 | 353 | 18 | 288 | 949 | 1.17<br>(1.98) |
| | Class | 262 | 142 | 9 | 72 | 485 | |
| | Type | 294 | 146 | 9 | 75 | 524 | |
| Scala | Field | 1 249 | 3 314 | 359 | 1 704 | 6 626 | 2.48<br>(6.53) |
| | Class | 490 | 2 064 | 96 | 1 173 | 3 823 | |
| | Type | 770 | 2 196 | 134 | 1 382 | 4 482 | |

dep. = dependently immutable, tra. = transitively immutable

merly mutable fields and 6.7% of formerly mutable classes exhibit stronger guarantees for assignability or immutability, respectively. The increased number of types with stronger immutability guarantees does not proportionally influence field immutability due to the high percentage of fields with primitive types or type `java.lang.String` (e.g, > 50% in OpenJDK). Third, runtime increased between 23% and 58%. This is because in an open-world scenario, we avoid performing expensive computations, e.g., for extensible types or for protected non-final fields in extensible classes, which are just *mutable*.

**Observation 9.5**

All immutability levels and flavors of our model are prevalent in real-world libraries. This means that (a) the definitions in our model reflect immutability in practice and (b) the versatile inference of *CiFi* is needed to consider fine-grained levels and diverse flavors of immutability.

**Observation 9.6**

Except for type immutability, applying an open-world assumption does not seem to significantly reduce precision while consuming significantly less computation time. Thus, it may be beneficial to use an open-world assumption even if all program code is available. *CiFi* gives users the flexibility to choose between an open- and a closed-world assumption.

## 9.6 SUMMARY

In this chapter, we proposed a comprehensive, fine-grained lattice model for field assignability and for field-, class-, and type immutability. Based on a literature survey, the model unifies the terminology of the research area, which has so far been used inconsistently. Unlike the state of the art, the model distinguishes between these different flavors of immutability and provides levels of immutability to represent relevant aspects such as lazily initialized fields and dependent immutability for generic classes and fields. As we have shown, our model covers a wider range of immutability than previous models. Accompanying this model, we provide *CiFi-Bench*, a handcrafted set of test cases to serve as a ground truth for class- and field-immutability analyses. We introduced *CiFi*, a set of modular analyses for each of the immutability flavors of our model. We used *CiFi-Bench* to showcase *CiFi*'s precision and recall, then used *CiFi* to study the prevalence of immutability in real-world libraries.

*CiFi* makes use of *OPAL*'s blackboard architecture by its reliance on the analyses from the previous case studies. In contrast to these, *CiFi* uses singleton-value-based lattices (cf. **R1**). *CiFi* has inherent cyclic dependencies between its analyses (cf. **R4**) and its analyses can be executed eagerly (**R11**), lazily (**R12**), or guided by the class hierarchy (**R13**).

# 10 MODULAR PURITY ANALYSIS

Our last case study is a family of analyses for identifying methods that are side-effect free and also deterministic, i.e., *pure* like mathematical functions. Such purity analyses for object-oriented programs have been the target of extensive research [108, 196, 222, 223]. Identifying pure methods helps to detect concurrency bugs [84] and to find security-related issues [84, 222]. Formal verification of programs also relies on pure methods for specifying expected behavior [15, 60, 61]. In that case, it is necessary to prove a method's purity to ensure that the formal specifications are correct. The identification of (mostly) pure methods further facilitates program comprehension [17, 69], provides opportunities for code optimizations [43, 137, 246], and supports automated model checking [235]. Pure methods have also become more relevant due to recent trends towards a more functional style of programming, which relies on pure methods [72].

However, no common terminology exists that describes the purity of methods [180]. Some terms (e.g., *pure* or *side-effect free*) are also used inconsistently. Further, all current approaches only compute a subset of possible purity levels and are thus each only suitable for a subset of potential use cases: while use cases like detecting concurrency bugs require *just* the absence of side effects, compiler optimizations and formal verification also need deterministic behavior. Code comprehension benefits from all levels and also weaker ones.

In this chapter, we present *OPIUM*, a family of three purity analyses that compute 13 different levels of purity organized in a fine-grained, unified lattice model. In the model, each lattice element has a well-defined semantics and is put into relation to the purity levels found in the literature. The model is extended by the level *Contextual Purity* which generalizes the so-called *External Purity* [17]. Additionally, we generalize the ability to ignore specific operations in specific contexts [222], e.g., logging in business applications, with the level *Domain-specific Purity*. Our model is sufficiently detailed for all identified use cases. Furthermore, the purity levels *External* and *Contextual Purity* support purity analyses to improve precision by identifying side effects that are limited. With our lattice model, modular purity analyses can reason about each method in isolation. We present a simple intraprocedural, bytecode-based purity analysis and two scalable purity analyses that produce more precise results ($> 4\%$) for real-world code than the state of the art. The purity analyses rely on information computed by all analyses discussed in Chapter 7, Chapter 8, and Chapter 9 and made available via the blackboard.

## 10.1 STATE OF THE ART

Prior research on method purity has introduced inconsistent terminology. In particular, some authors only focus on the absence of side effects and name this *purity*, while others employ a definition that additionally requires deterministic behavior.

We discuss previous research on purity in this section, showing the inconsistent terminology. We present the purity definitions and terminology found in the literature alongside the proposed analyses. We start with approaches that focus on identifying side-effect-free methods—independent of the question of whether the methods are also deterministic or not. After that, we discuss approaches that employ stricter *purity* definitions.

### 10.1.1 *Side-Effect-Free Methods*

One of the most commonly used definitions of method purity is given by Sălcianu and Rinard [223, 224] who characterize a method as pure if *"it does not mutate any object that exists in the pre-state, i.e., the program state right before the method invocation*". This definition does not capture deterministic behavior, only the absence of side effects. Their analysis is combined with a pointer analysis ([106]) and also supports the identification of individual parameters that are *read-only* or *safe*. Read-only characterizes parameters where the method does not mutate any object transitively reachable through the parameter. Safe parameters additionally require that the method does not create new, externally visible pointers to objects reachable through the parameter. To classify the parameters, their analysis is able to determine which memory locations may be modified by a method. The analysis is used in the Korat [30, 157] tool to check the purity of methods that define the behavior of data structures.

The same definition of purity is used by Huang et al. [108, 109]. They propose to extend the Java type system and present a type inference algorithm to annotate references as *read-only*. In the type system, pure methods are those that do not modify global state through static fields and that do not have any parameters inferred as mutable. The authors suggest using the approach to find errors or to do optimizations in concurrent programs.

Pearce [178] uses a similar definition. His *JPure* system checks that methods annotated as *pure* do not modify previously existing program state. It is also capable of inferring purity annotations for code that is not annotated.

Genaim and Spoto [90] also refer to a method as pure, if it does not modify the heap structures reachable from any of its parameters. Their constancy analysis identifies the parameters which are not used to modify the reachable heap. The analysis uses alias relationships between the parameters expressed as boolean formulas.

In the approach proposed by Ierusalimschy and Rodriguez [112] side-effect-free methods are allowed to allocate new objects and return them as long as the pre-state is not modified. They rely on manual annotations that mark methods as side-effect free. Their goal is to extend the type system and to automatically check the respective methods for conformance at compile-time.

In contrast, a dynamic analysis to find *pure* methods is described by Dallmeier [58]. The analysis explicitly deals with multithreading and especially the fact that constructors, while they may assign to fields of the currently initialized object, can be pure. His analysis results are used for *ADABU*, a tool for mining object behavior that requires information about side-effect-free methods for classifying methods as observers and mutators [59].

The definition of side-effect-free methods used by Rountev [196] is stricter than the previous ones. While it does allow allocation of new objects, these may not escape to the caller. The proposed model assumes single-threaded execution as the pre-state of the method could be modified by concurrently executing methods otherwise. They describe two analyses based on RTA ([12]) and a points-to analysis to identify side-effect-free methods in partial programs.

Naumann [166] and Barnett et al. [14, 15] introduce the idea of *observational purity*. Such methods are allowed to have side effects that are not observable by their callers. This definition especially allows for caching (intermediate) results, as is done in memoization. It is only valid in languages without unrestricted pointer arithmetic where noninterference properties can be proven [13]. Methods that are *observationally pure* could be used in program specifications written in, e.g., *ESC/Java* [85] and *JML* [36]. Traditionally, these languages required stronger restrictions (no use of methods in *ESC/Java* at all and only provably pure methods in *JML*). The analysis they propose to determine observational purity is built upon an information-flow analysis [197].

*ESC/Java2* [46] uses side-effect-free methods for specifications but relies on programmer-specified annotations to identify them. The authors also recognize that determinism is required for specifications but do not provide a way of identifying methods that are deterministic and side-effect free.

Table 10.1 summarizes the different approaches to detecting side-effect-free methods and the terminology as it is used by the respective authors.

Table 10.1: Summary of Analyses for Side-Effect-Free Methods

| **Authors** | **Analysis type** | **Purity levels (as named by authors)** |
|---|---|---|
| Sălcianu/ Rinard [223, 224] | pointer analysis | pure |
| Huang et al. [108, 109] | type system extension | pure |
| Pearce [178] | annotations | pure |
| Genaim/Spoto [90] | parameter mutability | pure |
| Ierusalimschy/ Rodriguez [112] | type system extension | side-effect free |
| Dallmeier [58] | dynamic purity analysis | pure |
| Rountev [196] | static purity analysis | side-effect free |
| Barnett et al. [15] | information flow analysis | strongly pure, observationally pure |

Table 10.2: Summary of Analyses for Deterministic Pure Methods

| **Authors** | **Analysis type** | **Purity levels (as named by authors)** |
|---|---|---|
| Finifter et al. [84] | type system/ restricted language | side-effect free, functionally pure |
| Xu et al. [245] | static & dynamic purity analysis | strong, moderate, weak, once-impure |
| Zhao et al. [246] | static purity analysis | pure |
| Benton/Fischer [17] | type-and-effect system | pure, read-only, externally pure, externally read-only |
| Stewart et al. [222] | type system extension | strict, strong, weak, externally pure |

### 10.1.2 *Deterministic Purity*

Aside from not having side effects, deterministic behavior (i.e., producing the same outputs whenever invoked with the same parameters) is a necessary condition for methods to be referentially transparent. This is required for compiler optimizations as well as formal specifications.

The term *functionally pure* for methods that are deterministic and side-effect free is introduced by Finifter et al. [84]. They use a subset of the Java language called *Joe-E* that restricts some features of Java that are non-deterministic or cause side effects, including mutable static state and access to the stack trace of exceptions. Pure methods are automatically thread-safe, as they can never interfere with other threads, and no synchronization is required. This allows for verifying security properties such as the correct behavior of encoding and decoding methods. They also suggest using side-effect-free methods for assertions and specifications. Similar to the concept of Huang et al. [109], methods that only have immutable parameters can never cause side effects or be non-deterministic, so pure methods can be easily identified.

In their work on dynamic purity analysis, Xu et al. [245] define several levels of purity. *Strongly pure* methods must be side-effect free and deterministic and are only allowed to have primitive parameters, thereby excluding reference type parameters completely. They may also not create any new objects or call impure methods, even if the effects of both are not visible to the caller. *Moderately pure* methods are similar in the constraints on their inputs but may create new objects as long as they do not escape the method execution context, similar to the definition of Rountev [196] above. They may also call impure methods if their effects are not observable by the caller. The restriction on reference types is partially lifted in *weakly pure* methods that may access fields of object type parameters. A rather unique concept is *once-impure purity* that allows methods to be impure on their first but not on subsequent invocations. The authors do not detail the use cases, but it seems that they want to support lazy initialization patterns. While their work focuses on dynamic analysis, they also present a static analysis for strong purity. The analysis divides the bytecode instructions executed by a method into impure and pure instructions. For weaker purity levels, some instructions are considered pure only when they are performed on locally allocated, non-escaping objects. The analysis results are used to support automated memoization of method results. This is possible because the results of pure methods are the same when invoked again with the same parameters.

Zhao et al. [246] explore different approaches to find pure methods: automated checking of programmer-supplied annotations and two static analyses based on a method's bytecode. The purity information is then used inside the *Jikes VM* [5] to support further analyses and optimizations such as the elision of method calls. Unnecessary synchronization can be removed as pure methods do not require synchronization with another thread.

*External purity* is introduced by Benton and Fischer [17]. Externally pure methods are allowed to read and modify mutable state but only on the receiver object of a call. Constructors that leak the reference to currently initialized objects are ignored as the authors consider this to be rare. The weaker purity level *externally read-only* allows methods to modify the state of the receiver object as above and to read any mutable state. Benton and Fischer show that a large percentage of methods in object-oriented programs fulfills these conditions using a type-and-effect system [151].

Stewart et al. [222] extend *ReImInfer* [108] by combining previous definitions of purity into five levels of side effects. For two of the proposed levels, *strict purity* (no local variable assignments are allowed) and *strong purity* (no allocations of objects are allowed), no use cases were identified and—as the authors admit—both are of no practical relevance. They also discuss three properties related to *Input*, *Output*, and *Determinism* but treat them as orthogonal to the purity levels. A coherent lattice model is not defined.

A summary of approaches identifying deterministic pure methods and the respective authors' terminology is given in Table 10.2.

## 10.2 MODEL

In the following, we present the unified lattice model by defining the different purity levels and their relations as well as by comparing them to the levels defined in the literature. The presented model is generic and can be used for any object-oriented programming language. Examples in the following sections are in Java.

### 10.2.1 *Purity Levels*

We define several levels of purity to allow for a fine-grained representation of purity and subsume the purity levels discussed above. Different analyses may derive only a subset of these, and clients may process only those purity levels relevant to their use case. We first introduce the level *side-effect free* as it builds the foundation for other levels. We will then proceed to discuss the level *pure* before we finally present weaker levels.

#### 10.2.1.1 *Side-effect Free Methods*

**Definition 10.1.** *A method is* side-effect free *if all object-graph manipulations performed by the method or its callees are visible only to the method while it is executed, i.e., all manipulations are invisible to a method's client.*

Here, the object graph is considered to also include the system's resources, e.g., the file system, network, etc., and therefore methods manipulating these resources are not *side-effect free*.

In contrast to definitions found in prior work, ours is applicable to multi-threaded execution. Other definitions rely on the *pre-state* of a method invocation, or, as Rountev [196], on the object graph before the method invocation. However, with concurrent threads it is unclear what exactly the (static) pre-state of an invocation is. Also, even temporary modifications of the program's state might affect the execution of concurrent threads, constituting a side effect. Thus, our definition explicitly refers to the visibility of side effects, i.e., it does not matter whether the method's pre-state is modified or not.

Our definition also allows *side-effect-free* methods to return newly allocated objects. While this changes the program state by making these objects accessible, this change only becomes visible to the method's immediate caller.

*Side-effect-free* methods may invoke other methods that are not *side-effect free*, if and only if the caused side effects are confined to the calling method, i.e., the side effect is invisible to callers of the *side-effect-free* method. An example is shown in the following Listing 10.1:

```
class CounterValue {
  public static int counter;
  private int value;
  public static CounterValue getCurrentValue(){
    CounterValue current = new CounterValue();
    current.setToCounter();
    return current;
  }
  private void setToCounter(){
    value = counter;
  }
}
```

Listing 10.1: A Side-Effect-Free Method

As the method `setToCounter` modifies its receiver object, it has a side-effect. However, when the method `setToCounter` is invoked by `getCurrentValue`, this side effect is confined to the method's scope, i.e., the side-effect occurs on a newly allocated object that is invisible to other methods than `getCurrentValue` until it finishes execution. Thus, `getCurrentValue` is *side-effect free*.

Another example of a *side-effect-free* method, returning but not modifying the program's state, is `java.lang.System.currentTimeMillis()`.

Further, methods that perform synchronization—except locking of objects inaccessible by other threads—are not *side-effect free* as they change the monitor's state causing changes to an object's object graph. Hence, *side-effect-free* methods are lock-free and cannot cause deadlocks, livelocks, or race conditions, since they do not modify any state that is visible outside their execution scope.

*Side-effect freeness* is the foundation of further purity definitions, and many authors refer to *side-effect-free* methods as *pure* (e.g., [109, 178, 223]). In order not to confuse side-effect freeness with stricter purity—that also requires deterministic behavior—, we refer to non-deterministic methods without side effects as *side-effect free* as in [112].

#### 10.2.1.2 *Pure Methods*

**Definition 10.2.** *A method is* pure *if it is* side-effect free *and additionally produces a structurally equal, deterministic result each time the method is invoked with identical parameters during one execution of a program. Two reference parameters are identical if they reference the same object. The results are structurally equal if the returned object graphs are isomorphic; ignoring aliasing relationships. Primitive parameters are identical and structurally equal if both values are the same.*

This definition is similar to that of *weak purity* as defined in literature [222, 245] and ensures referential transparency, i.e., one can replace the method's invocation with its result. Strong purity, by contrast, disallows allocation of new objects.

```java
final static double PI = 3.1415;
static double getArea(double radius){ return radius*radius*PI; }
```

Listing 10.2: A Pure Method

The method `getArea`, shown in Listing 10.2, is *pure* by our definition. It deterministically computes the circle's area with the given radius and is free from side effects.

Deterministic behavior can be achieved when neither mutable global state is used nor non-deterministic methods are called. Accessing immutable global state like the mathematical constant `PI` in Listing 10.2 does not impede determinism.

In multi-threaded programs, fields of reference-type parameters may change during the execution of the method due to concurrent threads. We therefore restrict accesses to immutable fields. Others have opted for more restrictive approaches, e.g., Xu et al. only allow value-type parameters for their definitions of *strongly* and *moderately* pure methods [245]. Alternatively, only immutable reference-type parameters can be allowed [84].

Our definition of purity only applies to a single execution of a program. Using immutable data that can be initialized to different values in different executions of a program is still considered *pure*. This is generally sufficient for the use cases of purity that we identified, although care has to be taken when used in compiler optimizations: in this case, the invocation of a pure method that returns global data may not be replaced with the result of evaluating the method at compile time (if even possible) but only with an access to the global data.

As variables can be *aliases* of each other, i.e., they can refer to the same object, it is necessary to define when parameters of different invocations are considered equal. We only require two invocations of a *pure* method to return the same result when the parameters passed to the method have the same identity, i.e., refer to the same objects. This restriction allows us to treat comparisons for reference (in)equality as deterministic.

On the other hand, it is not necessary that fields accessible via a parameter have the same identity, as *pure* methods can only access immutable fields and thus changes to mutable fields cannot be observed by *pure* methods.

For a method's result, we require structurally equal object graphs, i.e., the object graphs must be isomorphic but object identity is not considered. Therefore, methods are allowed to return a newly allocated object for every execution. A potentially counterintuitive consequence is that it is unsound to reuse a *pure* method's result instead of reevaluating it when the result may be subject to a reference equality test as in Listing 10.3:

```
Object a = pureMethod();
Object b = pureMethod();
if(a == b) [...]
```

Listing 10.3: A Problematic Reference Equality Check

To decide whether these two invocations can be replaced by either one object or two distinct objects, further analysis beyond purity analysis is necessary.

On the other hand, a method may not be *pure* even if it is guaranteed to always return the same object. Consider `getArray` in Listing 10.4: it always returns the same array, but as the array's contents can be changed between invocations of `getArray`, the resulting object graphs are not guaranteed to be isomorphic. While it is difficult to prove structural equality of the object graph in general, many practical cases can be identified easily. In particular, allocating and deterministically initializing fresh objects will always result in isomorphic object graphs.

```
final static int[] array = new int[]{ 1 };
static int[] getArray(){
  return array;
}
```

Listing 10.4: A Method that is Not Pure

#### 10.2.1.3 *External Purity*

**Definition 10.3.** *A method is* externally pure *(*externally side-effect free*) if its invocation may lead to a modification of its receiver but is* pure *(*side-effect free*) otherwise.*

Methods that call *externally pure* methods can be *pure* or *side-effect free* when the receiver object is confined to their context, i.e., the receiver of the *externally pure* method does not escape the calling method's scope. Furthermore, a client analysis can use this purity level to trivially identify methods that break abstraction boundaries by, e.g., mutating global state. As recognized by Benton and Fischer [17], it is beneficial to identify such methods to improve program comprehension.

```
public class A {
  public int f;
  public void setField(int value){
    f = value;
  }
}
```

Listing 10.5: A Field's Setter that is Externally Pure

Listing 10.5 gives an example of an *externally pure* method. It is deterministic, and its only side effect is the write to the field `f` which belongs to the same receiver object as the method.

Finding methods that are *externally pure* is essentially a specialized form of side-effect analysis. It is cheaper than a full side-effect analysis as it focuses only on the receiver object and does not have to take aliasing into account. Also, the representation of the results is simpler, as being externally pure is just a single property of a method.

*External purity* also applies to methods that use synchronization on the receiver object. This includes methods that have the `synchronized` modifier as well as methods with explicit synchronization on the receiver object. This follows directly from the above definition, as the monitor, that is used to perform the synchronization, is modeled as a property of the object.

In contrast to Benton and Fischer [17], we treat methods reading the receiver's mutable state as *side-effect free* instead of *externally pure*. This enables us to classify methods with calls on non-confined receiver objects as *side-effect free* rather than *impure*. The drawback is that a caller that invokes such methods on a confined receiver object cannot be *pure* anymore.

#### 10.2.1.4 *Contextually Pure*

**Definition 10.4.** *A method is* contextually pure *(*contextually side-effect free*) if its invocation may lead to a modification of at least one of its parameters but is* pure *(*side-effect free*) otherwise.*

We define *contextual purity* as an extension of *external purity*. It captures methods that potentially modify any of their parameters instead of only their receiver. While *contextually pure* methods break abstraction boundaries, they are still less problematic than methods with side effects on (static) global state. Also, they allow identifying more side effects as confined if it is known that none of their actual parameters are visible outside of their caller.

Consider `modifyA` in Listing 10.6. Its only side effect is to modify parameter `a`'s state deterministically. Thus, it is *contextually pure*. An example of a very frequently used *contextually pure* Java method is `System.arraycopy()`. It modifies the given target array and is used in the implementations of many core data structures.

```
public class A {
  public int f;
  public static void modifyA(A a, int value){
    a.f = value;
  }
}
```

Listing 10.6: A Contextually Pure Field Setter

### 10.2.1.5 *Domain-specific Purity*

**Definition 10.5.** *A method is* domain-specific pure *(*domain-specific side-effect free*) if it is* pure *(*side-effect free*) when the effects of certain instructions—belonging to a certain domain—are ignored. Here, two parameterized instructions with different parameterizations are considered different instructions, e.g., two field access instructions on different fields.*

Sometimes, there are methods that have side effects, yet we know that these side effects are harmless for the use case of our analysis. For instance, a method that writes a log file admittedly causes a side effect. If we are only interested in direct effects on the execution of other methods, this kind of side effect can be tolerated. Nevertheless, such methods cannot be called *side-effect free* or even *pure*. Steward et al. [222] previously identified the benefit of not classifying such methods as *impure*. We call them *domain-specific pure* (or *domain-specific side-effect free* if they are non-deterministic) as their classification depends on the use case of the purity analysis. An analysis used, e.g., for compiler optimization may consider different methods as *domain-specific pure* than one used for code comprehension.

```
static final double PI = 3.1415;
static double getArea(double radius){
  System.out.println("called getArea");
  return radius*radius*PI;
}
```

Listing 10.7: A Domain-Specific Pure Method Performing Logging

Consider the method `getArea` in Listing 10.7. Usually, the program's execution will not depend on the output from the `println` statement. Thus, the method can be considered *domain-specific pure*. The possibility of classifying logging as pure is important for the analysis of enterprise applications that include logging in many methods.

Another example of a side effect that is harmless for many use cases is raising exceptions. This is not pure, as newly constructed exceptions contain the current stack trace and that is not deterministic w.r.t. parameters of the method invocation. For example, two method invocations with identical parameters may not lead to exceptions with the same stack trace. However, as the stack trace is usually not inspected by the program, treating it as *domain-specific pure* allows classifying further methods as effectively deterministic.

In contrast to previous work (e.g., [245]), which only treats explicit exceptions and ignores implicit ones (e.g., `NullPointerExceptions` raised by the JVM), we consistently treat both implicit and explicit exceptions.

In this work, we always consider logging and raising exceptions to be *domain-specific pure*. However, our analyses allow the user to configure this based on his use case, following the domain-specific nature of this property.

#### 10.2.1.6 *Orthogonal Purity Properties*

The previously defined purity levels capture four different properties: (1) deterministic and non-deterministic behavior (*pure* vs. *side-effect free*), (2) modification of the receiver object (*external purity*), (3) modification of formal method parameters (*contextual purity*), and (4) non-deterministic or impure actions that may be considered pure in some circumstances (*domain-specific purity*). These properties are orthogonal to each other, i.e., every property combination is possible except for *external* and *contextual purity*, as the latter subsumes the first.

We define the combinations of *externally pure* and *contextual pure* with *domain-specific pure* to apply this concept to methods that modify their receiver or parameters.

**Definition 10.6.** *A method is* domain-specific externally pure *(*domain-specific externally side-effect free*) if it is* externally pure *(*externally side-effect free*) and ignores the effects of specific instructions.*

**Definition 10.7.** *A method is* domain-specific contextually pure *(*domain-specific contextually side-effect free*) if it is* contextually pure *(*contextually side-effect free*) and ignores the effects of specific instructions.*

#### 10.2.1.7 *Impurity*

When a method does not have any of the previously described properties and, therefore, no previous purity level can be assigned, we refer to it as *Impure*.

### 10.2.2 *Purity Lattice*

We arrange the purity levels defined above into a single, unified lattice that captures their relationships. The lattice enables a monotone framework for increasingly precise purity analyses that are able to refine previous analyses' results.

The purity lattice is depicted in Figure 10.1. Its bottom element is *pure*, the strictest purity level. Each step up the lattice loosens exactly one restriction on the method: *Side-effect free* allows for non-deterministic behavior, *domain-specific* allows domain-specific side effects like logging, and *externally* allows modifications on the implicit `this` parameter. *Contextually* further loosens restrictions, allowing modification of all formal method parameters (including `this`). *Impure* is the top value that places no restrictions on the method.

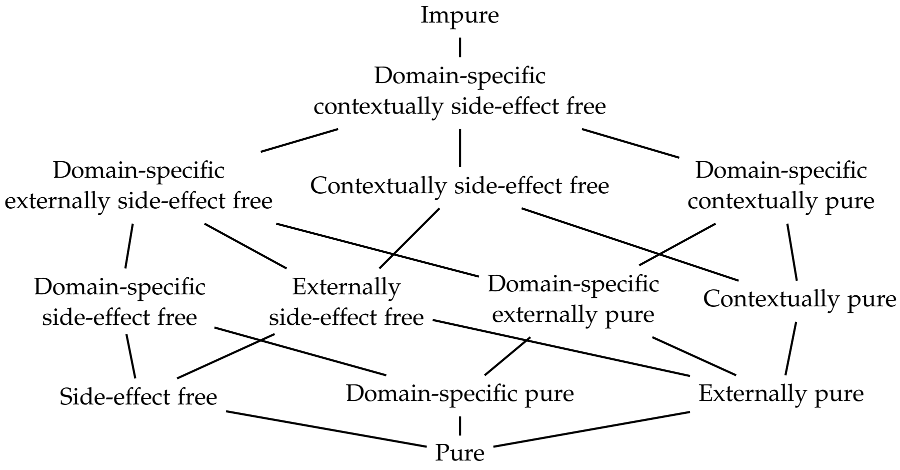

Impure
Domain-specific contextually side-effect free
Domain-specific externally side-effect free
Contextually side-effect free
Domain-specific contextually pure
Domain-specific side-effect free
Externally side-effect free
Domain-specific externally pure
Contextually pure
Side-effect free
Domain-specific pure
Externally pure
Pure

Figure 10.1: The unified lattice for purity information

## 10.3 PURITY ANALYSIS

*OPIUM* consists of three purity analyses with different trade-offs between precision and scalability. Figure 10.2 provides an overview of the analyses and their dependencies. *Purity₀* is a simple, intraprocedural analysis that directly analyses bytecode. It only needs field-assignability and type-immutability information from *CiFi* (Chapter 9). *OPIUM*'s advanced analyses (*Purity₁* and *Purity₂*) use the three-address-code intermediate representation from Chapter 7, the call graphs from Chapter 8, and *CiFi*'s class-immutability information. *OPIUM*'s most precise analysis (*Purity₂*) uses further analyses for the locality of fields and the freshness of return values. These analyses, in turn, use escape information for local variables provided by another analysis.

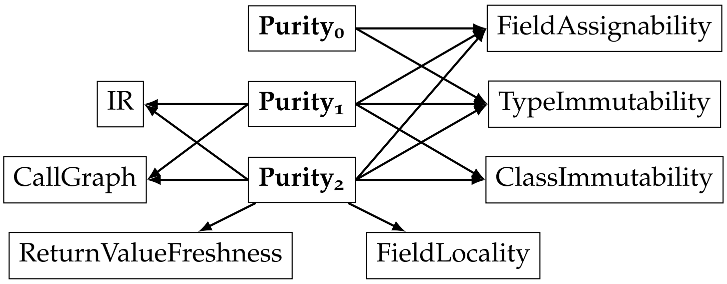


Figure 10.2: Dependencies of the OPIUM Purity Analyses

With *OPIUM* providing three different purity analyses, users can choose either of them and also exchange the analyses that *OPIUM* depends upon in a plug-and-play fashion. This allows them to achieve different trade-offs of precision and scalability (cf. **R2**). If analyses such as the immutability or escape analyses are not executed, *OPIUM* uses sound fallback values instead of their results (cf. **R3**).

*OPIUM* cannot analyze native methods for their purity, but we provide manually determined purity levels for many important native methods in the Java class library (cf. **R8**).

In the following, when we talk about *the analysis*, we are always referring to the two advanced purity analyses that differ only in how they handle locality (cf. Section 10.3.4). We do not describe *Purity₀* in more detail as it is simplistic and not a contribution of this thesis.

### 10.3.1 *Analysis Workflow*

The *OPIUM* purity analysis determines the purity level (cf. Figure 10.1) for each non-abstract method. It works as follows: When we start analyzing a method, we assume that it is *pure*—the lattice's bottom value (i.e., the purity analysis is *optimistic*). We then check each statement of the method if it violates the currently assumed purity level and—if so—reduce the purity level to the next best state. For example, when the currently assumed level is *pure* and an exception object is created, the level is decreased to *side-effect free*. Statements that access fields or other methods query the blackboard for respective properties; if necessary, a dependency on these properties is recorded. After analyzing all statements, the initial result, which consists of the currently assumed purity level and all dependencies, is passed to the blackboard. The information about the dependencies is used by the blackboard to call back the analysis when the required information regarding dependencies is updated. The analysis' continuation function then updates the assumed purity level as well as the set of (still) relevant dependencies and passes both back to the blackboard. The analysis of a method has finished if there are no more dependencies that may affect the assumed purity level; the latter is then the final purity level. If a cycle is found by the blackboard, it is automatically resolved by assuming the current derived purity level for the methods within the cycle. The latter is correct, as it is the best solution that takes all dependencies into account.

Listing 10.8 gives simplified pseudocode for the initial analysis function, ignoring dependency handling and configurable handling of *domain-specific purity*. The symbol ⊔ denotes the *join*, i.e., the least upper bound, of lattice values. The details are explained in the following sections.

```
result = Pure

if method is synchronized:
  result = result $\sqcup$ ExternallyPure

if method.cfg has abnormal return node:
  result = result $\sqcup$ DomainSpecificPure

for inst in method.instructions:
  if inst is GetStatic of field f or GetField of field o.f or ArrayLoad of
      array o:
    if blackboard.get(f, FieldAssignability)[1] $>$ LazilyInitialized $\wedge$
        blackboard.get(o, Locality)[2] $\neq$ Local:
      result = result $\sqcup$ SideEffectFree
  else if inst is PutStatic:
    result = Impure
  else if inst is PutField of field o.f or ArrayStore of array o or
      MonitorEnter on object o or MonitorExit on object o:
    if blackboard.get(o, Locality) $\neq$ Local:
      if o is this of method:
        result = result $\sqcup$ ExternallyPure
      else if o is i$-$th parameter of method:
        result = result $\sqcup$ ContextuallyPure(i)
      else:
        result = Impure
  else if inst is ReturnValue of value o or Throw of exception o:
    if blackboard.get(o.declaredType, TypeImmutability) $>$
        TransitivelyImmutable $\wedge$ blackboard.get(o, Locality) $\neq$ Local:
      result = result $\sqcup$ SideEffectFree
  else if inst is Call c with receiver object r and parameters ps:
    for callee in blackboard.get(c, Callees):
      calleePurity := blackboard.get(callee, Purity)
      if calleePurity is Impure:
        result = Impure
      if calleePurity has modifier SideEffectFree:
        result = result $\sqcup$ SideEffectFree
      if calleePurity has modifier Domain Specific:
        result = result $\sqcup$ DomainSpecificPure
      if calleePurity has modifier Contextual for indices is or External:
        for p := ps[i][3] for i in is:
          if p is this of method:
            result = result $\sqcup$ ExternallyPure
          else if p is i$-$th parameter of method:
            result = result $\sqcup$ ContextuallyPure(i)
          else if blackboard.get(r, Locality) $\neq$ Local:
            result = result $\sqcup$ SideEffectFree
    else if inst is CaughtException of implicit exception:
        result = result $\sqcup$ DomainSpecificPure
```

Listing 10.8: Core logic of the purity analysis

[1] Not for ArrayLoad

[2] Not for GetStatic

[3] p := r instead for External

### 10.3.2 *Effect of Instructions*

In the following, we discuss the instructions that may affect a method's purity. Line numbers refer to Listing 10.8. Instructions related to mathematical operations and constants, type checks and casts, or to the control flow (e.g., `add`, `if`, or `goto`) never affect a method's purity and thus are ignored.

**Field Accesses**: Field reads (`GetField` or `GetStatic`) introduce non-determinism when the values of the accessed fields may change; i.e., when the fields are not (effectively) non-assignable. Therefore, the best possible purity level will be *side-effect free* unless the receiver object of the field access is local to the method (Lines 10 to 12). For example, accesses to a field of a newly created, non-escaping object are ignored. A field's assignability and locality are determined by the respective analyses and queried from the blackboard.

Note that it is sufficient to require that fields that are accessed are (effectively) non-assignable; this ensures that the read value is never changed. While the read value may be a reference to a mutable object or array, that mutability can only be observed if another field/array access is performed on the acquired reference.

Writing to static fields (`PutStatic`) always reduces a method's purity to *impure* (Lines 13 and 14). Instance field writes (`PutField`) affect the purity if the written object is not local (Lines 15 to 22, cf. Section 10.3.4). If the receiver of the field access is the self-reference (`this`) of the method, the best purity level will be *external purity*. If the receiver is a formal parameter of the method, the method is at most *contextually pure*.

**Array Accesses**: We consider arrays as objects where all fields (array entries) are assignable. `ArrayLoad` and `ArrayStore` instructions are hence handled equivalently to instance field accesses, but array entries are always considered assignable (cf. Lines 10 to 12 and 15 to 22). No effort is taken to identify array entries that are effectively non-assignable, i.e., always refer to the same value.

**Synchronization**: We treat explicit acquisitions and releases of monitors (`Monitorenter`, `Monitorexit`) as writes of an (implicit) field `monitor` (cf. Lines 15 to 22).

**Return from Method**: A method that returns a reference (via the `ReturnValue` or `Throw` instructions) is deterministic only if the returned object graph is guaranteed to be structurally equal (cf. Section 9.2) across method invocations. Otherwise, the best possible level is *side-effect free* (Lines 23 to 25). Structural equality is guaranteed if the returned reference is fresh and non-escaping, i.e., *local*, or if the returned object is immutable. The latter property is derived by independent class- and type-immutability analyses (cf. Chapter 9) and once again queried from the blackboard.

**Method Calls**: *Unimocg* from Chapter 8 is used to resolve method calls and call targets are queried from the blackboard (Line 27). The purity of called methods is acquired from the blackboard independent of the underlying instruction (e.g., `StaticCall`, `VirtualCall`, cf. Line 28). It affects the purity of the caller as follows: If the callee is *pure*, *side-effect free*, *domain-specific pure*, or *impure*, the best possible purity level for the caller will be that of the callee (Lines 29 to 34). For an *externally pure* callee, the best possible level for the caller depends on the receiver object of the call (Lines 35 to 42). The caller can be *pure* if the receiver is local and non-escaping. If it is the receiver object of the caller or one of its formal parameters, the caller's purity cannot be better than *externally pure* or *contextually pure*, respectively. In all other cases, the caller must be *impure*. *Contextually pure* callees are handled in the same way, except that all of their parameters, including their receiver object, must be local and non-escaping. Callees that have a combined purity level (cf. Section 10.2.1.6) require the caller to respect all of the combined modifiers individually. For example, the caller of a method that is *domain-specific side-effect free* can be at most *domain-specific* and *side-effect free*.

We manually assigned purity levels to some native methods (e.g., `StrictMath.sqrt`, `System.arraycopy`) to improve the precision of the analysis. Similar to other purity analyses [109, 224], we also use this mechanism to specify the following methods as *pure*: `hashCode`, `equals`, and `compareTo`. However, unlike others, we do not handle `toString` in a special way since assigning correct purity levels for `StringBuilder` and `StringBuffer`'s `append` and `toString` methods suffices to correctly classify most `toString` methods.

In our default configuration of the analyses, calls that are part of logging or console output are treated as *domain-specific*, i.e., *domain-specific pure* is used instead of the callee's actual purity level. This behavior can be configured by the analysis' user, e.g., to include other methods as *domain-specific* for their individual use case.

**Allocation of Objects**: Allocations of objects in general do not influence a method's purity, besides the effect that invoking their constructor has as a method call. The constructor of `Throwable`—the superclass of all exceptions—is, however, *impure* by definition because it invokes `fillInStackTrace` which might be impurely overridden by a subclass. This is a problem as exceptions occur frequently and should not in general result in *impurity*. We thus treat `Throwable`'s constructor as *side-effect free*[4] and individually examine all subclasses of `Throwable` that override `fillInStackTrace`. Unless configured differently by the analysis user, exception constructors are treated as *domain-specific pure*.

[4] The constructor is not *pure* as the included stack trace depends on the currently executed method's context and not only on its parameters

### 10.3.3 *Special Cases*

Besides explicit statements, there are implicit effects on a method's purity that are also checked by the analysis.

**Synchronized Methods**: If a method has the modifier `synchronized`, the purity level will be at most *externally pure*; it is equivalent to writing an (implicit) field `monitor` on the method's receiver object (Lines 3 and 4).

**Implicit Exceptions**: Exceptions may not only be allocated explicitly, the JVM also raises exceptions on several occasions, e.g., a `NullPointerException` is raised when the receiver of a method call is `null`. We recognize these implicit exceptions by examination of the control-flow graph (for exceptions that terminate the method's execution; Lines 6 and 7) as well as `CaughtException` instructions (Lines 43 and 44). Implicit exceptions are again treated as *domain-specific pure* in our default configuration, as we know that the constructors of all exception types that are potentially created by the JVM do not have any side effects.

### 10.3.4 *Locality*

Modifications of objects that are constrained to the scope of a specific method `m` can be ignored when computing the purity of `m`; in such cases, the side effect is confined and the method can still be pure. Such objects are called *Local* and in order to identify them, we used an escape analysis. The analysis derives three different escape values, namely *No Escape*, *Escape Via Return*, and *Global Escape* (cf. [40]). These values form an escape lattice with the following order: $NoEscape < EscapeViaReturn < GlobalEscape$. Objects with a lifetime that is bounded to the scope of the method that created them have the property *No Escape*. If such an object is returned by a method, the escape value is *Escape Via Return*. *Global Escape* is used for all other objects.

The escape property is computed for each intraprocedural definition site, i.e., for all formal method parameters, every object and array allocation site, all calls of methods that return an object, and all field reads. The def-use information provided by *TACAI* (cf. Chapter 7) is used to identify every use site and to propagate the effect of the use site back to the def site. For example, if the only use site is a return statement, the escape state of the respective object is *Escape Via Return*. The analysis is field-insensitive, i.e., whenever an object is stored in a field, it assumes a *Global Escape*.

*OPIUM*'s most precise analysis uses the results of the escape analysis to identify local objects by examining all their definition sites. A reference is considered local if it is: (1) freshly allocated, (2) a *fresh return value* [88], or (3) a *local field* [178]. Whereas the locality for new allocations is trivially computed, we designed separate analyses—both depending on the escape analysis' results—that identify fresh return values and local fields. A method's return value is considered fresh if it is a newly allocated object or the result of a call to a method with a fresh return value. Furthermore, it must only *Escape Via Return*. When retrieving information for a virtual call where the precise type of the receiver is not known, we aggregate the results for all potential call targets. A field is considered local [178] when its owning instance is local, all objects stored in the field are local, and no read value escapes (*No Escape*). The latter analysis requires special handling of `java.lang.Object`'s `clone` method, which creates a shallow copy of an object—including all private fields of the object. Hence, a field might escape even if no `GetField` instruction is present. As a mitigation, the analysis determines whether the object's class overrides `clone` and stores a local object into the field of the new object. For classes that neither override `clone` nor implement the `Cloneable` interface, we can assume that the field is local if the class is final. In the case of a non-final class, the field is also considered to be local if the runtime type is precisely known. Finally, we extended the field and return value analyses to deal with getter methods.

### 10.3.5 *Threats to Soundness*

*OPIUM* considers unanalyzed methods like native methods as *impure* unless explicitly specified differently. Therefore, *OPIUM* is sound even in the presence of local use of reflection or `sun.misc.Unsafe` to invoke native methods. Non-local effects of such calls as well as reflective field writes may, however, lead to unsound results.

## 10.4 VALIDATION

In order to validate whether the purity levels of our model are actually found in real-world applications, we executed our analysis on the Oracle JDK 8 update 151, Scala 2.12.4, and all applications included in XCorpus [67]. The latter contains 76 programs: 70 programs from the Qualitas Corpus [233] and six additional ones that make use of modern dynamic language features of the JVM. We had to exclude *jasperreports-1.1.0* due to invalid bytecode.

The first column of Table 10.3 shows the different purity levels grouped by similar expressiveness. The last line lists the remaining impure methods. Columns two to four show (cf. Table 10.3) the total number of methods as well as the percentage of methods that were derived per purity level for XCorpus, JDK, and Scala.

The analysis shows that *pure* and *side-effect-free* methods are commonplace. In all three cases, ≈ 25% of all methods are in these two categories. Additionally, up to 12.76% of the methods are *domain-specific pure* or *side-effect free*. The analysis also identifies between 1.49%

Table 10.3: Purity Results on XCorpus, JDK, and Scala

| **Program** | **XCorpus** | **JDK** | **Scala** |
|---|---|---|---|
| *Total methods* | 469 727 | 253 282 | 174 881 |
| Pure | 73 701 (15.69%) | 44 378 (17.52%) | 28 502 (16.23%) |
| Domain-specific pure | 9 628 (2.05%) | 8 250 (3.26%) | 6 625 (3.79%) |
| Side-effect free (SEF) | 45 268 (9.64%) | 18 234 (7.20%) | 10 793 (6.17%) |
| Domain-specific side-effect free | 22 056 (4.70%) | 14 717 (5.81%) | 15 690 (8.97%) |
| Externally pure | 19 467 (4.14%) | 4 229 (1.67%) | 2 519 (1.44%) |
| Externally side-effect free | 2 380 (0.51%) | 3 414 (1.35%) | 82 (0.05%) |
| Domain-specific externally pure | 639 (0.14%) | 354 (0.14%) | 11 (0.01%) |
| Domain-specific externally SEF | 2 467 (0.53%) | 1 738 (0.69%) | 2 339 (1.34%) |
| Contextually pure | 4 (0.00%) | 7 (0.00%) | 0 (0.00%) |
| Contextually side-effect free | 7 (0.00%) | 48 (0.02%) | 1 (0.00%) |
| Domain-specific contextually pure | 522 (0.11%) | 547 (0.22%) | 31 (0.02%) |
| Domain-specific contextually SEF | 1 523 (0.32%) | 1 277 (0.50%) | 80 (0.05%) |
| Impure | 292 065 (62.18%) | 156 089 (61.63%) | 108 208 (61.87%) |

and 4.65% of all methods as being *externally pure/side-effect-free*. We only found 82 *externally side-effect-free methods* in Scala but a higher number of *domain-specific externally side-effect-free* methods when compared to Java projects. This deviation is at least partly due to the different treatment of exceptions in these programming languages. In Scala exceptions never need to be caught, and therefore fewer exceptions are explicitly caught. Hence, many more instructions may cause an abnormal return from a method. The same effect, albeit smaller, can be seen for *(domain-specific) side-effect-free* methods in Scala.

Furthermore, we found multiple *contextually pure/side-effect-free* methods. If we also take the *domain-specific* levels into account, we found 1879 methods in the JDK with a *contextual* purity level, which is about 0.74% of all methods. On the other hand, only 0.43% of all methods in the XCorpus and less than 0.1% in Scala have one of the respective levels. Thus, we conclude that the prevalence of these purity levels strongly depends on the analyzed target. In general, the effects of *contextual purity* may improve when better contextual information is provided by supporting analyses.

**Observation 10.1**

All purity levels defined in our model are actually found in real-world applications. In particular, this is also true for our newly defined *contextually pure/side-effect free* levels.

## 10.5 SUMMARY

In this chapter, we proposed a fine-grained unified lattice model for purity, covering all use cases from the literature. We also provide precise definitions for each lattice element to establish a common terminology. Based on this, we implemented *OPIUM*, three different purity analyses, which derive some or all properties of the lattice.

*OPIUM* heavily depends on the previous case studies and the precision and scalability of *OPIUM* can be fine-tuned by choosing either of its three analyses as well as by enabling, disabling, or exchanging the analyses it depends on. *OPAL*'s blackboard architecture supports this in a plug-and-play manner (cf. **R2**) to easily explore the different trade-offs. If analyses that *OPIUM* depends on are not executed, fallback values that replace the missing properties still allow *OPIUM* to be executed soundly (**R3**). Also, *OPIUM* uses pre-computed values for native and hard-to-analyze methods to increase its precision (**R8**).

# Part III

# EVALUATION

In Chapter 1, we defined as the main goal of this thesis that:

> A general framework for modular, collaborative program analysis should allow for complex systems of analyses that offer good soundness, precision, and scalability, including exploring the trade-offs between these qualities.

In this part, we will use the case studies discussed in Part II—and additional minor case studies where appropriate—to evaluate whether *OPAL* succeeds in fulfilling our main goal. In particular, we aim to answer the following three core research questions:

**RQ1** Applicability and Modularity: Does *OPAL* support the modular implementation of a broad range of static analysis kinds with varying requirements?

**RQ2** Precision and Soundness: Can modular, collaborative analyses in *OPAL* improve over the state of the art in terms of precision and soundness?

**RQ3** Scalability, Parallelization, and Scheduling: How does the scalability of modular, collaborative analyses in *OPAL* compare to the state of the art and how can automated parallelization and scheduling impact this?

The individual chapters in this part each focus on answering one of these core research questions. The chapters are as follows:

APPLICABILITY AND MODULARITY Chapter 11 looks at the applicability of *OPAL*'s modular design to implement a broad range of static analyses (**RQ1**). While some existing frameworks support some forms of modularity, they are often limited to specific kinds of analyses: the framework by Johnson et al. [116]), e.g., is built for alias analysis, and *Doop* [34] focuses on points-to and call-graph analyses. We also study whether the modular design enables exploring trade-offs between soundness, precision, and scalability and assess whether the *RA2*, our implementation of *OPAL*'s modularity based on Reactive Async (cf. Chapter 4) is also applicable to different analyses.

> **Finding III.1**
>
> We show that *OPAL*'s design—as well as our alternative implementation *RA2*—is applicable to a broad range of dissimilar static analyses built from modularly composed sub-analyses and that the plug-and-play exchangeability of these sub-analyses benefits both analysis developers and end users.

PRECISION AND SOUNDNESS Chapter 12 considers the precision and soundness of our case studies (**RQ2**). Improved precision and soundness over state-of-the-art analyses is part of the main goal of this thesis. We thus aim to show that the modular design of *OPAL* does indeed allow analyses to be more precise and sound than respective monolithic state-of-the-art analyses.

**Finding III.2**

We find that each of our case-study analyses improves over respective state-of-the-art analyses in terms of soundness, precision, or both. We also show that our analyses do not trade soundness for precision or vice-versa compared to the state of the art.

SCALABILITY, PARALLELIZATION, AND SCHEDULING Chapter 13 focuses on the runtime performance of our case studies. Decoupled, modular analyses may impose some communication overhead that could impede scalability compared to tightly integrated state-of-the-art analyses. At the same time, improvements in precision and soundness can often come with an increase in runtime as well. We show that *OPAL*'s automated parallelization and support for specialized data structures as well as task scheduling have a significant impact on the runtime performance of analyses.

**Finding III.3**

Overall, we show that our case-study analyses are on par with or even outperform respective state-of-the-art analyses in terms of runtime performance.

# 11 APPLICABILITY AND MODULARITY

In this chapter, we answer **RQ1**, that is, we evaluate whether *OPAL* supports implementing a broad range of different analyses in a modular, collaborative manner. We first take another look at the different case studies and how they pose different requirements for *OPAL* in order to show how *OPAL* enables their modular implementation. We then evaluate how the plug-and-play modularity of *OPAL*'s analyses benefits both end users and analysis developers in exploring trade-offs between soundness, precision, and scalability. Finally, we also consider the applicability of our alternative implementation based on *Reactive Async* (cf. Chapter 4).

## 11.1 SUPPORT FOR VARIOUS ANALYSES

In Part II, we implemented the case studies from Section 2.2 using *OPAL*. We argue that these are representatives of a broad range of different analysis kinds. The first case study represents pessimistic analyses in the context of improving the precision of a three-address code representation (TAC)—it shows how basic analyses can be extended by analyses that are specialized to increase the precision of sub-problems' solutions. The modular call graph of the second case study involves tightly interacting yet decoupled analyses (e.g., points-to and call graph) and demonstrates how one can plug in further modular analyses that handle special cases of Java in order to increase the call graph's soundness. The third and fourth case studies introduced several exchangeable analyses for different high-level properties (immutability, escape information, purity). The individual analyses are relatively simple and can focus on their respective property but by using the results of other analyses, they can be more precise than a corresponding monolithic analysis of medium complexity.

As discussed in Section 2.2, to achieve this modularity, several requirements need to be satisfied (cf. Table 2.1). Chapter 3 explained how *OPAL* supports all of them. Meanwhile, as we argue in Section 2.2.4, no current imperative or declarative framework supports all of these requirements.

We additionally implemented a solver for *interprocedural, finite, distributive subset problems* (IFDS) [190], a well-known general framework for dataflow problems based on graph reachability. Many implementations of IFDS exist, e.g., in *WALA* [111], *Heros* [25], *Flix* [154], or *IFDS-A* [192]).

Similar to other IFDS solvers, e.g., *Heros* [25], users of our solver provide a domain for their dataflow facts and four flow functions that together specify the IFDS problem. The solver starts one computation per pair of method and entry dataflow fact and these tasks need to communicate their results. We chose IFDS as it is a general framework that allows implementing many dataflow analyses and it requires a dissimilar implementation style compared to our other case-study analyses. In particular, it shows *OPAL*'s support for implementing general solvers as individual analyses.

**Observation 11.1**

*OPAL*'s programming model enables the implementation of dissimilar analyses, fostering their modularization into a set of comprehensible, maintainable, and pluggable units. *OPAL* is the only static analysis framework satisfying all requirements from Section 2.2.4.

## 11.2 SUPPORT FOR MODULAR CALL GRAPHS

Besides supporting dissimilar analyses, *OPAL* should also support individual analyses that are highly modularized. This enables breaking down complex analyses into simple sub-analyses, each of which is responsible for one clearly defined task, e.g., handling a single programming language feature.

We validate this using *Unimocg*, our modular architecture for call graphs. In *Unimocg* (cf. Chapter 8), we implemented various *type producers* that define a particular call-graph algorithm, *type iterators* that make type information accessible, and *call resolvers* that handle different language features that affect call graphs. We used these components to derive ten different call-graph algorithms.

The implemented *type producers* are: (i) a global instantiated-types analysis, (ii) a parameterized propagation-based instantiated-types analysis, (iii) a parameterized points-to analysis augmented by type producers for Java APIs that require special handling, including, e.g., `java.lang.System.arraycopy` and `sun.misc.Unsafe`.

The implemented *type iterators* are: (i) CHA and RTA iterators from Listing 8.4, (ii) an iterator that is parameterized to support the propagation-based algorithms XTA, MTA, FTA, and CTA [234], (iii) traits for different context sensitivities and points-to-set representations of *k*-*l*-CFA call graphs.

The implemented *call resolvers* are: (i) a virtual and non-virtual call resolver, (ii) call resolvers for static initializers, reflection, threads, serialization, finalizers, the `doPrivileged` API that is provided by the class `java.security.AccessController` and allows indirect calls, and a number of important native methods from the JDK's class library,

(iii) an alternative call-resolver for reflection that uses information from the dynamic analysis *Tamiflex* [27].

Combining the type producers, type iterators, and call resolvers from above, we derived ten different call-graph algorithms: CHA, RTA, XTA, MTA, FTA, CTA, 0-CFA, 0-1-CFA, 1-0-CFA, and 1-1-CFA. For CHA, we used the CHA iterator without a type producer, as it solely depends on the precomputed class-hierarchy type information. To derive RTA, we combined the RTA iterator with the global instantiated-types type producer. For XTA, MTA, FTA, and CTA, we combined the respective iterators with the parameterized propagation-based instantiated-types type producer. Lastly, to derive 0-CFA, 0-1-CFA, 1-0-CFA, and 1-1-CFA, we combined the respective CFA type iterator traits with the points-to type producers. Importantly, we were able to reuse the same call resolvers across all algorithms although the algorithms produce different type information and different call resolvers require different information.

Due to the abstraction introduced by the common type-iterator interface, we can easily derive a family of call-graph algorithms with varying levels of precision and scalability. By selecting an appropriate type iterator and corresponding type producer(s), users select the precision of type information to be computed, i.e., the precision of the call graph. The selection of call resolvers for language features is orthogonal to the selection of type iterators and corresponding type producers. This makes it easy for analysis developers to add new algorithms (as a combination of type iterator and type producer) or call resolvers for new features and reuse all existing components with them.

**Observation 11.2**

*Unimocg* enables deriving families of different call graphs by composing individual components in a modular way.

**Observation 11.3**

All call resolvers are reusable across all algorithms.

**Observation 11.4**

*OPAL* supports the implementation of highly modularized call graphs from reusable modules.

## 11.3 effects of the exchangeability of analyses

Our approach strictly decouples property kinds from analyses computing them. Thus, it can provide different analyses that compute the same property kind to cover a wide range of precision, sound(i)ness,

Table 11.1: Purity Results for Different Configurations (hsqldb)

| Configuration | #Pure | #SEF | #Other | #Impure | ⏲ Analysis |
|---|---|---|---|---|---|
| PA$_2$/FIA$_1$/E$_1$ | 417 | 482 | 245 | 2 635 | 2.42 s |
| PA$_2$/E$_1$ | 363 | 536 | 245 | 2 635 | 2.40 s |
| PA$_2$/FIA$_1$/E$_0$ | 417 | 481 | 241 | 2 640 | 1.93 s |
| PA$_2$ | 362 | 504 | 225 | 2 688 | 0.98 s |
| PA$_1$/FIA$_1$ | 415 | 431 | 0 | 2 933 | 0.93 s |
| PA$_0$/FIA$_1$ | 104 | 0 | 0 | 3 675 | 0.70 s |
| PA$_0$ | 100 | 0 | 0 | 3 679 | 0.13 s |

and scalability trade-offs. Using two experiments, we examine how this exchangeability fosters rapid probing, thus benefiting the analysis' developer and end user alike: We explore the impact on precision and scalability in one experiment and that on soundness in the second. The analyses were executed on the *DaCapo* benchmark [24]. We only show one application each here for brevity[1].

In our first experiment, we run various configurations of our purity analyses (PA$_0$, PA$_1$, and PA$_2$; cf. Chapter 10) with or without analyses for field immutability (FIA$_1$, cf. Chapter 9) and/or an intra- or interprocedural escape analysis (E$_0$ and E$_1$ respectively) to achieve different precision-scalability trade-offs. No other tool supports similar exchangeability of collaborative purity, immutability, and escape analyses.

Table 11.1 shows the results for the *hsqldb* program. Higher indices indicate more precise analyses. Comparing the least precise analysis PA$_0$ with the most precise PA$_2$/FIA$_1$/E$_1$, we observe a reduction in the number of reported impure methods by ~28% but a runtime slowdown by 18.6x. Some configurations have a significant impact on runtime for almost no gain in precision. As an example, compare the most precise configuration with that where E$_1$ is replaced with the simpler escape analysis E$_0$.

In the second experiment, we evaluate an RTA call graph (cf. Chapter 8) with different supporting modules for different JVM features. Results for *Xalan* are shown in Table 11.2, displaying the modules configured to be executed, the numbers of reachable methods and call edges, and the respective analysis time. Compared to the baseline, RTA with support for preconfigured native methods (RTA_C), reaches 21 more methods and ~200 more call edges. Reflection support (RTA_R) brings over 2 000 more RMs and 16 000 call edges; at the same time, construction time increases by about 15%. Exchanging the reflection module for the Tamiflex (RTA_X) module increases call graph size (and soundness) more but introduces further slowdown. With all modules enabled, we reach 111% more methods and 127%

[1] The full results are available at https://doi.org/10.5281/zenodo.3972736

Table 11.2: Results for Different Call-Graph Modules for Xalan

| **Configuration** | **#Reachable Methods** | **#Edges** | **⏱ Analysis** |
|---|---:|---:|---:|
| RTA_S_T_F_C_X | 12 970 | 106 778 | 13.35 s |
| RTA_C_X | 12 958 | 106 743 | 12.86 s |
| RTA_X | 12 937 | 106 516 | 12.99 s |
| RTA_R | 8 404 | 63 821 | 10.07 s |
| RTA_C | 6 162 | 47 154 | 8.76 s |
| RTA | 6 141 | 46 946 | 8.58 s |

C=Configured native methods; R=Reflection; X=Tamiflex;
S=Serialization; T=Threads; F=Finalizer;

more call edges, at the cost of a 55% increased runtime. Moreover, the data from our full analysis suggests that different modules benefit different projects. Tamiflex impacted *Xalan* and *jython*, reflection *fop*, and serialization *hsqldb*. Thus, which modules are more relevant than others may differ between different programs, and it may be worth investigating trade-offs even at the level of individual projects. Note though, that while some configurations discover more methods and edges than others, they may discover different sets of methods and edges. A configuration is only guaranteed to be strictly more sound if it uses a strict superset of modules.

Overall, both experiments confirm that *OPAL* maintains exchangeability benefits from Datalog-based analyses while generalizing these results to a broader range of lattices.

**Observation 11.5**

*OPAL* facilitates systematic investigation of different configurations, supporting users and developers in finding the best trade-off between precision, sound(i)ness, and scalability.

## 11.4 Implementation Based on Reactive Async

In order to assess the applicability of *RA2*, our implementation of *OPAL*'s core concepts based on the Reactive Async framework (cf. Chapter 4), we take another look at our simple purity analysis from Section 4.5. The implementation of this analysis, including the definition of the lattice and an execution harness, takes just about 80 lines of code (excluding whitespace and comments). We observe that *RA2*'s programming model allows implicit parallelization in this case: No explicit handling of parallelization, e.g., creation of threads, locks, or tasks, is required by the actual analysis[2]. In particular, no rethinking of the algorithm is required, as it is, e.g., for GPU-based solutions [160,

[2] Only within the `main` function one `Future` needs to be awaited.

181] to map the execution to the parallel hardware. Extending the analysis to a more powerful purity analysis would require changes only to the analysis and continuation functions (and potentially an extension of the lattice for a finer granularity) and would not introduce any additional complexity related to parallelization.

In order to showcase a more complex analysis, we adapted the IFDS solver from Section 11.1 to *RA2*. This required only minor changes to support the different dependency handling. IFDS has been parallelized in the past and we evaluate our scalability against the state-of-the-art IFDS solver *Heros* in Section 13.4.4. Our implementation makes use of `MonotonicUpdaters`, which reduces the number of large set operations. It requires sequential updates as it maintains mutable state between the continuation invocations to keep track of dataflow edges already known. Thus, the parallelization is semi-implicit in the case of the IFDS solver.

Using our IFDS solver, we implemented a taint analysis as a client analysis used in the rest of the evaluation. This analysis is inspired by *FlowTwist* [139] and identifies public or protected methods in the Java Runtime 7 (`rt.jar`) with return type `java.lang.Object` or `java.lang.Class` that have a string parameter that is later used in an invocation of `java.lang.Class.forName`. If such flows are found, attackers may get the ability to load a class of their choice. The analysis tracks local variables and is field-sensitive.

**Observation 11.6**

*RA2* is able to provide (semi-)implicit parallelization to the simple purity analysis as well as the significantly more complex IFDS solver.

**Observation 11.7**

Both analyses are very different in their kind and there is no specialized support for any of them implemented in *RA2*, indicating that *RA2* is applicable to different kinds of static analyses.

## 11.5 SUMMARY

In this chapter, we answered **RQ1** by showing that *OPAL*'s modular architecture supports the implementation of a broad range of static analysis kinds as evidenced by our case studies. The modularity of these analyses benefits both end users and analysis developers by facilitating experimentation with different trade-offs between precision, soundness, and scalability. *RA2* finally provides (semi-)implicit parallelization for different kinds of static analysis implemented in *OPAL*.

# 12 PRECISION AND SOUNDNESS

This chapter considers **RQ2**, that is, we evaluate whether modular analyses in *OPAL* can improve over the respective state of the art in terms of precision and soundness. We do so by comparing the analyses of each case study from Part II with respective state-of-the-art analyses. We show that our analyses match and even outperform this state of the art, confirming that their modular, collaborative design allows for improved results.

## 12.1 TACAI

We perform two experiments to evaluate *TACAI*, our analysis for a three-address-code intermediate representation based on abstract interpretation (cf. Chapter 7): first we look directly at the type information provided by the intermediate representation, then we consider its impact on the precision of a call graph built upon it.

### 12.1.1 *Precision of Type Information*

In order to evaluate the precision of *TACAI*, we compare the type information available in the *OPAL* IR to that available in *Shimple*[1], an intermediate representation of the *Soot* static analysis framework. We chose *Shimple* because it also is an SSA-based three-address-code representation and is thus closer to *TACAI*'s IR than *Soot*'s other IR, Jimple. The IR of the *WALA* framework does not provide any refined type information over the types available directly in the bytecode. We compare *TACAI*'s most basic domain $TACAI^{L0}$ which only uses static type information and the most precise domain $TACAI^{L2}$ which, among others, performs constant propagation and folding as well as inlining of monomorphic calls, computes union and intersection types, and has special support for `System.arraycopy`.

We analyzed five programs from the *XCorpus* [67]: *jasml*, *javacc*, *jext*, *proguard*, and *sablecc*. They all have a `main` method, which is necessary for the call graphs in the second experiment. To perform the experiment, we compare each representation's receiver-type information of all potentially polymorphic method invocations, i.e., virtual and interface invocations.

The comparison between *Soot*'s *Shimple* and *OPAL*'s *TACAI* is carried out as follows: First, we generate *Shimple* for each method within the target program. Afterward, we traverse each method's *Shimple* lin-

[1] https://github.com/soot-oss/soot/wiki/A-brief-overview-of-Shimple

early and memorize for each polymorphic invocation its surrounding method, the invoked method's signature, the line number[2] it occurred in, and its receiver type. As we traverse each method linearly, we can distinguish multiple invocations within the same line. Then, we run *TACAI* in its current configuration and match each call site with those recorded by *Soot*. Next, the call site's receiver types are compared to each other to determine if *Shimple*'s type information is more precise than ours or vice versa. If both types are equal, we consider them equally precise if *TACAI* does not know that its type information is precise. In the case of precise type information, *TACAI* is only considered more precise when the precise type has subtypes. When intersection types are inferred, we always consider them to be more precise. However, when *TACAI* reports union types, we only consider them to be more precise if each type contained in the union is more precise than *Shimple*'s receiver type. If a call site cannot be matched, e.g., because it is not present in either representation, we record that this call site is incomparable.

Table 12.1: Comparison of Receiver Type Information: Shimple vs. TACAI

| Program | Repr. | Call Sites | Failed | Not Null | Precise | Equal | +Shimple | +TACAI |
|---|---|---|---|---|---|---|---|---|
| **jasml** | L0 | 2 094 | 37 | 0 | 843 | 2 057 | 0 | 0 |
| | L2 | 2 094 | 37 | 838 | 1 028 | 1 987 | 0 | 70 |
| **javacc** | L0 | 9 883 | 0 | 0 | 4 709 | 9 878 | 0 | 5 |
| | L2 | 9 722 | 20 | 3 551 | 4 925 | 9 546 | 0 | 164 |
| **jext** | L0 | 15 457 | 2 | 0 | 2 803 | 15 450 | 0 | 5 |
| | L2 | 15 455 | 2 | 5 709 | 3 406 | 14 986 | 0 | 467 |
| **proguard** | L0 | 9 961 | 520 | 0 | 3 560 | 9 439 | 0 | 2 |
| | L2 | 9 959 | 520 | 3 694 | 4 168 | 9 083 | 0 | 356 |
| **sablecc** | L0 | 35 717 | 0 | 0 | 4 542 | 35 716 | 0 | 1 |
| | L2 | 35 715 | 0 | 4 143 | 5 180 | 35 262 | 0 | 453 |

+Shimple = Shimple more precise, +TACAI = TACAI more precise

The experiment's results are reported in Table 12.1, which shows the evaluated program in the first column and the compared representations in the second column followed by the project's total number of call sites[3]. The column *failed* gives the number of call sites that could not be matched between *TACAI* and *Shimple*. Column *not null* gives the total number of call sites where *TACAI* finds the receiver to never be null, whereas column *precise* gives the number of call sites where *TACAI* could compute a single possible runtime type for the receiver. The last three columns show for how many call sites the receiver-type

[2] We were not able to map the call site to its original bytecode program counter, therefore we chose to use the line number.

[3] Differences in the number of call sites are the result of dead code elimination

information provided by *Shimple* is equally, more, or less precise when compared to *TACAI*.

Table 12.1 shows that we were able to match most call sites across *Shimple* and *TACAI*'s representation. While comparing both intermediate representations on *proguard*, we were not able to match 520 call sites. A closer investigation revealed that *Shimple* erroneously assumes that some implicit exception (e.g., `DivisionByZeroException` or `ArrayIndexOutOfBoundsException`) cannot be raised. This leads to a large number of call sites that cannot be matched in *proguard*. In *javacc*, $TACAI^{L2}$ inlined some calls, resulting in the respective call sites being removed and thus not matchable to *Shimple*.

When we only consider matchable call sites, we observe that the receiver-type information across *Shimple*, $TACAI^{L0}$, and $TACAI^{L2}$ is mostly equal. Whereas *Shimple* never provides more precise type information than even $TACAI^{L0}$, $TACAI^{L2}$ can maximally improve on *jext* where it has more precise information for 467 receivers. However, the overall number of improvements pertaining to receiver-type information is small. When comparing the availability of nullness information, i.e., the number of cases where we definitively know that a receiver is non-null and no `NullPointerException` can be thrown, between $TACAI^{L0}$ and $TACAI^{L2}$, we observe that non-nullness information is at least available in 11% of all cases in *sablecc* and up to 40% of all cases in *jasml*.

**Observation 12.1**

Our approach improves little over *Shimple* w.r.t. receiver-type information. Beyond receiver-type information, however, $TACAI^{L1}$ and $TACAI^{L2}$ provide additional information useful for static analysis, e.g., nullness. Such information is not present in *Shimple*.

### 12.1.2 *Impact on Call-Graph Precision*

As a second experiment, we evaluate how the use of different abstract domains used to create the IR affects the precision of a call graph built upon this IR. To measure the effect, we use a class hierarchy analysis (CHA) to construct the call graph. CHA is solely based on the declared types and, therefore, is best suited for our experiment. However, other algorithms such as RTA potentially also benefit from more precise type information.

In Table 12.2 the columns for *Shimple* and *TACAI* give the respective number of call-graph edges. The final column shows the reduction in the number of call edges constructed based on $TACAI^{L2}$ as compared to *Shimple*. As the table shows, the reduction in the number of call edges between $TACAI^{L0}$ and $TACAI^{L2}$ is minuscule. Compared to

Table 12.2: Number of Call Edges: Shimple vs. TACAI

| Program | Classes | Methods | Shimple | TACAI$^{L0}$ | TACAI$^{L1}$ | TACAI$^{L2}$ | Reduction |
|---|---|---|---|---|---|---|---|
| **jasml** | 50 | 265 | 181 581 | 5 195 | 5 065 | 5 065 | -97% |
| **javacc** | 154 | 2 151 | 218 222 | 71 515 | 71 003 | 70 985 | -67% |
| **jext** | 466 | 2 799 | 481 643 | 17 335 | 17 297 | 17 291 | -96% |
| **proguard** | 645 | 5 237 | 503 190 | 46 218 | 46 096 | 43 535 | -91% |
| **sablecc** | 286 | 2 274 | 230 638 | 52 076 | 50 939 | 50 939 | -78% |

*Soot*'s CHA created with *Shimple*, our *TACAI* has a significant reduction in call edges with any of the three domains: between 67% for *javacc* and 97% for *jasml*.

**Observation 12.2**

*TACAI* significantly improves the size of a CHA call graph compared to *Soot*'s *Shimple* IR.

## 12.2 unimocg

Recall that *Unimocg*'s main goal was to enable consistently high soundness across multiple dissimilar call-graph algorithms (cf. Chapter 8). In addition to validating that this goal is achieved, we also take a look at *Unimocg*'s impact on precision to ensure that high soundness is not achieved at the cost of compromising precision.

### 12.2.1 *Soundness Consistency*

To assess the soundness consistency of *Unimocg*'s call graphs, we executed the benchmark of Reif et al. [186] on five of *Unimocg*'s algorithms—CHA, RTA, XTA, 0-CFA, 1-1-CFA—which are representative for the different families of algorithms; other algorithms from the same families (e.g., MTA instead of XTA) would show similar results. The benchmark by Reif et al. measures missing edges (false negatives) caused by insufficient support of individual language features. We used this benchmark instead of measuring recall on real-world applications, as there is no suitable ground truth for recall on real-world applications. Also, this would only provide a global view on false negatives, i.e., the overall number of missing call-graph edges independent of language features; thus, we would not be able to answer whether *Unimocg* improves on the consistency of language feature support.

Table 12.3 shows the results, and we show Table 8.1 here again as Table 12.4 for easier comparison. *Unimocg*'s call-graph algorithms exhibit high and consistent language feature support. They soundly pass be-

Table 12.3: Soundness of Unimocg's Call-Graph Algorithms

| **Feature** | **CHA** | **RTA** | **XTA** | **0-CFA** | **1-1-CFA** |
|---|---|---|---|---|---|
| Non-virtual Calls | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Virtual Calls | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Types | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Static Initializer | ● 8/8 | ● 8/8 | ● 8/8 | ● 8/8 | ● 8/8 |
| Java 8 Interfaces | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 |
| Unsafe | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 |
| Invokedynamic | ◐ 11/16 | ◐ 11/16 | ◐ 11/16 | ◐ 11/16 | ◐ 11/16 |
| Class.forName | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Reflection | ◐ 10/16 | ◐ 10/16 | ◐ 10/16 | ◐ 8/16 | ◐ 11/16 |
| MethodHandle | ● 9/9 | ◐ 8/9 | ◐ 7/9 | ◐ 7/9 | ◐ 7/9 |
| Class Loading | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 |
| DynamicProxy | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 |
| JVM Calls | ◐ 3/5 | ◐ 3/5 | ◐ 3/5 | ◐ 3/5 | ◐ 3/5 |
| Serialization | ◐ 9/14 | ◐ 9/14 | ◐ 9/14 | ◐ 7/14 | ◐ 7/14 |
| Library Analysis | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 |
| Sign. Polymorph. | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 |
| Java 9+ | ● 2/2 | ● 2/2 | ● 2/2 | ● 2/2 | ● 2/2 |
| Non-Java | ● 2/2 | ● 2/2 | ● 2/2 | ● 2/2 | ● 2/2 |
| **Sum** (out of 123) | 97 (79%) | 96 (78%) | 95 (77%) | 91 (74%) | 94 (76%) |

Algorithms are ordered by increasing precision
Soundness: all ●, some ◐, or no ○ test cases passed soundly

tween 74% and 79% of all test cases. In contrast, *WALA* passes between 41% and 53% and *Soot* between 53% and 66% of test cases. Consistent feature support in *Unimocg* is due to its call-graph algorithms sharing the same call resolvers.

Like *Soot* and *WALA*, *Unimocg* shows some degradation in soundness with increased precision, as, e.g., its 0-CFA algorithm passes only 74% of test cases compared to 79% for its CHA algorithm. But there are significant differences: First, the gap between the most and the least sound algorithms of *Unimocg* is only 5 percentage points (pp), as opposed to 11 pp and 13 pp for *WALA* and *Soot*, respectively. Second, and more importantly, unsoundness in *Unimocg* is either (a) to be expected for certain algorithms, (b) due to unsoundness of some type producers, or (c) due to not yet implemented call resolvers. For instance, it is not surprising that 0-CFA handles reflection less soundly than less precise algorithms that over-approximate some calls; on the other hand, 1-1-CFA has access to actual allocation sites, which enables more sound and precise resolution of reflective calls. Unsoundness for MethodHandles in XTA and CFA, on the other hand, is due to their current type producers not modeling field accesses fully soundly. Currently, we still lack resolvers for class loading and dynamic proxies.

Table 12.4: Soundness of Call Graphs for Different JVM Features

| | **WALA** | | | **Soot** | | |
|---|---|---|---|---|---|---|
| **Feature** | **CHA** | **RTA** | **0-CFA** | **CHA** | **RTA** | **SPARK** |
| Non-virtual Calls | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Virtual Calls | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Types | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 | ● 6/6 |
| Static Initializer | ◐ 4/8 | ◐ 7/8 | ◐ 6/8 | ● 8/8 | ● 8/8 | ● 8/8 |
| Java 8 Interfaces | ● 7/7 | ● 7/7 | ● 7/7 | ● 7/7 | ◐ 6/7 | ● 7/7 |
| Unsafe | ● 7/7 | ● 7/7 | ○ 0/7 | ● 7/7 | ● 7/7 | ○ 0/7 |
| Invokedynamic | ○ 0/16 | ◐ 10/16 | ◐ 10/16 | ◐ 11/16 | ◐ 11/16 | ◐ 11/16 |
| Class.forName | ◐ 2/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 | ● 4/4 |
| Reflection | ◐ 2/16 | ◐ 3/16 | ◐ 6/16 | ◐ 12/16 | ◐ 12/16 | ◐ 10/16 |
| MethodHandle | ◐ 2/9 | ◐ 2/9 | ○ 0/9 | ◐ 3/9 | ◐ 3/9 | ◐ 1/9 |
| Class Loading | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 | ○ 0/4 |
| DynamicProxy | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 | ○ 0/1 |
| JVM Calls | ◐ 2/5 | ◐ 3/5 | ◐ 3/5 | ◐ 4/5 | ◐ 4/5 | ◐ 3/5 |
| Serialization | ◐ 3/14 | ◐ 1/14 | ◐ 1/14 | ◐ 5/14 | ◐ 1/14 | ◐ 1/14 |
| Library Analysis | ◐ 2/5 | ◐ 2/5 | ◐ 1/5 | ◐ 2/5 | ◐ 2/5 | ◐ 2/5 |
| Sign. Polymorph. | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 | ○ 0/7 |
| Java 9+ | ● 2/2 | ◐ 1/2 | ◐ 1/2 | ● 2/2 | ● 2/2 | ● 2/2 |
| Non-Java | ● 2/2 | ● 2/2 | ● 2/2 | ○ 0/2 | ○ 0/2 | ○ 0/2 |
| **Sum** (out of 123) | 51 (41%) | 65 (53%) | 57 (46%) | 81 (66%) | 76 (62%) | 65 (53%) |

Algorithms within each framework are ordered by increasing precision
Soundness: all ●, some ◐, or no ○ test cases passed soundly

These complex features are not supported by *WALA* or *Soot* either. Once the respective call resolvers are implemented, they can be used consistently for all algorithms. To sum up, the sources of unsoundness in *Unimocg* are explainable and unsoundness due to imprecise type producers or missing call resolvers mainly requires further engineering effort—without requiring any changes to existing call resolvers.

**Observation 12.3**

*Unimocg* shows consistently high soundness compared to other frameworks. This is the result of reusing the same call-resolver modules across all call-graph algorithms.

### 12.2.2 *Impact on Precision*

Next, we show that *Unimocg*'s consistent soundness does not compromise precision.

Following common methodology [216, 234], we measure precision by counting the number of reachable methods—more precise call

Table 12.5: Reachable Methods for *Unimocg*'s Algorithms

| **Program** | **CHA** | **RTA** | **XTA** | **0-CFA** |
|---|---|---|---|---|
| jasml | 130 249 | 10 719 | 10 086 | 8 901 |
| javacc | 131 079 | 11 511 | 10 901 | 9 693 |
| jext | 132 885 | 23 266 | 21 215 | 19 237 |
| proguard | 134 895 | 14 766 | 14 149 | 12 773 |
| sablecc | 132 120 | 12 436 | 11 790 | 10 512 |

graphs have fewer reachable methods. We compare the numbers of reachable methods found by *Unimocg*'s CHA, RTA, XTA, and 0-CFA algorithms for five Java applications[4] from XCorpus [67].

The number of reachable methods decreases significantly when we use more precise type producers; e.g., RTA identifies around 90% fewer reachable methods than CHA. This matches the expectations of call-graph users and shows that reusing the same call-resolver modules across call-graph algorithms does not impair their relative precision. This is explainable: individual call resolvers use the type information gathered by the chosen type producer and thus work at a consistent level of precision. This is not possible if the computation of type information is tightly coupled to the resolution of virtual calls; in this case, modules for other language features would rely on a fixed, potentially ad-hoc, method of computing type information.

**Observation 12.4**

*Unimocg*'s modular architecture does not compromise on precision; this is indicated by the consistent relative precision across different algorithms.

### 12.2.3 *Impact on Type Consumers*

Finally, we performed a case study with our field-immutability analysis (cf. Chapter 9). In particular, the analysis depends on whether the types of objects stored in the field are immutable.

Our original implementation of the immutability analysis relied on the declared type of a field (i.e., it has CHA precision) plus some additional ad-hoc precision improvements (we refer to this as *ad-hoc CHA*). Subsequently, we implemented a new version of the field-immutability analysis that uses *Unimocg*'s type iterator interface. Our hypothesis is that the field-immutability analysis benefits directly from using the type iterator in terms of both improved precision and reduced code size. The precision of the immutability analysis depends on precise type information, because `final` fields are either *transitively*

[4] The same applications we used in prior work [186].

Table 12.6: Field Immutability Results for OpenJDK

| **Algorithm** | **mutable** | ⊒ **non-trans.** | ⊒ **depen.** | ⊒ **trans.** |
|---|---:|---:|---:|---:|
| Ad-hoc CHA | 23 195 | 24 296 | 108 | 46 368 |
| CHA | 23 195 | 25 252 | 20 | 45 500 |
| RTA | 23 195 | 7 352 | 316 | 63 104 |
| XTA | 23 195 | 2 871 | 316 | 67 585 |

depen. = dependently immutable, trans. = transitively immutable
Higher numbers in columns to the right = more precise

*immutable*, if they can only refer to objects that are immutable, or otherwise *non-transitively immutable*, if that is not the case; *dependent immutability* (for fields with generic types) is located between these two levels. More precise type information thus allows assigning the more precise value *transitively immutable* to more fields. The code size is expected to be reduced because the immutability analysis does not need to implement (ad-hoc) logic for inferring type information.

We compare the ad-hoc CHA implementation with the *Unimocg*-based one using different call-graph algorithms. Previously, such an exploration of different call-graph algorithms would not have been possible as the precision was hard-coded into the analysis (ad-hoc CHA). We analyze the Adoptium OpenJDK 1.8.0_342-b07 here.

Table 12.6 shows the results concerning precision, clearly showing that using more precise call-graph algorithms (in particular, more precise type information) significantly improves the precision of the immutability analysis implemented as a *Unimocg* type consumer. For instance, using the RTA type iterator results in 17604 more fields, i.e., 18.8% of all fields, found to be transitively immutable compared to CHA. Using XTA further improved the precision, with 4481 more transitively immutable fields compared to RTA. The CHA type iterator results in less precision than the baseline, because of the removed ad-hoc logic of the baseline for improving precision upon CHA. Yet, (a) the precision reduction is small (0.9% of all fields get a less precise result), and (b) one could avoid even this small imprecision (at the cost of some performance) by using Unimocg's *foreachAlloc* method instead without adding complexity to the implementation of the immutability analysis.

Moreover, by implementing the field immutability analysis as a *Unimocg* type consumer, we could replace the baseline's *ad-hoc CHA* logic (95 lines of code, or 20% of the total size of the field immutability analysis) by 26 lines of code for using the type iterator while achieving higher precision and enabling experimentation with different call-graph algorithms. The positive effect on code quality is more pronounced than the mere numbers may suggest: The removed code was complex and a clear violation of the principle of separation of concerns, as it was not concerned with the actual task of analyzing immutability. Using *Unimocg*, duplication of functionality is reduced, and separation of concerns is re-established.

**Observation 12.5**

The precision of the field-immutability analysis implemented as a type consumer is always consistent with the employed call-graph algorithm and directly benefits from more precise call-graph algorithms.

**Observation 12.6**

Implementing the field-immutability analysis as a type consumer leads to less complexity and a better separation of concerns compared to an ad-hoc implementation.

## 12.3 CIFI

Next, we evaluate the precision and sound(i)ness of our immutability analyses *CiFi* (cf. Chapter 9). To do so, we use our *CiFi-Bench* from Section 9.4, a handcrafted benchmark of standard and corner cases for class and field immutability. We also compare *CiFi* to the state-of-the-art immutability-enforcement tool *Glacier*.

### 12.3.1 *CiFi-Bench Results*

We first use *CiFi* on *CiFi-Bench*. For each test case *tc*, *CiFi* either produces the precise value annotated in *tc* (in five categories), or a soundy over-approximation, i.e., a value further up in the respective lattice (cf. Section 9.2), which is less precise than possible but can be soundily used by further analyses/optimizations. *CiFi* did not produce any unsound results, i.e., values further down in the lattice. In some more detail, the results are as follows:

- *CiFi* inferred immutability properties precisely for the categories: *Assignability*, *General*, *Known Types*, *Generic/Simple*, *Arrays/Non-Transitive*, *Lazy Initialization/Arrays*, and *Lazy Initialization/Objects*.
- In category *Generic/Extended*, *CiFi* soundily over-approximates some complex test cases such as doubly nested generic classes (`Gen<Gen<T>>`), generic cases with bounds, and more complex lazy initialization patterns than the one we described in Section 9.3.2. For doubly nested generics, the approximation is not *mutable*, but *non-transitively immutable*, retaining some precision. Generic classes with bounds are soundily over-approximated as *dependently immutable*.

- In *Arrays/Transitive*, *CiFi* soundily over-approximates all tests to *non-transitively immutable*, and in *Lazy Initialization/Primitive Types* to *unsafely lazily initialized* or *assignable*. All test cases in *Lazy Initialization/Scala Lazy Val* and *String* are soundily over-approximated to *assignable*, except for the field referring to the final `char` array in the category *String* which is soundily over-approximated to *non-transitively immutable*.

**Observation 12.7**

*CiFi* matches the annotated properties of the benchmark either precisely or soundily over-approximates them. The observed over-approximations are due to missing support for the respective features. Leaving complex features out of scope when the expected benefit is small is in line with other immutability analyses [179]. Handling these complex features precisely would not lead to considerably more immutability being recognized, as they represent rare corner cases. Handling each of these corner cases would only prevent few over-approximations.

### 12.3.2 *Comparison to Glacier*

Next, we run *Glacier* [44], the state-of-the-art tool for enforcing class and field immutability on *CiFi-Bench*. As *Glacier* only considers transitive immutability, we can only evaluate it w.r.t. this level of immutability. Hence, we annotated all classes and fields of *CiFi-Bench* with *Glacier*'s `@Immutable` annotation. We consider *Glacier* to pass a test if either of the following holds: (a) it does not output an error for transitively immutable fields and classes, (b) it outputs such an error for fields and classes that are not transitively immutable (since *Glacier*, does not handle non-transitive or dependent immutability, respective fields have to be considered mutable). The results for each category are as follows:

- For category *Known Types/Multiple*, *Glacier* can enforce transitive immutability.

- For two cases in *General* resp. *Known Types/Single*, *Glacier* produces unsound results. First, *Glacier* treats both `@Immutable` and `@MaybeMutable` classes as subtypes of `java.lang.Object`. Thus, a mutable object can be assigned to an `@Immutable` field of type `Object`. Second, while *Glacier* prohibits assignments to fields outside of the constructor, it does not check whether a field being assigned in a constructor belongs to the object being constructed. Thus, `@Immutable` fields can be mutated while constructing other objects. Both cases are shown in Listing 12.1.

```
@Immutable class C {
  @Immutable private Object o;
  public C(C parent, Object o){ parent.o = o; }
}
```

Listing 12.1: *Glacier* Unsoundness Example

- *Glacier* was unsound in three *Assignability* tests. Two are again due to `@MaybeMutable` being a subtype of `java.lang.Object`, but the third one revealed another issue: *Glacier* ignores compound-assignment operators like +=. Thus, fields of primitive types or type `java.lang.String` can be mutated outside of constructors. In *CiFi*, such accidental omissions are less likely as it analyzes bytecode. Additionally, *Glacier* could not handle the clone pattern cases properly because of assignments outside of constructors.
- *Glacier* passed all test cases in *Generic* as it enforces that only `@Immutable` types are used for type parameters of `@Immutable` classes and only `@Immutable` classes can extend `@Immutable` classes. But this means that *Glacier* cannot handle dependent immutability, resulting in lost opportunities for being more precise.
- In category *Arrays*, non-transitively immutable fields are handled correctly. Some transitively immutable fields are also enforced correctly but require four annotations: `@Immutable int @Immutable[] arr = new @Immutable int @Immutable[5];` *Glacier* cannot enforce transitive immutability where array elements are not mutated, despite not being `@Immutable`.
- In category *Lazy Initialization*, *Glacier* cannot enforce transitive immutability due to its rule that in `@Immutable` classes, fields may only be assigned in constructors.
- In category *String*, *Glacier* handles the case concerning the `char` array shared between identical strings precisely, but it cannot enforce immutability for the lazily initialized field caching the `hashCode` method's result.

**Observation 12.8**

Compared to *CiFi*'s fine-grained immutability results, *Glacier* can only recognize transitive immutability.

**Observation 12.9**

*Glacier* shows three cases of unsoundness.

**Observation 12.10**

While *Glacier* strictly enforces transitive immutability for generics, including nested and bounded generic types, it lacks the flexibility of dependent immutability to allow generic classes to be treated differently depending on whether they are instantiated with transitively immutable types or not. Additionally, *Glacier* does not handle lazy initialization.

To recap, *CiFi* is more soundy and often more precise than *Glacier* without requiring manual effort for annotations. As a result, *CiFi* can be applied easily to existing codebases and third-party code, even if source code is not available.

## 12.4 OPIUM

To evaluate the purity analyses of *OPIUM* (cf. Chapter 10), we compared the results of our most precise analysis, *Purity*$_2$, against *JPPA* [223, 224], *JPure* [178], and *ReIm* [108, 109]. *ReIm* is the most recent of these tools, representing the state of the art. All three tools identify *side-effect-free* methods and were downloaded from the authors' websites.

We first use a synthetic benchmark to study the differences between the four tools, then compare *OPIUM* against the most precise competitor, ReIm, on two real-world applications.

### 12.4.1 *Synthetic Benchmark*

For a first comparison, we use the *JOlden* [38] benchmark. It consists of ten small Java programs from different domains that all tools were able to analyze with only one exception.

The sets of all methods that are analyzed by the tools have small differences. *JPPA* only analyzes methods transitively invoked by the main method, *JPure* and *ReIm* analyze all methods present in the source code, and our approach analyzes all methods present in the class files, which in particular includes static initializers and automatically generated default constructors. Furthermore, the reports of *JPure* and *ReIm* also include aggregated purity results for abstract methods. Aggregated information is in our case provided by the *Virtual Method Purity* analysis, not by the base purity analysis.

The analysis results are shown in Table 12.7. For each of the *JOlden* projects, it gives the number of methods that each tool identified as *side-effect free* (including *pure* methods for *OPIUM*) and the number of methods that the toll has analyzed. *JPure* failed to analyze *TSP*. For *OPIUM*, the table additionally gives the number of *pure* (including *domain-specific pure*) methods identified by our analysis and the number

Table 12.7: At Least Side-Effect-Free (SEF) Methods in JOlden

| | **JPPA** | | **JPure** | | **ReIm** | | **OPIUM** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Program** | Σ | #SEF | Σ | #SEF | Σ | #SEF | Σ | #SEF | #Pure | #Cont. |
| BH | 59 | 24 | 69 | 10 | 69 | 33 | 70 | 33 | 11 | +18 |
| BiSort | 13 | 4 | 13 | 3 | 13 | 5 | 15 | 7 | 6 | +2 |
| Em3d | 20 | 5 | 19 | 1 | 19 | 8 | 23 | 8 | 5 | +3 |
| Health | 26 | 6 | 26 | 2 | 26 | 11 | 29 | 13 | 9 | +4 |
| MST | 31 | 15 | 33 | 12 | 33 | 16 | 36 | 19 | 8 | +6 |
| Perimeter | 36 | 27 | 42 | 31 | 42 | 38 | 45 | 38 | 21 | +1 |
| Power | 29 | 4 | 29 | 2 | 29 | 10 | 32 | 11 | 7 | +13 |
| TreeAdd | 5 | 1 | 10 | 1 | 10 | 6 | 12 | 7 | 6 | +1 |
| TSP | 14 | 4 | 0 | – | 14 | 1 | 16 | 4 | 3 | +0 |
| Voronoi | 60 | 40 | 71 | 30 | 71 | 47 | 73 | 49 | 12 | +5 |

Σ = analyzed methods, #SEF = at least SEF methods, #Pure = pure methods,
#Cont. = additional contextually/externally pure/SEF methods

of methods with additional purity levels (i.e., *external* and *contextual purity* and variants thereof).

For this comparison, we treat *pure* and *side-effect free* as well as their *domain-specific* variants as *side-effect free*. Our analysis is competitive with *ReIm* and significantly outperforms *JPPA* and *JPure*. Hence, our analysis is competitive with state-of-the-art analyses for *side-effect-free* methods. Additionally, our analysis identifies *pure* methods and a significant number of *externally* and *contextually pure/side-effect-free* methods, which are not found by the other tools including *ReIm*.

The differences between *JPPA* and *JPure* when compared to *ReIm* and our analysis in programs like *Power* and *TreeAdd* are due to a high number of constructors. These are not identified as *side-effect free* by *JPPA* and *JPure*. For *TSP*, *JPPA* classifies several methods using `java.util.Random` as *side-effect free*, even though they are not—`Random` modifies global state when an instance is created—while *ReIm* fails to identify a pure constructor. We manually verified that all classifications performed by *OPIUM* were sound and identified further potential for improvements where we would be able to identify methods as pure if the supporting analyses were more precise.

**Observation 12.11**

*OPIUM* identifies at least as many side-effect-free methods as any of the other tools.

**Observation 12.12**

*OPIUM* identifies a significant part of these methods as *pure*, adding to precision.

**Observation 12.13**

Additionally, *OPIUM* identifies several methods with external or contextual purity levels.

### 12.4.2 *Real-World Applications*

To evaluate our analysis on real-world applications, we compared it with *ReIm* on *Batik* and *Xalan*. Table 12.8 again presents the number of methods identified as side-effect free (or even pure) by *ReIm* and our analysis alongside the number of analyzed methods. The number of *pure* methods and the number of additional, *externally* or *contextually pure/side-effect-free* methods is given for *OPIUM*.

Table 12.8: At Least Side-Effect-Free Methods in Batik/Xalan

| **Program** | **Batik** | | **Xalan** | |
|---|---|---|---|---|
| **ReIm** *#Analyzed methods* | 16 029 | | 10 386 | |
| *At least Side-Effect-Free Methods* | 6 072 | (37.88%) | 3 942 | (37.95%) |
| **OPIUM** *#Analyzed methods* | 15 911 | | 10 763 | |
| *At least Side-Effect-Free Methods* | 6 780 | (42.61%) | 4 390 | (40.79%) |
| *Pure methods* | 4 009 | (25.20%) | 2 492 | (23.15%) |
| *Ext./Context. Pure/SEF methods* | +987 | (6.20%) | +748 | (6.95%) |

On these applications, our analysis outperforms ReIm: we identify up to 5% more side-effect-free methods and up to 7% of all methods as being externally or contextually pure/side-effect-free methods.

**Observation 12.14**

On two real-world applications, *OPIUM* outperforms *ReIm*, the state of the art, in terms of precision, finding more side-effect-free methods. It is also more expressive, additionally identifying both a significant number of *pure* methods as well as methods with external or contextual purity levels.

## 12.5 SUMMARY

In this chapter, we answered **RQ2** by showing that modular analyses implemented in *OPAL* improve upon the respective state of the art in terms of precision and soundness. We attribute these improvements to *OPAL*'s support for modularity, which allows solving complex analysis problems by combining several simpler modules that each are concerned with a clearly delineated sub-problem. This fosters the implementation of modules that improve precision by providing information that would otherwise have to be over-approximated (e.g., escape information for immutability and purity analyses) or improve soundness by adding support for complex language features (e.g., reflective calls for call graph analyses).

# 13 SCALABILITY, PARALLELIZATION AND SCHEDULING

This chapter answers our final research question **RQ3**, i.e., how parallelization and scheduling affect the scalability of modular analyses in *OPAL* and how their runtime performance compares to the respective state of the art. *OPAL*'s blackboard architecture allows for automated parallelization and scheduling that can utilize the hardware resources of modern multicore processors and reduce runtimes. We first evaluate the impact of these specific features such as parallelization, the possibility of using specialized data structures, and different strategies for scheduling the execution order of tasks in the blackboard. We then compare the case-study analyses of Part II as well as the IFDS analysis from Chapter 11 against respective state-of-the-art tools. In Chapter 12, we showed that analyses in *OPAL* outperform their competition w.r.t. soundness and precision. In this chapter, we show that this does not come at the cost of inferior scalability. Instead, analyses in *OPAL* are on par or even outperform the respective state of the art also in terms of runtime performance. This is despite *OPAL* being more general than individual analyses or specialized frameworks optimized for specific kinds of analyses.

When reporting runtimes, we present the median value of several executions as mentioned in the respective experiments. This is because runtime measurements often had high variance with some outliers that would distort a mean in an unrepresentative manner.

## 13.1 PARALLELIZATION

In order to assess how automated parallelization impacts the runtime performance of analyses in *OPAL*, we implemented a proof-of-concept parallel version of our blackboard. Using this, we measured the execution time for the points-to-based call graph (cf. Chapter 8) with different numbers of threads. All measurements were performed on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. The JVM was given 32 GB of heap memory. We report only excerpts of the results here.[1]

Results for five projects from the *DaCapo* benchmark [24] are shown in Figure 13.1. The projects were selected to have similar runtimes to facilitate graph readability; the other projects show similar behavior. The experiments were run seven times and we report their median runtime. Benefits of parallelization over one thread appear at two

[1] The entire results can be found at https://doi.org/10.5281/zenodo.3972736

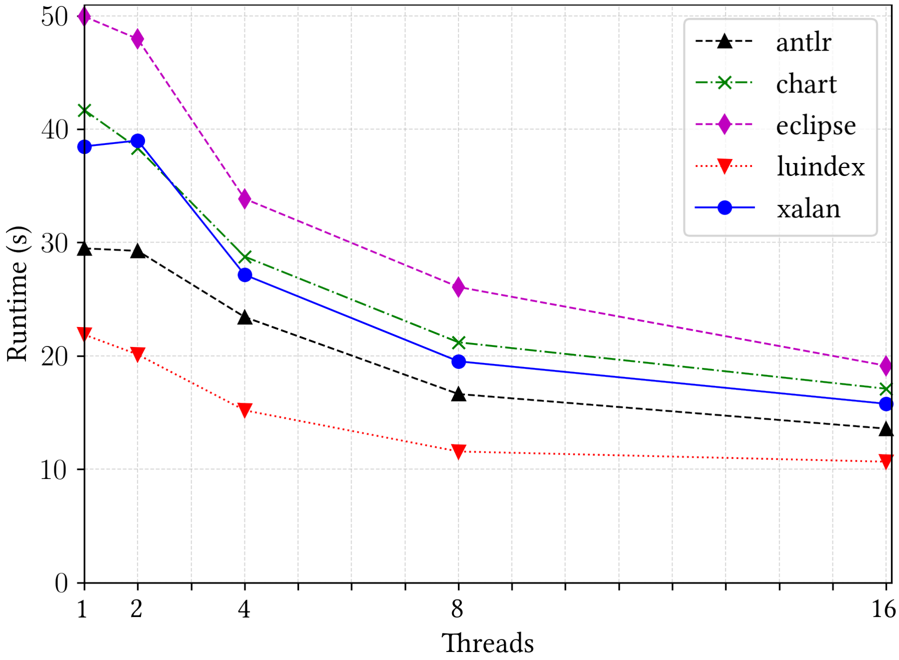


Figure 13.1: Parallel Architecture Scalability

to four threads and we achieve speedups of up to 2x for 16 threads. Beyond this, further improvement is negligible; instead, it slightly decreases due to growing communication overhead. These results are encouraging, given that the parallel version is not at all optimized. An optimized version of it is expected to scale better. Designing such an optimized version requires further research to identify the optimal way to parallelize the computation.

**Observation 13.1**

*OPAL*'s computation can be parallelized automatically, improving scalability.

## 13.2 BENEFITS OF SPECIALIZED DATA STRUCTURES

*OPAL* allows developers to freely choose appropriate data structures for each analysis individually. This is in contrast to other frameworks for modular analyses that enforce the use of particular data structures—e.g., the choice of Datalog solver fixes the data structures used for Datalog relations in the *Doop* framework [34].

In order to evaluate the benefits of using specialized data structures on scalability, we compare two versions of the same points-to-based call-graph algorithm (cf. Chapter 8). Both encode points-to, caller, and callee information as integer values. The first version uses specialized trie-based data structures, while the second one uses standard Scala sets. The evaluation setup was the same as in the experiment above.

Table 13.1: Runtime of Points-To-Based Call Graphs with Specialized and Standard Data-Structures

| **Program** | #RM | 🕒 specialized | 🕒 Scala | speedup |
|---|---:|---:|---:|---:|
| antlr | 8 653 | 28.36 s | 305.90 s | 10.8x |
| bloat | 10 000 | 34.43 s | 266.08 s | 7.7x |
| chart | 12 268 | 40.13 s | 516.37 s | 12.9x |
| eclipse | 13 429 | 44.89 s | 343.69 s | 7.7x |
| fop | 7 509 | 18.87 s | 56.64 s | 3.0x |
| hsqldb | 7 455 | 19.65 s | 55.69 s | 2.8x |
| jython | 13 161 | 77.65 s | 3 341.62 s | 43.0x |
| luindex | 7 972 | 19.34 s | 62.57 s | 3.2x |
| lusearch | 8 540 | 21.03 s | 70.55 s | 3.3x |
| pmd | 9 028 | 21.47 s | 75.47 s | 3.5x |
| xalan | 13 330 | 35.59 s | 246.97 s | 6.9x |
| geometric mean | | 29.68 s | 191.26 s | 6.44x |

#RM = number of methods found to be reachable

Results are given in Table 13.1. The columns *specialized* and *Scala* give the respective runtimes. Due to its high memory consumption, we had to run the version using Scala's data structures with 128 GB of heap space; analysis of *jython* even required 256 GB. Using tailored data structures, *OPAL*'s runtime was 2.8 up to 43 times faster (i.e., reducing the runtime between 65% and 98%) compared to naively using Scala's standard sets.

**Observation 13.2**

Selecting suitable data structures adapted to the specific analysis' needs is an important factor for analysis scalability. While the analysis developer can freely select optimized data structures in *OPAL*, strictly declarative approaches like *Doop* do not support such choices.

## 13.3 SCHEDULING STRATEGIES

*RA2*, our implementation based on *Reactive Async* (cf. Chapter 4) allows selecting a *scheduling strategy* to influence the order in which analysis tasks in the blackboard are executed. In order to assess the impact of these scheduling strategies, we evaluated different scheduling strategies for both the IFDS analysis introduced in Chapter 11 and the simple purity analysis from Chapter 4. We used a machine with an Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz (10 cores / 20 threads) and 128 GB RAM. The JVM was given 16 GB of heap memory. We analyzed the JRE 1.7.0 update 95 from the publicly available

Table 13.2: Runtime of Scheduling Strategies for IFDS

| **Strategy** | **🕘 [s]** | **Speed-up (a)** | **Speed-up (b)** |
|---|---|---|---|
| DefaultScheduling | 27.29 | 0.00% | 13.3% |
| SourcesWithManyTargetsLast | 21.00 | 23.1% | 33.3% |
| TargetsWithManySourcesLast | 19.84 | 27.3% | 37.0% |
| TargetsWithManyTargetsLast | 29.31 | -7.40% | 6.86% |
| SourcesWithManySourcesLast | 21.40 | 21.6% | 32.0% |

Speed-up shown compared to (a) default and (b) slowest strategy

*Doop benchmarks* project [213] to ensure reproducibility. The purity analysis was executed on the complete JRE, while the IFDS analysis was executed on the runtime jar only, which it was developed to be executed on. For each experiment, we report the median runtime of seven executions.

### 13.3.1 *Scheduling at Fixed Thread Count*

We first evaluated different scheduling strategies at a fixed level of parallelization with ten threads each, corresponding to the number of physical cores of our system. In Section 13.3.2, we look at the scalability of the different scheduling strategies at different levels of parallelization.

IFDS ANALYSIS We evaluated the performance of the analysis-independent strategies described in Section 4.4. Table 13.2 shows the median execution times for each strategy. The percentages show the speed-up of each strategy compared to (a) the default strategy and (b) the slowest strategy.

Out of these analysis-independent strategies, we did not further measure the `xFirst` strategies, as initial tests showed them to perform poorly for IFDS. For example, using the scheduling strategy `TargetsWithManySourcesFirst` took more than 900 seconds using one thread, thereby performing more than 1100% worse than the inverse `TargetsWithManySourcesLast` which was the best-performing strategy.

The data shows that using a suitable scheduling strategy can have a significant impact on execution time. Considering the evaluated strategies, `TargetsWithManySourcesLast` is the best strategy for our IFDS analysis. The difference between the best and worst strategies is 37.0% (excluding the above-mentioned poorly performing strategies). This shows the importance of choosing an appropriate strategy.

The effect of each strategy is application-dependent and may differ with the number of cells, the number of dependencies and cycles,

Table 13.3: Runtimes and Speed-Ups for the Purity Analysis

| **Strategy** | **🕓 [s]** | **Speed-up (a)** | **Speed-up (b)** |
|---|---|---|---|
| DefaultScheduling | 0.42 | 0.00% | 40.8% |
| SourcesWithManyTargetsFirst | 0.70 | -66.7% | 1.41% |
| SourcesWithManyTargetsLast | 0.70 | -66.7% | 1.41% |
| TargetsWithManySourcesFirst | 0.71 | -69.0% | 0.00% |
| TargetsWithManySourcesLast | 0.71 | -69.0% | 0.00% |
| TargetsWithManyTargetsFirst | 0.69 | -64.3% | 2.82% |
| TargetsWithManyTargetsLast | 0.68 | -61.9% | 4.23% |
| SourcesWithManySourcesFirst | 0.68 | -61.9% | 4.23% |
| SourcesWithManySourcesLast | 0.69 | -64.3% | 2.82% |
| LatticeValueStrategy | 0.71 | -69.0% | 0.00% |

Speed-up shown compared to (a) default and (b) slowest strategy

and the costs of the used continuation functions. The advantage of `TargetsWithManySourcesLast` for the IFDS analysis can be explained by considering the aggregation of results, i.e., avoiding notifications of dependers. In the case of cells with many sources, it pays off to hold back the update as long as possible because this potentially allows aggregation with updates from other sources. Recall that before a continuation is actually invoked, the most current value(s) is(are) queried again and passed to the continuation in an aggregated form. That way, the target cell can compute its result on a larger batch of information in one step as opposed to multiple small steps, which would be needed, if the cell was informed prematurely. This strategy works well for IFDS, because all propagations need to be handled in the same way and there are no special values, which would lead to early finalization of cells and could therefore be advantageous, as in the case of the purity analysis.

PURITY ANALYSIS In addition to the analysis-independent strategies, we used the `LatticeValueStrategy` adapted to the purity analysis that we introduced in Section 4.4. It uses the specific effect an update of a source cell may have on a target cell: if a dependee is *impure*, we can immediately decide on the purity of the depender and complete it with the value *impure*. The strategy gives such propagations a high priority as they lead to final results quicker. In contrast, if a cell is completed with *pure*, a target cell can *just* remove the dependency, because that update will not affect the current cell's value.

Table 13.3 shows the results for all strategies along with their relative speed-ups. The `DefaultStrategy` is significantly faster than the other strategies for this experiment. The other strategies, including the LatticeValueStrategy that prioritizes impure updates, show no significant differences in runtime. Relative standard deviations are between 1.0% and 6.1%. The HandlerPool uses a ThreadPoolExecutor from the java.util.concurrent package for pluggable scheduling strategies as this allows the necessary priority queue to be supplied. For DefaultStrategy, a ForkJoinPool from the same package is used instead, which uses a work-stealing LIFO queuing scheme. We believe that the DefaultStrategy is faster because the continuation tasks created by the simple purity analysis are tiny. This gives the default ForkJoinPool an advantage over the more complex ThreadPoolExecutor used for the other strategies.

**Observation 13.3**

The best scheduling strategies differ between the purity and IFDS analyses, emphasizing the need for pluggable, user-supplied strategies.

**Observation 13.4**

While the analysis-specific strategy did not benefit the simple purity analysis, it has been shown in the past [192] that such strategies can further improve scalability.

### 13.3.2 *Scalability with Thread Count*

As a second experiment, we measured the speed-ups that *RA2* achieves for different thread counts compared to the nearly identical solver that uses *OPAL*'s single-threaded, but highly optimized blackboard solver. We used the IFDS analyses introduced in Chapter 11.

Figure 13.2 shows how the performance changes with the number of threads used for the IFDS analysis. Depending on the scheduling strategy, the speed-up with two threads compared to one thread is between 1.6x and 2.0x. Note that better strategies in general show lower speed-ups. When increasing the number of threads further, the speed-up increases up to 6.2x for 20 threads. With speed-ups of more than 4.8x for 10 threads, according to Amdahl's law, more than 88% of the execution is parallelized. Relative standard deviations for the reported measurements are between 1.8% and 13.7%.

The *RA2*-based implementation of the IFDS analysis is 28% slower than the sequential implementation in *OPAL* when a single thread is used. The latter yielded runtimes of 58.0 seconds, compared to 74.3 seconds for our best strategy. As expected, *RA2* is slowed down by overhead related to enabling concurrency, which in this case is not needed. However, *RA2* clearly outperforms *OPAL*'s optimized single-threaded blackboard as soon as multiple threads are used. For the

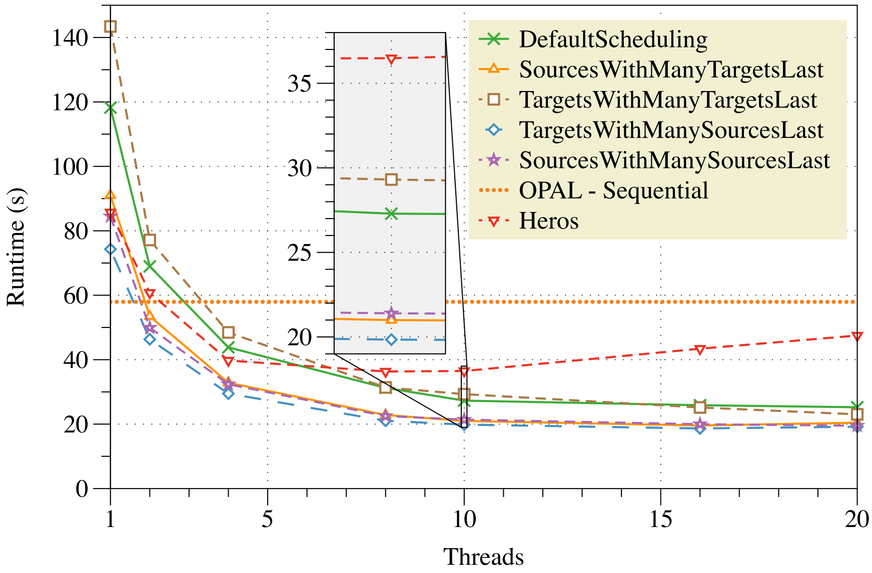


Figure 13.2: Scalability with Different Numbers of Threads

best strategy, the speed-up over *OPAL* ranges between 1.3x with two threads, 2.9x with 10 threads, and 3.0x with 20 threads.

**Observation 13.5**

Different scheduling strategies scale similarly with thread count, outperforming the optimized sequential blackboard at two to four threads and achieving a speedup of almost 3.0x at 10 threads.

## 13.4 SCALABILITY OF THE CASE-STUDY ANALYSES

In this section, we compare the scalability of our case-study analyses from Part II and the IFDS solver from Chapter 11 to respective state-of-the-art analyses. As there was no suitable state of the art for field- or class-immutability inference tools, we did not evaluate *CiFi* in isolation. However, our evaluation of *OPIUM* makes use of *CiFi*.

### 13.4.1 TACAI

Our first experiment assesses the runtime performance of *TACAI* (cf. Chapter 7) and evaluates how exchanging the abstract interpretation domains affects *TACAI*'s output as well as its transformation performance. In order to compare the results, we generate *Soot*'s *Shimple* (cf. Section 7.1 representation as well as our $TACAI^{L0}$, $TACAI^{L1}$, and $TACAI^{L2}$ (cf. Section 7.2) representations for all methods of our five evaluation programs from the *XCorpus* [67]: *jasml*, *javacc*, *jext*, *proguard*, *sablecc*. All measurements were taken on a Mac Pro with a Xeon E5 CPU with 8 cores@3GHz. The JVM was given 24GB of heap space.

Table 13.4: Runtime: Shimple vs. TACAI

| **Program** | **Classes** | **Methods** | **Representation** | **Instructions** | **Avg.** | **Med.** | **St.dev.** | 🕒 |
|---|---|---|---|---|---|---|---|---|
| **jasml** | 50 | 265 | *Shimple* | - | - | - | - | 7.6s |
| | | | $TACAI^{L0}$ | 14 164 | 53.5 | 12 | 307.5 | 3.5s |
| | | | $TACAI^{L1}$ | 14 163 | 53.5 | 12 | 307.5 | 3.9s |
| | | | $TACAI^{L2}$ | 14 066 | 53.4 | 12 | 307.5 | 6.9s |
| **javacc** | 154 | 2 151 | *Shimple* | - | - | - | - | 10.9s |
| | | | $TACAI^{L0}$ | 81 917 | 38.1 | 11 | 150.2 | 4.2s |
| | | | $TACAI^{L1}$ | 81 683 | 38.0 | 11 | 150.2 | 5.4s |
| | | | $TACAI^{L2}$ | 81 651 | 38.0 | 11 | 150.0 | 11.5s |
| **jext** | 466 | 2 799 | *Shimple* | - | - | - | - | 19.2s |
| | | | $TACAI^{L0}$ | 73 428 | 26.2 | 6 | 119.8 | 4.6s |
| | | | $TACAI^{L1}$ | 73 358 | 26.2 | 6 | 119.8 | 5.0s |
| | | | $TACAI^{L2}$ | 73 334 | 26.2 | 6 | 119.7 | 6.4s |
| **proguard** | 645 | 5 237 | *Shimple* | - | - | - | - | 26.3s |
| | | | $TACAI^{L0}$ | 70 203 | 13.4 | 5 | 140.4 | 4.4s |
| | | | $TACAI^{L1}$ | 70 194 | 13.4 | 5 | 140.4 | 4.7s |
| | | | $TACAI^{L2}$ | 69 859 | 13.4 | 5 | 140.4 | 5.8s |
| **sablecc** | 286 | 2 274 | *Shimple* | - | - | - | - | 10.3s |
| | | | $TACAI^{L0}$ | 35 717 | 15.7 | 5 | 50.6 | 4.1s |
| | | | $TACAI^{L1}$ | 35 715 | 15.7 | 5 | 50.6 | 5.0s |
| | | | $TACAI^{L2}$ | 35 715 | 15.7 | 5 | 50.6 | 6.3s |

Table 13.4 shows the results. The first three columns show the analyzed project and the number of its classes and methods, respectively. Column *representation* shows to which intermediate representation the values in the remaining columns belong. Those columns present the total number of instructions, the average instruction count per method, its median, and its standard deviation. The last column presents the time it takes to generate the intermediate representation.

Comparing the runtimes reveals that $TACAI^{L0}$, $TACAI^{L1}$, and $TACAI^{L2}$ are computed significantly faster than *Shimple*. One exception is *javacc* where $TACAI^{L2}$ took slightly longer to be generated than *Shimple*. The best speedup of $TACAI^{L2}$ compared to *Shimple* is more than 4.5x, achieved on *proguard*.

Table 13.5: Runtime of Different Call-Graph Algorithms

| | CHA | | | RTA | | | CFA | |
|---|---|---|---|---|---|---|---|---|
| **Project** | **Soot** | **WALA** | **Unimocg** | **Soot** | **WALA** | **Unimocg** | **WALA** | **Unimocg** |
| jasml | 21 s | 7 s | 37 s | 110 s | 532 s | 14 s | 15 s | 17 s |
| javacc | 24 s | 7 s | 35 s | 121 s | 518 s | 15 s | 16 s | 20 s |
| jext | 52 s | 7 s | 36 s | 430 s | 513 s | 18 s | 2 176 s | 25 s |
| proguard | 52 s | 7 s | 37 s | 406 s | 540 s | 15 s | 18 s | 19 s |
| sablecc | 23 s | 7 s | 39 s | 134 s | 509 s | 14 s | 17 s | 19 s |
| average | 34.4 s | 7.1 s | 36.6 s | 240.1 s | 522.2 s | 15.2 s | 448.4 s | 19.9 s |

**Observation 13.6**

Transforming bytecode using abstract interpretation in *TACAI* is feasible and faster than the generation of *Shimple*, even when $TACAI^{L2}$, our most precise configuration, is used (about half the mean runtime).

**Observation 13.7**

The overhead of $TACAI^{L1}$ compared to $TACAI^{L0}$ is almost negligible. Computing the more precise information in $TACAI^{L2}$ takes 70% more time on average. However, when the extra information (e.g., nullness) provided by $TACAI^{L1}$ and $TACAI^{L2}$ are required by an analysis, this time consumption is justified.

### 13.4.2 Unimocg

After we have already shown that *Unimocg*'s consistently high soundness does not come at the expense of imprecision (Section 12.2.2), we also validate whether it does not come at the price of scalability.

We compare the performance of three call-graph algorithms—CHA, RTA, and 0-CFA—which are available in Unimocg, *Soot*, and *WALA* (0-CFA only for *WALA* and Unimocg) by running them on the same five Java applications used for the evaluation of *Unimocg*'s precision. We report analysis runtime in seconds as the median of three executions on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. We used the Adoptium OpenJDK 1.8.0_342-b07 that worked with all frameworks.

Table 13.5 shows the results (*Unimocg*'s columns are labeled OPAL for brevity), evidencing that *Unimocg*'s call-graph algorithms do not suffer scalability degradations, either. The most notable finding is the difference in RTA performance. *Unimocg*'s RTA is on average 18x and 40x faster than the RTA of *Soot*, respectively *WALA*. This is noteworthy because to enable reusing call-graph modules, e.g., for the one resolving reflection *Soot* and *WALA* emulate RTA by a points-to analysis. Our evaluation indicates that this approach seems to come at a significant performance cost. *Unimocg*'s 0-CFA shows similar performance as *WALA*'s 0-CFA across applications in the benchmark except for *jext*, on which *WALA*'s 0-CFA took more than 45 minutes. *Unimocg*'s CHA is on average comparable to that of *Soot*'s CHA, but both are significantly slower than *WALA*'s CHA. We attribute this to OPAL's intermediate representation, which employs abstract interpretation [188] to provide refined type information but needs more computation time per reachable method; *Soot*'s Jimple [239], while not based on abstract interpretation, offers similar information.

We conclude that *Unimocg*'s modular architecture does not compromise performance. This is explainable: the main indirection we add to OPAL is cheap—two method calls on the type-iterator object.

**Observation 13.8**

*Unimocg*'s modular architecture does not compromise on performance; this is indicated by the comparison to state-of-the-art frameworks.

**Observation 13.9**

Crucially, *Unimocg* enables reuse of modules across different algorithms without relying on inefficient emulation; this benefit is indicated by *Unimocg* significantly outperforming *Soot* and *WALA*'s RTA implementations, which are emulated by points-to algorithms.

#### 13.4.2.1 *Comparison with Declarative Approaches*

Additionally, we compare *Unimocg*'s scalability to the *Doop* framework [34]. *Doop* is a highly optimized state-of-the-art tool for declarative Java points-to and call-graph analyses on top of the *Soufflé* [118] Datalog engine. Its declarative approach supports similar modularity and configurability and good trade-offs between pluggable precision/recall. Also, *Doop*'s and *Soufflé*'s authors repeatedly claimed its good scalability [33, 34, 118, 215]. Specifically, we compare our 0-CFA call graph's runtime from Chapter 8 to *Doop*'s.

For better comparability, we disabled the reflection support in both tools, because the respective approaches are different. The applications were analyzed together with OpenJDK 1.7.0_75 (used for the *TamiFlex* data in *Doop*'s benchmarks). Minor differences (less than 5% difference in the number of reachable methods, except for *eclipse* and *xalan*) remain, but these are in *Doop*'s favor since they result in more work

Table 13.6: Runtime and Size of Points-To-Based Call Graphs

| | **OPAL** | | **DOOP** | | | |
|---|---|---|---|---|---|---|
| **Program** | ⏲ | #RM | Compile | Facts | Analysis | #RM |
| antlr | 28.36 s | 8 653 | 107 s | 35 s | 41 s | 8 402 |
| bloat | 34.43 s | 10 000 | 109 s | 21 s | 33 s | 9 644 |
| chart | 40.13 s | 12 268 | 109 s | 38 s | 45 s | 12 058 |
| eclipse | 44.89 s | 13 429 | 109 s | 19 s | 17 s | 7 163 |
| fop | 18.87 s | 7 509 | 110 s | 41 s | 35 s | 7 300 |
| hsqldb | 19.65 s | 7 455 | 109 s | 38 s | 32 s | 7 097 |
| jython | 77.65 s | 13 161 | 108 s | 24 s | 90 s | 12 901 |
| luindex | 19.34 s | 7 972 | 108 s | 21 s | 19 s | 7 608 |
| lusearch | 21.03 s | 8 540 | 108 s | 21 s | 20 s | 8 281 |
| pmd | 21.47 s | 9 028 | 109 s | 39 s | 36 s | 8 817 |
| xalan | 35.59 s | 13 330 | 108 s | 37 s | 30 s | 7 111 |
| geo. ∅ | 29.68 s | | 108.54 s | 29.09 s | 32.51 s | |

#RM = number of methods found to be reachable

to be done by *OPAL*.[2] Still, the sixth column of Table 13.6 shows that our complete analysis, including all preprocessing, is often faster than *Doop*'s analysis (9% in the geometric mean). Further, *Doop* additionally requires time for rule compilation and fact generation.

We used *OPAL*'s single-threaded implementation since it seems that *Doop* is hardly parallelized (fact generation was done with 128 threads but did not significantly vary with other values for the `fact-gen-cores` parameter; the `souffle-jobs` parameter did not show any effects at all). Using a parallel version, *OPAL* should be able to outperform *Doop* even more as shown in Section 13.1.

**Observation 13.10**

Despite being more general, i.e., not tuned for points-to analyses but supporting many different kinds of analyses, *OPAL* clearly outperforms *Doop* (half the runtime compared to *Doop*'s fact generation and analysis).

### 13.4.3 OPIUM

To evaluate the scalability of *OPIUM* (cf. Chapter 10), we report runtime results for the same two real-world applications as in Section 12.4.2—*batik* and *xalan* from *XCorpus*—and compare them to *ReIm*, the state-of-the-art tool for purity analysis. The evaluation was per-

[2] For instance, *OPAL* does handle some cases of reflection more soundly even with reflection handling disabled in order to process the *DaCapo* benchmark correctly.

formed on a Mac Pro with a Xeon E5 CPU with 8 cores @ 3GHz. The JVM was given 24 GB of heap space.

Table 13.7: At Least Side-Effect-Free Methods in Batik/Xalan

| **Program** | **Batik** | **Xalan** |
|---|---|---|
| **ReIm** *#Analyzed methods* | 16 029 | 10 386 |
| *At least Side-Effect-Free Methods* | 6 072 (37.88%) | 3 942 (37.95%) |
| *Execution time (seconds)* | 103 | 140 |
| **OPIUM** *#Analyzed methods* | 15 911 | 10 763 |
| *At least Side-Effect-Free Methods* | 6 780 (42.61%) | 4 390 (40.79%) |
| *Pure methods* | 4 009 (25.20%) | 2 492 (23.15%) |
| *Ext./Context. Pure/SEF methods* | +987 (6.20%) | +748 (6.95%) |
| *Execution time (seconds)* | 103 | 104 |

Table 13.7 extends Table 12.8 with the respective execution times. We executed *OPIUM* with 8 threads, leading to execution times of 103*s* for *batik* (the same time as *ReIm*) and 104*s* for xalan (26% less time than *ReIm*'s 140*s*). At the same time, our analysis derives more fine-grained results and identifies a significantly higher number of side-effect-free methods than *ReIm* (cf. Section 12.4.2).

**Observation 13.11**

*OPIUM* scales to large projects: the execution time is less than two minutes for ≈170 000 methods—including the JDK and library dependencies—and similar or lower than *ReIm*, the previous state of the art.

### 13.4.4 *IFDS Solver*

Finally, we compare our *RA2* IFDS analysis to several other state-of-the-art systems using the same setup as in Section 13.3.

First, we compare the scalability and speed-ups achieved to the *Heros*[3] [104] IFDS solver. We used *Heros* because it is parallelized and also independent of the static analysis framework. It is very mature, widely used, and freely available. As we also compared to the IFDS solver with *OPAL*'s sequential blackboard implementation and as our analysis is based on our three address code representation (cf. Chapter 7), we again used this representation as the basis for our analysis in *Heros*. By using the exact same base technology stack for both analyses, we ensure that we compare the raw performance of

[3] Commit id: 46dda652

the solvers and that the results are not skewed by other technical differences. The IFDS client analysis was adapted to *Heros*' interfaces but performs the same analysis. As shown in Figure 13.2, *Heros* had lower speed-ups than our best strategy (which is also the one with the lowest speed-ups) for all thread counts, with a maximum of 2.36x at 8 threads. In comparison, our best strategy had a speed-up of 3.53x at 8 threads. Using more than 8 threads, *Heros*' performance decreased significantly, while *RA2*'s performance increased until 16 threads (with a speed-up of 3.98x for the best strategy) and did not decrease significantly for 20 threads. Relative standard deviations for Heros were between 2.1% and 5.7%.

A direct empirical comparison with a parallelized implementation of IFDS using Actors [192] is unfortunately not possible. The solution was never publicly available and—according to the authors whom we contacted—is now practically impossible to get working again due to dependencies on unavailable and outdated beta versions of libraries. However, they have also benchmarked their solution against a sequential implementation and we compare their speed-ups with our speed-ups against the sequential implementation using *OPAL*. The authors of *IFDS-A* reported a speed-up of 3.35x on 16 threads on eight cores compared to their own sequential implementation. Our implementation, on the other hand, using again 16 threads (on 10 cores), achieves a speed-up of 3.11x over *OPAL*'s highly optimized sequential implementation. Therefore, the scalability of our implementation seems to be comparable to theirs, while our programming model is analysis-independent and, therefore, not specialized to IFDS. Among other things, the fact that two different baselines are used makes it obvious that this comparison must be considered with caution and only as a workaround because a real comparison is not possible.

Finally, we evaluate against *WALA* 1.5.2 [111] as it provides another mature single-threaded IFDS implementation. As with *Heros*, we adapted the IFDS client analysis, this time with more effort because *WALA*'s IFDS solver is not framework-independent. The analysis was, however, thoroughly checked to be equivalent to the one used with *RA2* and *OPAL*. To overcome framework differences related to the underlying call graphs used by the different frameworks [187], we generated and serialized *WALA*'s RTA call graph using Judge [186] and deserialized it with *OPAL*. This ensures that the analyzed state space is equal. As *WALA* timed out after 10 hours when analyzing the JDK, we performed a comparison with *WALA* on *javacc*. For this setup, *WALA* took 12.7 seconds, while *RA2*, using the `DefaultScheduling` strategy, took 4.2 seconds on one thread and gave a speed-up of 4.5x (0.9 seconds) using 16 threads. `TargetsWithManySourcesLast`, *RA2*'s best strategy, took 0.70 seconds on 16 threads.

**Observation 13.12**

*RA2* outperforms state-of-the-art parallel IFDS solvers.

**Observation 13.13**

Compared to *Heros*, our implementation is faster on one thread, achieves higher speed-ups and scales to more threads.

**Observation 13.14**

It also outperforms *WALA*'s IFDS solver significantly and seems to provide at least similar speed-ups as a specialized actor-based IFDS solver, despite being semi-implicit and analysis-independent.

## 13.5 SUMMARY

In this chapter, we answered **RQ3** by showing how *OPAL*'s architecture allows improving the scalability of static analyses implemented in *OPAL*. We showed that our case-study analyses are on par with or outperform respective state-of-the-art analyses, thanks in particular to parallelization and specialized data structures. Scheduling strategies also impact the scalability of analyses in *OPAL* and an appropriate choice of strategy can make significant differences in execution time. While a specialized strategy for a simple purity analysis did not offer benefits over generic strategies, we have shown that such specialized strategies can be used and show potential for future research.

Part IV

# CONCLUSION

In this final part, we wrap up this thesis with a summary of the results, ideas for future work, and a conclusion. The individual chapters of this part are:

**SUMMARY OF RESULTS** Chapter 14 summarizes the main contributions and findings of this thesis.

**FUTURE WORK** In Chapter 15, we present ideas for possible extension of this work and future research directions. In particular, we discuss possibilities to further extend the applicability of our blackboard analysis architecture and possible ways to improve the scalability of analyses implemented in it.

**CLOSING DISCUSSION** Chapter 16 concludes this thesis with a closing discussion, putting the thesis into the context of current and future research.

# 14 SUMMARY OF RESULTS

In this thesis, we presented a novel approach to enable static analysis frameworks to cater to modular, collaborative analyses. We built this approach based on the blackboard architecture that allows fully independent modules to interact closely in solving a common problem. We showed the feasibility of this approach using two distinct implementations. We also implemented several case-study analyses, that in their own right advance the respective research areas, and evaluated them against the respective state of the art. They showcased the broad applicability of our approach and are on par with or outperform the state of the art in terms of soundness, precision, and scalability. Our individual contributions are:

BLACKBOARD ANALYSIS ARCHITECTURE We proposed the blackboard analysis architecture, a novel architecture for static analysis frameworks. It builds upon the blackboard architecture that was originally invented for problems of artificial intelligence. Independent modules communicate intermediate results via a central data storage, the blackboard, and improve upon other modules' results until the overall problem is solved. This allows each module to be implemented in the optimal way to solve a particular sub-problem and to freely combine the necessary modules to achieve the overarching goal.

We started out by distilling the requirements to a modular, collaborative static analysis framework from a series of case-study analyses (Chapter 2). Based on these requirements, we systematically developed the blackboard architecture and described its implementation in the *OPAL* static analysis framework (Chapter 3).

Like with declarative frameworks such as *Doop*, *OPAL*'s analyses, while developed in isolation, can be easily composed into complex analyses by collaboratively computing results during interleaved executions. Sub-analyses can be reused in various complex analyses and one can easily exchange sub-analyses of a complex analysis for fine-tuning precision, sound(i)ness, and scalability.

But, instead of relying on a general-purpose solver, *OPAL* combines imperative and declarative features to overcome limitations of fully declarative frameworks. Individual analyses can be implemented in an imperative style making use of whatever data structures and implementation strategies are appropriate for their specific needs. Interdependencies and other characteristics important for guiding their interleaved execution are specified declaratively and managed automatically by a custom solver. Due to its approach, *OPAL* (a) is more general in terms of the analyses supported—it is in particular the first framework to explicitly support lazy collaboration of optimistic and pessimistic analyses—and (b) enables analysis-specific optimizations, which lead to outperforming state-of-the-art declarative analyses.

*OPAL* is available under the open-source BSD 2-clause license and is the basis for our implementations of the case-study analyses and for our evaluation.

RA2 In addition to the implementation in *OPAL*, we developed *RA2*, an alternative implementation of our approach (Chapter 4). This showcases that the blackboard analysis architecture can be realized using different underlying concepts and thus is flexible enough to be included in frameworks based on different paradigms.

*RA2* is a new programming model for parallelizing static analyses based on the ideas and concepts of reactive programming, that (a) is semi-implicit, requiring only minor adaptions to analyses to benefit from parallelization, and (b) is analysis-independent, lending itself very well towards the implementation of a wide range of static analyses, including purity and data-flow analyses. *RA2* also provides support for using different scheduling strategies.

We used *RA2* to evaluate the impact of different scheduling strategies and the scalability of parallel execution of analysis tasks in the blackboard analysis architecture, showing promising results that should be expanded upon in future research.

MODULAR SOUNDNESS PROOFS We gave a formal definition of the core concepts of our approach (Chapter 5) and developed a theory for compositionalsoundness proofs for static analyses implemented in our blackboard analysis architecture. We proved that soundness of an analysis follows directly from independent soundness proofs of each module. Furthermore, we extended our theory to enable the reuse of soundness proofs of existing modules across different analyses. We evaluated our approach by implementing four analyses and proving them sound: A pointer, a call-graph, a reflection, an immutability analysis, and a demand-driven reaching definitions analysis.

This narrows the gap between impractical academic analyses, which can be proven sound, and useful applied analyses, which are too complex to be proven sound. We believe that some complexity of applied analyses is incidental and can be better managed by modularizing their implementation, which makes a compositional soundness proof more feasible. This is a significant step forward from traditional techniques for modular soundness proofs that often require analysis modules to be tied together by additional glue code which must be reasoned about and from techniques that require modules to follow a particular structure, such as being derived from a generic abstract interpreter.

UNIMOCG Using the flexibility provided by the blackboard analysis architecture, we developed *Unimocg* (Chapter 8), a novel architecture to build call graphs using independent modules for computing type information and for resolving calls resulting from distinct programming language features such as reflection or (de)serialization. Traditional call-graph algorithms suffer from the conflation of these two concerns, resulting in vastly different soundness with respect to the support of different language features. Thus, modular call-graph construction that decouples the computation of types of local variables from the resolution of call targets is sorely needed.

*Unimocg* solves this issue by separating the resolution of calls into individual modules corresponding to individual language features. This decoupling enables the modular composition of different analyses that contribute to both type computation and call resolution, making it possible to model different language features and APIs in individual modules. These analyses can then be reused fully across a multitude of different call-graph algorithms. With individual modules, feature support can be implemented and reasoned about in isolation. This is necessary to facilitate support for a multitude of such features that are relevant to call-graph construction. As a result, users of call graphs can rely on consistent feature support and analysis developers can easily add new algorithms or language features while reusing existing components. A common interface, the *type iterator* provides the connection between type resolution on the one hand and call resolution and further analyses dependent on type information (e.g., immutability analyses) on the other hand, retrieving and interpreting type information gathered by the call-graph algorithm and stored in the blackboard. Within *Unimocg*, we implemented ten different call-graph algorithms from vastly different families of algorithms: CHA, RTA, the XTA family including MTA, FTA, and CTA as well as several *k*-*l*-CFA-based algorithms.

With this architecture, we have shown that it is possible to implement call-graph algorithms from different families with a shared, consistently sound support for individual language features, improving significantly over the state of the art. We also found that this consistent soundness does not come at the expense of either precision or scalability compared to the state of the art in call-graph construction.

CIFI Next, we presented *CiFi*, a holistic and unified model for the analysis of field-, class-, and type immutability (Chapter 9). Immutability analysis can help developers in catching bugs and adhering to secure coding guidelines, and it can also provide vital information to other analyses such as purity analyses. In addition to concepts found in the literature, we also explicitly consider lazy initialization of fields and immutability that depends on the concretization of generic type parameters.

We show that *CiFi* is more expressive than state-of-the-art approaches to immutability inference and checking and also introduce *CiFi-Bench*, a handcrafted benchmark tailored to test and evaluate immutability analyses. Using this, we confirm that *CiFi* is both more sound and more precise than the state-of-the-art immutability checker *Glacier*.

*CiFi* benefits from *OPAL*'s blackboard architecture by being separated into four different modules for field assignability and field-, class-, and type immutability. This allows exchanging each module individually to optimize or trade off soundness, precision, and scalability.

OPIUM As our final case-study analysis, we developed *OPIUM*, three purity analyses of different precision and scalability characteristics (Chapter 10). *OPIUM* is built upon a novel, fine-grained, unified model of purity that simultaneously captures the absence of side effects as well as deterministic behavior methods for the first time. It captures all use cases found in the literature and provides precise definitions of all purity levels. Additionally, we introduced the concepts of *contextual purity*—an extension of *external property* that allows finding more cases of confined side effects—and *domain-specific purity* to capture side effects that are not relevant to the end user's use case. Using *OPIUM*, we find that all of these purity levels occur in real-world software.

We compared *OPIUM* to the state of the art in purity inference and found that *OPIUM* was significantly more precise than all competitors, identifying more methods without side effects, differentiating from them methods that also behave deterministically, and also find additional methods with confined side effects. At the same time, using *OPAL*'s parallel fixed-point solver, *OPIUM* was on par or outperformed the state-of-the-art purity analysis ReIm on real-world software.

*OPAL* allows *OPIUM*'s different analyses to be exchanged in a plug-and-play manner to explore trade-offs between precision and scalability and also enables *OPIUM* to use the information computed by all of our other case-study analyses as well as additional escape analyses.

# 15 FUTURE WORK

This thesis has laid out a novel framework architecture for modular, collaborating static analyses. Still, there are several directions in which this work can be extended upon, some of which we discuss in this chapter. In particular, we discuss possible ways to broaden the scope of analyses and present several interesting ideas on how the scalability of analyses could be improved significantly.

## 15.1 EXPANDING FRAMEWORK SCOPE AND APPLICABILITY

We demonstrated that *OPAL*'s blackboard architecture allows implementing static analyses from dissimilar domains. However, all of our case-study analyses are still implemented in Scala in a similar, functional-imperative style, and they all are tailored to analyze programs compiled to Java Virtual Machine bytecode. We believe that our blackboard architecture can be the foundation for a much broader set of analyses and sets of analyses.

MULTI-PARADIGM ANALYSES *OPAL*'s blackboard architecture poses little requirements to analyses: it must be possible to invoke them for a single property, and, if they have dependencies, they must be able to specify them and provide a continuation function to handle updates of dependencies. As no requirements are posed on the internal structure of analyses, it is presumably possible to implement analyses using different paradigms and still have them interact using the blackboard. Analyses can be agnostic of the way other analyses they depend upon are implemented. Possible paradigms to implement analyses could include Datalog-based declarative analyses, analyses based on SMT solving, or machine-learning-based analyses, significantly broadening the scope of analyses possible in *OPAL*. Connecting static to dynamic analyses would enable further use cases but also poses additional problems, as dynamic analyses can not simply be suspended and resumed or directed towards computing specific properties. However, we have already experimented with using data from a dynamic analysis by implementing a module for our call-graph construction framework *Unimocg* (cf. Chapter 8) that parses data on reflective calls produced by the Tamiflex ([27]) dynamic analysis. On the other hand, we believe that it is possible to use *OPAL*'s static analyses on demand during the execution of a dynamic analysis. Whether a full connection allowing cyclical dependencies between static and dynamic analyses is possible remains to be seen in future research.

CONNECTING EXISTING ANALYSES A significant problem in constructing complex analyses from smaller building blocks is providing these building blocks. Implementing sound, precise, and scalable static analyses is a laborious task, requiring deep knowledge of the questions to be analyzed. It also involves extensive engineering. While *OPAL*'s modular architecture makes implementing complex analyses easier by enabling them to be encoded in independent modules that can be implemented by different experts at different times and that can be improved upon or replaced at any time independent of other analyses, the individual modules currently have to be implemented explicitly for *OPAL*.

With the few requirements posed by the blackboard architecture, we assume that it is possible to connect existing analyses to *OPAL* without rewriting them from scratch. Instead, depending on the structure of the existing analysis, it may be possible to either directly invoke the analysis with minimal glue code or to modify only the analysis execution scheme, e.g., a worklist algorithm or fixed-point solver, to conform to *OPAL*'s requirements. In either case, the actual analysis code would not need to be re-implemented, saving developer time and effort. Pursuing this would be particularly helpful if complete frameworks could be connected to work in lockstep with *OPAL*, making all existing analyses in these frameworks available to be combined with *OPAL*'s analyses.

CROSS-LANGUAGE ANALYSES Modern software systems often make use of multiple programming languages for different purposes. They might use a scripting language like JavaScript for the front end and a compiled language like Java for the back end, potentially using native code written in a systems programming language like C or C++ for low-level tasks. Analyzing such hybrid software systems is hard, because semantics between programming languages differ.

Traditionally, for hybrid software systems only parts of the software written in a single language are analyzed, treating the effects of other parts either by ignoring, conservatively over-approximating, or manually modeling them. It is also possible to execute analyses for different parts of the software system sequentially, using the output of the analysis of one part as an input to the analysis of the next part. This may need to be repeated in order to deal with cyclic dependencies, e.g., data flows going back and forth between the front and back end. Another approach is to use an intermediate representation to which all parts of the software system are transformed. This does work well for some languages, e.g., languages like Java, Kotlin, or Scala all readily compile to Java Bytecode and many languages can be compiled to LLVM Bitcode, but it is difficult to create an intermediate representation that can both fully represent the semantics of all programming languages to be analyzed and to still be reasonably easy to analyze.

Using *OPAL*'s blackboard architecture, analyses for different languages could be executed simultaneously using the language-agnostic blackboard and fixed-point solver. Analyses communicate solely via values from explicitly reified lattices, enabling analyses for different languages to cooperate as long as the chosen lattice does faithfully represent the respective semantics of all languages in question. Designing individual lattices that represent the semantics of different languages would be easier than a full intermediate representation that needs to capture all semantics of each language.

In order to analyze hybrid software systems, it is of particular interest to avoid re-implementing analyses that already exist for these languages (see the paragraph above). Even if existing analyses use different lattices, small glue analyses can be used to translate information from one lattice to the other. This can be used in conjunction with *OPAL*'s option to trigger analyses based on the computation of particular properties (cf. **R10**) in order for analyses to trigger each other while being fully unaware of each other. A proof-of-concept adapter is currently being developed to connect to IFDS analyses in the *WALA* framework [111], enabling taint analysis of JavaScript code. Work is in progress to connect the *Sturdy* library [122] to *OPAL*, which would enable reusing its analyses including analyses for languages beyond Java Bytecode such as WebAssembly.

## 15.2 FURTHER SCALABILITY IMPROVEMENTS

While our case-study analyses show on-par or even superior performance compared to respective state-of-the-art analyses, we see several possible ways to improve runtime- as well as memory performance.

IMPROVED SCHEDULING We described several generic scheduling strategies as well as one optimized for a particular analysis in Section 4.4 and evaluated them in Section 13.3. While these scheduling strategies showed promising results in improving scalability, further research is necessary to find optimal strategies for different analysis scenarios. Such research should study further scheduling strategies as well as heuristics for selecting optimal scheduling strategies for both an individual analysis as well as a set of analyses executed concurrently. In particular, the selection of a scheduling strategy should be done automatically by the analysis framework to alleviate the end user from manually selecting a suitable strategy. Scheduling could further be improved if several scheduling strategies could be used at the same time, applied to different analyses executing concurrently.

FULL LAZINESS *OPAL* supports analyses that are lazy in the sense that they are only executed to compute some property if that property is actually required either by the end user of the analysis or by another analysis. However, once the computation of a property has been started, it is never stopped before the maximum fixed point of all analyses is reached. This is not strictly necessary, especially for properties required only by another analysis: The requesting analysis may not be interested in the full result of the property and instead be able to complete its own computation with only a partial result. Consider, for example, a purity analysis (cf. Chapter 10) requesting information on whether a value escapes the analyzed method. There may be no need for the purity analysis to distinguish whether the value only escapes to the immediate caller or potentially to any point in the program, so no further computation is required once the escape analysis has concluded the former to be possible. In a different scenario, an analysis might only care for a particular property dependent on the value of some other property. For the example of the purity analysis, an access to a field might impede purity only if either the field is mutable or the field's parent object is not local (i.e., locally instantiated and non-escaping) to the current method. In this case, computation of either property is no longer needed once the other property has already been determined to be true.

Shortcutting computations that are not at all or no longer required could save significant computation resources if these cases appear frequently. Thus, we suggest exploring how the blackboard and fixed-point solver can be extended to track which properties must still be computed and eschew invoking continuation functions for properties that need not be computed further. The blackboard and fixed-point solver already have the necessary information at hand, as analyses always report the full set of dependencies that are still to be resolved. Going further, it might also be possible to indicate already in the initial query to the blackboard whether the computation of the queried property is only required up to a certain point so that the analysis responsible for this computation can avoid further computation already during the initial analysis. Whether the performance overhead of tracking and checking which dependencies still exist is worth the possible benefit is an open question for future research.

ANALYSIS BATCHING So far, analyses in *OPAL* are all executed concurrently. If the end user wishes to execute some set of analyses only after computing the fixed point of a different set of analyses, they have to manually execute two batches of analyses sequentially. However, such batching of analysis execution can be done automatically based on the analyses' declared dependencies. This could positively impact scalability as it avoids repeated invocation of continuation functions for analyses that depend on other analyses in a non-cyclical manner. It could also enable the use of different scheduling strategies for different analysis batches if automated selection of scheduling strategies is implemented as described above. On the other hand, small batches of analyses executed could severely limit the number of tasks available for parallel execution, reducing the performance benefit achievable from utilizing multicore processors. Also, this conflicts with the idea proposed in the paragraph above and with lazy computation in general, as starting or stopping property computations is not possible if the requesting analysis is only executed in a later batch. A preliminary implementation to perform automated analysis batching exists in *OPAL*, but further research is necessary to study its benefits and drawbacks.

STORE CLEANUP When executing a set of analyses in *OPAL*, typically many of the analyses' results are not of direct interest to the end user but computed to support the computation of other properties, which in turn either have been requested by the end user or contribute to yet other properties. As an example, consider the purity analysis from Chapter 10: it depends on a call graph (cf. Chapter 8) but not necessarily on auxiliary data used in the computation of that call graph, such as points-to sets. Such supporting properties need not be kept in the blackboard's data store until the computations fixed-point is reached but could be discarded once they can no longer be of use. With many properties computed only for other analyses, this could free significant amounts of memory for use by later analyses. Implementing this would not only require the end user to specify which analysis results they are actually interested in but also require batching of analysis (see previous paragraph) as properties can only be purged from the blackboard once no analysis that might depend on them can be executed anymore.

IMPROVED PARALLELIZATION Both our implementations of Chapter 3 and Chapter 4 include fixed-point solvers that execute different analysis functions in parallel. However, both are only proof-of-concept and not yet highly optimized. In particular, both use simple worklist algorithms, distributing tasks between a set of equal threads. This means that computations pertaining to a single property can be executed by different threads at different times. While not necessarily problematic in systems with uniform memory access times—we believe the effects of caching to be insignificant due to typical analysis data working sets easily exceeding cache capacities by orders of magnitude—, this could create excessive long-latency memory traffic on non-uniform memory access (NUMA) or distributed systems that might be necessary to scale beyond double-digit thread counts. As a potential remedy, we suggest that it might be possible to bind computations for each property to specific threads, borrowing from the ideas of distributed hashtables. However, it is necessary to communicate the dependencies between property computations and the results of dependent computations, creating additional overhead while the potential for parallel execution is reduced if tasks can not be executed anywhere. Thus, particular benefits could be achieved if computations for properties with many interdependencies were kept on the same thread/computation node to minimize communication overhead. Whether this is achievable should be answered in future research.

PARTIAL ANALYSIS AND RESUMPTION A major problem in scaling static analyses to large software systems and to the analysis of whole repositories like Maven Central or the Google Play store is the amount of code to be analyzed. Most of this code is found in third-party libraries that applications include directly or transitively (e.g., Wang et al. [240] and Orikogbo et al. [175] found that Android respectively iOS apps consisted of about 60% code from libraries and only 40% actual application code). This opens up the potential to analyze each library only once, reusing results for the subsequent analysis of multiple applications that make use of the library. We already demonstrated that *OPAL* supports deserializing properties that have been determined manually or by a previously executed analysis; e.g., *OPIUM* uses manual annotations for a number of important native methods from the Java Development Kit.

However, for many analysis problems, the overall result is more than just the combination of partial results. Call graphs, for example, can have cyclic dependencies between library and application code; thus it is not possible to just provide a call graph for all used libraries when analyzing an application, but it is necessary to resolve further calls also inside the library code. We suppose that it is possible to support such use cases in the blackboard architecture by serializing not only the final results of a partial (e.g., library) analysis but also enough of its state to support the analysis' resumption later on. For a call-graph analysis, this can mean serializing computed type sets and the set of potentially polymorphic call sites. Preliminary research into this direction has already been performed [126].

Even more challenging would be the resumption of analyses to support incremental computations. In this case, not only new properties have to be computed, already computed properties can become invalid as well. A possible way to deal with this could be the addition of a cleanup phase that purges invalidated properties from the blackboard before the analysis is resumed. This, however, may require additional data on the dependencies between properties in order to invalidate all properties that depended on other invalidated properties as well.

# 16 CLOSING DISCUSSION

The work presented in this thesis is closely related to trends and challenges in current and future research and software engineering. Static analyses are an important tool in ensuring software quality and security, in providing developers and end users with insights into their applications, and in performing optimizing compilation. With the growing amount of software being produced and used, and with software used with growing ubiquity in our everyday lives and in contexts where software failures or vulnerabilities can threaten entire economies and the lives of millions of people, static analysis is only going to be ever more important.

As a result, the challenges for static analyses are increasing: They must be sound in order not to miss critical vulnerabilities or lead to incorrect optimizations. They must be precise in order to not overwhelm developers with false-positive results and to open up as many opportunities for optimization as possible. And finally, they must be scalable in order to cope with the ever-increasing amount of software written, distributed, and used every day.

These three qualities are often in conflict, and thus, trade-offs between them must be made, fine-tuned to different use cases. Experimenting with and fine-tuning such trade-offs is not easily possible with monolithic analyses that follow a one-size-fits-all approach. Instead, recent trends show interest in modularized static analyses, where modules that exhibit different qualities can be added, removed, or exchanged to adapt the overall analysis to the use case at hand. One example for this trend are modularized abstract interpreters that enable capturing different aspects of the analyzed programming language in distinct modules. This allows for analyses for different languages to be combined from the modules applicable, and at the same time, allows proving the soundness of the overall analysis from simpler soundness proofs of individual modules. Other works have focused on modularizing specific types of analyses such as alias analysis.

Declarative static analyses based on Datalog are another recent development that brings some modularity by design—Datalog rule sets can be exchanged to fine-tune analyses. However, traditional Datalog only supports set-based relations, limiting expressivity for static analyses. Extensions that allow the use of arbitrary lattices have not yet gained widespread adoption. Despite relying on highly optimized solvers and manual as well as automated optimization of rules, Datalog analyses also show inferior scalability compared to imperative approaches.

We have introduced *OPAL*'s blackboard analysis architecture, a novel, generic approach to modular, collaborative static analyses. Based on requirements distilled from a series of dissimilar case studies, we built *OPAL* to pose few restrictions on analyses and support the implementation of a broad range of static analyses without focusing on a particular problem domain. The blackboard architecture enables dissimilar analyses to collaborate while being ignorant of each other, which allows composing intricate systems of analyses from a multitude of modules. This enables more complex analyses that achieve more soundness and/or more precision than respective state-of-the-art analyses, and it facilitates fine-tuning trade-offs between these qualities.

In order to achieve scalability on par or superior compared to the state of the art, the blackboard architecture allows for automatic, semi-implicit parallelization of analyses to utilize the resources of modern multicore processors. Support for different strategies for scheduling the execution order of analysis tasks can improve scalability as our evaluation has shown, as does the ability to use optimized data structures.

We presented four case-study analyses to showcase *OPAL*'s broad applicability. Each of these analyses in its own right advanced the state of the art in their respective area of research: *TACAI* is an intermediate representation for Java Bytecode that uses abstract interpretation to provide important information for further analyses in a precise way, in particular for call-graph analyses. Research into sound call graphs and language feature support has seen a spike in recent years after decades of call-graph research mainly focusing on aspects of precision and scalability in the resolution of regular virtual calls. *Unimocg* addresses the issue of inconsistent soundness in the support of language features between different call-graph algorithms implemented in the same static analysis framework. Call-graph algorithms are the foundation of all interprocedural analyses and thus of vital importance for sound, precise, and scalable analyses that are meant to compute intricate properties. This applies, e.g., to immutability and purity analyses, both of which can be important tools to ensure software quality and security and to enable compiler optimizations. With *CiFi* and *OPIUM*, we provide precise, unified, and fine-grained definitions for immutability and purity properties, respectively. Both of these research areas had inconsistently used and vaguely defined terminology before, often catering to only a specific use case. Our analyses' flexible, modular implementation allows fine-tuning trade-offs and achieving superior soundness, precision, and scalability compared to the respective state of the art.

All of these case studies tie together and form an intricate system of analyses, again highlighting *OPAL*'s applicability to solving complex analysis problems by composing a multitude of distinct, dissimilar sub-analyses as shown in Figure 16.1:

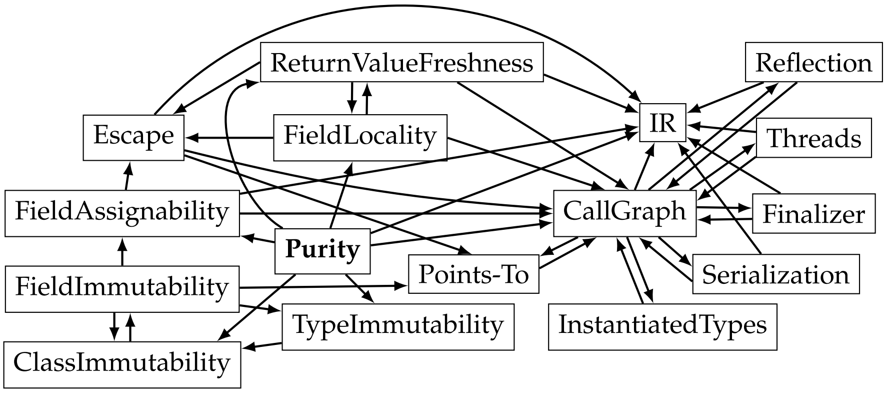


Figure 16.1: Dependencies Between Sub-problems for a Purity Analysis

Our case studies highlighted how *OPAL* enables dealing with three major challenges for static analyses: Complex language features that impact, e.g., call-graph construction, are handled using individual modules to capture one feature at a time. Complex analysis problems are solved by composing intricate systems of analyses from a multitude of pre-existing, independent sub-analyses. Complex trade-offs between soundness, precision, and scalability finally are balanced off and fine-tuned by providing a plug-and-play-like system where individual sub-analyses can be added, removed, or exchanged with different implementations with changing as little as a single line of setup code.

While our case-study analyses showed on par or better scalability than respective state-of-the-art analyses, we believe that it is possible to further improve the scalability of *OPAL*. This is imperative to scale complex analyses to the millions of applications available today, e.g., via repositories like Maven Central or app stores like Google Play and to analyze complex software that can not only consist of hundreds of thousands of lines of code but often include significant numbers of direct and indirect dependencies on third-party libraries. Only recently researchers have started to conduct large-scale analysis of such software repositories and to create tools to continuously monitor all changes and additions. Scalable analyses will be a major contributing factor to allow such projects to provide valuable insight into the vast amount of software we use and to discover vulnerabilities or malware in a timely manner.

Expanding the scope and applicability of static analyses is an even more important but also more challenging task for the future. As modern software systems consist of multiple distinct components, often implemented using different programming languages and frameworks, it will be necessary to develop static analyses that can fully capture such systems. This includes cross-language analyses that are able to simultaneously analyze connected software components written in different languages but also cross-paradigm analyses that combine the power and applicability of different static analysis and general problem-solving paradigms (e.g., machine learning or SMT solving) to answer ever more complex questions. It is vital that this does not require full (re-)implementation of every required sub-analysis but allows integrating existing analyses with each other as much as possible in order to benefit from decades of previous research and engineering.

Wrapping up, we conclude that *OPAL* fulfills this thesis' main goal:

> A general framework for modular, collaborative program analysis should allow for complex systems of analyses that offer good soundness, precision, and scalability, including exploring the trade-offs between these qualities.

Also, *OPAL*'s blackboard architecture helps to address the major challenges that static analyses will face going forward, from increasing demands on performance and scalability to the necessities of analyzing complex software systems.

Part V

# APPENDIX

## SOUNDNESS PROOFS OF THE CASE STUDIES FROM CHAPTER 5

We derive both the analysis modules and dynamic modules from the same parametric implementation[1]. The parametric modules can be found in file `GenericModules.scala`. Each case study is implemented in a separate trait containing the interface and the modules. Each module is parameterized and has a type according to Definition 5.11. The static and dynamic instances of the interfaces can be found in `AnalysisModules.scala` and `DynamicModules.scala` respectively.

Soundness of the generic modules follows as a free theorem of parametricity:

**Theorem V.1** (Soundness of Extensible Modules by Parametricity)**.** *Let $f$ be a generic module with type*

$$\begin{aligned} f : \forall E' \supseteq E,\ K' \supseteq K,\ P : K' \to \mathsf{Set}.& \\ F[P[K']] \Rightarrow E' \times \mathsf{Store}[E', K', P] &\to \mathsf{Store}[E', K', P]. \end{aligned}$$

*If $f$ is implemented in a language that enjoys parametricity, then for all $\widehat{E}', E' \supseteq E$, $\widehat{K}', K', \supseteq K$, $\widehat{P} : \widehat{K}' \to \mathsf{Lattice}$, $P : K' \to \mathsf{Set}$, $\widehat{I} \in F[\widehat{P}[\widehat{K}']]$, $I \in F[P[K']]$,*

$$\mathsf{sound}(I, \widehat{I}) \implies \mathsf{sound}(f[E', K', P](I), f[\widehat{E}', \widehat{K}', \widehat{P}](\widehat{I}))$$

*Proof.* Follows as a free theorem of parametricity. Module $f$ is parametric in the type of entities and kinds because types $E' \supseteq E$ can be viewed as a disjoint sum type $E + (E' \setminus E)$ which is discretely ordered. □

Hence, it suffices to prove soundness of the static instance $\widehat{I}$ and dynamic instance $I$ of the interface. The concrete and abstract domains and their orderings are defined in the code.

### *Soundness of Pointer Analysis and Call-Graph Analysis*

**Definition V.2** (Concretization Functions)**.** *We define the concretization functions on values and call targets:*

$$\begin{aligned} &\gamma : \widehat{Val} \to \mathcal{P}(\mathtt{Val}) \\ &\gamma(v) = \mathtt{NonAnalyzedValues} \cup \\ &\qquad\qquad \begin{cases} \{\mathtt{Obj}(o), \mathtt{NullVal} \mid o \in O\} & \textit{if } v = \widehat{Obj}(O) \\ \varnothing & \textit{if } v = \bot \end{cases} \\ &\ \textit{where } \mathtt{NonAnalyzedValues} = \\ &\qquad\quad \mathtt{BoolVal} \cup \mathtt{NullVal} \cup \mathtt{Val} \cup \mathtt{MethodVal} \cup \mathtt{ClassVal} \end{aligned}$$

[1] Implementation and tests available at https://doi.org/10.5281/zenodo.7311078

*The analysis does not analyze values such as booleans. To be able to prove boolean operations sound, the concretization function adds all boolean values to the set of concrete values.*

$$\begin{aligned} &\gamma : \widehat{CallTarget} \to \mathcal{P}(\texttt{CallTarget}) \\ &\gamma(\widehat{CallTarget}(Targets)) = \{\texttt{CallTarget}(target) \mid target \in Targets\} \end{aligned}$$ □

**Lemma V.3.** $\mathsf{sound}(\texttt{nullPointer}, \widehat{nullPointer})$

*Proof.* To show: $\{\texttt{nullPointer}()\} \subseteq \gamma(\widehat{nullPointer}())$

$$\{\texttt{nullPointer}()\} = \{\texttt{NullVal}\} \subseteq \gamma(\bot) = \gamma(\widehat{nullPointer}())$$ □

**Lemma V.4.** $\mathsf{sound}(\texttt{newObj}, \widehat{newObj})$

*Proof.* To show: $\{\texttt{newObj}(class, ctx) \mid ctx \in \gamma(\widehat{ctx})\} \subseteq \gamma(\widehat{newObj}(class, \widehat{ctx}))$ for all $class$ and $\widehat{ctx}$.

$$\begin{aligned} &\{\texttt{newObj}(class, ctx) \mid ctx \in \gamma(\widehat{ctx})\} \\ =\ &\{\texttt{Obj}(class, ctx) \mid ctx \in \gamma(\widehat{ctx})\} \\ =\ &\subseteq \gamma(\widehat{Obj}(\{class, \widehat{ctx}\})) \\ =\ &\gamma(\widehat{newObj}(class, \widehat{ctx})). \end{aligned}$$ □

**Lemma V.5.** $\mathsf{sound}(\texttt{forObj}, \widehat{forObj})$

*Proof.* Assume $\{f(class, ctx) \mid ctx \in \gamma(\widehat{ctx})\} \subseteq \gamma(\widehat{f}(class, \widehat{ctx}))$ for all $class$ and $\widehat{ctx}$. To show: $\{\texttt{forObj}(val)(f) \mid val \in \gamma(\widehat{val})\} \subseteq \gamma(\widehat{forObj}(\widehat{val})(\widehat{f}))$ for all $\widehat{val}$.
In case $\widehat{val} = \widehat{Obj}(X)$:

$$\begin{aligned} &\{\texttt{forObj}(val)(f) \mid val \in \gamma(\widehat{val})\} \\ =\ &\{f(className, ctx) \mid (className, \widehat{ctx}) \in X, ctx \in \gamma(\widehat{ctx})\} \\ \subseteq\ &\bigcup_{(className, \widehat{ctx}) \in X} \gamma(\widehat{f}(className, \widehat{ctx})) && \text{(by soundness of } \widehat{f}\text{)} \\ \subseteq\ &\gamma(\bigsqcup_{(className, \widehat{ctx}) \in X} \widehat{f}(className, \widehat{ctx})) && \text{(by monotonicity of } \gamma\text{)} \\ =\ &\gamma(\widehat{forObj}(\widehat{Obj}(X))(\widehat{f})). \end{aligned}$$

For all other $v \in \gamma(\widehat{val})$, $\texttt{forObj}$ returns a type error. □

**Lemma V.6.** $\mathsf{sound}(\texttt{newCallTarget}, \widehat{newCallTarget})$

*Proof.* To show: $\{\texttt{newCallTarget}(class, ctx, method) \mid ctx \in \gamma(\widehat{ctx})\} \subseteq \gamma(\widehat{newCallTarget}(class, \widehat{ctx}, method))$ for all $class$, $\widehat{ctx}$, and $method$.
Analogous to Lemma V.4. □

**Lemma V.7.** $\mathsf{sound}(\texttt{forCallTarget}, \widehat{\mathit{forCallTarget}})$

*Proof.* Assume that $\{f(\mathit{class}, \mathit{ctx}, \mathit{method}) \mid \mathit{ctx} \in \gamma(\widehat{\mathit{ctx}})\} \subseteq \gamma(\widehat{f}(\mathit{class}, \widehat{\mathit{ctx}}, \mathit{method}))$ for all $\mathit{class}$, $\widehat{\mathit{ctx}}$, and $\mathit{method}$. To show: $\{\texttt{forCallTarget}(\mathit{target})(f) \mid \mathit{target} \in \gamma(\widehat{\mathit{target}})\} \subseteq \gamma(\widehat{\mathit{forCallTarget}}(\widehat{\mathit{target}})(\widehat{f}))$ for all $\widehat{\mathit{target}}$.
Analogous to Lemma V.5. □

**Lemma V.8.** $\mathsf{sound}(\texttt{method}, \widehat{\mathit{method}})$.

*Proof.* Module $\widehat{\mathit{method}}$ recurses over all statements in a method and registers them in the store. Module `method` executes a subset of the statements in the order of the control flow. □

**Lemma V.9.** $\mathsf{sound}(\texttt{boolLit}, \widehat{\mathit{boolLit}})$ *and* $\mathsf{sound}(\texttt{equals}, \widehat{\mathit{equals}})$

*Proof.* To show $\{\texttt{boolLit}(b)\} \subseteq \gamma(\widehat{\mathit{boolLit}}(b))$ for all booleans $b$.

$$\{\texttt{boolLit}(b)\} \subseteq \gamma(\bot) = \gamma(\widehat{\mathit{boolLit}}(b))$$

The proof of $\widehat{\mathit{equals}}$ is analogous. □

### *Soundness Reflection Analysis*

The following proofs refer to the allocation information of the objects `String` ($\mathit{strAlloc} = \{(\texttt{String}, \varnothing)\}$), `Class` ($\mathit{classAlloc} = \{(\texttt{Class}, \varnothing)\}$), and `Method` ($\mathit{methodAlloc} = \{(\texttt{Class}, \varnothing)\}$).

**Definition V.10** (Concretization Functions).

$$\gamma : \widehat{\mathit{Val}} \to \mathcal{P}(\texttt{Val})$$
$$\gamma((\widehat{o}, \widehat{s}, \widehat{c}, \widehat{m})) := \texttt{NonAnalyzedValues} \cup \gamma(\widehat{o}) \cup \gamma(\widehat{s}) \cup \gamma(\widehat{c}) \cup \gamma(\widehat{m})$$
$$\textit{where } \texttt{NonAnalyzedValues} = \texttt{BoolVal} \cup \texttt{NullVal}$$

**Lemma V.11.** $\mathsf{sound}(\texttt{newObj}, \widehat{\mathit{newObj}})$ *and* $\mathsf{sound}(\texttt{forObj}, \widehat{\mathit{forObj}})$ *for the extended value types.*

*Proof.* Analogous to Lemma V.4 and Lemma V.5. □

**Lemma V.12.** $\mathsf{sound}(\texttt{newString}, \widehat{\mathit{newString}})$

*Proof.* To show: $\{\texttt{newString}(s)\} \subseteq \gamma(\widehat{\mathit{newString}}(s))$ for all strings $s$.
Analogous to Lemma V.4. □

**Lemma V.13.** $\mathsf{sound}(\texttt{classForName}, \widehat{\mathit{classForName}})$

*Proof.* To show:
$\{\texttt{classForName}(\mathit{val}) \mid \mathit{val} \in \gamma(\widehat{\mathit{val}})\} \subseteq \gamma(\widehat{\mathit{classForName}}(\widehat{s}))$ for all values $\mathit{val}$ and class tables $\mathit{table}$. We continue by case distinction on $\widehat{s}$ in $\widehat{\mathit{val}} = (\widehat{o}, \widehat{s}, \widehat{c}, \widehat{m})$.

- In case $\widehat{s} = \mathsf{Constant}(s)$ and there is a class $c$ with name $s$ in the class table, then $\mathsf{StrVal}(s, o) \in \gamma(\widehat{val})$ and

$$\begin{aligned}
&\texttt{classForName}(\mathsf{StrVal}(s, o)) \\
= {}&\mathsf{ClassVal}(c, \mathit{classAlloc}) \\
\in {}&\gamma((\mathit{classAlloc}, \varnothing, \{c\}, \varnothing)) \\
= {}&\gamma(\widehat{\mathit{classForName}}((\widehat{o}, \mathsf{Constant}(s), \widehat{c}, \widehat{m})))
\end{aligned}$$

  For all other $v \in \gamma(\widehat{val})$, `classForName` returns a type error.

- In case $\widehat{s} = \top$, then

$$\begin{aligned}
&\{\texttt{classForName}(\mathit{val}) \mid \mathit{val} \in \gamma(\widehat{val})\} \\
= {}&\{\mathsf{ClassVal}(c, \mathit{classAlloc}) \mid c \in \mathit{table}\} \\
\subseteq {}&\gamma((\mathit{classAlloc}, \varnothing, \top, \varnothing)) \\
= {}&\gamma(\widehat{\mathit{classForName}}((\widehat{o}, \top, \widehat{c}, \widehat{m}))) \qquad \Box
\end{aligned}$$

**Lemma V.14.** $\mathsf{sound}(\texttt{getMethod}, \widehat{\mathit{getMethod}})$

*Proof.* To show:

$$\begin{aligned}
&\{\texttt{getMethod}(\mathit{classVal}, \mathit{methodVal}) \mid \\
&\qquad \mathit{classVal} \in \gamma(\widehat{\mathit{classVal}}), \mathit{methodVal} \in \gamma(\widehat{\mathit{methodVal}})\} \\
&\subseteq \gamma(\texttt{getMethod}(\widehat{\mathit{classVal}}, \widehat{\mathit{methodVal}}))
\end{aligned}$$

for all values $\widehat{\mathit{classVal}} = (\_, \_, \widehat{c}, \_)$, $\widehat{\mathit{methodVal}} = (\_, \widehat{s}, \_, )$ and class tables *table*. We continue by case distinction on $\widehat{c}$ and $\widehat{s}$:

- In case $\widehat{c} \neq \top$ and $\widehat{s} = \mathsf{Constant}(s)$, then

$$\begin{aligned}
&\{\texttt{getMethod}(\mathsf{ClassVal}(c, \mathit{classAlloc}), \mathsf{StrVal}(s, \mathit{strAlloc})) \mid \\
&\qquad c \in \gamma(\widehat{c})\} \\
= {}&\{\mathsf{MethodVal}(\mathit{table}(c, s), \mathit{methodAlloc}) \mid c \in \widehat{c}\} \\
= {}&\gamma((\mathit{methodAlloc}, \varnothing, \varnothing, \{\mathit{table}(c, s) \mid c \in \widehat{c}\})) \\
= {}&\gamma(\texttt{getMethod}(\widehat{\mathit{classVal}}, \widehat{\mathit{methodVal}}))
\end{aligned}$$

- In case $\widehat{c} = \top$ or $\widehat{s} = \top$, then

$$\begin{aligned}
&\{\texttt{getMethod}(\mathit{classVal}, \mathit{methodVal}) \mid \\
&\qquad \mathit{classVal} \in \gamma(\widehat{\mathit{classVal}}), \mathit{methodVal} \in \gamma(\widehat{\mathit{methodVal}})\} \\
= {}&\{\mathsf{MethodVal}(\mathit{method}, \mathit{methodAlloc}) \mid \mathit{method} \in \mathit{table}\} \\
\subseteq {}&\gamma((\mathit{methodAlloc}, \varnothing, \varnothing, \top)) \\
= {}&\gamma(\texttt{getMethod}(\widehat{\mathit{classVal}}, \widehat{\mathit{methodVal}})) \qquad \Box
\end{aligned}$$

**Lemma V.15.** $\mathsf{sound}(\texttt{methodInvoke}, \widehat{methodInvoke})$

*Proof.* To show:

$$\begin{aligned}&\{\texttt{methodInvoke}(receiver, methodVal, args) \mid \\ &receiver \in \gamma(\widehat{receiver}), methodVal \in \gamma(\widehat{methodVal})\} \subseteq \\ &\qquad \gamma(\widehat{methodInvoke}(\widehat{receiver}, \widehat{methodVal}, args))\end{aligned}$$

for all values $\widehat{receiver} = (\widehat{o}, \_, \_, \_)$, $methodVal = (\_, \_, \_, \widehat{m})$, class tables *table*, and arguments *args*. We perform a case distinction on $\widehat{m}$.

- In case $\widehat{m} \neq \top$,

$$\begin{aligned}&\{\texttt{methodInvoke}(receiver, methodVal, args) \mid \\ &\qquad receiver \in \gamma(\widehat{receiver}), methodVal \in \gamma(\widehat{methodVal})\} \\ &= \{\mathsf{CallTarget}(c, ctx, method, args) \mid \\ &\qquad (c, ctx) \in \gamma(\widehat{o}), m \in \gamma(\widehat{m}), method = table(c, m)\} \\ &\subseteq \gamma(\widehat{\mathsf{CallTarget}}(\{(c, ctx, method, args) \mid \\ &\qquad (c, ctx) \in \widehat{o}, m \in \widehat{m}, method = table(c, m)\})) \\ &= \gamma(\widehat{methodInvoke}(\widehat{receiver}, \widehat{methodVal}, args))\end{aligned}$$

- In case $\widehat{m} = \top$,

$$\begin{aligned}&\{\texttt{methodInvoke}(receiver, methodVal, args) \mid \\ &\qquad receiver \in \gamma(\widehat{receiver}), methodVal \in \gamma(\widehat{methodVal})\} \\ &= \{\mathsf{CallTarget}(c, ctx, method, args) \mid \\ &\qquad (c, ctx) \in \gamma(\widehat{o}), method \in table(c)\} \\ &\subseteq \gamma(\widehat{\mathsf{CallTarget}}(\{(c, ctx, method, args) \mid \\ &\qquad (c, ctx) \in \widehat{o}, method \in table(c)\})) \\ &= \gamma(\widehat{methodInvoke}(\widehat{receiver}, \widehat{methodVal}, args))\end{aligned}$$ □

*Soundness Immutability Analysis*

For better readablilty, we use `TI` and `NTI` for `TransitivelyImmutable` and `NonTransitivelyImmutable` respectively.

**Definition V.16** (Concretization Function).

$$\gamma : \widehat{Assignability} \to \texttt{Assignability}$$

$$\gamma(x) = \begin{cases} \{\texttt{Assignable}, \texttt{NonAssignable}\} & \textit{if } x = \widehat{Assignable} \\ \{\texttt{NonAssignable}\} & \textit{if } x = Non\widehat{Assignable} \end{cases}$$

$$\gamma : \widehat{Mutability} \to \texttt{Mutability}$$

$$\gamma(x) = \begin{cases} \{\texttt{Mutable}, \texttt{NTI}, \texttt{TI}\} & \textit{if } x = \widehat{Mutable} \\ \{\texttt{NTI}, \texttt{TI}\} & \textit{if } x = \widehat{NTI} \\ \{\texttt{TI}\} & \textit{if } x = \widehat{TI} \end{cases}$$

**Lemma V.17.** $\mathsf{sound}(\texttt{assignable}, \widehat{assignable})$, $\mathsf{sound}(\texttt{nonAssignable}, \widehat{nonAssignable})$, $\mathsf{sound}(\texttt{mutable}, \widehat{mutable})$, $\mathsf{sound}(\texttt{immutable}, \widehat{immutable})$.

*Proof.*

$$\begin{aligned}
\{\texttt{assignable}\} &= \{\texttt{Assignable}\} \\
&\subseteq \{\texttt{Assignable}, \texttt{NonAssignable}\} \\
&= \gamma(\widehat{Assignable}) \\
&= \gamma(\widehat{assignable})
\end{aligned}$$

The soundness proofs of $\widehat{nonAssignable}$, $\widehat{mutable}$, $\widehat{immutable}$ are analogous. □

**Lemma V.18.** $\mathsf{sound}(\texttt{getFieldMutability}, \widehat{getFieldMutability})$

*Proof.* To show $\{\texttt{getFieldMutability}(assign, mut) \mid assign \in \gamma(\widehat{assign}), mut \in \gamma(\widehat{mut})\} \subseteq \gamma(\widehat{getFieldMutability}(\widehat{assign}, \widehat{mut}))$ for all $\widehat{assign}$ and $\widehat{mut}$. We continue by case distinction on $\widehat{assign}$ and $\widehat{mut}$:

- In case $\widehat{assign} = \widehat{Assignable}$, then

$$\begin{aligned}
&\{\texttt{getFieldMutability}(assign, mut) \mid \\
&\qquad assign \in \gamma(\widehat{Assignable}), mut \in \gamma(\widehat{mut})\} \\
= &\{\texttt{getFieldMutability}(\texttt{Assignable}, mut), \\
&\qquad \texttt{getFieldMutability}(\texttt{NonAssignable}, mut) \mid \\
&\qquad\quad mut \in \gamma(\widehat{mut})\} \\
= &\{\texttt{Mutable}, \texttt{NTI}, \texttt{TI}\} \\
= &\gamma(\widehat{Mutable}) \\
= &\gamma(\widehat{getFieldMutability}(\widehat{Assignable}, \widehat{mut}))
\end{aligned}$$

- In case $\widehat{Assign} = \widehat{NonAssignable}$ and $\widehat{mut} = \widehat{Mutable}$, then

$$\begin{aligned}
&\{\texttt{getFieldMutability}(assign, mut) \mid \\
&\qquad assign \in \gamma(\widehat{NonAssignable}), mut \in \gamma(\widehat{Mutable})\} \\
= &\{\texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{Mutable}), \\
&\qquad \texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{NTI}), \\
&\qquad \texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{TI})\} \\
= &\{\texttt{NTI}, \texttt{TI}\} \\
= &\gamma(\widehat{NTI}) \\
= &\gamma(\widehat{getFieldMutability}(\widehat{NonAssignable}, \widehat{Mutable}))
\end{aligned}$$

- In case $\widehat{Assign} = \widehat{NonAssignable}$ and $\widehat{mut} = \widehat{NTI}$, then

$$\begin{aligned}
&\{\texttt{getFieldMutability}(assign, mut) \mid \\
&\qquad assign \in \gamma(\widehat{NonAssignable}), mut \in \gamma(\widehat{NTI})\} \\
= &\{\texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{NTI}), \\
&\qquad \texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{TI})\} \\
= &\{\texttt{NonTransitivelyImmutable}, \texttt{TI}\} \\
= &\gamma(\widehat{NTI}) \\
= &\gamma(\widehat{getFieldMutability}(\widehat{NonAssignable}, \widehat{NTI}))
\end{aligned}$$

- In case $\widehat{Assign} = \widehat{NonAssignable}$ and $\widehat{mut} = \widehat{TI}$, then

$$\begin{aligned}
&\{\texttt{getFieldMutability}(assign, mut) \mid \\
&\qquad assign \in \gamma(\widehat{NonAssignable}), mut \in \gamma(\widehat{TI})\} \\
= &\{\texttt{getFieldMutability}(\texttt{NonAssignable}, \texttt{TI})\} \\
= &\gamma(\widehat{TI}) \\
= &\gamma(\widehat{getFieldMutability}(\widehat{NonAssignable}, \widehat{TI}))
\end{aligned}$$ □

**Lemma V.19.** $\texttt{sound}(\texttt{joinMutability}, \widehat{joinMutability})$

*Proof.* To show $\{\texttt{joinMutability}(Mut) \mid Mut \in \gamma(\widehat{Mut})\} \subseteq \gamma(\widehat{joinMutability}(\widehat{Mut}))$ for all $\widehat{Mut}$. We continue by case distinction on $\widehat{Mut}$:

- In case $\bigsqcup \widehat{Mut} = \widehat{Mutable}$, then

$$\begin{aligned}
&\{\texttt{joinMutability}(Mut) \mid Mut \in \gamma(\widehat{Mut})\} \\
\subseteq &\gamma(\widehat{Mutable}) \\
= &\gamma(\widehat{joinMutability}(\widehat{Mut}))
\end{aligned}$$

- In case $\bigsqcup \widehat{Mut} = \widehat{NTI}$, then $Mut \in \gamma(\widehat{Mut})$ contains elements NTI or TI. Hence

$$\begin{aligned}
&\{\texttt{joinMutability}(Mut) \mid Mut \in \gamma(\widehat{Mut})\} \\
= &\{\texttt{NTI}, \texttt{TI}\} \\
\subseteq &\gamma(\widehat{NTI}) \\
= &\gamma(\widehat{joinMutability}(\widehat{Mut}))
\end{aligned}$$

- In case $\bigsqcup \widehat{Mut} = \widehat{TransitivelyImmutable}$, then $Mut \in \gamma(\widehat{Mut})$ contains only elements TI. Hence

$$\begin{aligned}
&\{\texttt{joinMutability}(Mut) \mid Mut \in \gamma(\widehat{Mut})\} \\
= &\{\texttt{TI}\} \\
\subseteq &\gamma(\widehat{TI}) \\
= &\gamma(\widehat{joinMutability}(\widehat{Mut}))
\end{aligned}$$ □

*Soundness of the Reaching-Definitions Analysis*

The analysis consists of two modules $\widehat{\mathsf{reachingDefs}}$ and $\widehat{\mathsf{controlFlowPred}}$. A similar version of module $\widehat{\mathsf{reachingDefs}}$ has been proven sound in Lemma 5.7. It remains to prove the control-flow module sound:

**Lemma V.20.** $\mathsf{sound}(\mathsf{controlFlowPred}, \widehat{\mathsf{controlFlowPred}})$.

*Proof.* The analysis module $\widehat{\mathsf{controlFlowPred}}$ computes the set of all immediate control-flow predecessors of a given statement. The dynamic module $\mathsf{controlFlowPred}$ uses a mutable variable `pred` to remember the previously executed statement. Only if variable `pred` is a potential control-flow predecessor of the given statement `stmt`, module $\mathsf{controlFlowPred}$ adds it as a control-flow predecessor to the store ($\sigma[\mathtt{stmt}, \kappa_{\mathsf{controlFlowPred}} \mapsto \mathtt{pred}]$). The analysis module $\widehat{\mathsf{controlFlowPred}}$ over-approximates the dynamic module $\mathsf{controlfFlowPred}$ because it adds all control-flow predecessors to the store, not just one. □

# BIBLIOGRAPHY

[1] Vitor Monte Afonso, Paulo L. de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna, Adam Doupé, and Mario Polino. “Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy.” In: *23rd Annual Network and Distributed System Security Symposium*. NDSS’16. San Diego, CA, USA: Internet Society, 2016.

[2] Gul A. Agha. “ACTORS - A Model of Concurrent Computation in Distributed Systems.” PhD thesis. University of Michigan, MI, USA, 1985.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Addison-Wesley, 2007.

[4] Alexander Aiken. “Introduction to set constraint-based program analysis.” In: *Science of Computer Programming* 35.2-3 (1999), pp. 79–111.

[5] Bowen Alpern et al. “The Jikes Research Virtual Machine project: Building an open-source research community.” In: *IBM Systems Journal* 44.2 (2005), pp. 399–417.

[6] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. “Porting Doop to Soufflé: a Tale of Inter-Engine Portability for Datalog-Based Analyses.” In: *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP’17. Barcelona, Spain: ACM, 2017, pp. 25–30.

[7] AppBrain. *Number of Android applications on the Google Play store*. https://www.appbrain.com/stats/number-of-android-apps. [Online; accessed 15-September-2022]. 2022.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. “FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps.” In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI’14. Edinburgh, UK: ACM, 2014, pp. 259–269.

[9] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. “Hypercollecting Semantics and Its Application to Static Analysis of Information Flow.” In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL’17. Paris, France: ACM, 2017, pp. 874–887.

[10] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. "Using Static Analysis to Find Bugs." In: *IEEE software* 25.5 (2008), pp. 22–29.

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing." In: *ACM Computing Surveys* 26.4 (1994), pp. 345–420.

[12] David F. Bacon and Peter F. Sweeney. "Fast Static Analysis of C++ Virtual Function Calls." In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'96. San Jose, CA, USA: ACM, 1996, pp. 324–341.

[13] Anindya Banerjee and David A. Naumann. "Secure Information Flow and Pointer Confinement in a Java-like Language." In: *Proceedings 15th IEEE Computer Security Foundations Workshop*. Vol. 2. CSFW'02. Los Alamitos, CA, USA: IEEE, 2002, pp. 253:1–253:15.

[14] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. "99.44% pure: Useful Abstractions in Specifications." In: *Proceedings of the ECOOP Workshop FTfJP 2004, Formal Techniques for Java-like Programs*. FTfJP'04. Oslo, Norway: University of Nijmegen, 2004.

[15] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. "Allowing State Changes in Specifications." In: *Proceedings of the 2006 International Conference on Emerging Trends in Information and Communication Security*. Vol. 3995. ETRICS'06. Freiburg, Germany: Springer, 2006, pp. 321–336.

[16] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. "Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering*. ASE'15. Lincoln, NE, USA: IEEE, 2015, pp. 669–679.

[17] William C. Benton and Charles N. Fischer. "Mostly-Functional Behavior in Java Programs." In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. VMCAI'09. Savannah, GA, USA: Springer, 2009, pp. 29–43.

[18] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[19] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." In: *Communications of the ACM* 53.2 (2010), pp. 66–75.

[20] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: *International Conference on Computer Aided Verification*. CAV'07. Berlin, Germany: Springer, 2007, pp. 504–518.

[21] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Program Analysis with Dynamic Precision Adjustment." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE'08. L'Aquila, Italy: IEEE, 2008, pp. 29–38.

[22] Gavin Bierman. *JEP 395: Records*. https://openjdk.java.net/jeps/395. [Online; accessed 21-November-2022]. 2022.

[23] Adrian Birka and Michael D. Ernst. "A Practical Type System and Language for Reference Immutability." In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'04. Vancouver, Canada: ACM, 2004, pp. 35–49.

[24] Stephen M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis." In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA'06. Portland, OR, USA: ACM, 2006, pp. 169–190.

[25] Eric 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. Beijing, China: ACM, 2012, pp. 3–8.

[26] Eric 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*. SOAP'18. Amsterdam, The Netherlands: ACM, 2018, pp. 85–93.

[27] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE'11. Honolulu, HI, USA: IEEE, 2011, pp. 241–250.

[28] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. "Skeletal Semantics and Their Interpretations." In: *Proceedings of the ACM on Programming Languages*. POPL'19 3.POPL (2019), pp. 44:1–44:31.

[29] Denis Bogdanas and Grigore Roşu. "K-Java: A Complete Semantics of Java." In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'15. Mumbai, India: ACM, 2015, pp. 445–456.

[30] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: Automated Testing Based on Java Predicates." In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'02. Roma, Italy: ACM, 2002, pp. 123–133.

[31] John Tang Boyland, James Noble, and William Retert. "Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only." In: *ECOOP 2001 — Object-Oriented Programming*. ECOOP'01. Budapest, Hungary: Springer, 2001, pp. 2–27.

[32] John Boyland. "Why we should not add readonly to Java (yet)." In: *Journal of Object Technology* 5.5 (2006), pp. 5–29.

[33] Martin Bravenboer and Yannis Smaragdakis. "Exception Analysis and Points-to Analysis: Better Together." In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA'09. Chicago, IL, USA: ACM, 2009, pp. 1–12.

[34] Martin Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses." In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA'09. Orlando, FL, USA: ACM, 2009, pp. 243–262.

[35] Antonio Brogi and Paolo Ciancarini. "The Concurrent Language, Shared Prolog." In: *ACM Transactions on Programming Languages and Systems* 13.1 (1991), pp. 99–123.

[36] Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. "An overview of JML tools and applications." In: *International Journal on Software Tools for Technology Transfer* 7.3 (2005), pp. 212–232.

[37] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*. Vol. 1. Pattern-Oriented Software Architecture. Wiley, 1996.

[38] Brendon Cahoon and Kathryn S. McKinley. "Data Flow Analysis for Software Prefetching Linked Data Structures in Java." In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. PACT'01. Barcelona, Spain: IEEE, 2001, pp. 280–291.

[39] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. "DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN'18. Luxembourg City, Luxembourg: IEEE, 2018, pp. 430–441.

[40] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. "Escape Analysis for Java." In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'99. Denver, CO, USA: ACM, 1999, pp. 1–19.

[41] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Collaborative Verification and Testing with Explicit Assumptions." In: *FM 2012: Formal Methods*. FM'12. Paris, France: Springer, 2012, pp. 132–146.

[42] Mandy Chung. *JEP 416: Reimplement Core Reflection with Method Handles*. https://openjdk.org/jeps/416. [Online; accessed 06-September-2022]. 2022.

[43] Lars R. Clausen. "A Java bytecode optimizer using side-effect analysis." In: *Concurrency and Computation: Practice and Experience* 9.11 (1997), pp. 1031–1045.

[44] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. "Glacier: Transitive Class Immutability for Java." In: *2017 IEEE/ACM 39th International Conference on Software Engineering*. ICSE'17. Buenos Aires, Argentina: IEEE, 2017, pp. 496–506.

[45] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. "Exploring Language Support for Immutability." In: *2016 IEEE/ACM 38th International Conference on Software Engineering*. ICSE'16. Austin, TX, USA: IEEE, 2016, pp. 736–747.

[46] David R. Cok. "Reasoning with specifications containing method calls and model fields." In: *Journal of Object Technology* 4.8 (2005), pp. 77–103.

[47] Daniel D. Corkill. "Design Alternatives for Parallel and Distributed Blackboard Systems." In: Perspectives in Artificial Intelligence (1989), pp. 99–136.

[48] Daniel D. Corkill. "Blackboard Systems." In: *AI expert* 6.9 (1991), pp. 40–47.

[49] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. "A Survey on Product Operators in Abstract Interpretation." In: Electronic Proceedings in Theoretical Computer Science 129 (2013), pp. 325–336.

[50] Patrick Cousot. "Syntactic and Semantic Soundness of Structural Dataflow Analysis." In: *Static Analysis*. SAS'19. Porto, Portugal: Springer, 2019, pp. 96–117.

[51] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL'77. Los Angeles, CA, USA: ACM, 1977, pp. 238–252.

[52] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks." In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL'79. San Antonio, TX, USA: ACM, 1979, pp. 269–282.

[53] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTREÉ Analyzer." In: *Programming Languages and Systems*. ESOP'05. Edinburgh, UK: Springer, 2005, pp. 21–30.

[54] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "Combination of Abstractions in the ASTRÉE Static Analyzer." In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*. ASIAN'06. Tokyo, Japan: Springer, 2006, pp. 272–300.

[55] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. "Static Detection of Vulnerabilities in x86 Executables." In: *2006 22nd Annual Computer Security Applications Conference*. ACSAC'06. Miami Beach, FL, USA: IEEE, 2006, pp. 269–278.

[56] Iain D. Craig. "Blackboard systems." In: *Artificial Intelligence Review* 2.2 (1988), pp. 103–118.

[57] Iain D. Craig. *A New Interpretation of The Blackboard Metaphor*. Tech. rep. CS-RR-254. Department of Computer Science University of Warwick, 1993.

[58] Valentin Dallmeier. "Mining and Checking Object Behavior." PhD thesis. Saarland University, Germany, 2010.

[59] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. "Mining Object Behavior with ADABU." In: *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*. WODA'06. Shanghai, China: ACM, 2006, pp. 17–24.

[60] Ádám Darvas and K. Rustan M. Leino. "Practical Reasoning About Invocations and Implementations of Pure Methods." In: *Fundamental Approaches to Software Engineering*. FASE'07. Braga, Portugal: Springer, 2007, pp. 336–351.

[61] Ádám Darvas and Peter Müller. "Reasoning About Method Calls in Interface Specifications." In: *Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2005*. FTfJP'05. Glasgow, UK: ETH Zurich, 2005, pp. 59–85.

[62] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis." In: *European Conference on Object-Oriented Programming*. ECOOP'95. Åarhus, Denmark: Springer, 1995, pp. 77–101.

[63] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. "Compiler Techniques for Code Compaction." In: *ACM Transactions on Programming languages and Systems* 22.2 (2000), pp. 378–415.

[64] Keith Decker, Alan Garvey, Marty Humphrey, and Victor R. Lesser. "Effects of Parallelism on Blackboard System Scheduling." In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*. Vol. 1. IJCAI'91. Sydney, Australia, 1991, pp. 15–21.

[65] Delphine Demange, Thomas Jensen, and David Pichardie. "A Provably Correct Stackless Intermediate Representation for Java Bytecode." In: *Asian Symposium on Programming Languages and Systems*. APLAS'10. Shanghai, China: Springer, 2010, pp. 97–113.

[66] Werner Dietl and Peter Müller. "Universes: Lightweight Ownership for JML." In: *Journal of Object Technology* 4.8 (2005), pp. 5–32.

[67] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. "XCorpus–An executable Corpus of Java Programs." In: *Journal of Object Technology* 16.4 (2017), pp. 1:1–1:24.

[68] Roland Dodd, Andrew Chiou, Xinghuo Yu, and Ross Broadfoot. "Industrial Process Model Integration Using a Blackboard Model within a Pan Stage Decision Support System." In: *2009 Third International Conference on Network and System Security*. NSS'09. Gold Coast, Australia: IEEE, 2009, pp. 489–494.

[69] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. "An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension." In: *IEEE Transactions on Software Engineering* 29.7 (2003), pp. 665–670.

[70] Julian Dolby. "Using Static Analysis For IDE's for Dynamic Languages." In: *The Eclipse Languages Symposium*. Kanata, Canada: Eclipse Foundation, 2005.

[71] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild." In: *Security and Privacy in Communication Networks*. SecureComm'18. Singapore, Singapore: Springer, 2018, pp. 172–192.

[72] *Dotty Documentation Overview – Effect Capabilities*. https://dotty.epfl.ch/docs/reference/index.html. [Online; accessed 21-April-2018]. 2018.

[73] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. "Defining and Continuous Checking of Structural Program Dependencies." In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE'08. Leipzig, Germany: ACM, 2008, pp. 391–400.

[74] Michael Eichberg, Florian Kübler, Dominik Helm, Michael Reif, Guido Salvaneschi, and Mira Mezini. "Lattice Based Modularization of Static Analyses." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. SOAP'18. Amsterdam, The Netherlands: ACM, 2018, pp. 113–118.

[75] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. "Integrating and Scheduling an Open Set of Static Analyses." In: *21st IEEE/ACM International Conference on Automated Software Engineering*. ASE'06. Tokyo, Japan: IEEE, 2006, pp. 113–122.

[76] Torbjörn Ekman and Görel Hedin. "The JastAdd Extensible Java Compiler." In: *Proceedings of the ACM on Programming Languages*. OOPSLA'07. Montreal, Canada: ACM, 2007, pp. 1–18.

[77] Conal Elliott and Paul Hudak. "Functional Reactive Animation." In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP'97. Amsterdam, The Netherlands: ACM, 1997, pp. 263–273.

[78] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers." In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI'94. Orlando, FL, USA: ACM, 1994, pp. 242–256.

[79] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." In: *ACM Computing Surveys* 12.2 (1980), pp. 213–253.

[80] Douglas Everson, Long Cheng, and Zhenkai Zhang. "Log4shell: Redefining the Web Attack Surface." In: *Measurements, Attacks, and Defenses for the Web Workshop 2022*. MADWeb'22. San Diego, CA, USA: Internet Society, 2022.

[81] Manuel Fähndrich and Francesco Logozzo. "Static Contract Checking with Abstract Interpretation." In: *Formal Verification of Object-oriented Software*. FoVeOOS'10. Paris, France: Springer, 2010, pp. 10–30.

[82] Rodney Farrow. "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars." In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN'86. Palo Alto, CA, USA: ACM, 1986, pp. 85–98.

[83] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Efficient Construction of Approximate Call Graphs for JavaScript IDE Services." In: *2013 35th International Conference on Software Engineering*. ICSE'13. San Francisco, CA, USA: IEEE, 2013, pp. 752–761.

[84] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. "Verifiable Functional Purity in Java." In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS'08. Alexandria, VA, USA: ACM, 2008, pp. 161–174.

[85] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java." In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI'02. Berlin, Germany: ACM, 2002, pp. 234–245.

[86] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. "Static Analysis of Java Dynamic Proxies." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'18. Amsterdam, The Netherlands: ACM, 2018, pp. 209–220.

[87] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Addison-Wesley, 1995.

[88] David Gay and Bjarne Steensgaard. "Fast Escape Analysis and Stack Allocation for Object-Based Programs." In: *Compiler Construction*. CC'00. Berlin, Germany: Springer, 2000, pp. 82–93.

[89] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. "AI$^2$: Safety and Robustness Certification of Neural Networks with Abstract Interpretation." In: *2018 IEEE Symposium on Security and Privacy*. SP'18. San Francisco, CA, USA: IEEE, 2018, pp. 3–18.

[90] Samir Genaim and Fausto Spoto. "Constancy analysis." In: *Formal Techniques for Java-like Programs*. FTfJP'08. Paphos, Cyprus: Radboud University, 2008, pp. 100–110.

[91] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. "Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy." In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS'20. Taipei, Taiwan: ACM, 2020, pp. 694–707.

[92] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. "Uniqueness and Reference Immutability for Safe Parallelism." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA'12. Tucson, AZ, USA: ACM, 2012, pp. 21–40.

[93] Neville Grech and Yannis Smaragdakis. "P/Taint: Unified Points-to and Taint Analysis." In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–28.

[94] David Grove and Craig Chambers. "A Framework for Call Graph Construction Algorithms." In: *ACM Transactions on Programming Languages and Systems* 23.6 (2001), pp. 685–746.

[95] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. "Immutable Objects for a Java-Like Language." In: *Programming Languages and Systems*. ESOP'07. Braga, Portugal: Springer, 2007, pp. 347–362.

[96] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. "Codequest: Scalable Source Code Queries with Datalog." In: *ECOOP 2006 – Object-Oriented Programming*. ECOOP'06. Nantes, France: Springer, 2006, pp. 2–27.

[97] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. "Reactive Async: Expressive Deterministic Concurrency." In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA'16. Amsterdam, The Netherlands: ACM, 2016, pp. 11–20.

[98] Philipp Haller and Martin Odersky. "Scala Actors: Unifying thread-based and event-based programming." In: *Theoretical Computer Science* 410.2 (2009), pp. 202–220.

[99] Görel Hedin. "Reference Attributed Grammars." In: *Informatica* 24.3 (2000), pp. 301–317.

[100] Pat Helland. "Immutability Changes Everything." In: *Communications of the ACM* 59.1 (2015), pp. 64–70.

[101] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. "A Unified Lattice Model and Framework for Purity Analyses." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE'18. Montpellier, France: ACM, 2018, pp. 340–350.

[102] Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. "A Programming Model for Semi-implicit Parallelization of Static Analyses." In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'20. Virtual Event, USA: ACM, 2020, pp. 428–439.

[103] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. "Modular Collaborative Program Analysis in OPAL." In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE'20. Virtual Event, USA: ACM, 2020, pp. 184–196.

[104] *Heros IFDS/IDE Solver*. https://github.com/Sable/heros. [Online; accessed 03-November-2022]. 2022.

[105] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: *Proceedings of the Third International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, CA, USA: Morgan Kaufmann, 1973, pp. 235–245.

[106] Michael Hind and Anthony Pioli. "Which Pointer Analysis Should I Use?" In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'00. Portland, OR, USA: ACM, 2000, pp. 113–123.

[107] David Van Horn and Matthew Might. "Abstracting Abstract Machines." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP'10. Baltimore, MD, USA: ACM, 2010, pp. 51–62.

[108] Wei Huang and Ana Milanova. "ReImInfer: Method Purity Inference for Java." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE'12. Cary, NC, USA: ACM, 2012.

[109] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. "Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA'12. Tucson, AZ, USA: ACM, 2012, pp. 879–896.

[110] John Hughes. "Generalising monads to arrows." In: *Science of Computer Programming* 37.1 (2000), pp. 67–111.

[111] IBM. *WALA - T. J. Watson Libraries for Analysis*. http://wala.sourceforge.net/. [Online; accessed 12-June-2020]. 2020.

[112] Roberto Ierusalimschy and Noemi Rodriguez. "Side-Effect Free Functions in Object-Oriented Languages." In: *Computer languages* 21.3 (1995), pp. 129–146.

[113] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler." In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'00. Minneapolis, MN, USA: ACM, 2000, pp. 294–310.

[114] Daniel Jackson and Martin Rinard. "Software Analysis: A Roadmap." In: *Proceedings of the Conference on the Future of Software Engineering*. ICSE'00. Limerick, Ireland: ACM, 2000, pp. 133–145.

[115] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Interprocedural Analysis with Lazy Propagation." In: *Static Analysis*. SAS'10. Perpignan, France: Springer, 2010, pp. 320–339.

[116] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. "A Collaborative Dependence Analysis Framework." In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO'17. Austin, TX, USA: IEEE, 2017, pp. 148–159.

[117] Larry G. Jones. "Efficient Evaluation of Circular Attribute Grammars." In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 429–462.

[118] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. "Soufflé: On Synthesis of Program Analyzers." In: *Computer Aided Verification*. CAV'16. Toronto, Canada: Springer, 2016, pp. 422–430.

[119] Jacques-Henri Jourdan. "Verasco: a Formally Verified C Static Analyzer." PhD thesis. Universite Paris Diderot-Paris VII, France, 2016.

[120] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer." In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'15. Mumbai, India: ACM, 2015, pp. 247–259.

[121] George Kastrinis and Yannis Smaragdakis. "Hybrid Context-Sensitivity for Points-to Analysis." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'13. Seattle, WA, USA: ACM, 2013, pp. 423–434.

[122] Sven Keidel. *Sturdy*. http://github.com/svenkeidel/sturdy/. [Online; accessed 06-September-2022]. 2022.

[123] Sven Keidel and Sebastian Erdweg. "Sound and Reusable Components for Abstract Interpretation." In: 3.OOPSLA (2019), pp. 176:1–176:28.

[124] Sven Keidel and Sebastian Erdweg. "A Systematic Approach to Abstract Interpretation of Program Transformations." In: *Verification, Model Checking, and Abstract Interpretation*. VMCAI'20. New Orleans, LA, USA: Springer, 2020, pp. 136–157.

[125] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. "Compositional Soundness Proofs of Abstract Interpreters." In: 2.ICFP (2018), pp. 72:1–72:26.

[126] Mehdi Keshani. "Scalable Call Graph Constructor for Maven." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*. ICSE'21. Virtual Event, Spain: IEEE, 2021, pp. 99–101.

[127] Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. "How Good is Static Analysis at Finding Concurrency Bugs?" In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. SCAM'10. Timişoara, Romania: IEEE, 2010, pp. 115–124.

[128] Fredrik Berg Kjolstad, Danny Dig, Gabriel Acevedo, and Marc Snir. "Transformation for Class Immutability." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE'11. Honolulu, HI, USA: ACM, 2011, pp. 61–70.

[129] Günter Kniesel and Dirk Theisen. "JAC—access right based encapsulation for Java." In: *Software: Practice and Experience* 31.6 (2001), pp. 555–576.

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

[131] Thomas Kotzmann and Hanspeter Mössenböck. "Escape Analysis in the Context of Dynamic Compilation and Deoptimization." In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. VEE'05. Chicago, IL, USA: ACM, 2005, pp. 111–120.

[132] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. "How Much Parallelism is There in Irregular Applications?" In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP'09. Raleigh, NC, USA: ACM, 2009, pp. 3–14.

[133] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. "Optimistic Parallelism Requires Abstractions." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'07. San Diego, CA, USA: ACM, 2007, pp. 211–222.

[134] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. "Freeze After Writing: Quasi-Deterministic Parallel Programming with LVars." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'14. San Diego, CA, USA: ACM, 2014, pp. 257–270.

[135] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. "Context-Sensitive Program Analysis as Database Queries." In: *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS'05. Baltimore, MD, USA: ACM, 2005, pp. 1–12.

[136] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. "Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study." In: *2017 IEEE/ACM 39th International Conference on Software Engineering*. ICSE'17. Buenos Aires, Argentina: IEEE, 2017, pp. 507–518.

[137] Anatole Le, Ondřej Lhoták, and Laurie J. Hendren. "Using Inter-Procedural Side-Effect Information in JIT Optimizations." In: *Compiler Construction*. CC'05. Edinburgh, UK: Springer, 2005, pp. 287–304.

[138] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. "Flexible Immutability with Frozen Objects." In: *Verified Software: Theories, Tools, Experiments*. VSTTE'08. Toronto, Canada: Springer, 2008, pp. 192–208.

[139] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. "FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE'14. Hong Kong, China: ACM, 2014, pp. 98–108.

[140] Sorin Lerner, David Grove, and Craig Chambers. "Composing Dataflow Analyses and Transformations." In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'02. Portland, OR, USA: ACM, 2002, pp. 270–282.

[141] Xavier Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

[142] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert - A Formally Verified Optimizing Compiler." In: *Embedded Real Time Software and Systems, 8th European Congress*. ERTS'16. Toulouse, France: SEE, 2016.

[143] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. "Putting Static Analysis to Work for Verification: A Case Study." In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'00. Portland, OR, USA: ACM, 2000, pp. 26–38.

[144] Ondřej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using Spark." In: *Compiler Construction*. CC'03. Warsaw, Poland: Springer, 2003, pp. 153–169.

[145] Ondřej Lhoták and Laurie Hendren. "Context-Sensitive Points-to Analysis: Is It Worth It?" In: *Compiler Construction*. CC'06. Springer. Vienna, Austria, 2006, pp. 47–64.

[146] Hongyi Li, Rudi Deklerck, Bernard De Cuyper, A. Hermanus, Edgard Nyssen, and Jan Cornelis. "Object Recognition in Brain CT-Scans: Knowledge-Based Fusion of Data from Multiple Feature Extractors." In: *IEEE Transactions on Medical Imaging* 14.2 (1995), pp. 212–229.

[147] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. ICSE'15. Florence, Italy: IEEE, 2015, pp. 280–291.

[148] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters." In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'95. San Francisco, CA, USA: ACM, 1995, pp. 333–343.

[149] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. "In Defense of Soundiness: A Manifesto." In: *Communications of the ACM* 58.2 (2015), pp. 44–46.

[150] Benjamin Livshits, John Whaley, and Monica S. Lam. "Reflection Analysis for Java." In: *Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer, 2005, pp. 139–160.

[151] John M. Lucassen and David K. Gifford. "Polymorphic Effect Systems." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'88. San Diego, CA, USA: ACM, 1988, pp. 47–57.

[152] Ami Luttwak and Alon Schindel. *Log4Shell 10 days later: Enterprises halfway through patching*. https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell. [Online; accessed 06-September-2022].

[153] Magnus Madsen and Ondřej Lhoták. “Safe and Sound Program Analysis with Flix.” In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA’18. Amsterdam, The Netherlands: ACM, 2018, pp. 38–48.

[154] Magnus Madsen, Ming-Ho Yee, and Ondřej 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*. Vol. 51. PLDI’16 6. Santa Barbara, CA, USA: ACM, 2016, pp. 194–208.

[155] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars — their evaluation and applications.” In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[156] Alessandro Margara and Guido Salvaneschi. “On the Semantics of Distributed Reactive Programming: The Cost of Consistency.” In: *IEEE Transactions on Software Engineering* 44.7 (2018), pp. 689–711.

[157] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. *An evaluation of exhaustive testing for data structures*. Tech. rep. MIT-LCS-TR-921. MIT Computer Science and Artificial Intelligence Laboratory, 2003.

[158] *Maven Repository*. https://mvnrepository.com/. [Online; accessed 23-September-2022]. 2022.

[159] Erik Meijer. “Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues.” In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP’10. Baltimore, MD, USA: ACM, 2010, pp. 11:1–11:1.

[160] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. “A GPU Implementation of Inclusion-based Points-to Analysis.” In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP’12. New Orleans, LA, USA: ACM, 2012, pp. 107–116.

[161] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. “Parallel Inclusion-based Points-to Analysis.” In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA’10. Reno/Tahoe, NV, USA: ACM, 2010, pp. 428–443.

[162] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. “Flapjax: A Programming Language for Ajax Applications.” In: *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA’09. Orlando, FL, USA: ACM, 2009, pp. 1–20.

[163] Ana Milanova and Yao Dong. “Inference and Checking of Object Immutability.” In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ’16. Lugano, Switzerland: ACM, 2016, pp. 6:1–6:12.

[164] Peter D. Mosses. “Modular structural operational semantics.” In: *The Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 195–228.

[165] Steven Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[166] David A. Naumann. “Observational Purity and Encapsulation.” In: *Fundamental Approaches to Software Engineering*. FASE’05. Edinburgh, UK: Springer, 2005, pp. 190–204.

[167] Stephen Nelson, David J. Pearce, and James Noble. “Profiling Field Initialisation in Java.” In: *Runtime Verification*. RV’12. Istanbul, Turkey: Springer, 2012, pp. 292–307.

[168] Allen Newell. *Some problems of basic organization in problem-solving programs*. Tech. rep. RM-3283-PR. RAND Corporation, 1962.

[169] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[170] H. Penny Nii. “The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures.” In: *AI Magazine* 7.2 (1986), pp. 38–53.

[171] H. Penny Nii, Edward A. Feigenbaum, and John J. Anton. “Signal-to-Symbol Transformation: HASP/SIAP Case Study.” In: *AI Magazine* 3.2 (1982), pp. 23–35.

[172] Jesper Öqvist and Görel Hedin. “Concurrent Circular Reference Attribute Grammars.” In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE’17. Vancouver, Canada: ACM, 2017, pp. 151–162.

[173] Oracle. *Map (Java SE 16 & JDK 16)*. `https://docs.oracle.com/en/\java/javase/16/docs/api/java.base/java/util/Map.html`. [Online; accessed 21-November-2022]. 2022.

[174] Oracle. *Secure Coding Guidelines for Java SE*. `https://www.oracle.com/java/technologies/javase/seccodeguide.html`. [Online; accessed 21-November-2022]. 2022.

[175] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. “CRiOS: Toward Large-Scale iOS Application Analysis.” In: *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM’16. Vienna, Austria: ACM, 2016, pp. 33–42.

[176] Software Engineering Group at Heinz Nixdorf Institute of Paderborn University. *Soot - A framework for analyzing and transforming Java and Android applications*. https://soot-oss.github.io/soot/. [Online; accessed 06-September-2022].

[177] Ya Pan, Xiuting Ge, Chunrong Fang, and Yong Fan. "A Systematic Literature Review of Android Malware Detection Using Static Analysis." In: *IEEE Access* 8 (2020), pp. 116363–116379.

[178] David Pearce. "JPure: A Modular Purity System for Java." In: *Compiler Construction*. CC'11. Saarbrücken, Germany: Springer, 2011, pp. 104–123.

[179] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. "Automatic Detection of Immutable Fields in Java." In: *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. CASCON'00. Mississauga, Canada: IBM, 2000, pp. 10:1–10:15.

[180] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. "Immutability." In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, pp. 233–269.

[181] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. "EigenCFA: Accelerating Flow Analysis with GPUs." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'11. Austin, TX, USA: ACM, 2011, pp. 511–522.

[182] Aleksandar Prokopec, Dmitry Petrashko, Miguel Garcia, Jason Zaugg, Hubert Plociniczak, Viktor Klang, and Martin Odersky. *SIP-20 - Improved Lazy Vals Initialization*. https://docs.scala-lang.org/sips/improved-lazy-val-initialization.html. [Online; accessed 08-August-2021]. 2021.

[183] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. "Inference of Reference Immutability." In: *ECOOP 2008 – Object-Oriented Programming*. ECOOP'08. Paphos, Cyprus: Springer, 2008, pp. 616–641.

[184] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Harvesting Runtime Values in Android Applications that Feature Anti-Analysis Techniques." In: *23rd Annual Network and Distributed System Security Symposium*. NDSS'16. San Diego, CA, USA: Internet Society, 2016.

[185] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. "Call Graph Construction for Java Libraries." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE'16. Seattle, WA, USA: ACM, 2016, pp. 474–486.

[186] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. "Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA'19. Beijing, China: ACM, 2019, pp. 251–261.

[187] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. "Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. SOAP'18. Amsterdam, The Netherlands: ACM, 2018, pp. 107–112.

[188] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. "TACAI: An Intermediate Representation Based on Abstract Interpretation." In: *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. SOAP'20. London, UK: ACM, 2020, pp. 2–7.

[189] Thomas W. Reps. "Demand Interprocedural Program Analysis Using Logic Databases." In: *Applications of Logic Databases*. Springer, 1995, pp. 163–196.

[190] Thomas W. Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability." In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'95. San Francisco, CA, USA: ACM, 1995, pp. 49–61.

[191] Martin Rinard. "Analysis of Multithreaded Programs." In: *Static Analysis*. SAS'01. Paris, France: Springer, 2001, pp. 1–19.

[192] Jonathan Rodriguez and Ondřej Lhoták. "Actor-Based Parallel Dataflow Analysis." In: *Compiler Construction*. CC'11. Saarbrücken, Germany: Springer, 2011, pp. 179–197.

[193] John R. Rose. "Bytecodes meet Combinators: invokedynamic on the JVM." In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. VMIL'09. Orlando, FL, USA: ACM, 2009, pp. 2:1–2:11.

[194] Grigore Roşu and Traian Florin Şerbănută. "An overview of the K semantic framework." In: *The Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434.

[195] Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. "CiFi: Versatile Analysis of Class and Field Immutability." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering*. ASE'21. Virtual Event, Australia: IEEE, 2021, pp. 979–990.

[196] Atanas Rountev. "Precise Identification of Side-effect-free Methods in Java." In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* ICSM'04. Chicago, IL, USA: IEEE, 2004, pp. 82–91.

[197] Andrei Sabelfeld and Andrew C. Myers. "Language-Based Information-Flow Security." In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[198] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from Building Static Analysis Tools at Google." In: *Communications of the ACM* 61.4 (2018), pp. 58–66.

[199] Mooly Sagiv, Thomas Reps, and Susan Horwitz. "Precise interprocedural dataflow analysis with applications to constant propagation." In: *Theoretical Computer Science* 167.1 (1996), pp. 131–170.

[200] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications." In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY'14. Lugano, Switzerland: ACM, 2014, pp. 25–36.

[201] Guido Salvaneschi and Mira Mezini. "Debugging for Reactive Programming." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE'16. Austin, TX, USA: ACM, 2016, pp. 796–807.

[202] Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. "Serialization-Aware Call Graph Construction." In: *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. SOAP'21. Virtual Event, Canada: ACM, 2021, pp. 37–42.

[203] Joanna C. S. Santos, Reese A. Jones, and Mehdi Mirakhorli. "Salsa: Static Analysis of Serialization Features." In: *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs*. FTfJP'20. Virtual Event, USA: ACM, 2020, pp. 18–25.

[204] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. "On Fast Large-Scale Program Analysis in Datalog." In: *Proceedings of the 25th International Conference on Compiler Construction*. CC'16. Barcelona, Spain: ACM, 2016, pp. 196–206.

[205] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. "Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?" In: *ACM Computing Surveys* 49.1 (2016), pp. 4:1–4:37.

[206] Philipp Dominik Schubert. *PhASAR*. https://phasar.org. [Online; accessed 19-September-2022].

[207] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. “PhASAR: An Inter-procedural Static Analysis Framework for C/C++.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’19. Prague, Czech Republic: Springer, 2019, pp. 393–410.

[208] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. “Into the Woods: Experiences from Building a Dataflow Analysis Framework for C/C++.” In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation*. SCAM’21. Luxembourg City, Luxembourg: IEEE, 2021, pp. 18–23.

[209] Olin Shivers. “Control Flow Analysis in Scheme.” In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI’88. Atlanta, GA, USA: ACM, 1988, pp. 164–174.

[210] Olin Shivers. “Control-Flow Analysis of Higher-Order Languages or Taming Lambda.” PhD thesis. Carnegie Mellon University, PA, USA, 1991.

[211] Robert Silvers et al. *Review of the December 2021 Log4j Event*. Tech. rep. Cyber Safety Review Board, 2022.

[212] Jagsir Singh and Jaswinder Singh. “Challenges of Malware Analysis: Obfuscation Techniques.” In: *International Journal of Information Security Science* 7.3 (2018), pp. 100–110.

[213] Yannis Smaragdakis. *Doop Benchmarks*. `https://bitbucket.org/yanniss/doop-benchmarks`. [Online; accessed 28-October-2022].

[214] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. “More Sound Static Handling of Java Reflection.” In: *Programming Languages and Systems*. APLAS’15. Pohang, South Korea: Springer, 2015, pp. 485–503.

[215] Yannis Smaragdakis and Martin Bravenboer. “Using Datalog for Fast and Easy Program Analysis.” In: *Datalog Reloaded*. Datalog 2.0’10. Oxford, UK: Springer, 2010, pp. 245–251.

[216] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. “Pick Your Contexts Well: Understanding Object-Sensitivity.” In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL’11. Austin, TX, USA: ACM, 2011, pp. 17–30.

[217] Yannis Smaragdakis and George Kastrinis. “Defensive Points-To Analysis: Effective Soundness via Laziness.” In: *32nd European Conference on Object-Oriented Programming*. ECOOP’18. Amsterdam, The Netherlands: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, pp. 23:1–23:28.

[218] Johannes Späth, Karim Ali, and Eric Bodden. “IDE$^{al}$: Efficient and Precise Alias-Aware Dataflow Analysis.” In: *Proceedings of the ACM on Programming Languages*. OOPSLA’17 1.OOPSLA (2017), pp. 99:1–99:27.

[219] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. “Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java.” In: *30th European Conference on Object-Oriented Programming*. Vol. 56. ECOOP’16. Rome, Italy: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, pp. 22:1–22:26.

[220] National Institute of Standards and Technology. *CVE-2021-44228*. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. [Online; accessed 06-September-2022]. 2021.

[221] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. “Demanded Abstract Interpretation.” In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI’21. Virtual Event, Canada: ACM, 2021, pp. 282–295.

[222] Arran Stewart, Rachel Cardell-Oliver, and Rowan Davies. “Fine-grained classification of side-effect free methods in real-world Java code and applications to software security.” In: *Proceedings of the Australasian Computer Science Week Multiconference*. ACSW’16. Canberra, Australia: ACM, 2016, pp. 37:1–37:7.

[223] Alexandru Sălcianu and Martin Rinard. *A Combined Pointer and Purity Analysis for Java Programs*. Tech. rep. MIT-CSAIL-TR-2004-030. MIT Computer Science and Artificial Intelligence Laboratory, 2004.

[224] Alexandru Sălcianu and Martin Rinard. “Purity and Side Effect Analysis for Java Programs.” In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 5. VMCAI’05. Paris, France: Springer, 2005, pp. 199–215.

[225] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. “On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation.” In: *Programming Languages and Systems*. APLAS’18. Wellington, New Zealand: Springer, 2018, pp. 69–88.

[226] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. “On the Recall of Static Call Graph Construction in Practice.” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE’20. Seoul, South Korea: ACM, 2020, pp. 1049–1060.

[227] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. "Practical Virtual Method Call Resolution for Java." In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'00. Minneapolis, MN, USA: ACM, 2000, pp. 264–280.

[228] Chungha Sung, Markus Kusano, and Chao Wang. "Modular Verification of Interrupt-Driven Software." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE'17. Urbana-Champaign, IL, USA: IEEE, 2017, pp. 206–216.

[229] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. "Incrementalizing Lattice-Based Program Analyses in Datalog." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 139:1–139:29.

[230] Tian Tan, Yue Li, and Jingling Xue. "Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting." In: *Static Analysis*. SAS'16. Edinburgh, UK: Springer, 2016, pp. 489–510.

[231] Tian Tan, Yue Li, and Jingling Xue. "Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'17. Barcelona, Spain: ACM, 2017, pp. 278–291.

[232] Jubi Taneja, Zhengyang Liu, and John Regehr. "Testing Static Analyses for Precision and Soundness." In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO'20. San Diego, CA, USA: ACM, 2020, pp. 81–93.

[233] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies." In: *2010 Asia Pacific Software Engineering Conference*. APSEC'10. Sydney, Australia: IEEE, 2010, pp. 336–345.

[234] Frank Tip and Jens Palsberg. "Scalable Propagation-Based Call Graph Construction Algorithms." In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'00. Minneapolis, MN, USA: ACM, 2000, pp. 281–293.

[235] Oksana Tkachuk and Matthew B. Dwyer. "Adapting Side Effects Analysis for Modular Program Model Checking." In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE'03. Helsinki, Finland: ACM, 2003, pp. 188–197.

[236] John Toman and Dan Grossman. "Taming the Static Analysis Beast." In: *2nd Summit on Advances in Programming Languages*. SNAPL'17. Asilomar, CA, USA: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 18:1–18:14.

[237] Matthew S. Tschantz and Michael D. Ernst. "Javari: Adding Reference Immutability to Java." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'05. San Diego, CA, USA: ACM, 2005, pp. 211–230.

[238] Raja Vallée-Rai. "Soot: A Java Bytecode Optimization Framework." PhD thesis. McGill University, Canada, 2000.

[239] Raja Vallée-Rai and Laurie J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Tech. rep. Sable Research Group. McGill University, 1998.

[240] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. "WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection." In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA'15. Baltimore, MD, USA: ACM, 2015, pp. 71–82.

[241] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. "A Large Scale Investigation of Obfuscation Use in Google Play." In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC'18. San Juan, PR, USA: ACM, 2018, pp. 222–235.

[242] John Whaley. "Context-sensitive pointer analysis using binary decision diagrams." PhD thesis. Stanford University, CA, USA, 2007.

[243] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. "Using Datalog with Binary Decision Diagrams for Program Analysis." In: *Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer, 2005, pp. 97–118.

[244] John Whaley and Monica S. Lam. "Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams." In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI'04. Washington DC, USA: ACM, 2004, pp. 131–144.

[245] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. "Dynamic Purity Analysis for Java Programs." In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE'07. San Diego, CA, USA: ACM, 2007, pp. 75–82.

[246] Jisheng Zhao, Ian Rogers, Chris Kirkham, and Ian Watson. "Pure Method Analysis Within Jikes RVM." In: *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS'08. Paphos, Cyprus, 2008.

[247] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. "Object and Reference Immutability using Java Generics." In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE'07. Dubrovnik, Croatia: ACM, 2007, pp. 75–84.

[248] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. "Ownership and Immutability in Generic Java." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA'10. Reno/Tahoe, NV, USA: ACM, 2010, pp. 598–617.