Jens Van der Plas

Incremental Static Program Analysis through Reified Computational Dependencies

Dissertation submitted in partial fulfilment of the
requirements for the degree of Doctor of Sciences

4 November 2024

Promotors:
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel, Belgium
Prof. Dr. Quentin Stiévenart, Université du Québec à Montréal, Canada

Jury:
Prof. Dr. Dominique Maes, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Jens Nicolay, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium
Prof. Dr. Lynn Houthuys, Vrije Universiteit Brussel, Belgium
Prof. Dr. Görel Hedin, Lunds universitet, Sweden
Prof. Dr. Ben Hermann, Technische Universität Dortmund, Germany

Vrije Universiteit Brussel
Faculty of Sciences and Bioengineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 978-94-6494-861-5
NUR CODE: 980
THEMA: UMZT

Abstract

Over the last few decades, computers have become an indispensable part of modern society. As the programs running on these computers play an essential role in everyday life, for example in banking and communication, it is crucial that they are reliable. To this end, developers have come to rely on static analysis tools to verify a wide range of program properties without actually running the program. Static analyses are typically integrated into modern software development environments and continuous integration systems to safeguard software quality throughout the entire development process.

During the development process, software developers continuously apply small changes to the program. It is therefore not only important for a static analysis to deliver precise feedback, but also to efficiently update its feedback whenever the program is modified. To this end, an incremental static analysis avoids a full recomputation of the analysis result. Instead, upon a change in the program, it reuses and updates the previously computed result, saving valuable analysis time.

It is challenging to develop an incremental analysis that is sufficiently powerful to verify a wide range of program properties and that supports complex language features found in contemporary programming languages. In this work, we present a novel generic approach to construct such analyses by using reified computational dependencies. We show how an analysis that reifies the computational dependencies within the analysed program can be rendered incremental. By relying on these dependencies, our incremental analysis restricts the impact of a change to the affected parts of the analysis result. Moreover, we impose minimal requirements on the analysis itself, allowing a broad applicability of our approach to incrementalisation.

To preserve the precision of the resulting incremental analysis, we introduce three complementary result-invalidation strategies that limit the loss in precision. These strategies are built around the core idea of interleaving invalidation with computation.

Samenvatting

Tijdens de laatste decennia werden computers een onmisbaar deel van de moderne samenleving. Gezien de programma's die op deze computers uitgevoerd worden een essentiële rol spelen in het alledaagse leven, bijvoorbeeld in bankieren en communicatie, is het cruciaal dat ze betrouwbaar zijn. Om deze reden begonnen ontwikkelaars op statische analyses te vertrouwen om een breed scala aan programma-eigenschappen te verifiëren zonder het programma daadwerkelijk uit te voeren. Statische analyses worden doorgaans in moderne "integrated development environments" en "continuous integration"-systemen geïntegreerd om de softwarekwaliteit te bewaken gedurende het hele ontwikkelingsproces.

Gedurende het ontwikkelingsproces maken softwareontwikkelaars voortdurend kleine veranderingen aan het programma. Het is daarom niet enkel belangrijk dat een statische analyse precieze feedback geeft, maar ook dat deze efficiënt haar feedback updatet telkens wanneer het programma aangepast wordt. Om deze reden vermijdt een incrementele statische analyse het volledig herberekenen van het analyseresultaat. In plaats daarvan hergebruikt en updatet ze het voorgaande resultaat telkenmale het programma gewijzigd wordt, waardoor kostbare analysetijd bespaard wordt.

Het is uitdagend om een incrementele analyse te ontwikkelen die voldoende krachtig is om een breed scala aan programma-eigenschappen te verifiëren en die de complexe eigenschappen van hedendaagse programmeertalen kan ondersteunen. In dit werk presenteren we een nieuwe generieke methode om dergelijke analyses te construeren door gereïficeerde computationele afhankelijkheden te gebruiken. We tonen hoe een analyse die de computationele afhankelijkheden in het geanalyseerde programma reïficeert incrementeel gemaakt kan worden. Door op deze afhankelijkheden te steunen, kan onze incrementele analyse de impact van een verandering begrenzen tot de getroffen delen van het analyseresultaat. Daarenboven stellen we minimale vereisten aan de analyse zelf, hetgeen onze incrementalisatiemethode breed toepasbaar maakt.

Om de precisie van de resulterende incrementele analyse te bewaren, introduceren we drie complementaire strategieën voor resultaatrevocatie die het precisieverlies beperken. Deze strategieën zijn ontworpen rond het kernidee van het afwisselen van revocatie en computatie.

Acknowledgements

A PhD is a journey one never goes through alone. The first points of support, feedback, encouragement, and sometimes of critiques, are the promotors. Therefore, I first want to thank my promotors, Coen and Quentin, for their support during the past years. At several points, when the road ahead was unclear, their advice was essential. Without Coen, I would never have started this PhD in the first place. I want to thank him for his guidance throughout this entire journey. I want to thank Quentin for his input and feedback. Whenever I ask him a question – no matter how random or detailed – he always comes back with a helpful answer and points me in the right direction. It's fair to say that I sometimes didn't make things easy, but my promotors never stopped believing in a good outcome for this adventure.

Second, I also want to thank my jury members, Ben Hermann, Dominique Maes, Görel Hedin, Jens Nicolay, Lynn Houthuys, and Wolfgang De Meuter, for taking the time to read this dissertation and for their insightful comments and constructive feedback during the private defence. I also want to thank Görel, Ben, and Quentin for squeezing a trip to Brussels into their busy schedules to attend the private defence in person.

Throughout my PhD, I had the privilege of working at the Software Languages Lab. I want to thank all SOFTies for all these years of companionship, for all our interesting and vibrant discussions, for their valuable feedback, and for the many Opinio(n)s that crossed my path. I cannot remember how often we solved all world problems during our discussions. I especially want to thank the people in the CAMP subgroup and in particular the MAFties Noah and Bram with whom I waded through the swamp that is called static analysis. One other person I need to thank in particular is Maarten, with whom I shared an office during these past years. I want to thank him for his companionship, for his advice and feedback, and for enduring my sometimes endless talking.

In these acknowledgements, I could not leave out my family, my parents and my brother, who have been supporting me, not only throughout this PhD but since

the day I was born. I want to thank them for the support, love, and help they have given me all this time. Finally, I also want to thank my friends, and in particular Ruben, for all our adventures together and for being there for me whenever I need it. No man is an island, especially not while endeavouring a PhD.

Contents

List of Figures

List of Listings

List of Tables

Nomenclature

The following lists compile the most important acronyms, abbreviations, symbols and notations used in this dissertation.

List of Acronyms and Abbreviations

ANF	Administrative Normal Form
AST	Abstract Syntax Tree
CFG	Control-flow Graph
CFL	Context-free Language
CI	Component Invalidation
DI	Dependency Invalidation
DSL	Domain-specific Language
EDB	Extensional Database
IDB	Intensional Database
IDE	Integrated Development Environment
SCC	Strongly Connected Component
WI	Write Invalidation

List of Symbols and Notations

$\mathcal{A}$	Atomic evaluation function, page 29

a	Address in the global store in formal notations, page 96
a_x	Address of the variable $\mathtt{x}$, page 94
α, β,...	Components in formal notations (except σ), page 22
$\bot$	Bottom element of a lattice, page 15
$\mathbb{C}$	Call cache, page 92
C_α	Set of components discovered/called by α , page 32
$\triangleleft$	Domain anti-restriction operator, page 102
D	Dependency map, page 23
Inter	Inter-component analysis, page 22
Intra	Intra-component analysis, page 22
$\sqcup$	Join operator, page 15
Name	Components in examples (denoted by their name in small caps), page 22
$\mathbb{P}$	Provenance (cache), page 98
$\mathcal{P}(S)$	Power set of S, page 15
$P_{a,\alpha}$	Contribution of α to a, page 96
P_a	Provenance of a, page 96
$\mathbb{R}$	Dependency cache, page 93
R_α	Set of read dependencies of α , page 93
σ	Global value store, page 22
$\top$	Top element of a lattice, page 15
$\cup$	Pointwise union, page 28
$\uplus$	Disjoint union, page 28
U_α	Set of write effects generated during the analysis of α , page 23

v	(Abstract) value, page 98
V	Visited set, page 23
$\mathbb{W}$	Write cache, page 97
WL	Worklist, page 22
W_α	Set of all addresses written to by α , page 97

1

Introduction

It is difficult to overemphasise the importance of software in modern society. Over the course of the past decades, and especially since the turn of the millennium, the use of software has soared. Now, software is present in almost every aspect of everyday life, from communication to banking and beyond. Together with the increased use of software, the potential consequences of software faults have increased as well. Especially in critical systems, the impact of a software error may cause a dramatic chain of events, thereby impacting the lives of millions of citizens. In addition, software must be made secure to protect it and its users from malicious adversaries.

Already today, there exists a multitude of examples where faulty software caused significant losses. One example is the *Northeast blackout of 2003* [90, 97, 142], where a race condition in an energy management system was a crucial step in a chain of events that would cause a blackout affecting approximately 50 million people spread over an area of around 24 000 square kilometers in the United States and Canada, with financial losses estimated between 4 billion and 10 billion dollars in the United States alone. Another example is the maiden flight of the Ariane 5 rocket in June 1996 [10, 34, 50], where a software error in the flight control system caused the rocket to significantly veer off course, eventually triggering the rocket's self-destruction at an altitude of merely 4 kilometers. The root cause of this disaster was the conversion of a 64-bit floating point number to a 16-bit signed integer, which caused an integer overflow. The piece of Ada code that caused the error, shown in Listing 1.1, had been reused from the earlier Ariane 4 rocket but had not been adapted to suit the conditions under which the newer Ariane 5 rocket would operate. In addition, part of the diagnostic data ended up in the central on-board

```
1  ...
2  declare
3    vertical_veloc_sensor: float;
4    horizontal_veloc_sensor: float;
5    vertical_veloc_bias: integer;
6    horizontal_veloc_bias: integer;
7    ...
8  begin
9    declare
10     pragma suppress(numeric_error, horizontal_veloc_bias);
11   begin
12     sensor_get(vertical_veloc_sensor);
13     sensor_get(horizontal_veloc_sensor);
14     vertical_veloc_bias := integer(vertical_veloc_sensor);
15     horizontal_veloc_bias := integer(horizontal_veloc_sensor);
16     ...
17   exception
18     when numeric_error => calculate_vertical_veloc();
19     when others => use_irs1();
20   end;
21 end irs2;
```

Listing 1.1: Excerpt from the code of the *Inertial Reference System* operating in the Ariane 501 rocket [50]. Once the horizontal bias computed by the system reached 32768.0, a critical overflow would happen during the type conversion on Line 15. This situation could never arise on an Ariane 4 rocket, which followed a different trajectory just after take off that resulted in lower horizontal velocities during the operation of this part of the software [10].

computer which interpreted it as flight data, causing the rocket to deviate from its supposed trajectory. The disaster with the Ariane 5 rocket incurred a loss of approximately 370 million dollars at the time.

Nowadays, the bugs that lead to these two disasters would probably not have gone unnoticed. They could, for example, be detected by *static analyses*, which are currently widely used by developers, especially during the development of safety-critical software. A static analysis reasons about and computes the behavioural properties of a program without executing the program. This allows to uphold certain quality guarantees for the software that is being developed and to monitor the code quality throughout the development process, thus even before the software is deployed. As a result, when static analysis is used appropriately, organisations can be more confident in the code produced by their developers.

1.1 Research Context

In the modern-day software development process, static analyses have become an indispensable tool for developers to detect mistakes in their programs even before the programs are executed. Static analyses form the foundation of code smell, bug, and vulnerability detection tools used in modern software engineering processes. Over the course of many years, a variety of different static analyses has been developed for a wide range of application domains. Application domains for static analysis include purity [95] and coupling [94] analyses for JavaScript, security analyses for web applications [140, 141], concurrency-bug detection [129, 167], resource usage analysis [68], alias and points-to analysis [33, 67, 170], worst-case execution time analysis [1, 20, 75], and many more [15, 87]. Alongside the development of new analyses, many static analysis frameworks have been introduced as well. Such analysis frameworks allow developers to instantiate their analyses, and may also provide various static analyses themselves, in a ready-to-use environment and therefore facilitate the use of these analyses. Examples of static analysis frameworks are Doop [18], OPAL [37], Soot [143], PhASAR [114–116], and Infer [21, 22].

Static analyses can be integrated into different stages of the software engineering process, such as in the IDE, in the continuous integration pipeline or during code review, for example [155]. However, the integration of a static analysis within the development pipeline entails a timing constraint: in order to be integrated, static analyses must be able to keep up with the development cycle and produce results in a timely manner. Recent literature has found that timely feedback from a static analysis can dramatically improve the fix rate of reported defects from 0% to 70% as this avoids expensive mental context switches for developers and thereby diminishes the effort required from developers to fix the signalled defects [14, 60]. Yet, the execution of complex analyses may require a significant amount of time, especially when run on large code bases.

During the software development cycle, often, multiple small changes that typically only impact a limited part of the program are made [5, 6, 61, 104]. Having to reanalyse a program in full upon every (small) change is costly, especially when the program under analysis is large. This may impede the fixing of defects by the developers or even prevent the integration of the analysis (early) in the development cycle altogether. Instead, incremental static analyses can be used. Such analyses aim to deliver feedback faster upon program changes by reusing and updating parts of the result obtained from the analysis of a prior version of the program. To this end, an incremental analysis must be able to *efficiently* link the

code changes to the affected parts of the analysis result and to efficiently update the analysis result in accordance with the code changes while guaranteeing its correctness.

1.2 Problem Statement and Research Question

Although incremental static analyses are not new, existing techniques have their limitations. On the one hand, bespoke incrementalisation techniques are developed for a specific purpose. Such incremental analyses are tailored to a specific analysis or application domain and can use domain-specific knowledge, such as e.g., a *transitivity property* [76], to efficiently create an incremental analysis. Some bespoke incremental analyses require the call graph of the program to be known upfront, for which they cannot analyse programs written in highly-dynamic languages with dynamic typing, polymorphism, and several forms of late binding and closures. If the analysis also requires the call graph to be unaffected by the program changes, it cannot handle all changes incrementally. For example, such an analysis could not incrementally process the addition or removal of function calls. On the other hand, incremental analysis frameworks provide a domain-specific language for the implementation of analyses that are then incrementalised by the framework. Frameworks are typically restricted to a specific class of analyses that they can render incremental.

The two approaches just mentioned have been well-explored but require the reimplementation of an analysis to make it incremental, either in a custom manner that is to be devised for the problem at hand or in a framework that offers incrementalisation capabilities. Depending on the problem domain of the bespoke analysis or on the assumptions made by the incrementalisation framework, the incrementalisation is only applicable to a certain specific class of analyses.

A recent survey of the broader field of incremental computation [77] categorises existing work in this field into three broad categories: incremental algorithms, incremental program-evaluation frameworks, and incremental-algorithm-and-program derivation methods. As mentioned, in the field of incremental static analysis, plenty of work exists that falls into the first two categories. These thus have already been studied well. However, the same seems not to be true for the third category: there seems to be a lack of work describing approaches to render existing analyses incremental. With the work presented in this dissertation, we take the first steps into exploring methods to make existing static analyses incremental.

We are unaware of the existence of any *systematic design method* for incremental analyses that results in general-purpose incremental static program analyses applicable to contemporary, highly-dynamic programs with non-deterministic behaviour. In this dissertation, we investigate a general method to render a general-purpose static analysis incremental. Concretely, we only assume the analysis to be effect-driven [96] and that it allows us to track which expressions are analysed, and investigate how to render effect-driven analyses incremental while preserving the generality of the approach we develop. An effect-driven analysis can be seen as a traditional machine-based analysis that performs a fixed-point computation and where inter-dependent (coarse-grained) states that rely on some global analysis state are analysed separately.

1.3 Contributions

In this dissertation, we study the first systematic method to render a static analysis incremental. Concretely, our contributions are the following:

- We start from an effect-driven analysis and show how the computational dependencies within a program, which are reified by the effect-driven analysis, can be leveraged to obtain incrementality and to bound the impact of changes on existing analysis results. The approach consists of two steps: a change-impact calculation and the updating of the analysis result.
- We formulate our approach as general as possible by imposing minimal requirements on the analysis itself and essentially treat the actual analysis as a black box. This assures the generality and broad applicability of our method.
- We show how precision can be regained by introducing three complementary strategies for result invalidation that do not impose restrictions on the analysis. Our novel method to remove outdated parts of the analysis result interleaves invalidation with recomputation, preventing the over-approximation of outdated results and upholding the view of the analysis as a black box. (We still require the analysis to be effect-driven as the effects form the interface between the analysis and our incrementalisation method.)
- We present a formal specification of our approach and implement it in MAF [152], a research framework developed at our lab to study effect-driven static analyses.
- Using our implementation, we perform a thorough experimental evaluation of the incrementalisation method by applying it to several effect-driven

analyses. To this end, we compile two benchmarking suites of programs with changes. We show that the resulting analyses are sound and measure their precision and performance.

The incrementalisation method presented in this dissertation does not fully preserve the precision of the original non-incremental analysis. We hypothesise that a small precision loss is acceptable in use cases where performance is more important than precision. For example, analyses run within the IDE need to produce results fast to avoid hampering the developer. Here, the incrementalisation of an analysis could enable running it within the IDE, whereas this may not have been possible previously. A more precise, from-scratch analysis can be run afterwards, e.g., as part of a continuous-integration system or after several incremental updates of the analysis result, or simultaneously in a separate thread. This includes analyses used for program comprehension, where an incremental analysis could quickly give developers an information about the code. For example, an incremental analysis may compute an updated call graph faster than a from-scratch analysis and identify dead code more rapidly. In these cases, a small number of false positives is acceptable and the resulting incremental analysis may not report all dead code reported by the original non-incremental analysis, for example. However, our method preserves soundness so no false negatives will occur in the result: the incremental analysis will not report live code as dead code. To keep the precision loss minimal, we propose three strategies for precision recovery in Chapter 5.

1.4 Supporting Publications

In this section, we list our work that was published during the course of our PhD and that supports the material presented in this dissertation. The work listed below has been peer reviewed and was published at various international venues. Conference papers have been presented during the respective events by the first author.

> Van Es, N., **Van der Plas, J.**, Stiévenart, Q., and De Roover, C. (2020). MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, Los Alamitos, CA, USA. IEEE Computer Society.

This paper presents the Modular Analysis Framework (MAF), a research framework for the study of effect-driven analyses. The paper describes the modular design of

the framework itself, which facilitates experimenting with a variety of analysis configurations. All our contributions presented in this dissertation have been implemented in MAF.

> **Van der Plas, J.**, Stiévenart, Q., Van Es, N., and De Roover, C. (2020). Incremental Flow Analysis through Computational Dependency Reification. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, pages 25–36, Los Alamitos, CA, USA. IEEE Computer Society.

In this paper, we present the idea of using reified computational dependencies to drive an incremental analysis. We introduce a general approach to incrementalise effect-driven analyses, which support programs written in highly-dynamic programming languages with higher-order functions and closures as well as various analysis configurations, including different lattices and context sensitivies, for example. The work presented in this paper is the topic of Chapter 4.

> Kursun, T. R., **Van der Plas, J.**, Stiévenart, Q., and De Roover, C. (2022). RacketLogger: Logging and Visualising Changes in DrRacket. In D. Verna, editor, *Proceedings of the 15th European Lisp Symposium, ELS 2022, Porto, Portugal, April 21-22, 2022*, pages 61–68. ELSAA.

This paper presents RacketLogger, a change logger for the DrRacket IDE supporting the logging of changes made to programs written in Scheme-like languages such as R5RS Scheme and Racket. RacketLogger can log changes on a textual level and on the level of the abstract syntax tree. The extraction and representation of program changes is the topic of Section 3.2.

> **Van der Plas, J.**, Stiévenart, Q., and De Roover, C. (2023). Result Invalidation for Incremental Modular Analyses. In C. Dragoi, M. Emmi, and J. Wang, editors, *Proceedings of the 24th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023, Boston, MA, USA, January 15-21, 2023*, volume 13881 of *Lecture Notes in Computer Science*, pages 296–319, Cham, Switzerland. Springer.

In this paper, we build on the work disseminated in the paper "Incremental Flow Analysis through Computational Dependency Reification" and present three complementary strategies to improve the precision of incrementalised effect-driven analyses. The work presented in this paper is the topic of Chapter 5.

> Wauters, C., **Van der Plas, J.**, Stiévenart, Q., and De Roover, C. (2023). Change Pattern Detection for Optimising Incremental Static Analysis. In L. Moonen, C. D. Newman, and A. Gorla, editors, *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, pages 49–60, Los Alamitos, CA, USA. IEEE.

This paper presents an optimisation for incremental static program analysis based on change patterns for which the impact on the analysis result can be predicted. The paper shows that for such patterns, the analysis result can be updated using domain-specific knowledge, making a traditional incremental update of the analysis result unnecessary. Our results show that, for three behaviour-preserving change patterns, the time needed by an incremental update can be reduced significantly. We reference this work in Section 4.1.

1.4.1 Other Publications

For completeness' sake, we also list our other peer-reviewed publications that were published during the course of our PhD but that do not support directly the material presented in this dissertation. As before, conference papers have been presented during the respective events by the first author.

> Van Es, N., Stiévenart, Q., **Van der Plas, J.**, and De Roover, C. (2020). A Parallel Worklist Algorithm for Modular Analyses. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, pages 25–36, Los Alamitos, CA, USA. IEEE Computer Society.

> Stiévenart, Q., Van Es, N., **Van der Plas, J.**, and De Roover, C. (2021). A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, **181**, 111042.

This paper introduces a parallelisation strategy for effect-driven modular analyses, which is based on a novel parallel worklist algorithm. The journal extension of this paper also presents a comparative study of several worklist algorithms for parallelised effect-driven analyses to further increase the attainable speed-ups.

> **Van der Plas, J.**, Nicolay, J., De Meuter, W., and De Roover, C. (2023). ModInF: Exploiting Reified Computational Dependencies for Information Flow Analysis. In H. Kaindl, M. Mannion, and L. A. Maciaszek, editors, *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023*, pages 420–427, Setúbal, Portugal. INSTICC, SciTePress.

In this paper, we employ an effect-driven analysis to perform an information-flow analysis. We show that the effects generated by an effect-driven analysis model the inter-component information flow and extend the intra-component analysis to obtain the full information flow data. We refer to this work in Section 6.3.3 when describing the future work.

1.5 Outline of the Dissertation

The remainder of this dissertation is structured as follows:

In Chapter 2, we introduce background material to aid the comprehension of the remainder of the dissertation. First, we introduce the broad concept of static analysis and the principle of approximation. We then introduce lattices as a way to represent abstract values in a static analysis. Finally, we introduce effect-driven static analysis, the analysis method for which we introduce an incrementalisation approach in this dissertation.

Next, Chapter 3 introduces incremental static analysis, a method to speed up the analysis of consecutive versions of a given program. Then, we discuss representations for program changes that are used in the literature on incremental static program analysis. Finally, we provide an overview of the literature on incremental static analysis, and of the literature in the related fields of staged and demand-driven static analysis and of incremental computation.

In Chapter 4, we first introduce change expressions, the representation for program changes used in this work. We then present an initial, monotonic approach to render an effect-driven static analysis incremental by exploiting the computational dependencies it reifies by means of effects. To demonstrate the generality of the approach, we apply it to both a function-modular and a thread-modular analysis. The presented approach, however, does not allow the invalidation of outdated parts of the analysis result, allowing imprecision to arise in the result.

Chapter 5 improves upon the above approach by reducing its precision loss. To this end, three complementary strategies for precision recovery are introduced. The core invalidation principle of these strategies is to *interleave* invalidation with recomputation. This has the advantage that our approach remains general, as the actual analysis can remain a black box.

Finally, in Chapter 6, we conclude by providing an overview of the work presented in this dissertation, by recapitulating our contributions, and by presenting possible directions for future work on this topic.

The research presented in this dissertation was partially supported by the Research Foundation – Flanders (FWO) under grant numbers 11F4820N and 11F4822N, by the Cybersecurity Initiative Flanders (CIF), and by the Cybersecurity Research Program Flanders (CRPF).

2

Supporting Material

This chapter introduces background material that supports the work presented in this dissertation. We assume the reader is familiar with the fundamentals of static program analysis.

This chapter is structured as follows. Section 2.1 first introduces some general yet important notions related to static analysis. Section 2.2 then introduces lattices, which are often used in static (dataflow) analyses as a means of value abstraction. Lattices are not only used in the modular static analysis approach that is used in this dissertation, but they also play an important role in the Write Invalidation technique introduced in Chapter 5. Third, Section 2.3 introduces modular static analysis. In this section, we specifically focus on effect-driven modular static analysis, a specific method to static analysis that reifies the computational dependencies within a program, which we leverage in our work to obtain incrementality. Finally, we conclude in Section 2.4.

An introduction to incremental static program analysis, which lies at the heart of this dissertation, is given in Chapter 3.

2.1 Fundamentals of Static Analysis

In this dissertation, we improve the run-time efficiency of a static analysis by rendering it incremental. To this end, we first briefly introduce some important notions concerning static analysis in general.

A static analysis is used to infer behavioural, semantic properties of programs without actually executing the programs under analysis. Instead, a static analysis reasons about the source code, thereby inferring the program properties of interest, such as, e.g., the absence of certain types of bugs. Unfortunately, computing any non-trivial property is undecidable in general [110]. To this end, a static analysis needs to approximate the actual program behaviour to guarantee its termination. This means that an analysis may not always produce accurate or conclusive answers in all situations: due to the approximation, situations may arise where the analysis is uncertain. To always produce an answer, two types of approximations are possible: an over-approximation and an under-approximation.

An *over-approximating analysis* may consider execution paths in the program that are not reachable, that is, it over-approximates the program behaviour: program behaviours for which the analysis cannot conclusively say that they cannot take place are considered during the analysis as well. For this reason, an over-approximating static analysis reports all programs that exhibit a certain behaviour, e.g., that contain certain defect, but it may also report programs that, in practice, can never exhibit the behaviour in question.

An *under-approximating analysis* only considers execution paths that are guaranteed to be taken in the program and therefore may not consider all possible paths of execution, that is, it under-approximates the program behaviour: program behaviours for which the analysis cannot conclusively say that they will take place during program execution are not considered during the analysis. For this reason, an under-approximating static analysis only reports programs that are guaranteed to exhibit a certain behaviour, but may not report all programs that can possibly exhibit the behaviour in question.

These two types of approximating behaviour lead to two types of analyses.

Definition 1 *A static analysis is **sound** if it is over-approximating. A static analysis is **complete** if it is under-approximating.*

In the literature, various definitions for soundness and completeness can be found, for example in terms of the computation of semantic properties [111, 144]. In these definitions, however, the property must be chosen carefully, to be e.g., the *absence* of certain types of defects. For this reason, we use Definition 1, which defines soundness and completeness in function of the approximation made by the analysis and is in line with definitions found in the literature [27, 70, 87].

In this dissertation, we assume and require soundness for all analyses. A sound analysis is guaranteed to find all program defects. Therefore, its application in a development pipeline is useful.

A sound analysis may give rise to false positives, as it may take into account program behaviour that cannot occur in practice. A complete analysis may, on the contrary, give rise to false negatives, as it may not take into account program behaviour that can occur in practice. Clearly, a sound static analysis generating fewer false positives is preferred over one generating many false positives as the latter reduces the usability of the analysis. Similarly, a complete static analysis generating fewer false negatives is preferred over one generating many false negatives.

Definition 2 *The **precision** of an analysis is a measure to indicate how closely the approximation made by the analysis approaches all possible program behaviours.*

When a static analysis raises alarms that can be inspected for false positives and false negatives, such as alarms indicating an insecure data flow, precision is often defined as the ratio of the number of true positives to the total number of alarms raised, and a related measure called *recall* can be defined as ratio of alarms raised to the true number of alarms that should have been raised. A complete analysis is then fully precise whereas a sound analysis then has a perfect recall. However, precision often is not or cannot be measured in this way, e.g., when there is no client analysis available that can generate alarms. As a result, an abundance of precision metrics can be found in the literature (see e.g., [32, 83, 96, 120, 137, 150]), reflecting that it is often difficult to quantify the precision of an analysis. Therefore, precision metrics are normally used comparatively, to quantify the difference in precision between two analyses on a given set of benchmarks. It is this approach that we follow in this dissertation.

2.2 Lattices as an Abstract Value Representation

Techniques for static dataflow analysis such as Abstract Interpretation [28] reason about an *abstract* state space of which each abstract state represents a subset of the actual, concrete, and possibly infinite state space in which the program operates. The abstract state space is conceived in such a way that, when the analysis operates in it, it is guaranteed to terminate. This can be achieved by designing

the abstract state space to be finite, to converge in a finite number of steps, or by providing other mechanisms to guarantee the convergence of the analysis, such as widening [28, 29].

One element of the infinite state space in which programs operate, are values. For example, integers are elements of the infinite set $\mathbb{Z}$. In this case, we call $\mathbb{Z}$ the *concrete* domain of integers. An analysis, in contrast, reasons about values that are part of an *abstract* domain of values. Such abstract domains are typically formed by a lattice structure. The analysis then reasons about the elements of the abstract domain, i.e., over *abstract values*, and each abstract value represents a (possibly infinite) set of concrete values the program can encounter during its execution. An example of an abstract domain for integers is the sign domain, where integers are represented by their sign (positive, negative or zero). In this domain, the abstract value "positive" represents the infinite set of all strictly positive integers, $\mathbb{Z}_0^+$.

In this section, we first introduce some elementary notions related to lattices, which have been studied and utilised extensively in the mathematical literature (see e.g., [13, 16, 54, 55]). We then explain how lattices are used in a dataflow analysis and present the basic structure of two lattices that are frequently used in this dissertation: the type lattice and the constant-propagation lattice. Finally, we present some requirements on lattices that ensure the termination of an analysis.

2.2.1 Mathematical Foundation

We first define the notion of a lattice and briefly mention some of its most relevant properties.

Definition 3 *A **partially ordered set** or **poset** is a set S on which a binary relation $(\sqsubseteq) \subset S \times S$ is defined that is*

- *reflexive:* $\forall x \in S : x \sqsubseteq x$,
- *anti-symmetric:* $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$, *and*
- *transitive:* $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.

*The relation $(\sqsubseteq)$ is called a **partial order** on S. We denote the partially ordered set as $(S, \sqsubseteq)$. Two elements $x, y \in S$ are **comparable** if $x \sqsubseteq y$ or $y \sqsubseteq x$. Otherwise, they are **incomparable**.*

Definition 4 *A partially ordered set $(S, \sqsubseteq)$ satisfies the **ascending chain condition** if any increasing chain terminates, i.e., if any chain of strictly increasing elements is finite.*

Definition 5 *An element $u \in S$ is an **upper bound** of $X \subseteq S$ if $\forall x \in X : x \sqsubseteq u$. An upper bound u of X is the **least upper bound** or **supremum** of X, denoted $\bigsqcup X$, if for every upper bound v of $X : u \sqsubseteq v$. We define a corresponding binary operator $\sqcup$, called the **join** operator, so that $\bigsqcup\{x, y\}$ can be denoted as $x \sqcup y$.*

Definition 6 *An element $l \in S$ is a **lower bound** of $X \subseteq S$ if $\forall x \in X : l \sqsubseteq x$. A lower bound l of X is the **greatest lower bound** or **infimum** of X, denoted $\bigsqcap X$, if for every lower bound k of $X : k \sqsubseteq l$. We define a corresponding binary operator $\sqcap$, called the **meet** operator, so that $\bigsqcap\{x, y\}$ can be denoted as $x \sqcap y$.*

Definition 7 *A **lattice** is a partially ordered set L in which every subset of two elements has a supremum and an infimum. A **complete lattice** is a partially ordered set L where every subset of L has a supremum and an infimum in L. Every non-empty complete lattice has a **bottom** element, denoted $\bot$, so that $\bot = \bigsqcap L$ and a **top** element, denoted $\top$, so that $\top = \bigsqcup L$.*

By definition, any lattice of finite height is complete. Due to the anti-symmetry of the partial order relation, the supremum and infimum of every $X \subseteq S$ are unique.

Multiple lattices can also be combined to form a new lattice:

Definition 8 *We define the **product lattice** or **direct product** $S \times L$ of two lattices S and L to be the partially ordered set of all tuples (s, l) with $s \in S, l \in L$ using the following partial order: $(s_1, l_1) \sqsubseteq (s_2, l_2)$ if and only if $s_1 \sqsubseteq s_2$ in S and $l_1 \sqsubset l_2$ in L.*

It is guaranteed that if S and L are lattices, then so is $S \times L$ [16, 55]. In this way, a product of an arbitrary number of lattices can be created.

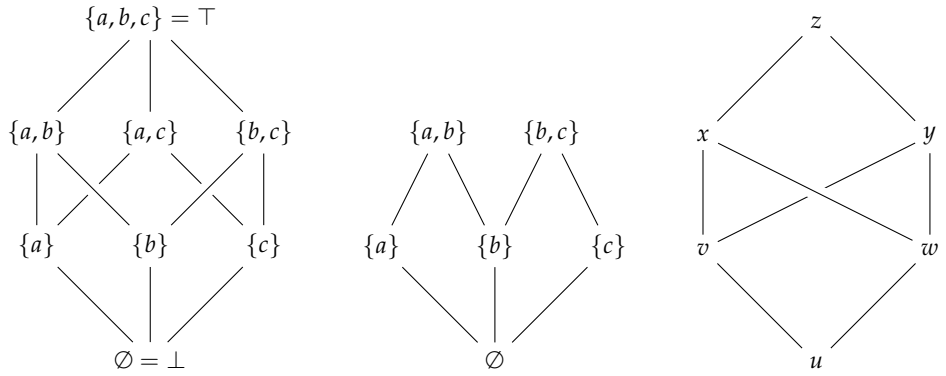
Definition 9 *We call a lattice a **power set lattice**, or simply a **set lattice**, if the elements of the partially ordered set form the power set $\mathcal{P}(S)$ of a given set S and the partial order is given by the set inclusion relation $\subseteq$.*

By definition, an ordered set where the elements form a power set and the partial order is given by the relation $\subseteq$ is a lattice: the ordering relation $\subseteq$ is reflexive, anti-symmetric and transitive, and therefore it is a partial order on $\mathcal{P}(S)$; the supremum of any subset of two elements can be computed by using the set union operator, $\cup$, which is the join operator of the lattice, and the infimum for every subset of two elements can be computed using the set intersection operator, $\cap$, which is the meet operator of the lattice. A set lattice is a complete lattice.

2.2.2 The Hasse Diagram

A partially ordered set $(S, \sqsubseteq)$ can be depicted by means of a *Hasse diagram*, which visualises the elements of the set and their partial order using a vertical structure. In the diagram, nodes represent elements of S. If $s_1 \sqsubset s_2$, then s_2 is drawn higher than s_1. A vertex is drawn between s_1 and s_2 if $s_1 \sqsubset s_2$ and there is no node $s_3 \in S$ for which $s_1 \sqsubset s_3 \sqsubset s_2$ holds. Thus, for two elements $s_1, s_2 \in S$ it holds that $s_1 \sqsubset s_2$ if there is an ascending chain from s_1 to s_2 in the diagram.

Example. Consider the Hasse diagram shown in Figure 2.1a, which represents the partially ordered set $(\mathcal{P}(\{a, b, c\}), \subseteq)$. As is shown in the diagram, the $\subseteq$ relation is a partial order on $\mathcal{P}(\{a, b, c\})$ and every subset of $\mathcal{P}(\{a, b, c\})$ has a supremum and an infimum. Therefore, the partially ordered set is a complete lattice. The join of two elements can be found by following the vertices in the diagram upwards. For example, $\{a\} \sqcup \{b, c\} = \{a, b, c\}$. In this case, the empty set is the bottom element of the lattice and the set $\{a, b, c\}$ is the top element of the lattice. On the contrary, the partially ordered set represented by the Hasse diagram shown in Figure 2.1b is not a lattice since not every subset of two elements has a supremum: there is no supremum for the subset $\{\{a, b\}, \{b, c\}\}$. Similarly, the partially ordered set shown in the Hasse diagram in Figure 2.1c is not a lattice because v and w do not have a supremum: x, y and z all are upper bounds for v and w, but none of them is the least upper bound.

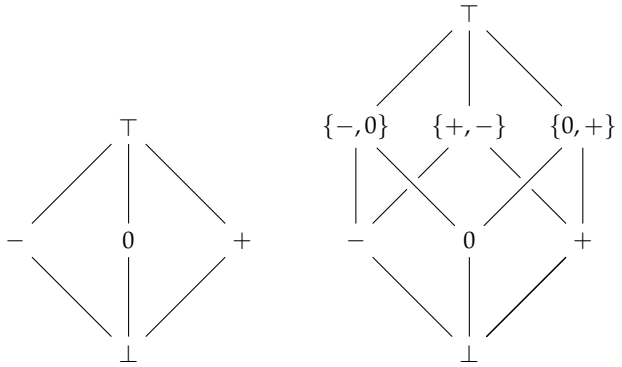


(a) Hasse diagram of the complete lattice $(\mathcal{P}(\{a, b, c\}), \subseteq)$.

(b) Hasse diagram of the partially ordered set $(\mathcal{P}(\{a, b\}) \cup \mathcal{P}(\{b, c\}), \subseteq)$.

(c) Hasse diagram of a partially ordered set with elements $\{u, v, w, x, y, z\}$.

Figure 2.1: Examples of Hasse diagrams as a means to visualise partially ordered sets.



(a) Three-layered sign lattice.

(b) Four-layered sign lattice.

Figure 2.2: Sign lattices for numbers.

2.2.3 Lattices in a Dataflow Analysis

During the conception of a dataflow analysis, an abstract domain is chosen that is able to represent the semantic property of interest for the client of the analysis. For example, to perform a sign analysis, which can e.g., detect division by zero errors, a special abstract domain that keeps track of the signs of numbers can be employed. A simple domain that can be used for this purpose is shown in Figure 2.2a. Then, an abstract value, which in this case corresponds to an element of the sign lattice, can be related to every variable in the program, for example, to obtain sign information for all variables.

In a static analysis, elements *lower* in the lattice (lower in the partial order) are regarded as representing *more precise* information, whereas elements *higher* in the lattice (higher in the partial order) are considered to represent *less precise* information. Consider again the lattice in Figure 2.2a. The bottom element $\bot$ represents the fact that the value is not a number; it is the most precise representation. The abstract value 0 indicates that the value is exactly zero, whereas the abstract values $+$ and $-$ indicate that the value is strictly positive resp. negative. When the sign of the number is not known exactly, the abstract value $\top$ can be used, which represents any number. Clearly, this is the least precise option.

A dataflow analysis also infers the effect of operations on the abstract values by specifying *abstract operations* on them, such as e.g., the addition of two abstract numbers. The exact implementation of these operations is lattice-dependent. Table 2.1 specifies multiplication for the sign lattice in Figure 2.2a. When one of the operands is not a number, represented by $\bot$, the result of the multiplication

```
1 (define x 0)      ; x = 0
2 (if condition     ; evaluates to ⊤
3     (set! x 1)    ; x = +
4     (set! x -1))  ; x = −
5 x                 ; x = ⊤
```

Listing 2.1: If the condition evaluates to $\top$, then the information in both branches needs to be combined and the value of `x` becomes $+ \sqcup - = \top$.

cannot be a number either and hence also is $\bot$. Multiplying a number by zero always yields zero. The product of two negative numbers is positive, as is the product of two positive numbers. On the contrary, the result of multiplying a positive and a negative number is a negative number. A multiplication with $\top$ always yields $\top$, except when the other operand is zero or $\bot$. After all, although $\top$ also represents the number zero, it represent any number, and thus the sign of the result cannot be derived more precisely as it may be positive, negative, or zero.

$*$	$\bot$	$-$	0	$+$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$+$	0	$-$	$\top$
0	$\bot$	0	0	0	0
$+$	$\bot$	$-$	0	$+$	$\top$
$\top$	$\bot$	$\top$	0	$\top$	$\top$

Table 2.1: Multiplication in the sign lattice.

In a dataflow analysis, the join operator of the lattice is used when information from multiple possible execution paths need to be combined, e.g., when information of two branches of an `if` expression needs to be combined to compute the result of the entire `if` expression. Consider for example the code in Listing 2.1, where a variable `x` is assigned a positive resp. negative value in the then or else branch of an `if` expression. If the value of the condition cannot be analysed exactly (i.e., if the analysis finds the value $\top$ for the condition), then the the analysis must take into account the information in both branches of the `if` expression as it cannot be certain which branch would actually be taken during the execution of the program. Thus, the final value for `x` becomes the join of its value in both branches, i.e., $\top$.

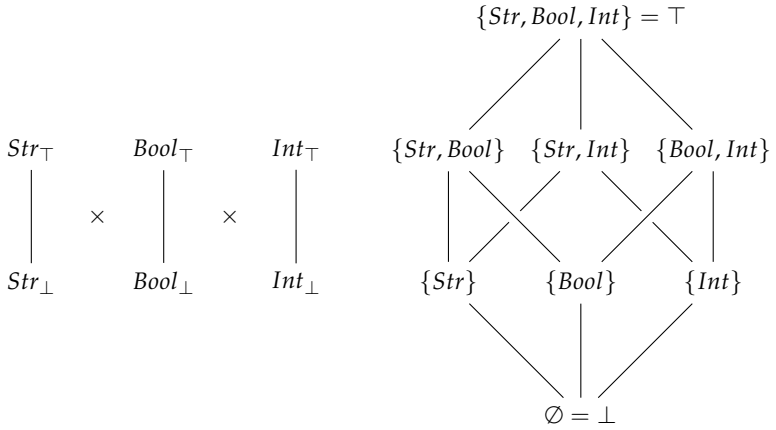
The choice of abstract domain impacts the precision of the analysis: the larger the (height of) the abstract domain, the more precise different concrete values can be approximated. For example, in the sign lattice of Figure 2.2a, we could add another level with the abstract values $\{-, 0\}$, $\{0, +\}$ and $\{+, -\}$, for numbers

that are negative or zero, zero or positive, or non-zero respectively, as is shown in Figure 2.2b. This, however, enlarges the state space, which can in turn influence the convergence speed of an analysis [17, 83, 84].

2.2.4 The Type Lattice and the Constant-propagation Lattice

In our evaluation, we make use of two specific lattices: the type lattice and the constant-propagation lattice. In this section, we briefly consider their general structure.

A *type lattice* is used to keep track of all possible types a variable can have, e.g., String, Boolean, Integer. A simple product type lattice for these types is shown in Figure 2.3a. For every type, $\bot$ represents that the value cannot be of the given type, whereas $\top$ represents that the value can be of the given type. We can also model this lattice as a set lattice, as shown in Figure 2.3b, where the elements of the partially ordered set $(\mathcal{P}(\{Str, Bool, Int\}), \subseteq)$ denote the possible types of the value.



(a) Product lattice. **(b)** Set lattice.

Figure 2.3: Hasse diagrams of a type lattice of strings, booleans and integers.

Another simple abstract domain is formed by the *constant-propagation lattice* [28, 159]. The abstract domain can precisely represent single values of the concrete domain as it has an abstract value for each possible concrete value. The top value of the lattice represents all possible values and is used when the exact value is not known. The bottom value represents the absence of a value. As there can be an infinite number of concrete values, e.g., in the case of integers, the width of the

lattice can be possibly infinite as well. However, its height is fixed to three. An example of a constant-propagation lattice for integers is shown in Figure 2.4. A constant-propagation lattice for multiple data types can be obtained by creating the product lattice of the constant-propagation lattices for different data types.

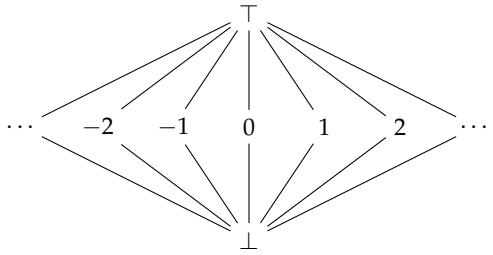


Figure 2.4: Constant-propagation lattice for integers.

All examples given in this dissertation make use of a type lattice, unless stated otherwise.

2.2.5 Lattices and the Termination of the Analysis

The choice of the lattice used to represent the abstract domain within the analysis has an impact on the precision and running time of the analysis. The closer the abstract domain resembles the concrete domain, that is, the larger the size of the abstract domain, the more precise the analysis result can be. However, for the analysis to be guaranteed to terminate, at least one of the following conditions must hold on the abstract domain [28, 139]:

- the lattice is finite, i.e., it contains only a finite number of elements, or
- the lattice satisfies the ascending chain condition, or
- the lattice is infinite and does not satisfy the ascending chain condition but a suitable widening operator is used to ensure the termination of the analysis,

provided that the analysis can be stated as a monotone (increasing) function on the lattice.

A lattice that is not finite but satisfies the ascending chain condition has an infinite width. As all ascending chains are finite, convergence of the analysis is guaranteed.

2.3 Modular Static Analysis

Modular static analysis [30] is an approach to static analysis that scales well, and that can achieve good precision with low memory consumption [53, 69, 85, 96, 126]. The incrementalisation approach presented in this dissertation is enabled by the design of modular analyses.

In a modular static analysis, the analysis of a program is decomposed into the analysis of elements of the program called *modules*. These modules correspond to static parts of the program and can for example be function definitions [96], classes, or thread definitions [85, 126]. A module may have multiple runtime instantiations, e.g., function calls, which the analysis might discern as well. We refer to the reification of these runtime instantiations in the analysis as *components*. For example, a component corresponding to a function definition is a function call. Other examples of components are class instances and threads. A component consists of a module and a context used to discern the different instantiations. Depending on the definition of contexts, more instantiations may be discerned, increasing the analysis precision (and complexity). Hence, depending on the *context sensitivity* of the analysis, one module can correspond to multiple components created by the analysis.

A modular analysis analyses its components in isolation. In the ideal case, the analysis result is obtained by composing the analysis results of all components. In this case, the analysis often is referred to as a *compositional analysis*. Since modules are usually a fraction of the program size, the analysis of each component is performed quickly and can be tuned to have a high precision.

In practice, however, components may interfere with each other: functions can call each other, objects interact, and threads may spawn other threads or read from shared variables. In a modular analysis, these interferences are reified as *dependencies* between components. When a new dependency is found or an existing dependency is updated, the analysis schedules the affected components for reanalysis, possibly triggering the analysis of other dependent components until a fixed point is reached. Thus, the analysis of one component may trigger the (re-)analysis of another.

2.3.1 Effect-driven Modular Static Analysis

Recently, ModF, an effect-driven formulation of function-modular analysis has been introduced [96]. ModF is a control-flow analysis also computing value

```
1  WL := {Main}; // The worklist, initially containing the Main component.
2  V := ∅; // The set of visited components.
3  D := λr.∅; // Map of dependencies (read effects).
4  σ := λa.⊥; // Global value store, initially all addresses map to bottom.
5  while WL ≠ ∅ do
6  |   α ∈ WL;
7  |   WL := WL \ {α};
8  |   (C', R', U', σ') = intra(α, σ); // Intra-component analysis.
9  |   σ := σ';
10 |   V := V ∪ {α};
11 |   WL := WL ∪ (C' \ V);
12 |   foreach r ∈ R' do D := D[r ↦ D(r) ∪ {α}];
13 |   foreach u ∈ U' do WL := WL ∪ D(u);
14 end
15 return (σ, V, D);
```

Algorithm 1: The inter-component analysis (Inter) of ModF.

information. It *reifies* the *computational dependencies* between components using *effects* and uses these to drive the fixed-point computation that constitutes the analysis, where an *inter-component analysis* schedules components for analysis and invokes an *intra-component analysis* which analyses individual components. The inter-component analysis, which we will henceforth refer to as Inter and which is shown in Algorithm 1, uses a worklist of components to be analysed. Initially, this worklist contains the Main component[1], a special component which represents the program's entry point (Line 1). In every analysis step, a component is removed from the worklist *WL* (Lines 6-7) and analysed by the intra-component analysis Intra (Line 8); the analysis terminates when the worklist is empty, indicating that a fixed point has been reached.

The *store* of the analysis maps abstract addresses to abstract values. It abstractly represents the heap of the program, in which values are stored at certain memory locations. ModF uses global-store widening [153], i.e., there is a single global value store σ within the analysis that is shared among all components [96]. For every component, σ also contains an abstract return value. Critically, upon a function call, ModF does not step into the function, but retrieves the stored return value of the component corresponding to the function call (or $\bot$ if no value has yet been stored previously). By not stepping into function calls, the function-modular character of ModF is obtained.

The intra-component analysis returns three sets of *effects* together with an updated store (Line 8). The effects are a reification of the component's computational

[1] In formal notations, we use lowercase Greek letters to denote components (e.g., α and β). Otherwise, we denote them by their respective names in small caps (e.g., Main and fib).

dependencies and influences on the global analysis state. They correspond to function calls (generating *call effects*), and reads from and writes to the store (generating resp. *read and write effects* – the latter is only generated when σ actually changes):[2]

- A call effect denotes the discovery of a component within the analysis. In ModF, this happens whenever a function call is encountered by the intra-component analysis. If the discovered component has not been encountered before, it is scheduled for analysis, meaning that it is added to the worklist. Note that no components are created for functions that are built into the analysis, such as primitive operators (e.g., +).
- A read effect on an address in the store indicates a *dependency*: the analysis of the component reading the address is dependent on the value at that address. These dependencies are used to determine the component(s) to be added to the worklist, causing components depending on updated information to be reanalysed.
- A write effect on an address is emitted when the value stored at that address in the global store changes. These effects thus indicate an update of a part of the global analysis state – in this case, an update of a value stored at a certain address in the global store – and cause all components with a dependency on this updated part to be scheduled for reanalysis.

Next, the state of the analysis is updated (Line 9) and the dependencies are registered in the dependency map D (Line 12).[3] Components are added to the worklist as follows: all components that have a dependency on an address that was changed by the analysis of the current component (indicated by the set of write effects U') need to be reanalysed (Line 13). This may possibly include the component just analysed as well. The current component is marked as visited (Line 10), and all new components, i.e., the components discovered except the ones that have been visited already, are added to the worklist as well (Line 11). Then, the algorithm proceeds with the next component in the worklist.

During the analysis, the inter-component analysis Inter keeps track of:

1. the analysis state, comprising solely the global store σ;
2. a set of visited components V;
3. a mapping of dependencies to components D;
4. a worklist of components to visit WL.

[2] For brevity, in pseudocode, the set C represents the set of all components corresponding to the emitted call effects, and the sets R and U represent the addresses corresponding to the emitted read and write effects respectively.

[3] The notation $M[k \mapsto v]$ denotes a map update: the map $M[k \mapsto v]$ is identical to the map M except for the key k which is mapped to the value v.

The analysis result is constituted by (1) the store σ, (2) the set of components V, and (3) the dependency map D (Line 15); the worklist will always be empty at the end of an analysis. We consider all parts of the result equally relevant, though in practice one might mostly be interested in σ.

Effect-driven flow analyses can be used with any abstract domain, given the conditions listed in Section 2.2.5, and with any context-sensitivity. All examples given in this dissertation use no context sensitivity, i.e., they use empty component contexts for which every function corresponds to one component at most, unless stated otherwise.

Example. We illustrate how ModF computes the control-flow and value information of the Scheme program in Listing 2.2. Assuming the use of a type lattice and no context sensitivity, ModF analyses the program as follows:

1. The analysis starts with the analysis of the MAIN component, which represents the entry point of the program. This component is analysed as follows:
   a) When the variable `x` is bound, its value in the analysis store is updated (`x` becomes `Int`) and, thus, a write effect for this variable is generated.
   b) Then, the variables `fun` and `inc` are bound. Both become bound to a closure and, thus, their values in the store are updated. Hence, for both variables, a write effect is generated.
   c) Then, the call to `fun` is analysed. To do so, the variable `fun` is read from the store, which returns the corresponding closure thus generating a read effect. The component corresponding to the function call, FUN, which has not yet been encountered by the analysis, is added to the worklist and a call effect is generated. Importantly, the analysis does *not* step into the component but retrieves its return value from the store. (If functions have arguments, which is not the case in this example, the calling component must write the argument values to the store so they can be used by the intra-component analysis of the called function.) As no return value has been computed previously for FUN, $\bot$ is read from the store; a read effect on the address of this return value is registered, which indicates that the component MAIN depends on this value.
   d) Finally, the return value of the MAIN component, $\bot$, is written to the return address of MAIN in the store. As this value does not change, no write effect is generated.
2. Next, FUN is analysed as this is the only component currently residing in the worklist:
   a) When analysing the call to `inc`, first, the value of the variable `inc` is read from the store, generating a read effect. As the component corresponding to the function call, INC, has not yet been encountered

```
1 (define x 0)                          ; Definition of a variable x.
2 (define (fun) (inc) x)                ; Function that reads x.
3 (define (inc) (set! x (+ x 1)) #t)    ; Function that reads and writes x.
4 (fun)                                 ; Call to the function fun.
```

Listing 2.2: Example Scheme program containing two functions.

previously, it is added to the worklist. A call effect is generated and the return value of INC, currently $\bot$, is read from the store, thereby generating a read effect.

b) Then, the variable reference is analysed, causing the generation of a read effect for `x`.

c) Finally, as the component FUN is now fully analysed, its return value is written to the store. As the return value in the store changes (it now becomes `Int` where it previously was $\bot$), a write effect is generated. This write effect causes the component MAIN to be added to the worklist as well, since MAIN previously emitted a read effect on the return address of FUN, indicating a dependency.

3. Either MAIN or INC can now be analysed. For the sake of the example, let's assume INC is analysed (the worklist order does not affect the result [96]):
   a) INC reads `x`, generating a read effect, and also writes to this variable. As the value of `x` in the store is not updated (it remains `Int`), no write effect is generated.
   b) As the return value of INC is written to the store and there updated to `Bool`, a write effect is generated and the dependent component, FUN, is added to the worklist again.
4. The analysis continues in this way until the worklist is empty, which indicates that a fixed point has been reached.

Note that in this example, for brevity, we omitted that to call the primitive function plus, first the corresponding function has to be retrieved from the store, thereby generating a read effect for the corresponding variable +.

Formal Specification of ModF

To clearly define the behaviour of a ModF analysis, we now formally specify a ModF analysis for a simple language, based on the untyped lambda calculus, with side effects that is restricted to Administrative Normal Form (ANF). In ANF, a distinction is made between two types of expressions: (1) *atomic expressions*, which can be evaluated to a value immediately and without altering the program state,

and (2) *complex expressions*, whose evaluation may require an infinite amount of time and whose evaluation may impact the program state; ANF requires all subexpressions of a function application, that is, all operators and operands, to be atomic expressions [44, 144].

Our formal specification is based on the one of Nicolay *et al.* [96] but differs, although not conceptually, to better match our description of the ModF algorithm and the pseudocode in Algorithm 1. Although a ModF analysis can work with different intra-component analyses, we present a formal specification for a small-step intra-component analysis to clearly define and exemplify the generation of effects during the intra-component analysis.

Syntax. The syntax of the input language we use for the formal definition of ModF is given in Figure 2.5. Expressions can be atomic expressions, function applications, `set!` expressions or `letrec` expressions. A `set!` expression modifies the value of a variable to the value resulting from the evaluation of a given atomic expression. A `letrec` expression binds a variable to the result of the evaluation of a given expression. The scope of this binding is limited to the binding expression of the variable and to the body of the `letrec` expression; the former allows recursive functions to be defined as it allows functions to refer to themselves. An atomic expression can be a variable reference, a lambda expression, i.e., an anonymous function, or an integer. As a program only contains a finite number of expressions, the set of expressions *Expr* is always finite during the analysis of a program.

$$
\begin{aligned}
e \in \mathit{Expr} &::= \mathit{ae} \mid (f\ \mathit{ae}) \mid (\texttt{set!}\ x\ \mathit{ae}) \mid (\texttt{letrec}\ ((x\ e))\ e) \\
f, \mathit{ae} \in \mathit{AExp} &::= x \mid \mathit{lam} \mid n \\
\mathit{lam} \in \mathit{Lam} &::= (\lambda\ (x)\ e) \\
x \in \mathit{Var} &= \textit{A finite set of identifiers.} \\
n \in \mathbb{N} &= \textit{The set of integers.}
\end{aligned}
$$

Figure 2.5: Syntax of the input language used for the formal definition of ModF.

State space. The state space of a ModF analysis is depicted in Figure 2.6. It contains two types of components: a Main component, corresponding to the program entry point, and Call components corresponding to function calls. The latter consist of the module (in this case, the closure corresponding to the called function) and a context. The store maps addresses to abstract values. In our formal specification, we represent the address corresponding to the return value of a component by means of the component itself. We leave the exact definition of contexts, values and addresses open. We assume, however, that the values are part of a lattice

which adheres to the termination requirements stated in Section 2.2.5. For the sake of the formal specification, we assume the lattice approximates integers by their type (**int**) and that it contains elements for closures (**clo**(lam, ρ)). We assume the existence of a function *clos* that retrieves all closures corresponding to an abstract value.

For the small-step intra-component analysis, the state space contains states that consist of a control component and a stack component. The control component can either contain an expression together with an environment in which the expression should be evaluated or a value. The stack is a list of frames; it represents the continuation of the program execution. ε denotes the empty stack. There are two types of frames: the **let** frame indicates that the binding expression of a `letrec` has been evaluated to a given value and that the body can be evaluated. For ModF, the **ret** frame indicates that a function body has been fully analysed and, thus, that the return value of the component should be written to the analysis store. Finally, the analysis uses an environment, which maps variables to addresses in the analysis store.

$$
\begin{aligned}
\alpha, \beta, \gamma \in \mathit{Cmp} &::= \textsc{Main} \mid \textsc{Call}(v, \kappa) \\
\epsilon \in \mathit{Eff} &::= \mathbf{w}(a) \mid \mathbf{r}(a) \mid \mathbf{c}(\mathit{cmp}) \\
\sigma \in \mathit{Store} &= (\mathit{Addr} + \mathit{Cmp}) \to \mathit{Val} \\
\kappa \in K &= \textit{A finite set of contexts.} \\
v \in \mathit{Val} &= \textit{Values from the abstract domain.} \\
a \in \mathit{Addr} &= \textit{A finite set of addresses.}
\end{aligned}
$$

$$
\begin{aligned}
\varsigma \in \Sigma &::= \langle c, \iota \rangle \\
c \in \mathit{Ctrl} &::= \mathbf{ev}(e, \rho) \mid \mathbf{val}(v) \\
\iota \in \mathit{Stck} &::= \phi : \iota \mid \varepsilon \\
\phi \in \mathit{Fram} &::= \mathbf{let}(a, e, \rho) \mid \mathbf{ret}(\alpha) \\
\rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Addr}
\end{aligned}
$$

Figure 2.6: State space of a ModF analysis with a small-step intra-component semantics.

Inter-component analysis. We now define the inter-component analysis Inter. It has the following signature:

$$\begin{aligned}\textsc{Inter}\colon\ & \overbrace{\mathcal{P}(\mathit{Cmp})}^{WL} \times \overbrace{(\mathit{Addr} \to \mathcal{P}(\mathit{Cmp}))}^{D} \times \overbrace{\mathit{Store}}^{\sigma} \\ & \to \underbrace{\mathcal{P}(\mathit{Cmp})}_{V} \times \underbrace{(\mathit{Addr} \to \mathcal{P}(\mathit{Cmp}))}_{D'} \times \underbrace{\mathit{Store}}_{\sigma'}\end{aligned}$$

Inter takes as argument the worklist WL, a mapping of dependencies to components D and the store σ. It produces the analysis result that consists of the set of visited components V, the collected dependencies D' and the final store σ'. We define Inter as follows:[4]

$$\textsc{Inter}\big(\varnothing, D, \sigma\big) = \Big(\big\{\alpha \mid \alpha \in \mathrm{dom}(\sigma) \wedge \alpha \in \mathit{Cmp}\big\}, D, \sigma\Big)$$

$$\textsc{Inter}\big(\{\alpha\} \uplus \alpha s, D, \sigma\big) = \textsc{Inter}\Big(\alpha s \cup \bigcup_{\substack{\mathbf{c}(\beta)\in C \\ \beta\notin\mathrm{dom}(\sigma)}} \beta \cup \bigcup_{\substack{\mathbf{w}(a)\in U \\ \gamma\in D(a)}} \gamma, D \uplus \biguplus_{\mathbf{r}(a)\in R} [a \mapsto \{\alpha\}], \sigma'\Big)$$

where $(C, R, U, \sigma') = \textsc{Intra}(\alpha, \sigma)$, αs are the remaining components in the worklist, $\uplus$ denotes the disjoint union and $\uplus$ is a pointwise union defined as follows:

$$d_1 \uplus d_2 = \bigcup_{\substack{a\in\mathrm{dom}(d_1) \\ a\in\mathrm{dom}(d_2)}} \Big(a, d_1(a) \cup d_2(a)\Big) \cup \bigcup_{\substack{a\in\mathrm{dom}(d_1) \\ a\notin\mathrm{dom}(d_2)}} \Big(a, d_1(a)\Big) \cup \bigcup_{\substack{a\notin\mathrm{dom}(d_1) \\ a\in\mathrm{dom}(d_2)}} \Big(a, d_2(a)\Big)$$

The inter-component analysis analyses all components in its worklist separately by using the intra-component analysis Intra. Whenever a new component is discovered, it is added to the worklist, and whenever the value of an address in the store is updated, dependent components are added as well.[5] This way, the inter-component analysis performs a fixed-point computation.

Collecting semantics. Given this definition of Inter, the abstract collecting semantics of a program, i.e., all flow and value information computed by the analysis, can be obtained by computing $\textsc{Inter}(\{\textsc{Main}\}, [], [])$, where $[]$ denotes an empty map.

Intra-component analysis. We now formalise a small-step intra-component analysis for the given input language to clearly define how effects are supposed to be

[4] To not clutter notation, we omit the curly braces of singleton sets when using the big union $\bigcup$.

[5] In the formal specification of Nicolay *et al.* [96], write effects are always emitted when an address is written to the store; the inter-component analysis only adds dependent components to the store when the value at that address changes. In our formal specification, without loss of generality, we assume write effects are only emitted when the value of an address in the store is changed.

generated during the analysis of components. We formalise this analysis (1) in terms of an atomic evaluation function $\mathcal{A}$, which specifies how atomic expressions are evaluated, (2) in terms of a transition function $\rightsquigarrow$, which specifies how complex expressions are evaluated, and (3) by means of a fixed-point computation for the intra-component analysis, Intra. They have the following signatures:

$$\begin{aligned}
\mathcal{A}&\colon \mathit{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathit{Val} \times \mathcal{P}(\mathit{Eff}) \\
\rightsquigarrow&\colon \Sigma \times \mathit{Store} \to \Sigma \times \mathit{Store} \times \mathcal{P}(\mathit{Eff}) \\
\textsc{Intra}&\colon \mathit{Cmp} \times \mathit{Store} \to \underbrace{\mathcal{P}(\mathit{Eff})}_{C} \times \underbrace{\mathcal{P}(\mathit{Eff})}_{R} \times \underbrace{\mathcal{P}(\mathit{Eff})}_{U} \times \mathit{Store}
\end{aligned}$$

The atomic evaluation function takes an atomic expression and an environment and store in which to evaluate this expression and returns the value resulting from the evaluation and a set of effects that were generated during this evaluation. The transition relation takes a state and a store and returns the subsequent state and a possibly updated store. It also returns the set of effects that are generated during the transition. Rather than adding this set to the result tuple, we annotate the transition relation with it as follows: $\overset{E}{\rightsquigarrow}$. Finally, the intra-component analysis Intra takes a component and a store and returns the effects generated during the analysis of the component together with an updated store.

Atomic evaluation function. The following rules specify the atomic evaluation function:

$$\begin{aligned}
\mathcal{A}(x, \rho, \sigma) &= (\sigma(\rho(x)), \{\mathbf{r}(\rho(x))\}) \\
\mathcal{A}(n, \rho, \sigma) &= (\mathbf{int}, \varnothing) \\
\mathcal{A}(\mathit{lam}, \rho, \sigma) &= (\mathbf{clo}(\mathit{lam}, \rho), \varnothing)
\end{aligned}$$

The evaluation of a variable causes its address to be looked up in the environment ρ, after which the value residing at that address in the store is returned. For this reason, a read effect is generated for the address. The evaluation of a number returns an abstract value, in this case **int**. Finally, the evaluation of a lambda returns a closure containing the lambda and its definition environment.

Transition function. The transition function describes how complex expressions are evaluated using the six rules shown in Figure 2.7. The transition function uses four helper functions: (1) the function *allocCtx* allocates a context, given the module and the value store, (2) the function *allocAddr* allocates a store address for a variable, (3) the function *clos* retrieves all closures corresponding to an abstract value, and (4) the function *write* writes a value to the store and generates a write

effect if needed, it is defined as follows:

$$write(\sigma, a, v) = \begin{cases} (\sigma[a \mapsto v], \{\mathbf{w}(a)\}) & \text{if } a \notin dom(\sigma), \\ (\sigma[a \mapsto (v \sqcup \sigma(a))], \{\mathbf{w}(a)\}) & \text{if } a \in dom(\sigma) \wedge v \neq \sigma(a), \\ (\sigma, \varnothing) & \text{otherwise.} \end{cases}$$

$$\frac{(v, E) = \mathcal{A}(ae, \rho, \sigma)}{\left(\langle \mathbf{ev}(ae, \rho), \iota \rangle, \sigma\right) \overset{E}{\rightsquigarrow} \left(\langle \mathbf{val}(v), \iota \rangle, \sigma\right)} \textsc{Atomic}$$

$$\frac{\begin{array}{c}(v_f, E_f) = \mathcal{A}(f, \rho, \sigma) \qquad \mathbf{clo}(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)}, \rho') \in clos(v_f) \\ a = allocAddr(x) \qquad (v_a, E_a) = \mathcal{A}(ae, \rho, \sigma) \qquad (\sigma', E_w) = write(\sigma, a, v_a) \\ mod = \mathbf{clo}(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)}, \rho'[x \mapsto a]) \qquad \kappa = allocCtx(mod, \sigma) \\ \alpha = \textsc{Call}(mod, \kappa) \qquad v_r = \sigma(\alpha) \qquad E = E_f \cup E_a \cup E_w \cup \{\mathbf{r}(\alpha), \mathbf{c}(\alpha)\}\end{array}}{\left(\langle \mathbf{ev}(\texttt{(}f\ ae\texttt{)}, \rho), \iota \rangle, \sigma\right) \overset{E}{\rightsquigarrow} \left(\langle \mathbf{val}(v_r), \iota \rangle, \sigma'\right)} \textsc{Call}$$

$$\frac{\begin{array}{c}(v_a, E_a) = \mathcal{A}(ae, \rho, \sigma) \\ a = \rho(x) \qquad (\sigma', E_w) = write(\sigma, a, v_a) \qquad E = E_a \cup E_w\end{array}}{\left(\langle \mathbf{ev}(\texttt{(set! } x\ ae\texttt{)}, \rho), \iota \rangle, \sigma\right) \overset{E}{\rightsquigarrow} \left(\langle \mathbf{val}(v_a), \iota \rangle, \sigma'\right)} \textsc{Assign}$$

$$\frac{a = allocAddr(x) \qquad \rho' = \rho[x \mapsto a] \qquad \iota' = \mathbf{let}(a, e_b, \rho') : \iota}{\left(\langle \mathbf{ev}(\texttt{(letrec ((}x\ e_x\texttt{))}\ e_b\texttt{)}, \rho), \iota \rangle, \sigma\right) \overset{\varnothing}{\rightsquigarrow} \left(\langle \mathbf{ev}(e_x, \rho'), \iota' \rangle, \sigma\right)} \textsc{Letrec-binding}$$

$$\frac{(\sigma', E) = write(\sigma, a, v_x)}{\left(\langle \mathbf{val}(v_x), \mathbf{let}(a, e_b, \rho) : \iota \rangle, \sigma\right) \overset{E}{\rightsquigarrow} \left(\langle \mathbf{ev}(e_b, \rho), \iota \rangle, \sigma'\right)} \textsc{Letrec-body}$$

$$\frac{(\sigma', E) = write(\sigma, \alpha, v)}{\left(\langle \mathbf{val}(v), \mathbf{ret}(\alpha) : \varepsilon \rangle, \sigma\right) \overset{E}{\rightsquigarrow} \left(\langle \mathbf{val}(v), \varepsilon \rangle, \sigma'\right)} \textsc{End}$$

Figure 2.7: The transition function for a small-step ModF analysis.

To evaluate an atomic expression, the atomic evaluation function is used and the transition function is annotated with the effects generated by the atomic evaluation function (rule Atomic).

The evaluation of a function application, shown in rule Call, is special, as here, ModF encounters the component boundary: the analysis may not step into the function body as components are to be analysed in isolation. First, the atomic evaluation function is used to evaluate the operator and operand; since our

language is restricted to be in ANF, both the operator and operand are guaranteed to be atomic expressions. Using the *clos* function, a closure is retrieved from the abstract value to which the operator evaluated. Then, the argument is bound by allocating an address in the store and by writing the value of the operand to this address. If this causes a store update, a write effect is generated; this is taken care of by the *write* function. Finally, the component corresponding to the function call is created. To this end, the module is constructed using the closure and the extended environment, which contains the binding for the operand, and a context is allocated using the function *allocCtx*. As ModF cannot step into the function body, a call effect is generated for the component and its return value is retrieved from the store, leading to the generation of a read effect. Logically, this return value is returned from the function call for the analysis to proceed with.

Rule Assign specifies how an assignment is evaluated. First, the value to which the variable is to be bound is evaluated using the atomic evaluation function (again, due to the restriction of the input language to ANF, this is guaranteed to be an atomic expression). Then, the address of the variable is retrieved from the environment and the new value is written, possibly causing a write effect. The result of an assignment is the value that was assigned to the variable.

A `letrec` is evaluated in two steps. First, the binding expression needs to be evaluated (rule Letrec-binding). To this end, a new address is allocated to store the value of the binding variable and the environment is extended to bind the variable to the newly allocated address. By using this extended environment also during the evaluation of the binding expression, the variable can be referenced within the binding expression, thereby enabling recursive functions. The stack is extended with a **let** frame that contains the body expression of the `letrec` and the extended environment, as well as the address to which the value of the binding variable needs to be written once it has been evaluated. Thus, once the binding expression has been evaluated, that is, when a value is reached and the top of the stack contains the **let** frame, the body of the `letrec` can be evaluated (rule Letrec-body). To this end, the value just computed for the binding expression is written to the address of the variable in the store and the **let** frame is removed from the stack, after which the body expression is evaluated.

Finally, rule End specifies what happens when the body of a function has been entirely evaluated. This is the case when a value has been reached and the top of the stack contains a **ret** frame. In this case, the return value of the component is written to the store at the return address of the component contained in the frame, the **ret** frame is removed from the stack and the value is returned.

Intra-component analysis. The transfer function prescribes how the intra-component analysis steps from one state to the next one during the evaluation (analysis) of a component. The intra-component analysis INTRA itself explores all reachable states, and returns the generated effects and the resulting store. (Of course, it is possible to compute more information depending on the exact analysis to be performed, such as, e.g., the set of all reachable states.) INTRA is defined as the least fixed point computation of a step function, which, for a given set of states, computes all successor states and adds them to the given set of states while computing the updated store and collecting all generated effects:

$$step(\varsigma s, \sigma, E) = \{(\varsigma s, \sigma, E)\} \cup \bigcup_{\substack{\varsigma \in \varsigma s \\ (\varsigma,\sigma) \overset{E_s}{\rightsquigarrow} (\varsigma_s,\sigma_s)}} (\{\varsigma_s\}, \sigma_s, E_s)$$

$$\textsc{Intra}(\alpha, \sigma) = \Big(\bigcup_{\mathbf{c}(\beta) \in E_i} \mathbf{c}(\beta), \bigcup_{\mathbf{r}(a) \in E_i} \mathbf{r}(a), \bigcup_{\mathbf{w}(a) \in E_i} \mathbf{w}(a), \sigma_i \Big)$$

where $(\varsigma s_i, \sigma_i, E_i) = lfp(step(\{\varsigma_0\}, \sigma, \varnothing))$ with $\varsigma_0 = \langle \mathbf{ev}(e, \rho), \mathbf{ret}(\alpha) : \varepsilon \rangle$ (e and ρ are derived from α), lfp the least fixed point operator and $\cup$ over tuples the pairwise union/join. The subscript s denotes the result of a step and the subscript i denotes the result of the intra-component analysis.

The Component Graph

The analysis of a component generates call effects, each corresponding to a component discovered by the intra-component analysis. After the analysis of a component α, INTER collects the set of components *called* by α, which we will henceforth denote as C_α. This gives rise to a cyclic directed graph, the *component graph*, representing how components are created: for every component $\beta \in C_\alpha$ there is an edge from α to β. Figure 2.8 depicts the component graph that arises from the analysis in the previous example. Components are depicted as rounded squares and the edges between components show the call effects generated by the analysis. For obvious reasons, in a ModF analysis, the component graph resembles the call graph of the program.

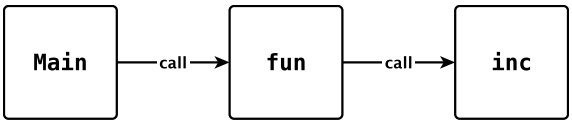


Figure 2.8: The component graph inferred by the analysis of the program in Listing 2.2: `inc` is called from `fun`, which is called from the program's entry point. Upon every call to a non-primitive function, a call effect is generated.

```
1 (define (fib n)
2   (if (= n 0) ; Incorrect end condition.
3       n
4       (let ((fib-n-1 (fib (- n 1)))
5             (fib-n-2 (fib (- n 2))))
6         (+ fib-n-1 fib-n-2))))
7 (fib 5)
```

Listing 2.3: Example Scheme program (incorrectly) computing the n^{th} Fibonacci number.

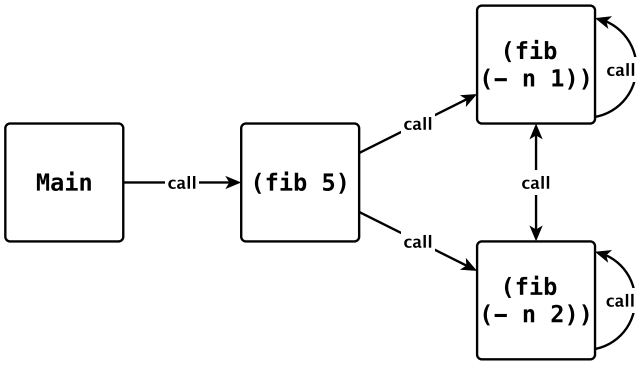


Figure 2.9: The ModF component graph for the program in Listing 2.3. The analysis uses call-site sensitivity, that is, component contexts consist of the call expression. In the figure, we therefore use the call expression to denote the components. Note that no components are created for calls to primitive functions such as +.

When context-sensitivity is used, multiple components may be created for the analysis of a single function. Thus, for one module, the component graph may show multiple components. This is illustrated in Figure 2.9, which depicts the component graph for an analysis of the program in Listing 2.3 which uses call-site sensitivity. In the example, the call expression is used as the component context, thus, components are distinguished by the function that is called and by the calling expression. As `fib` is called in three different locations, the analysis result contains three components for this function.

The Analysis Graph

The component graph shows how components are discovered during the analysis. However, it does not give any information regarding the use of the global store. The store, however, is a key component in an effect-driven analysis, as it is the

state shared by all components and changes to the store lead to the reanalysis of components. Therefore, knowing how information flows to and is retrieved from the store is crucial to understanding the functioning of the analysis. To this end, we can extend the component graph to also include the addresses from the global store, as well as the corresponding read and write effects, to denote how the store is used by the different components. We call the resulting graph the *analysis graph*.

The analysis graph shows how components are created and how values are used and updated by the components within the analysis. We visualise the analysis graph by depicting components using rounded squares, as in the component graph, and by depicting store addresses using circles. The edges visualise the effects inferred by the analysis, i.e., call effects, read effects, and write effects in the case of a ModF analysis. Optionally, we can annotate the effects with extra information (e.g., we can annotate read and write effects with the most recent (abstract) values read resp. written by the component from/to the given address). Writes to the store that do not cause the generation of a write effect may be shown using a dashed line.

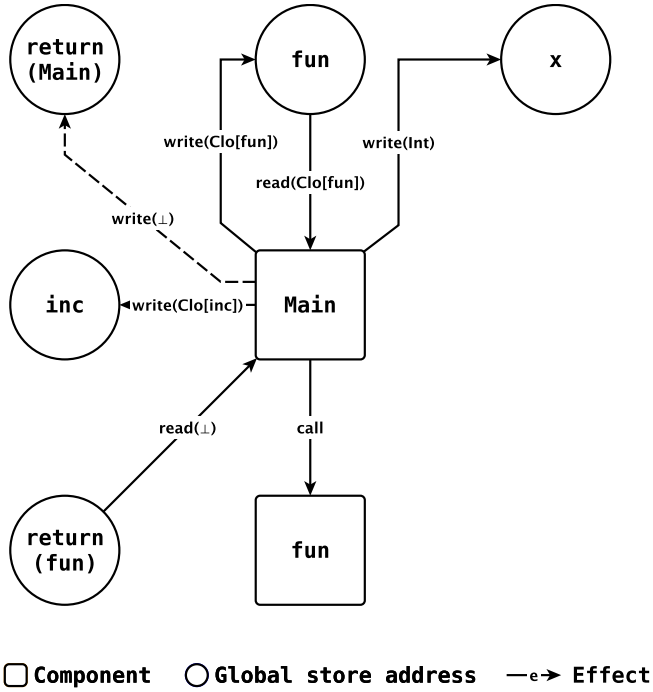
Figure 2.10 shows the analysis graph becoming larger after different intra-component analyses that are performed during the analysis of the program in Listing 2.2, following the analysis steps described in the example on page 24 and again omitting the lookup of the variable +.

Other Module Granularities

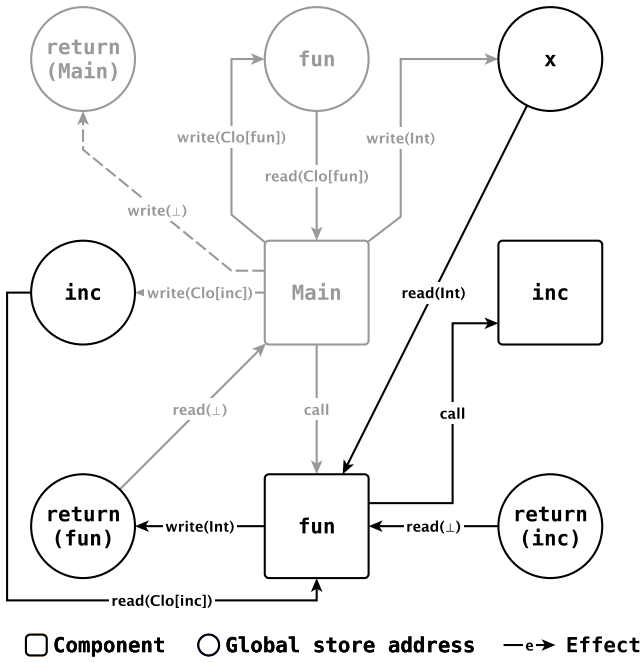
The principle of effect-driven flow analysis, just presented using ModF, is applicable to different module granularities, such as threads, as well. In a thread-modular analysis, a module corresponds to a thread definition, and a component corresponds to a spawned thread. Thread-modular effect-driven analyses are referred to as *ModConc* analyses in the literature [126, 128].

In a ModConc analysis, a component is created when a new thread is spawned. For a concurrent extension of Scheme, a component consists of the expression to be evaluated concurrently, the lexical environment of this expression, and an optional context. As in the case of ModF, ModConc knows three types of dependencies: *spawn*, *read* and *write*, where *read* effects indicate a dependency and *spawn* dependencies denote the discovery of a (new) component (similar to *call* effects in ModF).

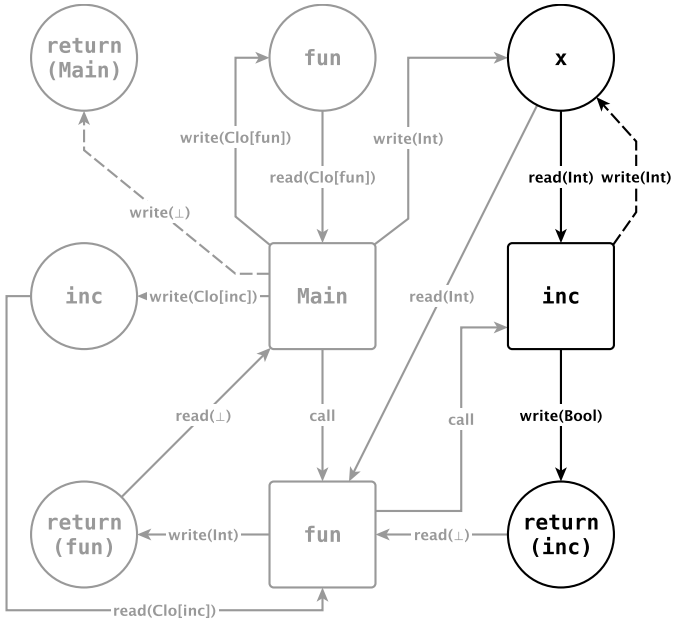
More generally, any effect-driven analysis needs the following three types of effects:



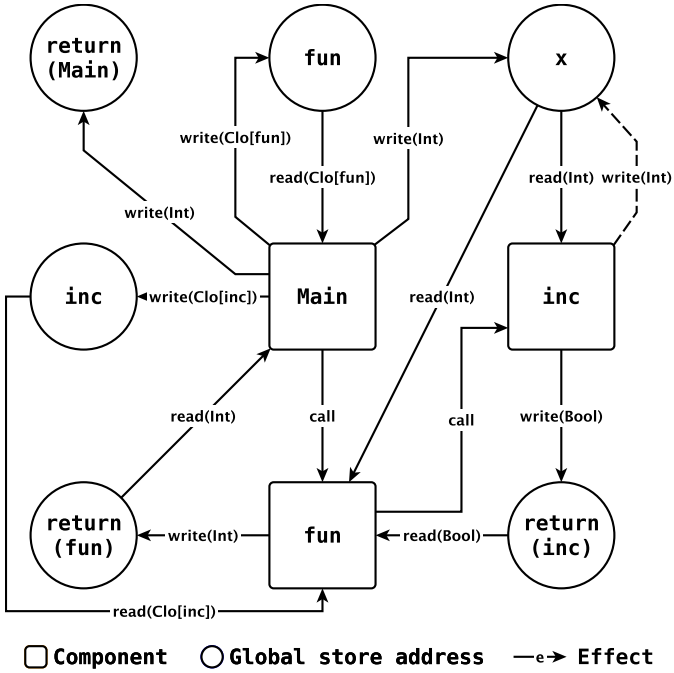
(a) After the analysis of MAIN.



(b) After the analysis of FUN.



(c) After the analysis of INC.



(d) After the entire analysis.

Figure 2.10: The analysis graph after different stages of the analysis of the program in Listing 2.2, following the steps described in the example on page 24. The read and write effects are annotated with the latest values read/written by the components and the lookup of the variable + is omitted.

```
1 (define (fib n)
2   (if (< n 2) ; Correct end condition.
3       n
4       (let ((fib-n-1 (fork (fib (- n 1))))
5             (fib-n-2 (fib (- n 2))))
6         (+ (join fib-n-1) fib-n-2))))
7 (fib 5)
```

Listing 2.4: Example Scheme program computing the n^{th} Fibonacci number in parallel.

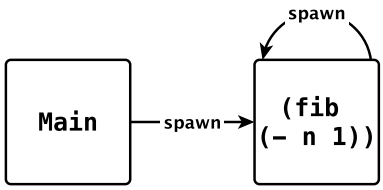


Figure 2.11: The ModConc component graph for program shown in Listing 2.4.

- effects that indicate the discovery (creation) of a component, e.g., call effects in ModF and spawn effects in ModConc;
- effects that indicate a dependency on some part of a shared analysis state, e.g., read effects in both ModF and ModConc;
- effects that indicate an update of some part of the shared analysis state, e.g., write effects in both ModF and ModConc.

Effects of the first kind cause the discovered components to be scheduled for analysis if the component has not yet been analysed before. Effects of the third kind cause dependent components, which are inferred using effects of the second kind, to be scheduled for reanalysis. This way, the updates made to the shared analysis state by a component are taken into account by dependent components and propagated.

Figure 2.11 depicts the component graph of a ModConc analysis of a program that computes the n^{th} Fibonacci number in parallel, shown in Listing 2.4. The figure shows the components and indicates how components are discovered. Again, in the example, component contexts are empty. The analysis starts with the Main component, and a second component is created upon the analysis of `fork`. This component encounters the `fork` expression again during its analysis, but since component contexts are empty, the same component is encountered and no new component is created. Therefore, the analysis result only contains two components.

2.4 Conclusion

In this chapter, we first presented static analysis as a means to infer semantic program properties and we defined the related concepts of soundness and precision. We then presented lattices as a fundamental abstraction of values in data-flow analysis. We presented how lattices are used in such analyses, introduced the structure of the two main lattices that are used in this dissertation: the type lattice and the constant-propagation lattice, and discussed termination guarantees. Subsequently, the concept of modular static analysis was described. In a modular static analysis, a program is divided into modules, which are reified in the analysis as components. These components are analysed separately, although the analyses of components can be inter-dependent. The work in this dissertation makes use of effect-driven modular analyses, where reified computational dependencies are used to infer inter-component dependencies. To this end, we presented ModF, a function-modular effect-driven analysis conceptually, formally, and visually using the component and analysis graphs. Lastly, we also considered other module granularities for effect-driven analyses.

3

State of the Art in Incremental Static Program Analysis

In this chapter, we introduce incremental static program analysis as a means to reduce the analysis time for programs that evolve over time. Upon an update to a program, an incremental analysis reuses and updates the result from the analysis of the previous program version rather than performing a from-scratch computation. Reusing the result from a previous analysis run avoids the recomputation of those parts of the result that were not impacted by the change and thus would be inferred anew.

The goal of this process is to reduce the time needed to compute the analysis result for new program versions. When static analyses are integrated in build processes such as continuous integration pipelines, their running time must not become prohibitively long. Fast analyses can also be integrated into IDEs, for example, and recent literature has shown that timely feedback of a static analysis significantly increases the fix rate of signalled problems, thereby reducing the cost needed to resolve the reported issues [14, 60]. In the ideal case, an incremental analysis scales in the size of the program change rather than in the size of the entire program.

This chapter is structured as follows. Section 3.1 first gives a general overview of incremental static analysis. Every incremental analysis needs a means to represent and receive modifications made to the subject program under analysis. Therefore, Section 3.2 gives an overview of change representations that are used in the literature. We then turn our attention to existing incremental static analyses in the literature and look at how existing analyses update their result incrementally. To this end, we divide the existing work into two broad categories, in line with the categorisation of Szabó *et al.* [133, 135] and Van Es *et al.* [149]:

Bespoke Incremental Analyses. Bespoke incremental analyses are designed with a specific purpose in mind, that is, they are tailored to a specific type of analysis and/or to a specific application domain. For these analyses, a custom, bespoke incrementalisation approach is proposed that is fitted specifically to the analysis and domain at hand. We discuss this type of incremental analyses in Section 3.3.

Incremental Analysis Frameworks. Framework-based incremental analyses are implemented or specified in a framework that takes care of the incrementalisation of the analyses. Incremental analysis frameworks often are focussed on, or restricted to, a specific class of analyses which they can render incremental. Analysis designers then use a framework's infrastructure, e.g., a DSL, to specify their analysis, and the framework internally provides facilities to execute the analysis incrementally. Incremental analysis frameworks are the focus of Section 3.4, in which we also describe some analyses that are specified in a framework.

Unfortunately, sometimes, there is no clear distinction between an analysis framework to incrementalise a certain class of analyses and a bespoke incremental analysis that can perform several different yet related analyses. We classify a work as a framework if it provides or relies on a DSL for the specification of analyses for which the incrementalisation is then provided by the framework. We also classify a work as a framework if it refers to itself as being one, if other literature refers to it as such, or – in the case of analyses specified in a logic programming language – if the work provides specific provisions for incremental analysis rather than merely an incremental evaluation engine.

After discussing the literature within these two categories, we also briefly discuss work on staged static analysis in Section 3.5 and on incremental computation in Section 3.6. Finally, we conclude in Section 3.7.

It is worth mentioning that many other fields of research also touch the field of incremental static analysis, e.g., work on incremental parsing [156], tree diffing and change distilling [39, 45, 124], change logging [73, 162], and on persisting analysis results [36]. As these topics fall outside the scope of this dissertation, we do not explore them further in this chapter.

3.1 Introduction to Incremental Program Analysis

The aim of an incremental program analysis is to produce results faster than a from-scratch analysis of the program. To this end, an incremental program

analysis stores the result that was produced by a previous analysis of the program and updates it where needed. To do so, an incremental program analysis uses the program changes, which can be extracted and represented in various ways (see Section 3.2). Szabó [130] states that, ideally, the updating of the analysis result within an incremental analysis should scale in the change size, and not in the size of the program. As the literature on software development finds that many changes are small [6, 61, 104], there should thus be a sufficient opportunity for incremental static analysis to outperform a full program analysis. However, this is challenging because a change may, in the worst case, impact the entire analysis result. For this reason, we say that, ideally, an incremental update of an analysis result should scale in the size of the *change impact*.

In this dissertation, we use the following terminology. We use the term *initial program* to refer to the version of the program before an update and use the term *initial analysis* to refer to the full, from-scratch analysis of the initial program. We refer to the program after the program update as the *updated program*. A non-incremental analysis of the updated program is referred to as a *full reanalysis*, whereas the incremental updating of the result of the initial analysis by the incremental analysis is simply referred to as the *incremental update*. To perform the incremental update, the incremental analysis makes use of the *initial result* and of the *program changes*. If the incremental update is fully precise, the *updated result* matches the result of a full reanalysis. If the incremental update is not fully precise, its result will be less precise than the result of a full reanalysis. The general principle of incremental static analysis, using the terminology just introduced, is visualised in Figure 3.1.

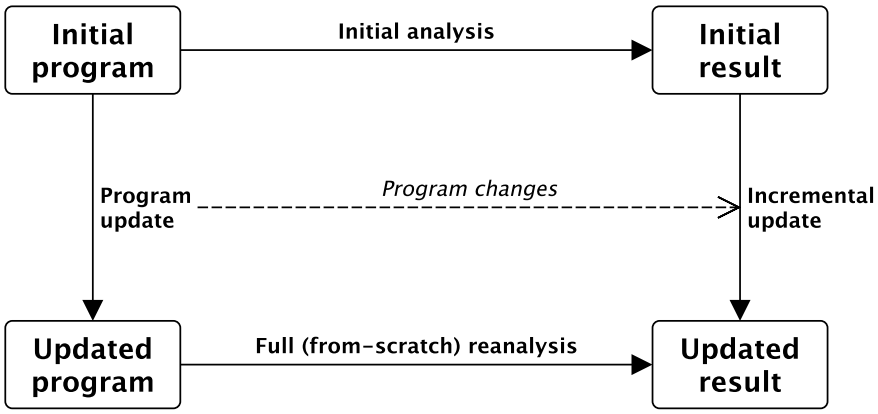


Figure 3.1: Schematic overview of a precise incremental program analysis.

In the literature, many approaches to incremental static analysis, some of which we will discuss in Sections 3.3 and 3.4, have been proposed. Fundamentally, an incremental update of an analysis result typically follows a two-step approach. In a first step, a change-impact analysis is employed to detect the parts of the analysis result that are outdated, which are then discarded. If the incremental update needs to uphold full precision, this detection phase may need to over-approximate which parts of the result are to be discarded, thereby ensuring that all outdated parts are removed. This, however, may cause excess parts of the result to be deleted that have to be recomputed later (see e.g., [12, 102, 113, 133]). In a second step, recomputation takes place and new analysis result is computed. This step often starts from the program points affected by a change, e.g., from the tuples added to the database for Datalog-based incremental analyses.

The incremental analysis presented in this dissertation does not follow this two-step approach but rather interleaves invalidation with recomputation. This is the topic of Chapter 5.

3.2 Extraction and Representation of Program Changes

An incremental program analysis updates its analysis result as the analysed program evolves. In order to do so, the analysis needs a means to identify how the program under analysis has changed, that is, there needs to be means to extract and represent changes. The extraction and representation of program changes are important for a static analysis, though form a different problem and are therefore not the focus of our work. Nevertheless, as there is a multitude of possibilities to infer and deliver changes to an analysis and some analysis implementations support multiple alternatives, this section describes how program changes are handled in the literature.

Program changes can, for example, be inferred by means of a *change distiller*, which produces a list of fine-grained AST changes corresponding to a code update (e.g., [24, 41, 45, 46, 124]), or by means of a *change logger*, which records changes while the developer edits the code in an IDE (e.g., [62, 91, 98, 162] and our own work Kursun, Van der Plas *et al.* [73] to log changes in the DrRacket IDE). Techniques for change distilling and change logging present a complex problem on their own and therefore fall outside the scope of this dissertation, and, as just stated above, the techniques for the extraction and representation of program changes also fall outside the scope of this dissertation.

The incremental static program analyses in the literature use a variety of change representations, depending on the analysis and its implementation. In the remainder of this section, we discuss some of the representations used in the literature. Our own work uses change expressions, which are based on the patch annotations of Palikareva *et al.* [101] and which we discuss in Section 4.1.

3.2.1 Incremental Analyses built upon or using Logic Programming

Analyses of logic programs can represent changes to the program under analysis as the addition, deletion or update of rules, which constitute the logic program [66]. In the work of Saha and Ramakrishnan [113], which uses logic programming to analyse C programs, the logic program contains clauses which define the analysis itself and facts which represent the program under analysis. Here, the changes are represented as the addition or deletion of the facts that represent the program under analysis.

3.2.2 Incremental Analyses built upon a Projectional Editor

Other incremental program analyses are built on top of a structured program editor, also referred to as a projectional editor, where AST changes are received immediately from the IDE, i.e., the editor expresses every change to the program as an AST change.

The incremental data-flow analysis of Zadeck [165] uses information on the AST node or block currently being edited and on the variable whose use or definition is being modified by the user, which is provided by the editor. IncA [130, 133–136] is built on top of the JetBrains MPS projectional editor, which notifies the analysis of AST changes. These AST changes lead to updates in the tuple relations that represent the AST of the program, and which form the extensional database (EDB) that IncA processes incrementally to compute the updated analysis result, which corresponds to the intensional database (IDB).

3.2.3 Other Change Representations

There exist numerous works on the topic of incremental (static) program analysis that use yet other representations for program changes. We now briefly list some of them.

In Szabó [130, Chapter 7], the applicability of IncA within a parser-based IDE is discussed as an alternative to the JetBrains MPS front-end. To this end, hdiff, a recent and efficient generic AST differencing algorithm applicable to typed ASTs [86], is used to compute AST differences. These differences are then used to compute the tuples that need to be inserted and deleted from IncA's EDB, which represents the program under analysis. As in this work we do not rely on the use of typed ASTs, hdiff cannot be used in our setting. Although IncA is Datalog-based, it does not require changes to be provided directly using the addition or deletion of facts, like e.g., the work of Saha and Ramakrishnan [113], but IncA also provides a front end that computes these additions and deletions using the results of the hdiff algorithm.

To find changes, Andromeda [141] is notified of changes in a compilation unit. It then employs a change-impact analysis that marks changed compilation units as modified, added or deleted, and uses this information to process the changes. The analysis of Reps *et al.* [108] represents programs as attributed trees and changes by means of derivation-tree operations like grafting, deriving and pruning.

Reviser [11, 12] uses a differencing algorithm to compare both code versions and to compute a structural diff, from which it extracts the addition and removal of nodes and edges in the program's inter-procedural control-flow graph. Reviser then uses these changes to the program's control-flow graph to incrementally update the analysis result. The graph differencing algorithm does not necessarily compute the smallest change set, but, at the cost of precision, aims to be fast. The incremental analysis of Conway *et al.* [26] also processes changes to the control-flow graph of the program, but in addition also uses changes to an automaton, which reflect changes to types and variables.

Nichols *et al.* [92] on the other hand match different versions of JavaScript programs at the level of an intermediary representation as part of their approach, by matching functions, blocks, and instructions. First, corresponding functions are matched based on an edit-distance calculation which is computed for pairs of functions in the old and new program. If the edit distance is below a certain threshold, the functions are matched and the blocks in the function bodies are matched using a similar approach. Once blocks have been matched, the instructions in the matched blocks are matched based on the type, allocation sites and variable names. The correctness of the matching process heavily influences the precision and performance of their incremental analysis.

3.3 Bespoke Incremental Analyses

Bespoke incremental analyses are built for a specific type of analysis, often in a specific domain. Therefore, the incremental algorithms underlying bespoke incremental analyses are tailor-made to the problem at hand and domain knowledge can be used. However, the effort required to develop a bespoke incremental analysis and the complexity involved in the development of this type of analyses is higher than when an incremental analysis framework is used: not only does the analysis itself need to be developed and implemented, a custom incrementalisation approach for the analysis needs to be devised as well, which is not a concern for analyses implemented in an incremental analysis framework. As terminology is not standardised, in the literature, bespoke incremental analyses are sometimes referred to as *manual approaches* [149], as *specialized algorithms* [135], or as *single-analysis algorithms* [134].

We first discuss incremental analyses that require a static call graph. Such approaches to incremental analysis rely on a statically available call graph that is not impacted by code changes or which is provided again after an update. Afterwards, analysis techniques that do not rely on such a call graph are discussed. In this category, we also classify analyses for logic programming languages because logic programs are declarative and thus do not have a call graph but rather rely on a precedence graph for their evaluation. Unfortunately, for some related work, it could not be derived with certainty whether the work requires a static call graph or not; we classify these approaches as requiring a static call graph, since we are not certain of the opposite. We opt for this choice, as approaches not requiring a static call graph are less restricting and thus could also be applied in situations where a static call graph is present, whereas this does not hold the other way around.

3.3.1 Bespoke Incremental Analyses with Static Call Graphs

Zadeck [165] presents a lattice-based monotonous dataflow analysis that uses a given control-flow graph and can be used in a structured program editor. The analysis splits the data-flow problem into several independent clusters. This technique can solve a limited set of data-flow problems, such as performing a reaching-definitions and live-variable analyses. In the incremental setting, Zadeck assumes the situation in which most programmer edits involve only a single statement. The analysis can process three types of changes: reference changes (changes that do not impact the control flow or block structure of the program), added or removed control-flow paths or added or removed basic blocks. To this

end, the *SearchL* algorithm is presented, which computes the nodes in the flow graph which are affected by a single change. This algorithm is then used as a fundamental building block in algorithms specific for each of the three types of changes.

Marlowe and Ryder [79] present an incremental dataflow analysis that produces a precise result. The approach is said to be hybrid as it uses a graph decomposition of the data flow graph. To this end, they compute the SCCs of the data flow graph and make computations locally, and then propagate information across SCCs to compute the final result. Their analysis can solve a large class of dataflow problems, but e.g., not constant propagation. The incremental algorithm detects which regions of the flow graph are impacted and classifies the changes. Then, again, computations are made locally and information is propagated. Other bespoke incremental analyses for classical data-flow problems in this category are [23, 102].

McPeak *et al.* [80] propose an incremental and parallel static analysis for C programs. The analysis is split into deterministic work units of which the results are cached. Upon a code change, the cache is updated so that stale results are removed. Special care is taken to avoid including in the cache results that may need to be updated upon code changes, by relying on *stable anchor points* in the source code.

Other bespoke incremental approaches relying on static call graphs comprise incremental adaptations of context-free-language (CFL) reachability analyses [78, 118], alias analyses [164], interval analyses [19], model checking [26], incremental analyses through novel analysis mechanisms such as diff-graphs [71] and SAT solving [88] or using function summaries of the callers rather than of the callees [72], and incremental analyses tailored to specific settings, such as probabilistic programming where changes are made to the probabilistic knowledge rather than to the program itself [168].

3.3.2 Bespoke Incremental Analyses without Static Call Graphs

Reps *et al.* [108] present an incremental analysis for attribute grammars, i.e., context-free grammars where attributes are attached to the symbols of the grammar. The authors want to support building language-based editors, i.e., editors which enforce syntactic correctness of the program by restricting editing operations. Programs are represented using attributed trees and attribute grammars allow context-dependent features of the analysed language to be expressed in the analysis. Upon a program change, an incremental analysis is performed to update

the attribute values in the tree. This way, the editor can efficiently flag or undo program changes that cause a violation of the language constraints. To this end, two naive algorithms are introduced (change propagation and nullification/re-evaluation), after which an optimal-time algorithm is introduced which works in a time linear to the number of affected attributes in the tree. Standard attribute grammars have several limitations, however. For example, the dependencies between attributes are limited by the shape of the syntax tree and can only involve the nodes of a single production rule [65].

Attribute grammars also have been used by Hedin [63] for incremental semantic analysis of statically-typed object-oriented languages in the Mjølner programming environment. In this work, constraints allow for side effects to be performed upon evaluation. The proposed approach makes use of these constraints to perform incremental updates and lookups in symbol tables. This avoids copying and propagating information through the tree and allows the incremental update to work in a single pass. However, the incremental evaluator must make sure to undo side effects that were performed during the evaluation of constraints. To this end, constraints performing side effects also have a *de-evaluation* operation which cancels the side effects of their *evaluation* operation. In addition, the propagation of evaluation is not limited to local nodes, which allows the evaluation to return when missing information for the evaluation of a declaration is added to the symbol table. Also, circular dependencies may exist, for which the author foresees a means to break circular evaluation.

In later work, Hedin [64, 65] formally introduces *Door attribute grammars* in which nodes in the AST may have attributes containing reference values pointing to semantic objects, which is disallowed by standard attribute grammars. More recent work using reference attribute grammars, where attributes can be references to AST nodes, includes an approach to incremental evaluation using dependency tracking [119] and incremental model validation [81, 82].

Liu *et al.* [76] propose an alternative to incremental points-to analysis that does not require expensive graph reachability computations. Instead, the authors rely on what they refer to as a *"fundamental transitivity property of Andersen's analysis"*, allowing them to avoid a graph reachability analysis and redundant computations, for which the proposed algorithm is more efficient. In addition, the fixed-point computation within each iteration of the algorithm is parallelised. The approach remains however limited to flow-insensitive analyses.

Seidl *et al.* [117] propose to use *generic local solvers* to provide incrementalisation for abstract interpretation in an analysis infrastructure without restricting the design of the analysis (modular, IDE, IFDS). This is achieved by modifying a top-down

solver to leverage dependencies for incrementality. To reuse parts of the result from the previous run of the analysis, functions are matched to functions with the same name in the previous version of the program. For every modified function, parts of the result corresponding to any of the program points of the function are invalidated.

Nichols *et al.* [92, 93] introduce *fixpoint reuse* to incrementally analyse JavaScript programs. The analysis creates a mapping of analysis results from the old program points to corresponding program points in the updated program, allowing reuse of analysis results for mapped points. The fixed-point computation can then be restarted and makes use of this mapping to reuse analysis results, thereby accelerating the convergence of the analysis. The mapping function plays a key role: more mapped points lead to more reuse and a faster analysis, but incorrect matches can cause the analysis to lose precision. To ensure soundness, all program points need to be reanalysed at least once to avoid a premature convergence of the fixed-point algorithm, implying that the impact of changes cannot be bounded effectively by this approach.

Andromeda [141] is an analysis tool used to perform taint analyses incrementally and demand-driven. Upon changes to the program, Andromeda performs a change impact analysis that computes the part of the analysis result to be invalidated, and the parts that need to be updated. The change impact analysis determines the affected data structures, and uses an auxiliary *support graph* to find outdated taint facts, which are then removed. This allows the analysis result to remain precise. Andromeda lends its incremental capabilities to its demand-driven nature. Therefore, the incremental analysis may not be suitable for use in a non-demand-driven setting.

Razafintsialonina *et al.* [107] present two complementary approaches to speed up the abstract interpretation of C programs. A first approach reuses function caches. For every function, a pair of abstract states representing the memory at the function entry and exit are kept. When a function and its entry memory remain unchanged, the exit memory can be reused to avoid the reanalysis of the function. The approach, implemented in the monolithic Eva abstract interpreter of the Frama-C framework, reuses the caches saved by Eva itself and transfers them to the analysis of the updated program. The caching, however, does not work for functions that perform dynamic memory allocations. The second approach reuses previously calculated fixed points for loops by restarting the fixed point of an updated loop from this previous result, rather than from the usual bottom value. This speeds up the convergence of the analysis for loops, at the cost of losing precision.

Albert *et al.* [7, 8] introduce an incremental resource usage analysis for a sequential Java-like language. A three-step process is used, consisting of a pre-analysis which includes a class analysis which also computes the CFG of the program, the computation of potentially recursive *cost relations*, and finally the computation of *cost functions* by solving the previously computed cost relations. A *multi-domain incremental fixed-point algorithm* that computes information for different domains separately, taking into account dependencies amongst them, is used. For each method, a *cost summary* is computed so that new results can be computed incrementally by only replacing the affected method costs. The analysis is abstract interpretation-based and works on a rule-based intermediate representation which is created during the class analysis. The authors state that their approach is modular as it stores method summaries although the program is not split into smaller parts. However, the speed-ups obtained by the incremental analysis are limited and range from 1.48 to 5.13. Moreover, the highest speed-ups are obtained in an experiment for which the implementation of a method is replaced by an identical implementation and no change has to be propagated to calling methods. The results of the experiments where information needs to be propagated show a maximal speed-up of 1.8 (on average over all benchmark programs used in an experiment), with the maximal speed-up for an individual benchmark program being under 3. Also, the presented analysis is limited to the analysis of sequential programs.

Yu *et al.* [163] present an abstract-interpretation-based incremental predicate analysis used for regression verification. Every point in the CFG is annotated with an assertion over the predicates that represent the abstract program state. The incremental analysis syntactically invalidates invalid assertions upon a program change, and only recomputes these invalidated assertions. To this end, a change-impact analysis detects at each program point which variables are impacted by the changes. Then, the algorithm (re-)computes the parts of the result that were affected. The presented analysis has been implemented for C and reaches, on average, a speed-up of 2.8. It is unclear how well the analysis would work when applied to programs written in more dynamic languages, in which data flow and control flow are intertwined.

Garcia-Contreras *et al.* [47] present a context-sensitive incremental modular analysis which achieves incrementality at the inter-modular and intra-modular level. The analysis requires an encoding of the program in constrained Horn clauses. The analysis does not divide the program into modules itself and does not use components. Instead, a programmer-defined lexical module partitioning is used, although it is claimed that any partitioning is possible. Thus, as the

partitioning is lexical, their analysis can, e.g., not be used with thread-modular analyses. Later work [48] presents an updated approach, also capable of handling external modules, together with a formal description and a further evaluation.

Other bespoke incremental analyses not relying on static call graph comprise incremental abstract interpretation based on the invalidation and adaptation of states [149], on a reified representation of the abstract interpretation [121–123], or on a system of equations supporting multithreaded C programs [40], specific analyses such as race detection [167] and use-before-initialisation bug detection in the Linux kernel [166], analyses of logic programs [66, 103] or for analyses specified in tabled Prolog [38], and call graph construction in industrial Java applications [169].

3.4 Incremental Analysis Frameworks

Incremental analyses frameworks are used for the incrementalisation of a specific class of analyses. An analysis can be implemented or specified in a framework, and the framework is responsible for the incrementalisation. This category thus does not comprise bespoke incremental analyses that are incorporated into a software framework. In this section, in addition to the description of incremental analysis frameworks, we also describe some framework usages that are present in the literature.

IncA [130, 133–136] is a Datalog-based DSL for the specification of incremental analyses. In IncA, analyses are written using pattern functions and an analysis is constructed by specifying graph patterns of interest. Pattern functions abstract over graph patterns which express relations between AST nodes using Datalog rules, where the extensional database (EDB) is formed by the relations describing the AST and the intensional database (IDB), i.e., the derived relations, is formed by evaluating the Datalog program specified by the graph patterns. IncA's runtime system uses an incremental Datalog solver based on differential dataflow to perform an incremental graph pattern matching on memoized computation graphs. Program updates are represented by the addition and deletion of tuples to the EDB. The IncA compiler translates pattern functions into inter-connected graph patterns and performs optimisations to ignore AST changes that do not affect the analysis result. IncA supports inter-procedural analyses with recursive, monotonic, user-defined aggregations of lattice values of custom lattices. Updating the lattice values requires an interleaving of the deletion and rederivation phases

to avoid the excess deletion of some tuples in the case of cyclic relations. IncA works in JetBrains MPS, a projectional editor, although a textual front end is available as well.

An application of the IncA framework is given by Szabó *et al.* [132], in which the IncA framework is used to resolve static overloading in Featherweight Java. The authors compare two approaches: co-contextual type checking and IncA. They find that co-contextual type checking cannot be used efficiently in an incremental setting, because this type of type checking requires global information, whereas only local information is present when checking an AST node. Instead, the IncA DSL is used to implement the overload resolution, relying on the incrementalisation provided by the framework. The evaluation of the approach on synthetic benchmarks shows significant speed-ups, where the incremental analysis is about 450 to about 900 times faster than a full analysis.

Pacak *et al.* [99] make use of IncA to provide the automated derivation of incremental type checking. Starting from the specification of a type checker, an incremental type checker is produced. The authors provide a DSL to encode the typing rules and a compiler. First, the compiler applies three transformations to the typing rules to make them computable in Datalog, to allow efficient incrementalisation, and to improve the handling of ill-typed terms. Then, the compiler compiles the rules to the IncA variant of Datalog. This way, IncA can take care of the incrementalisation of the analysis. A preliminary evaluation considers five types of artificial changes that affect a single subterm for two simple programs written in an extended simply-typed lambda calculus, four types of which result in ill-typed programs, and shows that the incremental analysis often is faster than a non-incremental type checker written in Java on incremental updates, but has a significantly higher initialisation time.

Another Datalog-based framework for incremental static analysis, created by one of the authors of IncA, is iQL [131]. iQL is a prototype incremental solver for analyses written in the CodeQL framework which are used by GitHub to provide feedback on pull requests. In CodeQL, analyses are specified in a language called QL, which is compiled to Datalog. The program under analysis is extracted into a relational database. iQL relies on an existing incremental Datalog solver, Viatra Query. However, the author finds that CodeQL scales better in terms of performance than Viatra Query as the latter was not built for analyses at such scale, for which iQL performs a hybrid analysis: the incremental analysis is combined with the non-incremental analysis of some parts of the code performed by CodeQL. Depending on which parts are evaluated incrementally, the analysis can benefit

more from the incrementality of Viatra Query or from the scalability of CodeQL. However, the author finds that the iQL framework requires a prohibitively high amount of memory.

Reviser [11, 12] is a framework for incremental, inter-procedural data-flow analyses, which is applicable to analyses implemented in the IDE [112] or IFDS [109] frameworks, where flow functions should be distributive over the merge operator. To this end, a new solver that handles program changes and solves flow functions written in a template-driven style is introduced. To perform an incremental update, first, a diff is generated, indicating how the inter-procedural control-flow graph of the program was modified. For each affected node in the graph, analysis information is removed. Then, analysis information is propagated again from all predecessor nodes of affected nodes. Reviser supports flow-sensitive and context-sensitive data-flow analyses. Upon a program update, however, Reviser recomputes the entire call graph from scratch.

Saha and Ramakrishnan [113] present a framework for incremental and demand-driven analyses based on logic programming. Analyses are specified as Horn clause rules and the programs under analysis need to be specified as Horn clauses. As with other analyses built using logic programming, program changes are represented by means of the addition or deletion of facts. The framework combines an incremental evaluation technique for logic programs with a goal-directed query evaluator. The addition of facts is managed by inferring a *delta relation* for every rule which defines the analysis. To handle deletions of facts, first a mark phase is performed, which marks all answers that are potentially affected by the deletion, and then a rederivation phase is executed, which attempts to rederive marked answers without the use of the deleted facts. Affected answers that cannot be rederived are deleted. Internally, a support graph, used to detect changes to derived relations when tuples of a base relation are deleted, is used to speed up the second phase.

Dura *et al.* [35] find that a common problem with static checker frameworks is that rely on imperative specifications of bug patterns. They present JavaDL, a declarative language to specify bug patterns for Java code using syntactic pattern matching and logical rules that allows both from-scratch and incremental analysis. The authors argue that white-box testing frameworks, which do not rely on imperative specifications, can automatically enhance bug detection specifications, e.g., by incrementalisation, which is in line with the fact that almost all frameworks discussed in this section are based on logic programming. JavaDL avoids fact extraction from the syntax by providing syntactic pattern matching on the AST.

It also provides an incremental evaluator for inter-procedural Datalog-based analyses. However, the incremental cannot consistently outperform the from-scratch analysis.

As becomes clear in this section, many of the frameworks for incremental static analysis are based on logic programming, where the program under analysis forms the EDB and in which the analysis developers should specify their analysis in a declarative, rule-like manner. An incremental evaluation technique for declarative languages is then used to obtain an incremental analysis. The only exception to this pattern which we discussed is Reviser, which incrementalises IDE/IFDS-based analyses.

3.5 Staged and Demand-driven Static Analysis

Related to incremental static analysis is *staged static analysis*. In this setting, a program is not analysed *all at once* but in different stages, e.g., when portions of the program become available part by part. In this case, the program gets updated (extended) incrementally as more code becomes available for analysis. However, the code that has been analysed does not change and thus no result invalidation is needed.

An example of a staged analysis is Gulfstream [56], a staged points-to analysis for web applications written in JavaScript. The authors find that an entire web application is only available in the browser, as code is downloaded by need. Gulfstream combines an offline, heavy-weight server-based analysis with an online, light-weight client-side analsyis. The former is used to analyse the parts of the code that are available for offline analysis, whereas the latter performs an on-demand analysis when code is loaded in the browser. The analysis is built in a declarative manner, where a program is represented as a database of facts and a solver derives information based on inference rules.

A *demand-driven static analysis* on the contrary has access to the full program from the beginning. However, this type of analysis does not compute the entire result immediately, but only computes the parts of the result that are requested, e.g., by means of a query. As such, the analysis saves times. As with a staged static analysis, no parts of the result need to be retracted as the program does not change. However, existing results are stored, so that when a new query is launched, they can be reused and the result grows incrementally.

In previous sections, we already discussed some incremental analyses that are also demand-driven, such as the framework of Saha and Ramakrishnan [113] and Andromeda [141]. An example of a non-incremental demanded analysis is the work of Yan *et al.* [161], who propose a demand-driven alias analysis for Java which expresses the analysis as a CFL reachability problem and does not rely on an underlying points-to analysis. The analysis is both field-sensitive and context-sensitive.

3.6 Incremental Computation

The work in this dissertation concerns incremental static analysis. Incremental static analysis is part of the field of incremental computation, which is used in many fields. Liu [77] generally describes the domain as the study on how to efficiently handle continually changing input by storing and reusing previously computed results, and dubs incrementalisation as *"the discrete counterpart of differentiation in calculus"*. We now give a very brief overview of some of the existing work on incremental computation, but by no means intend or claim to be complete. A categorised overview of (by now) early work in the field is given by Ramalingam and Reps [105].

Liu [77] classifies existing works into three broad categories: incremental algorithms, incremental evaluation frameworks, and incremental program-derivation methods. The first category comprises algorithms for particular functions, potentially applicable to a broad class of problems. The second category concerns frameworks in which (classes of) non-incremental programs can be expressed and which automatically handle input changes to these programs incrementally. Finally, the third category concerns methods to derive incremental algorithms based on a given algorithm and on an expected change representation using a set of semantics-preserving program transformations. The incremental analysis frameworks discussed in Section 3.4 can be classified in the second category, whereas the bespoke incremental analyses discussed in Section 3.3 can be classified in the first category. We classify the work presented in this dissertation in the third category, as we will present a method to use reified computational dependencies to achieve incrementality in static program analysis.

Acar [3] presents self-adjusting computation, and later a brief overview thereof in [4], which is a language-based approach to incremental computation based on memoised dynamic dependence graphs. In this setting, a program consists of two components: a *core* which runs the application with a fixed input and a *mutator* which modifies the input and requests a change propagation that propagates

the data modifications through the core. The change-propagation mechanism is provided by the language, so that programmers do not need to handle incremental updates themselves. The change-propagation mechanism relies on memoised dynamic dependence graphs as well as on *modifiable references* that store data that changes over time and which can only be read or written by specific primitives. To handle side effects, dependencies between certain versions of such modifiable references are kept.

The work of Acar [3] only considers so-called *monotone* traces, meaning that the order of reused subcomputations cannot be changed. Ley-Wild *et al.* [74] remove this restriction for pure programs. Their solution is built upon partial traces called *trace slices* to support the reordering of subcomputations. A pure source language is introduced, which is compiled into a low-level language in a continuation-passing style for which non-monotonic change propagation is supported.

Adapton [58] is a framework for composable and demand-driven incremental computation. The work is underlain by a *demanded computation graph* (DCG) which stores dependencies between computations and mutable reference cells. When the value in such a cell is updated, recomputation of dependent computations is needed. However, this computation is postponed until its result is actually needed, i.e., it is made into a thunk, making the framework demand-driven. In Adapton, values of thunks are memoized. Together with the DCG, this enables reusing the results of already evaluated computations. The framework is bi-layered: *inner computations* can read reference cells but not modify them, whereas *outer computations* can allocate and modify reference cells. As such, inner-layer computations can be reused incrementally. When a change is made, the DCG is used to mark computations that need to be re-evaluated when their value is demanded again. The approach, having explicit primitives for handling thunks and mutable state, is formalised in a calculus applying demand-driven semantics to incremental computing. In later work, the framework is extended with first-class names for identifying computations to increase the possibilities for their reuse [59].

Gupta *et al.* [57] present two algorithms to incrementally update the result of a Datalog program, i.e., the IDB, after a change to the tuples in the EDB or to the rules defining the IDB. The approach supports recursion, negation, union, and aggregation. Two algorithms for incrementally maintaining views are introduced. For non-recursive programs, the authors propose the *counting* algorithm, which keeps track of the number of derivations for each tuple that are used to infer which tuples should be deleted upon a change to the EDB. For recursive programs, the authors present the *DRed* (delete and rederive) algorithm. DRed works in three steps. First, the algorithm computes an over-estimate of the tuples in the IDB that have to be deleted. All tuples for which one derivation becomes invalid

are deleted. Second, alternative derivations are taken into account and deleted tuples that have alternative derivations are restored. Third, new tuples that need to be added are computed. In Section 3.4, we discussed two incremental analysis frameworks that use adaptations of the DRed algorithm, namely the framework of Saha and Ramakrishnan [113] and IncA [130, 133, 134, 136].

Recent work on incremental computation includes incremental parsing of natural language [25], a functional incremental computation framework based on monadic computations [43], a study of incremental graph computations [42], and an approach to incremental software architecture recovery based on a Bayesian classifier [157].

To the best of our knowledge, no work exists on incremental static analyses that have been implemented in an incremental computation framework such as Adapton. A reason for this may be that incremental computation frameworks are general and therefore not tailored to the domain of incremental static analysis. This lack of domain-specific knowledge within the framework may lead to analyses with a sub-optimal performance in comparison to bespoke incremental analyses or analyses implemented in dedicated incremental analysis frameworks. In addition, it may be cumbersome to implement an analysis in an incremental computation framework, especially in comparison to incremental analysis frameworks which provide a DSL for this purpose. For example, in Adapton, computations are lazy and should be written using mutable reference cells and thunks, which adds an extra layer of complexity to the implementation of an analysis.

3.7 Conclusion

This chapter introduced incremental static program analysis as a means to speed up static analysis by reusing and updating the result obtained for a previous program version. First, we discussed different change representations that enable an incremental analysis to detect how the program under analysis has been changed. Second, this chapter reviewed existing work on incremental static analysis. In line with existing literature, we divided the existing work into two broad categories: bespoke incremental analyses, which are built for a specific analysis and setting, and incremental analysis frameworks, in which analyses can be specified which are then rendered incremental by the framework. A further distinction between analyses that require the presence of a static call graph and those that do not was made. Finally, we briefly presented the literature on staged static analysis and on the general domain of incremental computation.

Currently, the literature on incremental static analysis thus presents analysis developers with two options, each belonging to a separate category in the realm of incremental computation: a bespoke method, that aims to solve a particular incrementalisation problem, e.g., a specific analysis for a clearly demarcated application domain, or a general incrementalisation framework wherein analyses can be specified and that handles the incrementalisation of a certain category of analyses. However, as far as we are aware, no work on static analysis exists that fits into the third category identified by Liu [77]: we did not encounter any existing work on static analysis that presents a general step-by-step incrementalisation method, applicable to a broad class of existing analyses, with which analysis developers can give their analysis incremental capabilities without having to reimplement it in a certain framework or without having to devise an incrementalisation tailored to their specific analysis themselves. Instead, analysis developers should be able to reuse as much as possible of their current implementation and they should not be concerned about devising a custom yet sound incrementalisation method for their analysis. This reduces their workload and the risks of implementation errors. In this dissertation, we study this missing option for the class of effect-driven analyses, i.e., for analyses for which dependencies on the global analysis state can be reified.

4

A Monotonic Approach to Incrementalisation

This chapter is based on our 2020 paper "Incremental Flow Analysis through Computational Dependency Reification" [145]. The text has been moderately modified and extended for clarity and completeness. In addition, a formal specification has been added (Section 4.3).

This chapter introduces a novel method for rendering effect-driven static analyses incremental. The presented incrementalisation method can be used to construct incremental effect-driven analyses that support dynamic, higher-order languages as well as changes to the call graph, independently of the specific analysis that is performed. The approach is instantiated with the modular effect-driven static analysis from Section 2.3.1, which divides the program under analysis into parts that are analysed in isolation and reifies the dependencies between these parts, such as e.g., the use of a shared variable [96, 126, 128]. We observe that these dependencies can be exploited to track the parts of the result impacted by a change, and to bound the impact of this change to only those parts of the result that are directly or indirectly affected.

Concretely, in this chapter, we make the following contributions:

- We present a novel method to incrementalise effect-driven analyses. Our method makes use of the intra-program dependencies which are reified by the analysis, and takes advantage of the division into modules and components to bound the impact of changes (Section 4.2).
- To demonstrate the generality of our approach, we instantiate our method for two context-insensitive modular effect-driven analyses: a function-modular (ModF) analysis and a thread-modular (ModConc) analysis.

- We perform a thorough evaluation of analysis time and analysis precision using the two instantiations described previously (Section 4.4). We find that our approach leads to a reduction of the analysis time from 6% to 99% for all but two benchmark programs, and that the impact on precision is limited for most programs.

Without loss of generality, we present our method from the viewpoint of a static analysis for μScheme programs and encode program changes using change expressions, as explained in Section 4.1. We claim that our method is applicable to other languages too. Even though we only use context-insensitive instantiations for the evaluation, the method is also applicable to context-sensitive analyses, as is shown by the example component graph in Section 2.3.1 (Figure 2.9 on page 33).

We first discuss how we represent program changes in our work.

4.1 Change Expressions

As discussed in Section 3.2, an incremental program analysis has to react to change in the program under analysis. Thus, every incremental analysis needs a means to represent or receive changes to which it can react. To represent changes in a program, we take inspiration from Palikareva *et al.*'s *patch annotations* [101], and annotate programs with *change expressions*. As such, both the original and updated version of a program are represented by a single annotated program. The use of these change expressions avoids the trouble of unifying different program versions to match corresponding program points in the different versions, allowing us to focus on the core problem of incrementalisation, rather than on the tasks of change distilling [41] and change analysis [89, 125]. We assume that, upon a program change or given information from a version control system, a change distiller inserts the required annotations in the program AST. To facilitate the evaluation of our approach, we currently insert change expressions manually in the program text, prior to the initial analysis.

As an example, consider the annotated program in Listing 4.1, fixing a bug in the program of Figure 2.9. In this program, the annotation denotes a change to the condition of the `if` expression: the condition has changed from `(= n 0)` in the original program to `(< n 2)` in the updated version.

Although the approach discussed in this work is language-independent, we express program changes in the μScheme programming language and use it for

```
(define (fib n)
  (if (<change> (= n 0) (< n 2))
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
        (+ fib-n-1 fib-n-2))))
```

Listing 4.1: An annotated Fibonacci, fixing a bug in the end condition.

the evaluation of our work.[1] We therefore now show how μScheme programs can be annotated with change expressions. Figure 4.1 shows a reduced formal syntax of the most prevalent syntactic forms of this language, based on [2], in an extended Backus-Naur form, where a star (*) indicates zero or more repetitions, a plus (+) indicates one or more repetitions and square brackets (`[.]`) indicate an optional element. Details and other syntactic forms have been omitted for simplicity. Our additions to support change expressions are shown in blue.

In contrast to the patch annotations of Palikareva *et al.*, our change expressions really are expressions and not annotations. This way, no invasive changes to the parser of the analysis are needed. However, some parts of the program cannot be edited as freely as with annotations: the entire program – including the change expressions – must still be a valid expression. It is therefore sometimes necessary to use more coarse-grained changes than with annotations. For example, to change the parameter list of a function, the entire function definition needs to be put inside a change expression, and to change a `let` special form into a `let*` special form, the entire `let` expression must be put inside a change expression. We also do not explicitly check for nor disallow nested change expressions in the formal grammar nor in our implementation. Due to their semantics, no issues can arise should change expressions (accidentally) be nested, although it is not meaningful to nest change expressions and such a situation is not expected to arise when change expressions are inserted automatically, e.g., by means of a change distiller.

[1] μScheme, the language supported by our implementation, is based on R5RS Scheme and contains all essential constructs of R5RS Scheme, such as closures, higher-order functions, lists, and quasiquoting. Our implementation supports many of the primitive functions of Scheme, but the numeric tower is limited to integers and real numbers. The `eval` primitive, macros, and delayed evaluation are not supported. Our implementation of ModF also supports the `call-with-current-continuation` primitive. Although we did not have to take any special measures for our incrementalisation method to support this primitive, `call-with-current-continuation` is not used in our benchmark programs (described later in Sections 4.4.1 and 5.4.1). We do note that the analysed language and the supported language features are a concern of the intra-component analysis, and are therefore orthogonal to the incrementalisation method presented in this work, on the condition that the effects are generated correctly by the intra-component analysis.

⟨*program*⟩ ::= ⟨*expression or definition*⟩* — *Programs*

⟨*expression or definition*⟩ ::= ⟨*expression*⟩
| ⟨*definition*⟩
| `(begin` ⟨*expression or definition*⟩*`)`

⟨*definition*⟩ ::= — *Definitions*
`(define` ⟨*variable*⟩ ⟨*expression*⟩`)` — *variable definition*
| `(define (`⟨*variable*⟩ ⟨*define formals*⟩`)` ⟨*body*⟩`)` — *function definition*

⟨*expression*⟩ ::= — *Expressions*
⟨*variable*⟩ — *variable*
| ⟨*literal*⟩ — *literal*
| `(`⟨*expression*⟩$^+$`)` — *application*
| `(lambda` ⟨*lambda formals*⟩ ⟨*body*⟩`)` — *lambda expression*
| `(if` ⟨*expression*⟩ ⟨*expression*⟩ `[`⟨*expression*⟩`])` — *conditional*
| `(cond` ⟨*clause*⟩$^+$`)` | `(cond` ⟨*clause*⟩* `(else` ⟨*sequence*⟩`))` — *conditional*
| `(set!` ⟨*variable*⟩ ⟨*expression*⟩`)` — *assignment*
| `(and` ⟨*expression*⟩*`)` | `(or` ⟨*expression*⟩*`)` — *conditions*
| `(`⟨*let*⟩ `(`⟨*binding*⟩*`)` ⟨*body*⟩`)` | `(let` ⟨*variable*⟩ `(`⟨*binding*⟩*`)` ⟨*body*⟩`)` — *bindings*
| `(begin` ⟨*expression*⟩*`)` — *sequencing*
| `(<change>` ⟨*expression*⟩ ⟨*expression*⟩`)` — *change expression*

⟨*define formals*⟩ ::= ⟨*variable*⟩* | ⟨*variable*⟩* `.` ⟨*variable*⟩ — *Formal parameters*

⟨*lambda formals*⟩ ::= `(`⟨*variable*⟩*`)` | ⟨*variable*⟩ | `(`⟨*variable*⟩$^+$ `.` ⟨*variable*⟩`)` — *Formal parameters*

⟨*body*⟩ ::= ⟨*definition*⟩* ⟨*sequence*⟩ — *Bodies*

⟨*sequence*⟩ ::= ⟨*expression*⟩$^+$ — *Sequencing*

⟨*clause*⟩ ::= `(`⟨*expression*⟩ ⟨*sequence*⟩`)` | `(`⟨*expression*⟩`)` — *Conditional clauses*

⟨*let*⟩ ::= `let` | `let*` | `letrec` — *Let names*

⟨*keyword*⟩ ::= — *Syntactic keywords*
`and` | `begin` | `cond` | `define` | `else` | `if`
| `lambda` | ⟨*let*⟩ | `or` | `set!` | `<change>`

⟨*variable*⟩ ::= Any identifier that is not a ⟨*keyword*⟩. — *Variables*

⟨*literal*⟩ ::= Boolean, number, character, string or quoted data. — *Literals*

Figure 4.1: Partial grammar of the μScheme language, based on [2], with change expressions.

In general, change expressions may be applied at several levels of granularity. Consider for example the program in Listing 4.2. Here, the two annotations together represent the same change as the single one in Listing 4.1.

```
1 (define (fib n)
2   (if ((<change> = <) n (<change> 0 2))
3       n
4       (let ((fib-n-1 (fib (- n 1)))
5             (fib-n-2 (fib (- n 2))))
6         (+ fib-n-1 fib-n-2))))
```

Listing 4.2: A Fibonacci with fine-grained annotations.

The usage of change expressions has an extra advantage. Often, the data structures used by the analysis contain syntactic elements containing e.g., positional information or other identifiers which change between program versions. For example, if a new line is added at a certain point in the program, the positional information stored in the syntactic elements within the inner data structures of the analysis does not match any more. In this case, the incremental analysis may consider all of this information outdated, thus requiring recomputation. To prevent this situation, it must be avoided that such kind of information ends up in the data structures of the analysis, or a mapping of old to new AST nodes is needed by the analysis, which may be used to update the analysis data structures or to retrieve information stored under an old syntactic element [49, 80, 100, 158].

The use of change expressions avoids this situation entirely, as both program versions are encoded within a single program and thus no positional information needs to be updated. This thus avoids the update or remapping step that in a real-life setting would be needed. However, as in Wauters, Van der Plas *et al.* [158], we find that the updating of the analysis data structures takes very little time, in a more complex setting than ours, we are confident that avoiding this update will not influence the performance results for our incremental analysis. Alternatively, a static analysis can also be designed to avoid storing syntactical information at certain places. For example, McPeak *et al.* [80] avoid storing defect reporting information in work units, which are self-contained parts of the input data that can be analysed independently, and store locations as offsets to some stable *anchor points* to avoid excess computations.

4.1.1 Change Types

There are two types of changes that can appear in a μScheme program and in programs written in other expression-based languages:

i. Adding an expression to a sequence of expressions.
ii. Removing an expression from a sequence of expressions.

Any change can be regarded as a combination of changes in these categories. For example, modifying an expression is a combination of removing and adding an expression.

The change expressions we use can represent all types of changes. Adding an expression is represented as `(<change> #f new-exp)`, where `#f` indicates the absence of an expression, and removing an expression is represented as `(<change> old-exp #f)`. A perhaps undesired side effect of our encoding using `#f` is that it may affect the semantics of a program, as every expression evaluates to a value in μScheme. For example, in case the last expression of a function body is added or removed, the return value would be `#f` in the initial resp. updated program. Although this may be inadequate in practice, this is acceptable in our setting as we will investigate the performance and precision of an incremental analysis in comparison to a full reanalysis, for which the sole requirement is that both analyses reason over the same program. Also, this situation does not occur when an expression is updated, since this can simply be encoded as `(<change> old-exp new-exp)`.

Some changes, such as the addition or removal of a variable, can modify the lexical environment of expressions. Such changes can be represented by enclosing all expressions affected by the change in the lexical environment inside the change expression. For example, defining a new variable may be represented as `(<change> (+ x 1) (let ((y 1)) (+ x y)))`. This, however, is cumbersome and may lead to changes that are too coarse-grained. For example, adding a new definition at the beginning of a program will affect the entire program, and the incremental analysis will degenerate into a full reanalysis.

Using this kind of coarse-grained changes can be avoided by updating the internal data structures of the analysis to account for the changes to the environment of expressions. These updates are difficult as environments may be stored in several of the internal data structures of the analysis. For example, in lexically-scoped languages such as μScheme, environments are stored in closures. As closures are (abstract) values within the analysis, they are present in the store σ of the analysis. The complexity of these updates depends, for example, on the parameters used to

tune the precision of the analysis. For example, if component contexts need to be updated, this may affect the number of components, and hence the dependency map D, as well as the addresses in the store. Hence, dealing with this kind of updates to the internal data structures of the analysis may not be trivial and this is exactly what our use of change expressions aims to prevent.

Therefore, we avoid this complexity by introducing new bindings in the original program or the updated program as follows. In case a new variable is defined, we may represent this as `(define x (<change> #f 1))`, where we again use `#f` to denote the absence of an expression. When a variable is removed, we similarly replace the value it is bound to by `#f`. The environments therefore remain unchanged, but the binding is updated. Hence, the data structures of the analysis do not need to be updated upon a program change.

4.2 Approach

In this section, we describe our method to incrementalise an effect-driven analysis, which we explain for a modular effect-driven analysis of which the intra-component analysis infers dependencies, in particular for the ModF analysis of Section 2.3.1, shown in Algorithm 1 on page 22.

To analyse the initial program, a full ModF analysis, as described in Section 2.3.1, is performed. As a matter of convenience, the initial analysis already takes a program annotated with change expressions, as explained in Section 4.1, but ignores these annotations and treats the program as if it does not contain any changes, that is, only the initial program is considered and the changes are ignored. Our method then consists of two steps, which are explained next: a change-impact calculation to detect which parts of the result have been affected by the program update and the actual updating of the analysis results.

4.2.1 Step 1: Change-impact Calculation

Upon a change to the program, the analysis result needs to be updated. The result of a modular effect-driven analysis consists of the global analysis state (in casu, the global store of the ModF/ModConc analysis), the set of visited components, and a mapping of dependencies to components. Hence, upon a change within the program, the analysis has to infer which components are impacted by the change and reanalyse them.

Due to the modular and effect-driven design of the analysis, only the components that are directly impacted by the change must be explicitly scheduled for reanalysis. Components that are transitively impacted by a change need to be reanalysed as well. These, however, will be scheduled for reanalysis by the effect-driven analysis itself when a dependency on which the components depend is triggered. For example, when the program update causes the value of an address in the global store of a ModF analysis to be updated, a write effect is generated and the dependent components, who earlier generated a read effect, are added to the worklist as well. Thus, the reanalysis of directly-impacted components can cause other components to be reanalysed as well, thereby propagating the updated information.

This step in our approach already demonstrates how our incremental analysis benefits from using reified effects: the analysis will not only ensure that *all* components that are impacted directly or indirectly are reanalysed, but also that *only* those components are reanalysed. After all, components which are not directly impacted and which do not depend on a part of the global state that is updated, are not scheduled for reanalysis. Therefore, our approach restricts the impact of changes to those components that are affected. An incremental update thus scales in the size of the change impact and not in the size of the entire program, which is the ideal scenario as described in Section 3.1.

To infer the components that are directly impacted by program changes, different approaches are possible depending on the type of analysis that is to be incrementalised. For example, when using a ModF analysis, the components directly impacted by a change can be inferred lexically from the source code, by inspecting which function definitions are impacted. All components related to updated function definitions can thus easily be added to the worklist. This is however not possible for a ModConc analysis, for example, as threads may execute code from multiple functions and it thus cannot always be inferred lexically which parts of the program a thread executes. We therefore propose a general tracking method, which is applicable to every type of effect-driven analysis with minimal effort.

Our tracking method works as follows. An intra-component analysis performs a fixed-point computation during which it steps through the code corresponding to the component. The analysis steps through the expressions one by one whilst reasoning over the semantic properties of the program. The goal of the tracking method is to track for every component the expressions that are encountered during its analysis. Typically, an `analyze` function that checks the type of the expression and acts accordingly is used, as shown in Algorithm 2. (Note that neither this `analyze` function nor the case splitting are needed by our method, but we use them to illustrate how tracking should be incorporated.) During

```
1  Function analyze(e: Expr, ρ: Env, cmp: Comp) is
2      // cmp is the current component.
3      // exprToCmps is globally available.
4      // exprToCmps :: Map[Expr → Set[Comp]]
5      exprToCmps := exprToCmps[e ↦ exprToCmps(e) ∪ {cmp}];
6      switch type of e do
7          case variable(id) do return lookup(id, env);
8          case fnCall(f, args) do
9              return analyzeCall(f, args, ρ);
10         end
11         case if(pred, then, else) do
12             return analyzeIf(pred, then, else, ρ);
13         end
14         case ... do ...;
15     end
16 end
```

Algorithm 2: The `analyze` function of the intra-component analysis. Our additions for the tracking of encountered expressions have been coloured blue.

the intra-component analysis, every expression that is encountered is registered (Line 5). To this end, a mapping of expressions to sets of components is created (`exprToCmps`), which links an expression to all components during the analysis of which the expression was encountered. Hence, an expression that is never encountered during an analysis will be mapped to an empty set. This also means that when an expression is not encountered during the analysis of a component, e.g., when a branch of an `if` expression is not reached, a change to this expression does not cause the reanalysis of the component.

After the annotation of the AST with changes, e.g., by a change distiller, the annotated AST can be traversed to collect all expressions that change (function `findUpdatedExpressions` in Algorithm 3). Finding these expressions is easy as the traversal only needs to retrieve the original expressions from all change expressions it encounters. Given these expressions, `exprToCmps` can be used to infer which components have been impacted directly by the change, by looking up the retrieved expressions and aggregating the resulting associated components (function `findAffectedComponents` in Algorithm 3).[2][3]

[2] In this chapter, we consider a single change and a single incremental update. When multiple successive incremental updates are possible, expressions may need to be removed again from `exprToCmps`, reflecting changes in the program behaviour, to avoid spurious reanalyses.

[3] A technical limitation of our tracking approach is that, in practice, added expressions cannot be tracked as they are not present in the initial program. Therefore, `exprToCmps` contains no information for these expressions. To find the components affected by an added expression, the

```
1  Function findUpdatedExpressions(e: Expr): Set[Expr] is
2      switch type of e do
3          case change(old, new) do return {old};
4          otherwise do
5              return subExpressions(e).flatMap(findUpdated);
6          end
7      end
8  end
9  Function findAffectedComponents() is
10     // program is globally available.
11     affectedExpr := findUpdated(program);
12     affectedComp := ∅;
13     foreach e ∈ affectedExpr do affectedComp := affectedComp ∪ exprToCmps.get(e);
14     return affectedComp;
15 end
```

Algorithm 3: The `findAffected` function.

```
1  (define (fib n)
2    (if (<change> (= n 0) (< n 2))
3        n
4        (let ((fib-n-1 (fib (- n 1)))
5              (fib-n-2 (fib (- n 2))))
6          (+ fib-n-1 fib-n-2))))
7  (fib 5)
```

Listing 4.3: Fix for the the incorrect Fibonacci number computation of Listing 2.3 on page 33.

Consider for example the change made to the `fib` function in Listing 4.3. The components whose analysis encountered the expression that is updated are `(FIB 5)`, `(FIB (- N 1))`, and `(FIB (- N 2))`. Our change tracking algorithm will thus infer that three out of the four components are directly impacted. Only the component Main is not directly impacted, as the analysis of this component did not encounter the changed expression. This is visualised in Figure 4.2. The second step of our approach will thus restart from these three components.

Due to the way the directly impacted components are computed, the granularity of the change expressions can have an influence on the number of components that are inferred to be impacted directly. The reason for this lies in the points of the program where the control flow may follow one of several branches, as is the case for an `if` expression for example. Recall that if the initial analysis for a component

expression whose execution would precede the execution of the added expression can be looked up, for example. As explained in Section 4.1, our change annotations do support the addition of expressions as we add them upfront, prior to the initial analysis.

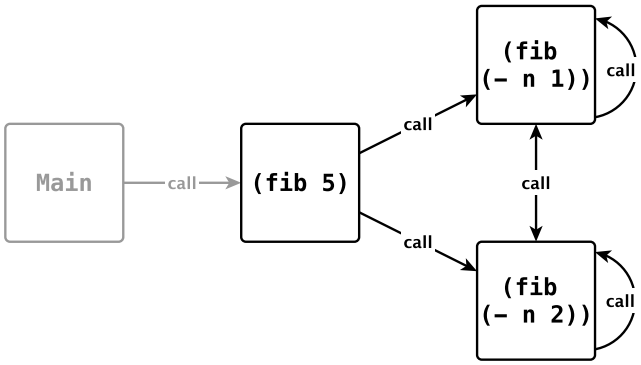


Figure 4.2: The ModF component graph corresponding to the program in Listing 4.3. Only the MAIN component, shown in grey, is not directly affected.

is precise enough to infer that only a specific branch is taken, the expressions of the other branches will not be related to the component by `exprToCmps`. Hence, if a change only spans a branch that was not taken, the corresponding component will not be inferred as being directly impacted. Therefore, changes spanning fewer branches may lead to fewer components being reanalysed.

4.2.2 Step 2: Updating the Analysis Result

After the set of directly impacted components has been computed, the analysis result can be updated, to compute and obtain an over-approximation of the behaviour of the new program version.

To update the result, the set of directly impacted components is added to the worklist of the analysis and the analysis is restarted, as shown in Algorithm 4 (the addition to the worklist is denoted using an assignment as the worklist is assumed to be empty at the beginning of an incremental update). In the example of Figure 4.2, these are the components corresponding to `(FIB 5)`, `(FIB (- N 1))` and `(FIB (- N 2))`. Remember that no other components need to be added to the worklist, as components that are indirectly impacted by the changes will be scheduled for reanalysis by the effect-driven analysis itself. For this, the analysis relies on the inter-component dependencies that were reified in D during the initial analysis of the program. Consider again the previous example. If the return value of the component `(fib 5)` changes, this will trigger a dependency causing MAIN to be reanalysed as well. In contrast, when this value does not change, MAIN is not impacted by the change and is not reanalysed. Note that even though a value within the program may change, the corresponding abstract value used by

```
1 Function incrementalUpdate() is
2 |   // V, D, and σ remain unchanged.
3 |   WL := findAffectedComponents();
4 |   computeFixedpoint();
5 end
```

Algorithm 4: Incremental update.

the analysis may remain the same. For example, if a value is represented by its type in the abstract, Main does not need to be reanalysed unless the type of the value changes.

During the updating phase, more dependencies can be inferred and new components can be discovered, as a regular effect-driven analysis is performed. The result of the analysis is monotonically updated until a new fixed point is reached, i.e., no parts of the analysis result are *cleared* or *invalidated* by the incremental update. Therefore, the result can never be more precise than the result computed by a full reanalysis: the incrementally updated result over-approximates both the old and new program version, whereas a full reanalysis only has to over-approximate the new program version. Because incremental analyses are run on small program changes, this is not expected to significantly deteriorate the precision of the analysis. However, throughout a series of incremental reanalyses, this can result in a more significant loss of precision. We therefore consider the invalidation of outdated parts of the analysis result in Chapter 5.

4.3 Formal Specification of the Approach

We now present a formal specification of the method outlined in Section 4.2, in line with the formal specification of ModF presented in Section 2.3.1 on page 25. We formalise the use of change expressions, the tracking of expressions during the intra-component analysis, the initial analysis, and the incremental update. Recall that we present the formal specification for a small input language, namely for a language based on the untyped lambda calculus with side effects and restricted to ANF.

Syntax. The syntax of the input language used for the formalisation of the analysis remains the same as shown in Figure 2.5. To express changes, however, we add a new type of expression, i.e., change expressions:

$$e \in \mathit{Expr} ::= \texttt{(<change>}\ e\ e\texttt{)} \mid \ldots$$

As introduced in Section 4.1, the first argument of a change expression is the expression in the original (old) program whereas the second argument represents the corresponding expression in the updated (new) program.

State space. The state space of the analysis also remains the same. We now however add the concept of an *analysis phase* to indicate which subexpression of a change expression should be taken into account when evaluating change expressions, i.e., the analysis phase distinguishes the initial analysis from the incremental update in our approach. We formalise this addition to the state space as follows:

$$\pi \in \Pi ::= \textsc{Old} \mid \textsc{New}$$

Intra-component analysis. The intra-component analysis is now modified to track the expressions that it encounters. As mentioned before, our approach aims to be general, putting minimal constraints on the intra-component analysis which is treated as a black box. To show this minimal dependency, we do not alter the small-step ModF transition function, $\rightsquigarrow$ specified in Figure 2.7 in Section 2.3.1, but introduce an incremental transition function, denoted $\rightsquigarrow_I$, that uses the transition function from Section 2.3.1 out of the box. The intra-component analysis function Intra itself now takes the analysis phase as an extra argument and also returns the set of encountered expressions. These functions now have the following signature:

$$\rightsquigarrow_I : \Sigma \times \mathit{Store} \times \Pi \to \Sigma \times \mathit{Store} \times \mathcal{P}(\mathit{Eff}) \times \mathcal{P}(\mathit{Expr})$$
$$\textsc{Intra} : \mathit{Cmp} \times \mathit{Store} \times \Pi \to \mathcal{P}(\mathit{Eff}) \times \mathcal{P}(\mathit{Eff}) \times \mathcal{P}(\mathit{Eff}) \times \mathit{Store} \times \mathcal{P}(\mathit{Expr})$$

As before, the transition relation returns the set of effects generated during the transition. In addition, it now also returns the set of expressions encountered. Due to the structure of the transition function, this set will either be empty or a singleton set. Again, rather than adding the set of effects and the set of expressions to the result tuple, we annotate the incremental transition function with them as follows: $\overset{E,X}{\rightsquigarrow}_I$. We use the symbol X to denote sets of expressions as we already use the symbol E to denote sets of effects.

Incremental transition function. The incremental transition function $\rightsquigarrow_I$ uses the transition function from Figure 2.7 to evaluate complex expressions while keeping track of the expressions encountered. In addition, the transition function also provides rules for evaluating change expressions given the current analysis phase. The incremental transition function is defined by the rules shown in Figure 4.3.

When an expression is evaluated, the ModF transition function is applied and the evaluated expression is tracked as it was encountered by the intra-component analysis (rule Eval). To this end, the function is annotated with a singleton set

$$\frac{\langle \mathbf{ev}(e, \rho), \iota \rangle = \varsigma \qquad e \neq (\texttt{<change>}\ e_o\ e_n) \qquad (\varsigma, \sigma) \overset{E}{\rightsquigarrow} (\varsigma', \sigma')}{(\varsigma, \sigma, \pi) \overset{E,\{e\}}{\rightsquigarrow}_I (\varsigma', \sigma')}\ \textsc{Eval}$$

$$\frac{\langle \mathbf{val}(v), \iota \rangle = \varsigma \qquad (\varsigma, \sigma) \overset{E}{\rightsquigarrow} (\varsigma', \sigma')}{(\varsigma, \sigma, \pi) \overset{E,\varnothing}{\rightsquigarrow}_I (\varsigma', \sigma')}\ \textsc{Kont}$$

$$\frac{\varsigma = (\langle \mathbf{ev}(e_o, \rho), \iota \rangle}{\big(\langle \mathbf{ev}(\texttt{(<change>}\ e_o\ e_n\texttt{)}, \rho), \iota \rangle, \sigma, \textsc{Old}\big) \overset{\varnothing,\varnothing}{\rightsquigarrow}_I (\varsigma, \sigma)}\ \textsc{Old}$$

$$\frac{\varsigma = (\langle \mathbf{ev}(e_n, \rho), \iota \rangle}{\big(\langle \mathbf{ev}(\texttt{(<change>}\ e_o\ e_n\texttt{)}, \rho), \iota \rangle, \sigma, \textsc{New}\big) \overset{\varnothing,\varnothing}{\rightsquigarrow}_I (\varsigma, \sigma)}\ \textsc{New}$$

Figure 4.3: The tracking transition function for a small-step ModF analysis.

containing the evaluated expression. However, the expression may not be a change expression, as this type of expression requires a specific treatment and is not supported by the transition function of Section 2.3.1. When the evaluation has reached a value, the transition function is applied as well. In this case, however, the analysis does not encounter an expression, so no expression needs to be tracked (rule Kont).

When a change expression is encountered, the expression to be evaluated is either the old expression, encoded as the first argument of the change expression, or the new expression, encoded as the second argument of the change expression. In this case, the ModF transition function is not used. Instead, a new evaluation state is created, containing the expression to evaluate next depending on the program version that is to be analysed. No effects are generated in either case. In addition, no expressions need to be tracked: the tracked expressions are used to find the components affected by changes, i.e., the components that rely on an expression that is the first argument of a change expression. For this reason, it is not needed to track entire change expressions as well because due to the transition rules, the subexpression representing the old resp. new expression is tracked in the next evaluation step as soon as it is encountered by the incremental transition function.[4]

[4]On this point, our formal specification deviates slightly from Algorithm 2 and from our implementation, which track every expression. It is possible, but not necessary, to track entire change expressions as well.

Intra-component analysis. The incremental transfer function defines how the intra-component analysis steps from one state to the next during the analysis of a component, while keeping track of the encountered expressions and while selecting the expressions corresponding to the correct program version. As before, the intra-component analysis Intra explores all reachable states. It now not only returns the generated effects and resulting store, but also the set of encountered expressions. Intra is again defined in terms of a step function. Both now take the analysis phase as an extra parameter, which is passed to the incremental transfer function. We now define both functions as follows:

$$step_\pi(\varsigma s, \sigma, E, X) = \{(\varsigma s, \sigma, E, X)\} \cup \bigcup_{\substack{\varsigma \in \varsigma s \\ (\varsigma,\sigma,\pi) \overset{E_s, X_s}{\rightsquigarrow}_I (\varsigma_s, \sigma_s)}} (\{\varsigma_s\}, \sigma_s, E_s, X_s)$$

$$\textsc{Intra}(\alpha, \sigma, \pi) = \Big(\bigcup_{\mathbf{c}(\beta) \in E_i} \mathbf{c}(\beta), \bigcup_{\mathbf{r}(a) \in E_i} \mathbf{r}(a), \bigcup_{\mathbf{w}(a) \in E_i} \mathbf{w}(a), \sigma_i, X_i \Big)$$

where $(\varsigma s_i, \sigma_i, E_i, X_i) = \mathit{lfp}(\mathit{step}_\pi(\{\varsigma_0\}, \sigma, \varnothing, \varnothing))$ with, as before, $\varsigma_0 = \langle \mathbf{ev}(e, \rho), \mathbf{ret}(\alpha) : \varepsilon \rangle$, *lfp* the least fixed point operator and $\cup$ over tuples the pairwise union/join. As before, the subscript s denotes the result of a step and the subscript i denotes the result of the intra-component analysis.

Inter-component analysis. The inter-component analysis now takes an analysis phase as an extra argument, to distinguish the initial analysis from the incremental update (the analysis phase is merely passed to the intra-component analysis). In addition, it produces a mapping of expressions to components, which has the function of `exprToCmps` in Algorithm 2. Since Inter is defined recursively, it also takes this kind of mapping as an extra argument. As a result, Inter now has the following signature:

$$\begin{aligned} \textsc{Inter}\colon\ & \mathcal{P}(Cmp) \times (Addr \to \mathcal{P}(Cmp)) \times Store \times (Expr \to \mathcal{P}(Cmp)) \times \Pi \\ & \to \mathcal{P}(Cmp) \times (Addr \to \mathcal{P}(Cmp)) \times Store \times (Expr \to \mathcal{P}(Cmp)) \end{aligned}$$

We update its definition to pass the analysis phase to the intra-component analysis Intra. Thus, the fixed-point computation that constitutes the inter-component

analysis Inter now is defined as

$$\textsc{Inter}\big(\varnothing, D, \sigma, T, \pi\big) = \Big(\big\{\alpha \mid \alpha \in \mathrm{dom}(\sigma) \wedge \alpha \in Cmp\big\}, D, \sigma, T\Big)$$

$$\begin{aligned}\textsc{Inter}\big(\{\alpha\} \uplus \alpha s, D, \sigma, T, \pi\big) = \textsc{Inter}\Big(&\alpha s \cup \bigcup_{\substack{\mathbf{c}(\beta)\in C \\ \beta\notin\mathrm{dom}(\sigma)}} \beta \cup \bigcup_{\substack{\mathbf{w}(a)\in U \\ \gamma\in D(a)}} \gamma,\\ &D \cup \dot{\bigcup_{\mathbf{r}(a)\in R}} [a \mapsto \{\alpha\}], \sigma',\\ &T \cup \dot{\bigcup_{e\in X}} [e \mapsto \{\alpha\}], \pi\Big)\end{aligned}$$

where $(C, R, U, \sigma', X) = \textsc{Intra}(\alpha, \sigma, \pi)$. We use the symbol T to denote a map from (tracked) expressions to sets of components.

Collecting semantics. Given this new definition of Inter, we can now specify the computation of the collecting semantics. We do this both for the initial analysis and for the incremental update.

The collecting semantics for the initial program can be obtained by computing $\textsc{Inter}(\{\textsc{Main}\}, [], [], [], \textsc{Old})$. The incremental update of the analysis result is computed by $\textsc{Inter}(\alpha s, D, \sigma, T, \textsc{New})$, where $(\beta s, D, \sigma, T)$ is the result of the initial analysis and αs is computed as follows:

$$\alpha s = \bigcup \{T(e) \mid e \textit{ is updated}\}$$

where an expression e is considered to be updated if it is the first argument of a change expression in the annotated program.

Memory overhead. We now use our formal specification to quantify the memory overhead of our method. The only memory that needs to be allocated extra in comparison to a non-incremental effect-driven analysis, is the memory to store the map T which maps expressions to sets of components:

$$T : Expr \to \mathcal{P}(Cmp)$$

We use the big O notation to express an upper bound for the memory overhead of our method. Using the function signature of T, we can compute its worst-case size: $T \in \mathcal{O}(|Expr| \times |Cmp|)$. The worst-case memory overhead of T is thus related to the number of expressions in components in the program, but in practice, not all components encounter all expressions and the number of components may be limited. In addition, as our implementation is written in Scala, a map data structure contains object references rather than the objects themselves. As the objects pointed to also exist in the non-incremental analysis, no extra memory is required to store them.

Consider now a function-modular analysis. In this case, every expression may only be encountered by a limited number of components, i.e., by the components corresponding to the function to which the expression belongs. If the function-modular analysis is context insensitive, every expression can only be encountered by at most one component and we can reduce the upper bound for T to $T \in \mathcal{O}(|Expr|)$. Thus, for a context-insensitive ModF analysis, the memory overhead grows linearly with the program size.

4.4 Evaluation

To evaluate our approach, we have applied our incrementalisation method to two different effect-driven analyses for μScheme, a function-modular (ModF) analysis and a thread-modular (ModConc) analysis. Using these instantiations, we aim to answer the following three research questions:

RQ 4.1. Does an incrementalised effect-driven modular analysis result in a reduction of analysis time in comparison to a full reanalysis of the modified program?

RQ 4.2. How precise is an incremental update compared to a full reanalysis of the modified program?

RQ 4.3. What is the impact of the granularity of the components on the effectiveness of our approach?

4.4.1 Set-up and Benchmark Programs

Our approach, including the change expressions described in Section 4.1 and the two instantiations described above, has been implemented for the μScheme language in the Modular Analysis Framework MAF [152], a research framework developed at the Software Languages Lab for studying effect-driven modular static analyses. The framework is implemented in Scala in a modular way, allowing different analysis specifications, e.g., various context sensitivities, different lattice implementations or different language semantics, to be used and swapped by changing the Scala traits that are mixed into the analysis. Therefore, the framework facilitates performing experiments using different analysis implementations and with various analysis configurations.

Our implementation has been made available online at `https://github.com/softwarelanguageslab/maf/releases/tag/SCAM2020`.

We now describe the two instantiations of the incremental analysis used for evaluation in more detail, together with the benchmark programs used for each instantiation. Both instantiations use a type lattice, approximating values by their type, except for functions and pointers which are approximated as sets of abstract closures or addresses respectively. Booleans are represented using their respective value if possible. For our experiments, no context sensitivity is used, i.e., the contexts of components are empty.

The benchmark programs used for our evaluation are available online at `https://github.com/jevdplas/SCAM2020-Benchmarks` [146].

ModF Analysis for µScheme

The first instantiation used to evaluate our method is a ModF analysis for µScheme [96]. The implementation of the intra-component analysis follows a *big-step semantics*, where the analysis is implemented using a recursive evaluation function to evaluate and analyse an expression, that is, to evaluate a subexpression of a given expression, the evaluation function is just called recursively.

With this instantiation, we use a set of seven benchmark programs to which change expressions have been added. These programs are described in Table 4.1a, which also indicates their size and explains the changes made to them. For example, in the `peval` benchmark, we abstracted duplicated code into a function and replaced all occurrences of that code by a function call to the newly introduced function.

ModConc Analysis for µScheme

The second instantiation used to evaluate our approach is a ModConc analysis for µScheme [126, 128]. The ModConc benchmark programs use a version of µScheme that contains threads and locks, which are the concurrency constructs used in the ModConc literature. The implementation of the intra-component analysis follows a *small-step semantics*, the type of semantics we have used for the formal specification of ModF in Section 2.3.1 (page 28). In a small-step semantics, the evaluation of an expression is performed in small steps. These are implemented using evaluation and continuation states which keep track of the continuation of the evaluation, manually pushing and popping stack frames on the continuation stack while progressing the evaluation step by step. Instead of a recursive evaluation function,

a small fixed-point computation is performed where a state is continuously removed from a worklist and *stepped*, except if the state has been encountered before, yielding one or more successor states which are then added to the worklist.

With this instantiation, we use a set of nine benchmark programs to which change expressions have been added. These programs are described in Table 4.1b, which also indicates their size and explains the changes made to them. For example, in the `mcarlo2` benchmark, we halve the number of threads created in the program.

By using a big-step semantics for the ModF analysis and a small-step semantics for the ModConc analysis, to which we have applied our incrementalisation method, we illustrate the generality of the incrementalisation method we have developed.

4.4.2 Evaluation Method

To answer the research questions posed above, we use the following metrics:

1. The *analysis time*: we measure the time needed by (1) the initial analysis of the program, (2) the incremental update of the analysis result and (3) a full reanalysis of the updated program. To gain certainty in our measurements, every measurement is repeated 35 times, of which the first 5 repetitions are considered warm-up and discarded.
2. The *precision* of values in the store of the analysis: for every address mapped in the store, we compare the abstract values computed by the incremental update and by the full reanalysis.
3. The *size of the store*.
4. The *number of components* discovered by the analysis.
5. The *number of dependencies* inferred by the analysis.
6. The *number of intra-component analyses* performed.

The size of the store and the number of dependencies and components give an indication of the number of program paths explored by the analysis and hence give another view of its precision. The number of intra-component analyses is an alternative measure for the amount of work required to complete an analysis.

Using the above metrics, we compare the analysis time and result obtained after an incremental update to those of a full analysis of the updated program. The lower the number of components and dependencies and the smaller the size of the store, the more precise the analysis result is. We also compare all abstract values mapped to in the store. As they are part of a lattice, the partial order relation of

Benchmark	LOC	Description of the Sequential Program	Changes	#Changes
mceval-dynamic	246	Meta-circular evaluator for Scheme, executing a small Scheme program.	Changed the evaluator so procedures become dynamically scoped.	4
leval	379	Lazy Scheme evaluator, used to perform some list computations.	Changes to the evaluator so that only specific arguments are evaluated lazily.	11
multiple-dwelling (fine)	404	Evaluator for a non-deterministic Scheme, used to solve an allocation problem.	Fine-grained changes to the input for the evaluator.	3
multiple-dwelling (coarse)	434	Evaluator for a non-deterministic Scheme, used to solve an allocation problem.	Changed the input for the evaluator.	1
peval	507	Partial evaluator for Scheme, used to evaluate multiple small programs on given input.	Abstracted repeated code to a function and replaced all occurrences by a call to this function.	38
nboyer	636	Version of the Boyer benchmark. Evaluator for logic programs, applied to a small logic program.	Rewrote conditionals with two branches. In one instance, a semantic difference was introduced.	2
machine-simulator	1116	Compiles a factorial into machine code, then uses a simulator to execute this code.	Modified the compiler to generate faster code for some primitive functions.	7

(a) The benchmark programs for the ModF evaluation.

Benchmark	LOC	Description of the Concurrent Program	Changes	#Changes
mcarlo2	28	Monte Carlo simulation.	Now creates less threads, to avoid waiting on a thread just created.	2
pc	43	Producer-Consumer problem.	Converted a variable into a function, and replaced variable references with function calls.	2
msort	44	Merge sort.	Updated implementation of `sorted?` to avoid creating useless threads.	3
pps	71	Parallel-Prefix Sum.	Swapped around the expressions in the body of a procedure.	1
sudoku	84	Sudoku checker.	Changed the sudoku board by replacing a number by `'oops`.	1
actors	103	An implementation of actors using threads.	Replaced `begin` expressions with a single expression in their body.	2
stm	138	Implementation of Software-Transactional Memory.	Updated definitions of `every?` and `map-contains?` to improve code style.	2
crypt	170	Implementation of Vigenère cipher cryptanalysis.	Changed the implementation of `fold`.	1
crypt2	174	Implementation of Vigenère cipher cryptanalysis.	Changed the implementation of `argmin` to use `foldl`.	1

(b) The benchmark programs for the ModConc evaluation.

Table 4.1: The benchmark programs used for the evaluation, adapted from various sources. Lines of code as counted with `cloc`. The benchmark programs for ModF and ModConc are available online at `https://github.com/jevdplas/SCAM2020-Benchmarks` [146].

Benchmark	Initial Analysis [ms]	Full Reanalysis [ms]	Incremental Update [ms]	Δ
ModF				
mceval-dynamic	226	124	72	-41.94%
leval	1407	1971	489	-75.19%
multiple-dwelling (fine)	8466	8822	2126	-75.90%
multiple-dwelling (coarse)	3527	3533	15694	+344.21%
peval	19753	17644	103	-99.42%
nboyer	1397	1271	98	-92.29%
machine-simulator	54124	57043	24093	-57.76%
ModConc				
mcarlo2	9	29	27	-6.90%
pc	21	16	11	-31.25%
msort	117	151	194	+28.48%
pps	421	423	1	-99.76%
sudoku	86	90	62	-31.11%
actors	1601	1595	354	-77.81%
stm	5384	5597	745	-86.69%
crypt	7568	7351	2812	-61.75%
crypt2	9315	10277	8340	-18.85%

Table 4.2: Timing results using a timeout. Every measurement is repeated 30 times, of which the average is shown. The delta shows how the time needed by the incremental update compares to the time needed by a full reanalysis.

the lattice is used to infer which values are more precise. All experiments were run on a 2015 Dell PowerEdge R730 with 2 Intel Xeon 2637 processors. We used OpenJDK 1.8.0_265, Scala 2.13.3 and a maximal heap size of 4GB.

4.4.3 Experimental Results

Table 4.2 contains the results for our evaluation of the analysis time. For ModF, we note a reduction of the analysis time from 40% up to 99% for all but one benchmark, `multiple-dwelling (coarse)`, for which the incremental update is a lot slower than a full reanalysis. For ModConc, we see reductions of the analysis time ranging between 6% and 99% on all but one benchmark, `msort`, for which the incremental update is slightly slower than a full reanalysis. These numbers indicate that our method overall results in reduced analysis times.

ModF				
Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)
mceval-dynamic	158	220	58.20%	10
leval	187	389	67.53%	10
multiple-dwelling (fine)	851	0	0.00%	0
multiple-dwelling (coarse)	231	817	77.96%	198
peval	919	2	0.22%	0
nboyer	2115	17	0.80%	1
machine-simulator	1676	14	0.83%	7

ModConc				
Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)
mcarlo2	28	2	6.67%	1
pc	35	4	10.26%	1
msort	27	9	25.00%	1
pps	99	0	0.00%	0
sudoku	101	0	0.00%	0
actors	136	0	0.00%	0
stm	156	0	0.00%	0
crypt	141	3	2.08%	3
crypt2	140	6	4.11%	6

Table 4.3: Precision results. The table indicates how many addresses in the store after an incremental update contain a value that is equal or less precise compared to a full reanalysis of the updated program. 66 addresses corresponding to built-in functions are ignored as they are never assigned and hence of equal precision in all cases. A fourth column indicates the number of addresses present in the incrementally updated store minus the number of addresses in the store after a full reanalysis.

There are two versions of `multiple-dwelling`, as, for this program, the same changes were applied using different granularities of change expressions. Hence, the difference between the performance of the incremental update on two versions is striking as they both represent the same program with the same code changes, though the granularity of the expressions used to encode the changes differs. In both versions, an input list is changed; in the coarse-grained version, the entire list is updated, whereas in the fine-grained version, the change expressions are put around the elements of the list that change. We find that this difference might be explained by the fact that our analysis cannot invalidate outdated parts of the result, which is exacerbated by the exact change: the change to `multiple-dwelling (coarse)` causes an entire new list to be allocated by the analyser, thereby creating a vast amount of pointers. We find that, after the incremental update, the store

ModF

Benchmark	Components			Dependencies			Intra-Component Analyses		
	I	R	U	I	R	U	I	R	U
mceval-dynamic	86	85	87	2647	2057	2742	1722	1529	273
leval	101	107	109	4999	6683	6840	4155	4413	971
multiple-dwelling (fine)	139	139	139	14138	14538	14538	7442	7442	1245
multiple-dwelling (coarse)	139	139	139	14138	14498	22418	5063	5115	7982
peval	90	91	91	23564	23570	24056	4816	5222	20
nboyer	45	45	45	20366	20364	20376	1360	1310	33
machine-simulator	282	289	289	55452	56166	56173	43460	48502	7633

ModConc

Benchmark	Components			Dependencies			Intra-Component Analyses		
	I	R	U	I	R	U	I	R	U
mcarlo2	3	2	3	90	62	91	7	4	4
pc	3	3	4	66	67	85	8	7	6
msort	3	2	3	105	77	112	11	6	9
pps	3	3	3	138	138	138	6	6	1
sudoku	30	30	30	1051	1051	1051	63	63	35
actors	2	2	2	233	233	233	4	4	1
stm	2	2	2	268	268	268	7	7	1
crypt	2	2	2	293	293	299	8	8	3
crypt2	2	2	2	293	291	303	8	8	6

Table 4.4: Number of components created, number of dependencies inferred and number of intra-component analyses performed by the initial analysis of the original program (I), the full reanalysis of the updated program (R) and the incremental update of the initial result (U).

of the analysis contains almost 60% more pointers for the coarse-grained version than for the fine-grained program version. As pointers cannot be efficiently joined by our implementation, this possibly causes the slowdown.

Table 4.3 shows the results of our precision evaluation, obtained by comparing the abstract values at each address in the store. Recall that the incremental analysis can never be more precise than a full reanalysis. We see however that on a majority of benchmarks, the precision loss is very small to none. Yet, on some benchmarks, the loss in precision is more important. For example, `multiple-dwelling (coarse)` sees a huge loss in precision, as more than 75% of the values in the store is less precise. This can again be linked to the fact that the incremental update creates a lot of pointers while being unable to remove outdated parts of the result, as can be seen in the fourth column of the table. For the `msort` benchmark, we find that the imprecision arises due to the fact that the incremental update does not remove

components: after the incremental update, the store contains abstract values at 9 addresses related to components that are not created by the full reanalysis. Hence, the values at these addresses computed by the incremental update are less precise than those computed by a full reanalysis.

Finally, we consider the results in Table 4.4, which shows the number of components and dependencies discovered by the analysis, as well as the number of intra-component analyses performed for the initial analysis, full reanalysis, and incremental update of the initial analysis result. On all but three benchmarks, the incremental update requires less intra-component analyses than a full reanalysis, and hence overall our approach reduces the work required to reach an updated fixed point. The result of an incremental update is less precise than that of a full reanalysis as more components and/or dependencies are inferred for most benchmarks. However as discussed before, the impact on the abstract values in the store is limited. The loss of precision can be mitigated by performing a full reanalysis, e.g., at regular intervals. The point where a full reanalysis is needed may depend on the actual analysis performed, and should be determined accordingly. In Chapter 5, we present three strategies to reduce the precision loss caused by incremental updates to alleviate the need to perform a full reanalysis every once in a while.

4.4.4 Discussion

Our results show that, in general, the incrementalisation method presented in this chapter leads to a reduction of the analysis time, compared to a full reanalysis of the program (**RQ 4.1**). This is also visible when comparing the number of intra-component analyses required to reach the fixed point. On two programs, a slowdown is seen, which is caused by an increased imprecision due to the incremental update. However, in general, the precision of an incremental update seems to be comparable to that of a full reanalysis (**RQ 4.2**). We see that for ModF, on average, our approach results in higher reductions of the analysis time than for ModConc (**RQ 4.3**). This is most likely caused by the fact that the ModF analyses create more components than the ModConc analyses, given our evaluation set-up. Also, the components created by ModConc, which correspond to spawned threads, are generally larger than the ones created by ModF, which correspond to function calls. Hence, ModConc leads to more coarse-grained incrementality for which the reduction of the analysis time may be smaller.

Our experiments show that our method leads to a reduction of the reanalysis time when applied on context-insensitive analyses for μScheme, a highly dynamic, higher-order language. We find that the approach is sufficiently general to be

applied to different types of effect-driven analyses, as we have demonstrated using our experiments. Hence, the method presented in this chapter improves upon current incremental analyses that require a statically known call graph or are tailored to specific analyses.

As is shown in the example of Figure 2.9 on page 33, which uses call expressions as component contexts, effect-driven analyses can also be instantiated with context-sensitive analyses. Our approach therefore also works for any context sensitivity as it does not put restrictions on the components, which therefore can contain arbitrary contexts. We restricted our evaluation to two context-insensitive analyses, where all components contain empty contexts. If more information is stored in a context, the analysis may create more components, which both impacts the incremental update and full reanalysis times. This may also depend on the abstract domain used, as a component context can contain abstract values. Hence, extending the evaluation to context-sensitive analyses would require investigating the influence of the abstract domain on the performance of the incremental analysis in comparison to a full reanalysis. However, the abstract domain only impacts the creation of components via the component contexts, hence we do not expect changes in the results for context-insensitive analyses when using different abstract domains.

4.4.5 Threats to Validity

We now briefly identify possible threats to the validity of our results, following the classification of Wohlin *et al.* [160].

Analysis Framework

A threat to the *external validity* comes from the framework in which our method has been implemented. This framework is based on ModF [96] and ModConc [126, 128], both inspired by the work of Cousot and Cousot [30]. Various precision-improving optimisations, such as abstract garbage collection [83, 150], exist, but have not been incorporated in our framework. We do not expect detrimental changes to our experimental results should they be integrated. Also, our method has only been incorporated in a research-oriented framework. An incorporation in a production quality tool may be required to show how our method performs in practice. We are however unaware of any industry-standard analysis frameworks for dynamic languages that offer heavyweight analyses and follow a modular design.

Evaluation

A threat to the *conclusion validity* of our experiments stems from the low number of benchmark programs used. In the literature, there is no standard set of benchmarks used to evaluate incremental static analyses of dynamic languages. We therefore had to compose a benchmark suite ourselves. To compose this benchmark suite, we added change expressions to the benchmark programs manually. Another approach would be to add such expressions programmatically, enabling more benchmark programs, but such changes might not reflect real changes made by developers. We nonetheless take this approach for completeness' sake in Chapter 5 to complement a suite of curated benchmarks.

To each program, we have manually added changes. We did not possess change histories for the programs. We however made sensible and varied changes that could reflect actual developer edits. For example, some of the ModF benchmark programs have been used during university classes. To these programs, the changes correspond to solving a course assignment. For ModConc, the changes could correspond to refactorings. We therefore believe that our changes are sufficiently varied and realistic to validate our approach, even though the number of changes to some programs is limited.

4.5 Conclusion

In this chapter, we first introduced change expressions, the change representation used in our work. We then introduced a method to incrementalise effect-driven analyses based on reified inter-component dependencies. To this end, the intra-component analysis registers the expressions encountered during the analysis of each component. A change impact calculation infers the components directly affected by a change, which are then reanalysed. The reified dependencies ensure that transitively affected components are reanalysed as well. Hence, the effect-driven nature of the analysis leads to a relatively straightforward incrementalisation, where only the parts of the analysis result for the components directly or indirectly affected by the changes are updated.

We applied our method to both a function-modular and a thread-modular effect-driven analysis for μScheme, a highly-dynamic, higher-order programming language. We found that an incremental update is faster than a full recomputation of the result on 14 out of 16 benchmark programs, reducing the analysis time by up

to 99%. A high precision is retained for most benchmark programs. To reduce the precision loss and improve the precision of incrementalised effect-driven analyses, we consider three strategies for precision recovery in the next chapter.

In comparison to existing work, our method results in general incremental effect-driven analyses that can support highly dynamic, higher-order languages and program changes that modify the program's call graph. Our method is not specific to any particular analysis, and allows the impact of changes to be bounded to the parts of the analysis results that are affected by program changes. Additionally, the analyses must not be reformulated, and only a single lightweight auxiliary data structure is required to keep track of the expressions encountered during the analysis of each component.

Method to incrementalise an effect-driven analysis.

To incrementalise an effect-driven analysis, add expression tracking to the intra-component analysis. For every expression, keep track of the components for which the intra-component analysis encountered the expression.

To perform an incremental update:

1. Collect the set of directly affected components and add them to the worklist of the analysis. All components for which the intra-component analysis encountered an expression that is affected by the source code changes are directly affected.
2. Restart the fixed-point computation of the inter-component analysis.

5

Three Strategies for Precision Recovery

This chapter is based on our 2023 paper "Result Invalidation for Incremental Modular Analyses" [148]. The text has been slightly modified and extended for clarity and completeness. In addition, a formal specification has been added (Section 5.3).

The previous chapter introduced a general method for rendering any effect-driven static analysis incremental. Such analyses divide a program into components which are (re-)analysed separately but whose analyses may be interdependent; these inter-component dependencies are reified using effects emitted by the analyses of the components. The effect-driven and modular nature of these analyses facilitates bounding the impact of changes.

While the evaluation presented in the previous chapter shows that incremental updates are often faster than a full reanalysis, the incremental updates may be less precise than a full reanalysis as the presented analysis cannot delete outdated parts of the analysis result. In this chapter, we improve upon the previous by making the following contributions:

- We introduce three complementary strategies to regain lost precision. The idea is to *interleave* invalidation with recomputation, to maximise reuse of the previously computed result. The invalidation is based primarily on the effects emitted by the intra-component analysis, which allows us to specify the strategies as general as possible without relying on a specific intra-component analysis, that is, we maintain the view of components and the intra-component analysis as *black boxes* to the maximum extent, as we did in the previous chapter.

- We implement these strategies and evaluate their impact on the precision and performance of the incremental analysis, when used alone or in combination.

Without loss of generality, we choose to present the three invalidation strategies for the incremental ModF analysis from Chapter 4. Our method remains applicable to other effect-driven analyses that have been incrementalised using the method of the previous chapter, such as e.g., a ModConc analysis, but for clarity, we now only focus on a single instantiation for our explanation, examples, and evaluation.

5.1 Sources of Imprecision

The incrementalisation method for effect-driven analyses presented in the previous chapter can be summarised as follows. For a given set of change expressions, the affected parts of the analysis result are computed and updated accordingly. The analysis tracks which expressions within the source code of a module were encountered during the analysis of the corresponding components. Every component for which the intra-component analysis encountered a expression that is changed is added to the worklist, after which the fixed-point computation is restarted. The effect-driven analysis design ensures that indirectly affected components are reanalysed as well.

Table 5.1 depicts the three parts of the result of an effect-driven analysis. The method presented in the previous chapter only updates the prior result monotonically: no outdated information can be removed; the result of the analysis over-approximates the behaviour of both the updated and the original program. All parts of the result may suffer from imprecision, as shown in Table 5.1. This means that components and dependencies no longer representing the program's behaviour cannot be removed. In the value store σ, values cannot become more precise. Note that imprecision in one part of the result may cause imprecision in other parts. For example, when a value in σ is imprecise, the analysis may explore more paths and thus infer more components and dependencies, which may in turn degrade the precision of the store.

Components	
Explanation	Set of components created during the analysis, each abstractly representing an aspect of the runtime behaviour of the program, e.g., a function call.
Imprecision	Components no longer representing the program's behaviour cannot be removed.
Solution	*Component Invalidation:* remove components that are no longer created.
Dependencies	
Explanation	Set of inter-component dependencies (read effects) computed during the analysis, each marking a link between a component and an address in the global value store σ. Using these dependencies, the analysis of one component takes into account information computed by the analysis of other components.
Imprecision	Dependencies that are no longer valid cannot be removed.
Solutions	*Dependency Invalidation:* remove dependencies that are no longer computed by the reanalysis of an impacted component.
	Component Invalidation: removing a component clears its dependencies.
Value Store σ	
Explanation	Over-approximates the heap. Mapping of abstract addresses to abstract values.
Imprecision	Values in σ are updated monotonically, since they are joined upon updates.
Solutions	*Write Invalidation:* improve the precision of values in the store σ by removing values that are no longer written.
	Component Invalidation: when write invalidation is enabled, the removal of a component may allow the value store σ to be refined.

Table 5.1: Overview of the parts of the result of an effect-driven analysis, of the sources of imprecision for each part, and of the corresponding strategies to invalidate outdated parts of the result.

5.2 Strategies for Precision Recovery

We now introduce three complementary strategies that improve the precision of an incremental analysis result by invalidating the information that corresponds to outdated program behaviour. The aim is to minimise the precision loss caused by monotonic updates to a prior analysis result, without increasing the analysis time.

5.2.1 Invalidation Principle

The incrementalisation method introduced in Chapter 4 was conceived to be as general as possible by solely requiring the intra-component analysis to emit effects when needed, together with tracking the expressions encountered, and by not

```
1  WL := {Main}; // The worklist, initially containing the Main component.
2  V := ∅; // The set of visited components.
3  D := λr.∅; // Map of dependencies (read effects).
4  σ := λa.⊥; // Global value store, initially all addresses map to bottom.
5  while WL ≠ ∅ do
6  |   α ∈ WL;
7  |   WL := WL \ {α};
8  |   (C', R', U', σ') = intra(α, σ); // Intra-component analysis.
9  |   σ := σ';
10 |   V := V ∪ {α};
11 |   WL := WL ∪ (C' \ V);
12 |   foreach r ∈ R' do D := D[r ↦ D(r) ∪ {α}];
13 |   foreach u ∈ U' do WL := WL ∪ D(u);
14 end
15 return (σ, V, D);
```

Algorithm 5: The inter-component analysis of an effect-driven analysis. (Repetition of Algorithm 1.) Invalidation happens after the intra-component analysis, i.e., after Line 8.

introducing any other fundamental restrictions. The strategies presented in this chapter treat the intra-component analysis as a black box as well and thus do not put any restrictions on the lattice nor on the context-sensitivity used by the analysis. Again, the intra-component analysis must only compute a set of effects.

The aim is to invalidate as few valid parts of the result as possible, so that the parts not impacted by a change need not be needlessly recomputed. As explained in Section 3.1, related work such as the work of Arzt and Bodden [12] and Nichols *et al.* [92] often consists of an *invalidation phase*, which *over-approximates* and clears outdated parts of the result, and a *recomputation phase*, which updates the analysis result. To avoid over-approximating the outdated parts of the result, we *interleave* invalidation with recomputation, maximising reuse. After an intra-component analysis, Inter computes which parts of the result have become obsolete and removes them; information is only removed when it is no longer computed by an intra-component analysis. Mapping this onto the algorithm of the inter-component analysis, which we have repeated in Algorithm 5, invalidation happens after Line 8. Our approach leads to a recompute-and-invalidate cycle: the analysis of a component may lead to a result invalidation, which in turn can lead to more analyses of components. It is important to note that the computations performed by the intra-component analysis remain monotonic as we do not alter it, that is, no invalidation happens during the analysis of a component.

Upholding the Black-box View

The interleaving of the invalidation phase and the recomputation phase, which sets it apart from the related work on incremental static analysis discussed in Chapter 3, also allows us to uphold the black-box view of components: the strategies presented in this chapter invalidate information merely based on the effects emitted by the intra-component analysis and on the interactions with the store. After all, to perform a precise a priori invalidation phase, precise knowledge on information flow within the intra-component analysis is required to infer which parts of the analysis result depend on which other parts and thus may need to be invalidated. However, this information cannot be inferred from the reified computational effects on top of which our incremental method is built. Hence, a precise a priori invalidation phase would require us to abandon the black-box view of components and to impose extra restrictions on the intra-component analysis, i.e., the intra-component analysis would need to compute and provide information-flow data. To preserve the generality of our approach, this is not an option we have taken. In addition, the result of such a precise a priori invalidation phase would still be an over-approximation.

The other alternative, which does manage to uphold a black-box view of the intra-component analysis, is to perform an a priori invalidation phase solely based on the observed effects. However, such an invalidation phase could only be imprecise, leading to a vast amount of needless invalidation and recomputation afterwards as there is no causal relation between the emitted effects themselves nor between the emitted effects and the program changes: the effects do not provide any information on which reads, writes or component creations are impacted by the changes in the program as there is no information-flow data available. Thus, this kind of imprecise invalidation would be very coarse-grained, where all creation and write effects could cause a cascade of invalidations among dependent components (even when the order of effects is taken into account or when effects can be related to specific program elements). For example, to perform a safe over-approximation of the parts of the analysis result to invalidate, the values in the store of all addresses written to by a directly affected component would need to be invalidated, as well as all information transitively written by components that read the values of any of the invalidated addresses. As this behaviour clearly is not desirable, this is not an option we have taken either.

As stated before, the interleaving of the invalidation phase and the recomputation phase avoids the over-approximation typically seen in the former phase, that is, our strategies only remove information which is no longer computed by an intra-component analysis and thereby try to remove as few valid parts of the analysis result as possible. Furthermore, the principle of interleaving which

```
1  (define (fac n)
2    (if (< n 2)
3        n
4        (* n (fac (- n 1)))))
5  (define (fac-loop n) ; Executes the `fac` function in a loop.
6    (define (loop i)
7      (if (< i n)
8          (begin
9            (display (fac i))
10           (display " ")
11           (loop (+ i 1)))))
12   (loop 0))
13 (<change> (fac-loop 10) (fac 10)) ; Updated to call `fac` directly.
```

Listing 5.1: A change causing components to be removed.

governs our invalidation also allows us to uphold the desired black-box view of the intra-component analysis, ensuring a broad and general applicability of our incrementalisation approach.

Table 5.1 outlines the developed strategies, one for each part of the result of an effect-driven analysis: component invalidation, dependency invalidation, and write invalidation. However, note that invalidations in one part of the result may also impact the other parts.

5.2.2 Component Invalidation (CI)

Component invalidation (CI) removes components from the analysis result that are no longer created by any other component, plus the dependencies related to these components. Consider e.g., the program in Listing 5.1. The initial analysis creates four components, shown by the component graph on top of Figure 5.1. The change expression replaces the call to `fac-loop` by a call to `fac`; `fac-loop` (and transitively `loop`) are no longer called. The reanalysis of Main now finds that fac-loop is no longer called, so that fac-loop can be removed. As this was the only component emitting a call effect for loop, this component can be removed as well.

Component invalidation uses the component graph to detect outdated components: all components no longer transitively reachable from Main, i.e., from the entry point of the program, can be removed. Algorithm 6 extends Inter with component invalidation. For every component α, Inter caches C_α, the set of components called by α's last analysis, using a cache $\mathbb{C}$. The set of dependencies R_α, cached in $\mathbb{R}$,

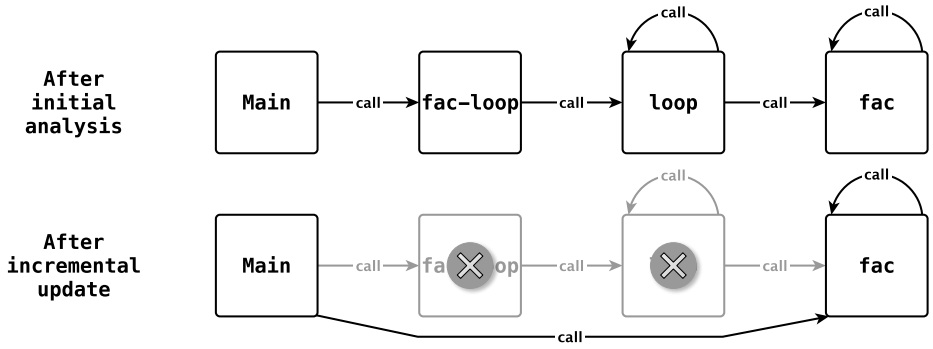


Figure 5.1: ModF components for the program in Listing 5.1. On top, the components after the initial analysis of the program; at the bottom, the components after the incremental update. Arrows depict generated call effects.

enables the efficient removal of the dependencies of deleted components ($\mathbb{R}$ holds the same information as D but in the reverse order, avoiding a full traversal of D). After the analysis of a component α, the set of components called by the analysis of this component, C'_α, is returned. Inter then retrieves C_α, the set of components called during the *previous* analysis of α, and updates the cache $\mathbb{C}$ (Lines 14-15). It then computes the set containing all components that are no longer called by α. If this set is non-empty, one or more edges were removed from the component graph and some components may have become outdated (Line 16). In this case, the transitive closure of $\mathbb{C}$ is computed, starting from Main; all components that are not part of it are removed (Lines 17-18). All dependencies of these components are removed too, avoiding the existence of dependencies of non-existent components. The transitive closure is needed because a component can only be removed if it is no longer created by any other component. Finally, $\mathbb{R}$ is updated (Line 20). Note that Lines 17 and 18 will never be executed during the initial analysis of the program. To avoid the needless but possibly expensive computation of set differences in the condition, we first check whether an incremental update is taking place (Line 16). For similar reasons, we do the same for dependency invalidation and write invalidation (see Sections 5.2.3 and 5.2.4).

5.2.3 Dependency Invalidation (DI)

The second result invalidation strategy, *dependency invalidation* (DI), removes outdated dependencies. Recall that dependencies correspond to emitted read effects and thus, in the case of ModF, indicate a dependency of a component on the value of an address in the global store. Removing dependencies ensures

```
1  // Assumes the existence of a cache for the sets C' returned by the
       intra-component analyses of different components, ℂ, initialised as
       ℂ := λα.∅ before the initial analysis, and the existence of a cache for
       the sets R' returned by the intra-component analyses of different
       components, ℝ, initialised as ℝ := λα.∅ before the initial analysis.
2  Function deleteComponent(β) is
3      foreach r ∈ ℝ(β) do D := D[r ↦ D(r) \ {β}]; // Delete dependencies.
4      // Remove β from all data structures.
5      V := V \ {β}; WL := WL \ {β}; ℝ := ℝ \ {β}; ℂ := ℂ \ {β};
6  end
7  while WL ≠ ∅ do
8      ... // Same as in Algorithm 5.
9      foreach u ∈ U' do WL := WL ∪ D(u);
10     if incremental update then
11         R := ℝ(α);
12         foreach r ∈ (R \ R') do D := D[r ↦ D(r) \ {α}];
13     end
14     C := ℂ(α);
15     ℂ := ℂ[α ↦ C']; // Update ℂ immediately to use the updated C'.
16     if incremental update and C \ C' ≠ ∅ then
17         reachable := ℂ(Main) ∪ {β | γ ∈ reachable ∧ β ∈ ℂ(γ)};
18         foreach β ∈ (V \ reachable) do deleteComponent(β);
19     end
20     ℝ := ℝ[α ↦ R']; // Both for component invalidation and dependency
       invalidation.
21 end
22 return (σ, V, D);
```

Algorithm 6: Inter extended with component invalidation (in blue) and dependency invalidation (in orange).

that components are not spuriously reanalysed. Consider, e.g., the program in Listing 5.2. Initially, the component read has a dependency on the address of the variable x, a_x. During the incremental update, the analysis of read will find a new dependency on a_y, whilst the dependency on a_x can be removed. This is visualised in Figure 5.2. Note that the store remains untouched and thus that a_x remains present in the store.

Algorithm 6 also extends Inter with dependency invalidation. The set of dependencies computed during the last analysis of every component α, R_α, is cached using the cache $\mathbb{R}$ (also used by component invalidation). After the (re-)analysis of a component α, Inter collects the computed dependencies, R'_α. It then fetches the dependencies computed during the *previous* analysis of α from $\mathbb{R}$ and computes the set of outdated dependencies which are then removed (Lines 11-12). Finally, as for component invalidation, $\mathbb{R}$ is updated (Line 20).

```
1 (define x 1)
2 (define y 2)
3 (define (write) (<change> (set! x 7) (set! y 7)))
4 (define (read) (<change> x y))
5 (read)
6 (write)
```

Listing 5.2: Example program with changing dependencies. Initially READ has a dependency on the address of variable `x`, a_x. In the new version of the program, READ solely has a dependency on a_y, the address of variable `y`.

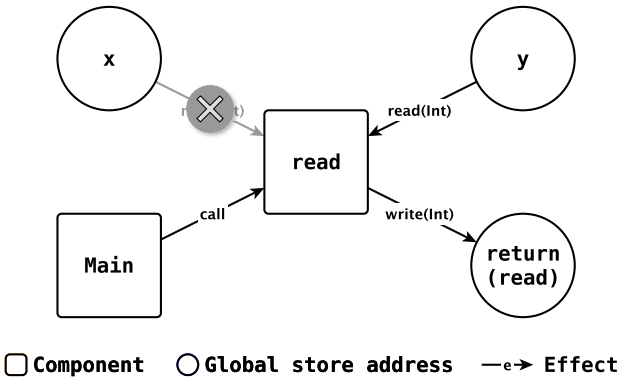


Figure 5.2: Partial ModF analysis graph for the program in Listing 5.2. After the program update, no read effect is emitted for the variable `x` any more. Hence, the dependency from READ on a_x can be removed.

5.2.4 Write Invalidation (WI)

Write invalidation (WI) aims to increase the precision of the abstract values in the store. It is motivated by the code in Listing 5.3. Variable `x` is changed from storing symbols to strings. A strong update would *overwrite* the abstract value `Symbol` by `String` in σ. A monotonic update instead joins the values together, resulting into the less precise value `{Symbol, String}`. Clearly, a strong update is desired.

The values in σ are part of an abstract domain, forming a complete lattice (see Section 2.2.1). Recall from Section 2.2.3 that the *higher* a value resides in the lattice, the less precise information it represents. Write invalidation aims to *lower* all values as much as possible by monitoring the values computed for every address in σ, and by lowering values that no longer correspond to the program's behaviour. We first describe the required monitoring.

```
(define (fromBool b)
  (if b
      (<change> 'aSymbol       "aString")
      (<change> 'anotherSymbol "anotherString")))
(define x (fromBool (some-complicated-predicate)))
(display x)
```

Listing 5.3: Example program. Initially, `x` only holds a symbol, whereas after the update it can only contain a string.

Provenance Tracking

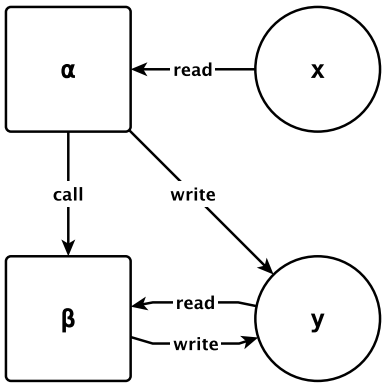
Values in the global store σ result from one or more writes, each monotonically updating the value. In this process, the analysis loses information w.r.t. the *constituents* and origins of the resulting values in the store. For example, when a component α writes `1` to the address a and β writes `-1` to the same address a, $\sigma(a)$ contains `{Int}`, without information about the values written by α and β, nor about which components wrote these values. We introduce *provenance tracking* to regain this information. For every component and address in $dom(\sigma)$, the analysis maintains the *contribution* of the component to the address, i.e., the join of all values written to the address during the analysis of the component. This requires intercepting *write* operations to the store but does not require modifications of the actual intra-component analysis Intra.

Consider the case in Figure 5.3: components α and β read and write two variables, `x` and `y`: both write `y`, α reads `x`, and β reads `y`. When α writes `Int` to `y` and β writes `Boolean` to `y`, σ holds join of these values, `{Int, Boolean}`, for `y`.

During the analysis of a component α, we track, for each written address a, the join of all values written to that address. We call this joined value the *contribution* of α to a, denoted $P_{a,\alpha}$. For every address, the contributions of all components are cached. We call this cache the *provenance* of the address, P_a. We define the *provenance value* of an address a as the join of all values in its provenance. For the example of Figure 5.3, this information is shown in (c).

Non-monotonic Store Updates

The intra-component analysis perform all updates monotonically. Inter thus has to restore precision after it has been lost. Provenance tracking enables write invalidation to perform non-monotonic updates to σ, improving its precision.



(a) Example analysis graph.

σ	
x	String
y	{Int, Boolean}

(b) The global store, σ.

$P_{a_y} : \{Int, Boolean\}$
$P_{a_y,\alpha} : Int$
$P_{a_y,\beta} : Boolean$

(c) The provenance and contributions for a_y.

Figure 5.3: Illustration of the interaction of intra-component analyses with two variables **x** and **y** and their values in σ. The corresponding provenance of and contributions to a_y are shown in (c).

This is possible when a previously written address is no longer written by a component, and when the contribution of a component to an address changes in a non-monotonic way.[1] The code for write invalidation is shown in Algorithm 7.

Outdated writes. The analysis of a component tracks all addresses written to. For every component α, Inter caches this set, W_α, using a cache $\mathbb{W}$. After the analysis of a component α, Inter collects the set of written addresses, W'_α, and computes the set containing all addresses previously written by the component that are no longer written (Line 30). Finally, the cache $\mathbb{W}$ is updated (Line 32).

When the contribution $P_{a,\alpha}$ of α to an address a is removed, its provenance value, no longer influenced by $P_{a,\alpha}$, is used as the new value for the address (Lines 3–4). If the provenance value equals the value at $\sigma(a)$, the value at this address does not change and deletion is completed. Else, the provenance value replaces the value σ, thereby effectively updating the store in a non-monotonic manner. All dependent components are scheduled for reanalysis (Line 6), allowing the new value to be taken into account during their reanalysis, possibly leading to further

[1] Conceptually, the first case corresponds to the second case for which the contribution of the component to an address has become $\bot$. We treat it separately since no write to the address is performed any more.

refinements of the result. When an address is no longer written by any component, all information in the analysis' data structures related to this address can be removed (Line 8).

More precise writes. After every intra-component analysis, INTER compares the contribution of the component for every written address to the corresponding contribution computed by the component's previous analysis. Based on this comparison, the value at the given address in σ may be updated, in which case all dependent components are added to the worklist (Line 34). The comparison may yield one of three possible results:

$P_{a,\alpha} = P'_{a,\alpha}$ The contribution of α to a does not change. The analysis did not compute new information and no information can be discarded (Line 14). The value at address a in the store does not change.

$P_{a,\alpha} \sqsubset P'_{a,\alpha}$ The new contribution of α to a subsumes the component's previous contribution to the address. The update is monotonic, for which no information can be discarded. The updated contribution is stored (Line 15) and a new value for a is computed (Line 17). As the update is monotonic, this value can be computed using a single join operation.

$P_{a,\alpha} \not\sqsubset P'_{a,\alpha}$ The contribution changes non-monotonically. In this case, the value for a in the store can possibly be updated non-monotonically, thereby improving its precision. The new value for a is computed by calculating its new provenance value (Line 17), taking into account the updated contribution $P'_{a,\alpha}$ (stored in $\mathbb{P}$ on Line 15).

In the second and third case, an updated value for a is computed. In the second case, this corresponds to a monotonic update, whereas in the third case, this corresponds to a non-monotonic update. However, in both cases there may not be an actual update of the value of a in σ as the new value computed for a (on Line 17) can be the same as the value already in σ. Only when the new value is different, the dependent components need to be scheduled for reanalysis.

Reinforcing Component Invalidation

Section 5.2.2 introduced component invalidation. However, component invalidation does not allow for the removal of information from σ: values written by removed components cannot be deleted, a limitation that can be remedied by combining component invalidation with write invalidation as follows. When a component α is removed, all addresses in the set $\mathbb{W}(\alpha)$ are treated as outdated

```
1  // Assumes the existence of a cache for the sets W' computed during the
       intra-component analyses of different components, 𝕎, initialised as
       𝕎 := λα.∅ before the initial analysis, and the existence of a cache ℙ,
       the provenance, initialised to ℙ := λa.(λα.⊥) before the initial
       analysis.
2  Function deleteContribution(α, a) is
3      ℙ := ℙ[a ↦ (ℙ(a) \ {α})];
4      v := ⨆_{β∈dom(ℙ(a))} ℙ(a)(β);
5      if v ≠ σ(a) then
6          WL := WL ∪ D(a);
7          // If an address is no longer written by any component, it is
               deleted. Otherwise, the store is updated.
8          if ℙ(a) = ∅ then σ := σ \ {a}; ℙ := ℙ \ {a}; D := D \ {a}; else σ := σ[a ↦ v];
9      end
10 end
11 // updateAddressIncremental compares the new contribution v' of α to a to
       the previous contribution v, and improves the store if possible.
12 Function updateAddressIncremental(α, a, v') is
13     v := ℙ(a)(α); // Previous contribution of α to a, P_{a,α}.
14     if v = v' then return false; // Identical contribution: no precision gain.
15     ℙ := ℙ[a ↦ (ℙ(a)[α ↦ v'])];
16     old := σ(a);
17     new := if v ⊑ v' then old ⊔ v' else ⨆_{β∈dom(ℙ(a))} ℙ(a)(β);
18     if old = new then return false;
19     σ := σ[a ↦ new]; // Update the store.
20     return true;
21 end
22 while WL ≠ ∅ do
23     ... // Same as in Algorithm 5.
24     // σ := σ'; // This line is now omitted.
25     ... // Same as in Algorithm 5.
26     // foreach u ∈ U' do WL := WL ∪ D(u); // This line is now omitted.
27     // W' computed during the intra-component analysis. W' is the set of
           all addresses written during the component's analysis (i.e.,
           U' ⊆ W').
28     if incremental update then
29         W := 𝕎(α);
30         foreach w ∈ (W \ W') do deleteContribution(α, w);
31     end
32     𝕎 := 𝕎[α ↦ W'];
33     // P computed during the intra-component analysis. P maps every written
           address to the join of all values written to it during the
           component's analysis.
34     foreach (a, v) ∈ P do if updateAddressIncremental(α, a, v) then WL := WL ∪ D(a);
35 end
36 return (σ, V, D);
```

Algorithm 7: Inter extended with write invalidation (in purple).

```
1 Function deleteComponent(β) is
2     foreach r ∈ ℝ(β) do D := D[r ↦ D(r) \ {β}]; // Delete dependencies.
3     // Remove β from all data structures.
4     V := V \ {β}; WL := WL \ {β}; ℝ := ℝ \ {β}; ℂ := ℂ \ {β};
5     W := 𝕎(β);
6     forall w ∈ W do deleteContribution(β, w);
7     𝕎 := 𝕎 \ {β};
8 end
```

Algorithm 8: Function `deleteComponent` of component invalidation reinforced with write invalidation (Lines 5–7 in purple).

writes as just described on page 97. This allows σ to become more precise, which may in turn invoke the analysis of other components. The updated code for component deletion is shown in Algorithm 8.

5.3 Formal Specification of the Approach

As in the previous chapter, we present a formal specification of our method. This formal specification builds on the formal specification of Section 4.3, which formalises the monotonic incrementalisation method presented in Chapter 4. As we treat the intra-component analysis Intra as a black box, we only need to update the formal specification of the inter-component analysis Inter. We present the formal extension to the inter-component analysis in two steps: first we formalise component invalidation and dependency invalidation (Section 5.3.1), before extending the formal specification further by adding write invalidation and the corresponding reinforcement of component invalidation (Section 5.3.2). In every step, we also show the updated formal specification of the collecting semantics, which depends on the inter-component analysis Inter.

5.3.1 Component Invalidation and Dependency Invalidation

Component invalidation and dependency invalidation each make use of a separate cache to store the set of called components resp. dependencies inferred during the component's last analysis. These caches, $\mathbb{C}$ and $\mathbb{R}$, and an explicit visited set V now become parameters of the inter-component analysis, which as a result thereof

now has the following signature:

$$\begin{aligned}\textsc{Inter}\colon\ & \overbrace{\mathcal{P}(Cmp)}^{WL} \times \overbrace{\mathcal{P}(Cmp)}^{V} \times \overbrace{(Addr \to \mathcal{P}(Cmp))}^{D} \times \overbrace{Store}^{\sigma} \times \overbrace{(Expr \to \mathcal{P}(Cmp))}^{T} \\ & \times \underbrace{\Pi}_{\pi} \times \underbrace{(Cmp \to \mathcal{P}(Cmp))}_{\mathbb{C}} \times \underbrace{(Cmp \to \mathcal{P}(Eff))}_{\mathbb{R}} \\ & \to \mathcal{P}(Cmp) \times (Addr \to \mathcal{P}(Cmp)) \times Store \times (Expr \to \mathcal{P}(Cmp)) \\ & \times (Cmp \to \mathcal{P}(Cmp)) \times (Cmp \to \mathcal{P}(Eff))\end{aligned}$$

For clarity, we again annotated the parameters with their role. The three extra parameters, V, $\mathbb{C}$, and $\mathbb{R}$, are passed to the inter-component analysis so that the visited set and caches can be used in the computation.

In what follows, we first give the entire formal specification of the inter-component analysis with component invalidation and with dependency invalidation, which is explained immediately afterwards. Thus, given the function signature above, we now define Inter as follows:

$$\textsc{Inter}\big(\varnothing, V, D, \sigma, T, \pi, \mathbb{C}, \mathbb{R}\big) = \big(V, D, \sigma, T, \mathbb{C}, \mathbb{R}\big) \tag{a}$$

$$\begin{aligned}&\textsc{Inter}\big(\{\alpha\} \uplus \alpha s, V, D, \sigma, T, \pi, \mathbb{C}, \mathbb{R}\big) = \\ &\quad \textsc{Inter}\Bigg(\Bigg[\alpha s \cup \bigcup_{\substack{\mathbf{c}(\beta)\in C' \\ \beta\notin \mathrm{dom}(\sigma)}} \beta \cup \bigcup_{\substack{\mathbf{w}(a)\in U' \\ \gamma\in D(a)}} \gamma\Bigg] \setminus C^-, && \text{(b)}\\ &\qquad \big(V \cup \{\alpha\}\big) \setminus C^-, && \text{(c)}\\ &\qquad \Big\{\Big(a, \big(\beta s \setminus \{\alpha\}\big) \setminus C^-\Big) \;\Big|\; (a, \beta s) \in D' \wedge \mathbf{r}(a) \in \big(\mathbb{R}(\alpha) \setminus R'\big)\Big\} && \text{(d)}\\ &\qquad\quad \dot\cup \Big\{\big(a, \beta s \setminus C^-\big) \;\big|\; (a, \beta s) \in D' \wedge \mathbf{r}(a) \notin \big(\mathbb{R}(\alpha) \setminus R'\big)\Big\}, && \text{(e)}\\ &\qquad \sigma', \\ &\qquad T \dot\cup \dot\bigcup_{e\in X} [e \mapsto \{\alpha\}], \\ &\qquad \pi, \\ &\qquad C^- \triangleleft \mathbb{C}[\alpha \mapsto C'], && \text{(f)}\\ &\qquad C^- \triangleleft \mathbb{R}[\alpha \mapsto R']\Bigg) && \text{(g)}\end{aligned}$$

where

$$(C', R', U', \sigma', X) = \textsc{Intra}(\alpha, \sigma, \pi)$$

$$\mathit{reachable}(\beta s) = \{\beta s\} \cup \bigcup_{\substack{\beta \in \beta s \\ \gamma \in \mathbb{C}[\alpha \mapsto C'](\beta)}} \gamma \tag{h}$$

$$C^- = V \setminus \mathit{lfp}(\mathit{reachable}(\textsc{Main})) \tag{i}$$

$$D' = D \cup \bigcup_{\mathbf{r}(a) \in R'} [a \mapsto \{\alpha\}]$$

and $\triangleleft$ is the domain anti-restriction operator, which is defined as follows:

$$A \triangleleft B = \{(a, c) \mid (a, c) \in B \wedge a \notin A\}$$

To avoid confusion, we define the precedence of this operator so that it has a lower precedence than the binding operator $[.]$ but a higher precedence than $\neq$ and $\in$.

The parts of the formal specification that are specific to component invalidation have been coloured blue and the parts specific to dependency invalidation have been coloured orange, matching the colours used in Algorithm 6. In the algorithm, the variables R and C are used. In the formal specification, they respectively correspond to $\mathbb{R}(\alpha)$ and $\mathbb{C}(\alpha)$. To match the names in Algorithm 6, the sets of effects returned by Intra are now called C', R' and U', in contrast to the formal specification of Section 4.3 where they are called C, R, and U.

Inter is still defined recursively, where every step of the recursion corresponds to the analysis of a component followed by invalidation. For some arguments of Inter, the set-builder notation is used. Here, the predicates correspond to conditions in the algorithm, e.g., in an `if` or a `foreach`. We now discuss the formal specification in more detail.

After the intra-component analysis, the set of components to be removed is computed and assigned to C^- on Line (i). These are the components not in the transitive closure of the updated $\mathbb{C}$ starting from Main, which is computed in (h). For every invalidated component, its dependencies are removed (shown in (d) and (e)), and the component is removed from the worklist (b), visited set (c), call cache (f) and dependency cache (g). To be able to compute the transitive closure of $\mathbb{C}$, the cache is updated after every intra-component analysis in (f).

The set of dependencies to be removed from D is the set $\mathbb{R}(\alpha) \setminus R'$, i.e., the set of read effects no longer emitted by the intra-component analysis of α. For all these dependencies, α is removed, which is formalised in (d); all other dependencies

remain untouched, as denoted in (e). Finally, to be able to perform this computation, the dependency cache $\mathbb{R}$ is updated after every intra-component analysis, which corresponds to (g) in our formal specification.

Up until now, we did not explicitly model the visited set V in the formal specification, but we relied on the return addresses present in the store to reconstruct a visited set in the first clause of the definition of Inter on Line (a). However, in the presence of component invalidation without write invalidation, this approach would result in a visited set containing too many components. The reason for this is that without write invalidation, the return addresses from deleted components cannot be deleted from the store. Hence, the visited set would contain deleted components as well. For this reason, we explicitly include a visited set V in this version of the formal specification, thereby ensuring that this set only contains non-invalidated components at the end of the analysis as shown in (c).

In line with Algorithm 6 on page 94, deleted components are not filtered out from T, the map from tracked expressions to sets of components. In practice, this is necessary to avoid spurious recomputations and to preserve precision in case there are multiple successive incremental updates. We omit this for brevity, focussing on the core aspects of the presented invalidation strategies. We also ignore the tests in the code to check whether an incremental update is taking place, as these are mere optimisations. Finally, to be complete, we pose that if an unmapped element of a map is looked up, the empty set or bottom is returned, whichever is applicable (e.g., $\mathbb{R}(\alpha)$ returns $\varnothing$ if α is has not explicitly been mapped in $\mathbb{R}$).

In the formal specification of the collecting semantics, we now take into account the extra arguments required by Inter and the extra values in the tuple it returns. The collecting semantics for the initial program can therefore be obtained by computing $\textsc{Inter}(\{\textsc{Main}\}, [], [], [], [], \textsc{Old}, [], [])$, and the incremental update of the analysis result is computed by $\textsc{Inter}(\bigcup\{T(e) \mid e \textit{ is updated}\}, V, D, \sigma, T, \textsc{New}, \mathbb{C}, \mathbb{R})$, where $(V, D, \sigma, T, \mathbb{C}, \mathbb{R})$ is the result of the initial analysis.

5.3.2 Write Invalidation

We now extend and adapt the formal specification of the inter-component analysis from the previous section to also incorporate write invalidation, shown in Algorithm 7, and reinforced component invalidation, shown in Algorithm 8.

In our formal specification of write invalidation, we ignore some of the branches and optimisations that are present in the pseudocode algorithms when they can safely be ignored. For example, in the formal specification of the updated store σ

and provenance cache $\mathbb{P}$, we can ignore the condition in the function *deleteContribution* which checks whether the value without the removed contribution differs from the value residing in the store (Algorithm 7, Line 5). However, we cannot ignore this condition in the formal specification of the worklist, as components only need to be added to the worklist when the values are different. Another example is the condition in the function *updateAddressIncremental* that checks whether the new contribution to an address subsumes the old contribution, and thus whether the new value for the address can be computed by a simple join of the contribution with the value in the store or whether all contributions need to be joined (Line 17). As this is just an optimisation, we only consider the else branch in the formal specification.[2]

Write invalidation makes use of two additional caches, i.e., the write cache $\mathbb{W}$ and the provenance cache $\mathbb{P}$. As before, these caches are added to the parameters and to the result of Inter. As we explicitly formalised the visited set V in the previous paragraph, we keep it although it is not strictly needed anymore in the presence of write invalidation. As a result, the incremental inter-component analysis with the three result invalidation strategies incorporated has the following signature:

$$\begin{aligned}
\textsc{Inter}\colon\ & \overbrace{\mathcal{P}(Cmp)}^{WL} \times \overbrace{\mathcal{P}(Cmp)}^{V} \times \overbrace{(Addr \to \mathcal{P}(Cmp))}^{D} \times \overbrace{Store}^{\sigma} \times \overbrace{(Expr \to \mathcal{P}(Cmp))}^{T} \\
& \times \underbrace{\Pi}_{\pi} \times \underbrace{(Cmp \to \mathcal{P}(Cmp))}_{\mathbb{C}} \times \underbrace{(Cmp \to \mathcal{P}(Eff))}_{\mathbb{R}} \times \underbrace{(Cmp \to \mathcal{P}(Addr))}_{\mathbb{W}} \\
& \times \underbrace{(Addr \to (Cmp \to Val))}_{\mathbb{P}} \\
& \to \mathcal{P}(Cmp) \times (Addr \to \mathcal{P}(Cmp)) \times Store \times (Expr \to \mathcal{P}(Cmp)) \\
& \times (Cmp \to \mathcal{P}(Cmp)) \times (Cmp \to \mathcal{P}(Eff)) \times (Cmp \to \mathcal{P}(Addr)) \\
& \times (Addr \to (Cmp \to Val))
\end{aligned}$$

in which we again annotated the parameters with their role. We now first show the entire formal specification of the inter-component analysis with all three invalidation strategies incorporated, and give the explanation immediately afterwards. Thus, we now define Inter as follows, colouring the parts of the formal specification specific to write invalidation purple and the parts for reinforced component invalidation blue:

$$\textsc{Inter}\big(\varnothing, V, D, \sigma, T, \pi, \mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{P}\big) = \big(V, D, \sigma, T, \mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{P}\big)$$

[2] One condition we did not ignore is the check to see whether the provenance of an address has become empty, meaning that an address is no longer written to by any component (Line 8). Although this condition only serves to avoid mappings to $\bot$ or empty sets, this changes the domain of the store and the provenance cache in the implementation. Therefore, we find it sufficiently important to formalise it as well.

$$
\begin{aligned}
&\textsc{Inter}\big(\{\alpha\} \uplus \alpha s, V, D, \sigma, T, \pi, \mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{P}\big) = \\
&\quad \textsc{Inter}\Bigg(\Bigg[\alpha s \cup \bigcup_{\substack{\mathbf{c}(\beta)\in C' \\ \beta\notin \mathrm{dom}(\sigma)}} \beta && \text{(a)}\\
&\qquad \cup \bigcup_{(a,v)\in P} \Big\{\gamma \;\Big|\; \gamma \in D(a) \wedge \sigma(a) \neq \bigsqcup_{\delta\in dom(\mathbb{P}(a)[\alpha\mapsto v])} \mathbb{P}(a)[\alpha \mapsto v](\delta)\Big\} && \text{(b)}\\
&\qquad \cup \bigcup_{a\in W^-} \Big\{\gamma \;\Big|\; \gamma \in D(a) \wedge \sigma(a) \neq \bigsqcup_{\delta\in dom(\mathbb{P}(a))\setminus\{\alpha\}} \mathbb{P}(a)(\delta)\Big\}\Bigg] \setminus C^-, && \text{(c)}\\
&\quad (V \cup \{\alpha\}) \setminus C^-,\\
&\quad \Big\{\Big(a, (\beta s \setminus \{\alpha\}) \setminus C^-\Big) \;\Big|\; (a,\beta s) \in D' \wedge \mathbf{r}(a) \in \big(\mathbb{R}(\alpha) \setminus R'\big)\Big\}\\
&\qquad \uplus \Big\{\big(a, \beta s \setminus C^-\big) \;\big|\; (a,\beta s) \in D' \wedge \mathbf{r}(a) \notin \big(\mathbb{R}(\alpha) \setminus R'\big)\Big\},\\
&\quad \Big\{\Big(a, \bigsqcup_{(\beta,v')\in C^- \lhd \mathbb{P}(a)[\alpha\mapsto v]} v'\Big) \;\Big|\; (a,v) \in P\Big\} && \text{(d)}\\
&\qquad \cup \Big\{\Big(a, \bigsqcup_{(\beta,v)\in (C^-\cup\{\alpha\})\lhd \mathbb{P}(a)} v\Big) \;\Big|\; a \in W^- \wedge (C^- \cup \{\alpha\}) \lhd \mathbb{P}(a) \neq \varnothing\Big\} && \text{(e)}\\
&\qquad \cup \Big\{\Big(a, \bigsqcup_{(\beta,v')\in C^- \lhd \mathbb{P}(a)} v'\Big) \;\Big|\; (a,v) \in W' \lhd \sigma \wedge a \notin W^- && \text{(f)}\\
&\qquad\qquad \wedge C^- \lhd \mathbb{P}(a) \neq \varnothing\Big\}, && \text{(g)}\\
&\quad T \cup \bigcup\!\!\!\!\cdot_{e\in X} [e \mapsto \{\alpha\}],\\
&\quad \pi,\\
&\quad C^- \lhd \mathbb{C}[\alpha \mapsto C'],\\
&\quad C^- \lhd \mathbb{R}[\alpha \mapsto R'],\\
&\quad C^- \lhd \mathbb{W}[\alpha \mapsto W'], && \text{(h)}\\
&\quad \Big\{\Big(a, C^- \lhd \mathbb{P}(a)[\alpha \mapsto v]\Big) \;\Big|\; (a,v) \in P\Big\} && \text{(i)}\\
&\qquad \cup \Big\{\big(a, (C^- \cup \{\alpha\}) \lhd \mathbb{P}(a)\big) \;\big|\; a \in W^- \wedge (C^- \cup \{\alpha\}) \lhd \mathbb{P}(a) \neq \varnothing\Big\} && \text{(j)}\\
&\qquad \cup \Big\{\big(a, C^- \lhd \beta s\big) \;\big|\; (a,\beta s) \in W' \lhd \mathbb{P} \wedge a \notin W^- \wedge C^- \lhd \beta s \neq \varnothing\Big\}\Bigg) && \text{(k)}
\end{aligned}
$$

where

$$
\begin{aligned}
(C', R', U', \sigma', X) &= \textsc{Intra}(\alpha, \sigma, \pi) \\
\mathit{reachable}(\beta s) &= \{\beta s\} \cup \bigcup_{\substack{\beta \in \beta s \\ \gamma \in \mathbf{C}[\alpha \mapsto C']}(\beta)} \gamma \\
C^- &= V \setminus \mathit{lfp}(\mathit{reachable}(\textsc{Main})) \\
D' &= D \uplus \bigcup_{\mathbf{r}(a) \in R'} [a \mapsto \{\alpha\}] \\
W' &= \mathit{dom}(P) \\
W^- &= \mathbb{W}(\alpha) \setminus W'
\end{aligned}
$$

and for which we assume the existence of a map $P : \mathit{Addr} \to \mathit{Val}$, which is computed during the intra-component analysis and which maps every written address to the join of all values written to that address during the intra-component analysis of α.

The definition of Inter is still recursive but has now become more complex. As write invalidation modifies the store, it does not only impact the store but also the worklist as impacted components need to be scheduled for reanalysis. In addition, the provenance cache $\mathbb{P}$ needs to be kept consistent. As addresses can be refined for two reasons (outdated writes and more precise writes), the updated worklist, store, and provenance cache are constructed from multiple subsets which each correspond to one situation. We now go into more detail on their construction.

As before, the worklist is extended with newly encountered components (shown in (a)). However, the components to be added to the work list are no longer computed using the dependencies of the addresses in U'. After all, more addresses may be updated due to write invalidation than the ones updated monotonically by the intra-component analysis. (This corresponds to the omission of Line 26 in Algorithm 7.) Instead, dependent components need to be added to the worklist for every written address that is updated (monotonically or non-monotonically) and for every address that is no longer written and which is therefore refined by write invalidation:

- In the first case, dependent components are added to the worklist when an address is written to by the intra-component analysis of α and causes the store to be updated (Line 34 of Algorithm 7). This corresponds to the set constructed on Line (b) in our formal specification. This set contains all components that depend on an address written by α and for which the value in the store changes due to the updated contribution of α.
- In the second case, dependent components are added to the worklist when the contribution of the component α to an address is deleted (outdated write,

see page 97) and causes the store to be updated (Line 6 of Algorithm 7). This corresponds to the set constructed on Line (c) in our formal specification. This set contains all components that depend on an address no longer written by α and for which the value in the store changes due to the retraction of the contribution of α.

The updating of the store is the core focus of write invalidation. In contrast to before, we no longer pass the updated store returned from the intra-component analysis, σ', to the recursive call of Inter as σ' has only been updated monotonically with regard to σ. (This corresponds to the omission of Line 24 in Algorithm 7.) Instead, the provenance is used as explained in Section 5.2.4:

- For all addresses that are written to by the analysed component, the new provenance value is computed using the new contribution of the component. This corresponds to (d) in our formal specification and to Line 19 of Algorithm 7 (ignoring the optimisation of Line 17).
- For all addresses that are no longer written by the analysed component, the new provenance value is computed, thereby omitting the contribution of the analysed component. This is formalised in (e) and corresponds to Line 8 of Algorithm 7.
- Finally, all other values in the store are retained, as formalised in (f).

To adhere to the condition on Line 8, in all three cases, extra conditions are placed on the addresses to be added to the updated store, as e.g., the condition in (g).

Similar to the construction of the updated store, the updated provenance cache is formalised in (i)–(k). Both formal specifications closely follow each other; their construction is entirely similar (consider e.g., the equal conditions when building the subsets forming the new store and provenance cache). In addition to the provenance cache, the write cache also needs to be updated. This is shown in (h) and corresponds to Line 32 of Algorithm 7.

Finally, the additions for the reinforcement of component invalidation with write invalidation, corresponding to Algorithm 8, are shown in blue in (d)–(k). The contributions from invalidated components are ignored when computing the updated store and information related to these components is deleted from the write cache and from the provenance cache.

In the final version of the collecting semantics, we again take into account the extra arguments required by Inter and the extra values in the tuple it returns. The collecting semantics for the initial program can now be obtained by computing

$\textsc{Inter}(\{\textsc{Main}\}, [], [], [], [], \textsc{Old}, [], [], [], [])$. The incremental update of the analysis result is computed by $\textsc{Inter}(\bigcup\{T(e) \mid e \textit{ is updated}\}, V, D, \sigma, T, \textsc{New}, \mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{P})$, where $(V, D, \sigma, T, \mathbb{C}, \mathbb{R}, \mathbb{W}, \mathbb{P})$ is the result of the initial analysis.

5.3.3 Memory Overhead

As in the previous chapter, we now use our formal specification to quantify the memory overhead of the three strategies presented in this chapter. To support these strategies, we introduced multiple data structures:

$$\begin{aligned}\mathbb{C} &: Cmp \to \mathcal{P}(Cmp)\\ \mathbb{R} &: Cmp \to \mathcal{P}(Eff)\\ \mathbb{W} &: Cmp \to \mathcal{P}(Addr)\\ \mathbb{P} &: Addr \to (Cmp \to Val)\end{aligned}$$

Based on their signature, we can again derive their worst-case memory overheads. For $\mathbb{C}$, we find that $\mathbb{C} \in \mathcal{O}(|Cmp|^2)$. In the case of $\mathbb{R}$, this is $\mathbb{R} \in \mathcal{O}(|Cmp| \times |Eff|)$ and thus $\mathbb{R} \in \mathcal{O}(|Cmp| \times |Addr|)$. For both $\mathbb{W}$ and $\mathbb{P}$ we find a worst-case complexity of $\mathcal{O}(|Cmp| \times |Addr|)$ (in $\mathbb{P}$ there is one value per combination of a component and an address so we do not add a term for it).

To get an indication about how much memory would be used in practice, we can use these worst-case complexities to compute an upper bound for space needed by the contents of the added data structures. Our implementation in MAF is written in Scala, so that the map data structures we use contain object references (pointers). The components and addresses pointed to also exist in the non-incremental analysis and thus require no extra memory; only the values stored in $\mathbb{P}$ may require extra memory as these values are not stored by the non-incremental analysis. As Scala runs on the JVM, pointers typically use 8 bytes of memory on a 64-bit architecture. Using the data on the number of addresses and components from the previous chapter (Tables 4.3 and 4.4), we estimate that the worst-case memory consumption of our method is in the range of (few tens of) megabytes and that therefore the memory consumption of our method does not become prohibitively high.

$\mathbb{R}$ contains the same information as D but in the reverse order. Should the memory footprint of the incremental analysis need to be reduced, only one of both can be kept, but the lookup of information will be slower. In addition, the information in the store σ can entirely be derived from $\mathbb{P}$ as well as the information in $\mathbb{W}$. At the cost of performance, σ and $\mathbb{W}$ could be removed as well.

5.4 Evaluation

We evaluated the presented strategies to answer the following research questions:

RQ 5.1. How well do the three invalidation strategies improve the precision of the analysis, both when applied individually and when applied in combination?
RQ 5.2. What is the impact of the invalidation strategies on the time needed to perform an incremental update?
RQ 5.3. How much does the incremental analysis reduce the analysis time compared to a full reanalysis of the program?

We tested soundness of the initial analysis and the incremental update experimentally (1) by ensuring that the analysis over-approximates multiple runs of a concrete interpreter [9, 152], and (2) by comparing the incremental analysis result to the result of a non-incremental analysis. We performed these tests for a thread-modular analysis for a concurrent μScheme, for a function-modular analysis, for all possible combinations of the invalidation strategies, and using both a constant-propagation lattice and a type lattice; no unsound results were encountered.

5.4.1 Experimental Design

Our evaluation uses a context-insensitive ModF analysis for μScheme, with a LIFO-ordered worklist and a product lattice[3]. As for Chapter 4, we implemented our contributions in the open-source MAF framework [152]. Our implementation can be found online at `https://github.com/softwarelanguageslab/maf` in the branch `incremental-experiments`.

Our evaluation is run on a 2015 Dell PowerEdge R730 with 2 Intel Xeon 2637 processors and 256GB of RAM, running OpenJDK 1.8.0_312 and Scala 3.1.0. The JVM was given a maximum of 32GB RAM, and all analyses used a timeout of 30 minutes.

To evaluate the precision of the incremental update (**RQ 5.1**), we inspect the store σ at the end of the analysis. For each address, we measure the precision of the incremental update by comparing its value to its counterpart in the store of a

[3]The lattice represents primitive values by their possible types, except booleans which are represented as their respective value when possible. Pointers are represented as sets of addresses (in $dom(\sigma)$); closures and primitives are represented using sets as well. A join of two values is the pointwise join of the corresponding elements of the product, where the join of two sets is their union.

full reanalysis. The proportion of addresses in the final store that contain values equally or less precise than the values obtained by a full reanalysis shows us how much precision can still be improved. We also compare to the store resulting from an incremental analysis without result invalidation. Here, the proportion of addresses in the final store that contain values equally or more precise than the values obtained by an incremental update without invalidation shows us how many addresses have an improved precision thanks to our strategies. We perform these comparisons for all possible combinations of the invalidation strategies.

To evaluate the performance of our strategies (**RQ 5.2** & **RQ 5.3**), we measure the time needed to (1) analyse the initial program, (2) fully analyse the updated program, and (3) perform the incremental update given a set of enabled strategies. For (1) and (2), no strategy is enabled; the analysis will not maintain the caches required by any strategy. For (3), the initial analysis initialises all caches used by the strategies. Each measurement is repeated 15 times preceded by a warm-up of up to 3 repetitions (every warm-up is limited to maximally 30 minutes). Garbage collection is forced prior to each analysis.

To investigate a trade-off between precision and performance, we compare the precision and performance of an incremental update using all strategies to (1) an (imprecise but fast) update without invalidation, (2) an update using only one or two strategies, and (3) a (precise but slow) full reanalysis.

Benchmarking Suites

Our evaluation uses two benchmarking suites.[4] Each benchmark program is a μScheme program containing real-world code, annotated with change expressions. As such, a benchmark corresponds to program changes.

Curated Benchmarks. We extended the set of benchmark programs used for the evaluation of the incremental ModF analysis in Chapter 4 (excluding the `machine-simulator` benchmark) and curated a suite of 32 programs to which we manually added changes resembling possible developer edits, shown in Table 5.2. The programs originate from different sources, e.g., courses at our university with programming exercises in Scheme, together with the solutions for solving particular exercises, and benchmarking suites used by other researchers. Example edits include changing representations of data structures (e.g., replacing

[4] In our online repository, which is available at `https://github.com/softwarelanguageslab/maf` in the branch `incremental-experiments`, the curated benchmarks can be found in the folders `/test/changes/scheme` and `/test/changes/scheme/reinforcingcycles`. The generated benchmarks can be found in the folder `/test/changes/scheme/generated`.

Benchmark	LOC	#Changes	Benchmark	LOC	#Changes
baseline	6	1	primtest	43	11
browse	164	1	cycleCreation	3	1
collatz	18	1	higher-order-paths1	4	2
fact	5	1	higher-order-paths2	4	1
fib-loop	15	1	implicit-paths	3	1
fib	5	2	ring-rotate	32	2
freeze	327	11	sat	16	4
gcipd	9	2	satCoarse	17	1
leval	379	11	satFine	13	3
matrix	617	3	satMiddle	16	3
mceval-dynamic	246	4	satRem	20	2
multiple-dwelling (coarse)	434	1	slip-0-to-1	123	6
multiple-dwelling (fine)	404	3	slip-1-to-2	117	3
nbody-processed	1252	10	slip-2-to-3	397	9
nboyer	636	2	tab-inc	317	3
peval	507	38	tab	307	3

Table 5.2: The curated benchmark suite, retrieved from various sources. For every benchmark, we list the lines of code as counted with `cloc` and the number of change expressions.

lists by vectors in `nbody-processed`, as shown in Listing 5.4), or updating a meta-interpreter (e.g., adding the ability to make variables immutable in `freeze` or making procedures dynamically scoped in `mceval-dynamic`). In programs like `slip-0-to-1`, `slip-1-to-2`, and `slip-2-to-3`, edits convert the program to a later version. A new abstraction is introduced and used throughout `peval`, which is shown in Listing 5.5. Some edits were constructed to be tricky for an incremental update to process accurately, as they trigger *cyclic reinforcement of lattice values* [130, 133] (see Section 5.4.2). Also, certain programs contain the same changes but use a different granularity of change expressions; this is e.g., the case for `multiple-dwelling (coarse)` and `multiple-dwelling (fine)`, and for `satFine`, `satMiddle`, and `satCoarse`. The runtimes of the initial analyses of the programs in the curated benchmarking suite vary from 0s to 117s.

Generated Benchmarks. We automatically generated 5 mutations for each of 190 programs, originating from various sources, obtaining 950 programs. We use a set of edit patterns of one or more change expressions that are inserted randomly, with a certain probability and at an arbitrary depth in the program. We consider the following patterns: expression deletion (7.5%), inserting a random sub-expression, possibly wrapped in a call to `display` (5%), swapping expressions (10%), wrapping an expression with a call to the identity function (7.5%), negating the predicate of an `if` (7.5%), and swapping the branches of an `if` (7.5%). A valid mutation has at least one edit, is unique, and does not lead to an error after

```
...
(define make-raw-pt
  (lambda (pt-1 pt-2 pt-3)
    ((<change> list vector) '<pt> pt-1 pt-2 pt-3)))
(define pt?
  (lambda (obj)
    (if ((<change> list? vector?) obj)
        (if (= ((<change> length vector-length) obj) 4)
            (eq? (<change> (car obj) (vector-ref obj 0)) '<pt>)
            #f)
        #f)))
(define pt-1
  (lambda (obj)
    (if (pt? obj)
        (void)
        (error 'pt-1 "~s is not a ~s" obj '<pt>))
    (<change> (cadr obj) (vector-ref obj 1))))
...
```

Listing 5.4: Excerpt from the `nbody-processed` benchmark program.

```
...
(define tagged-list? (<change> #f (lambda (l tag) (eq? (car l) tag))))
...
(define (beta-subst exp env)
  (define (bs exp)
    (cond ...
          ((or (<change> (eq? (car exp) 'let) (tagged-list? exp 'let))
               (<change> (eq? (car exp) 'letrec) (tagged-list? exp
                ↪ 'letrec)))
           (list (car exp)
                 (map (lambda (x) (list (car x) (bs (cadr x)))) (cadr
                  ↪ exp))
                 (bs (caddr exp))))
          ...
          (else
           (map bs exp))))
  (bs exp))
...
```

Listing 5.5: Excerpt from the `peval` benchmark program.

running it with a μScheme interpreter for one minute. The runtimes of the initial analyses of the programs in the generated benchmarking suite vary from 0s to 148s, most programs complete in under 10s.

An excerpt from a benchmark program with generated code changes is shown in Listing 5.6.

```
1  (letrec ((false #f)
2           (true #t)
3           (create-stack
4            (lambda (eq-fnct)
5              (let ((content ()))
6                (letrec (...
7                         (dispatch
8                          (lambda (m)
9                            (if (eq? m 'empty?) empty?
10                               (if (eq? m 'push) push
11                                   (if (eq? m 'pop) pop
12                                       (if (eq? m 'top) top
13                                           (if (eq? m 'is-in) is-in
14                                               (error "unknown request --
                                                     ↪ create-stack"
                                                     ↪ m)))))))))
15                 dispatch)))))
16    (let ((stack (create-stack =)))
17      (if ((stack 'empty?))
18          (if (<change> (begin ((stack 'push) 13) (not ((stack 'empty?))))
19                        (not (begin ((stack 'push) 13) (not ((stack
                            ↪ 'empty?))))))
20              (if ((stack 'is-in) 13)
21                  (<change>
22                   (if (= ((stack 'top)) 13)
23                       (begin
24                         ((stack 'push) 14)
25                         (= ((stack 'pop)) 14))
26                       #f)
27                   #f)
28                  (<change>
29                   #f
30                   (if (= ((stack 'top)) 13)
31                       (begin
32                         ((stack 'push) 14)
33                         (= ((stack 'pop)) 14))
34                       #f)))
35              #f)
36          #f)))
```

Listing 5.6: Excerpt from the R5RS_ad_stack-4 benchmark program. In this program, two change patterns were inserted: the predicate of an `if` expression is negated (Lines 18–19) and the branches of another `if` expression were swapped (Lines 21–34).

5.4.2 Precision Evaluation (RQ 5.1)

We evaluate the precision improvement caused by our invalidation strategies as follows. On every benchmark program, and for all possible configurations, we count the percentage of addresses in σ that are less precise than a full reanalysis. Figure 5.4 depicts the results of our precision evaluation. These allow us to (1) evaluate the precision improvement caused by the application of the presented strategies, and (2) to see whether additional opportunities for precision improvement are possible. As a precision improvement of σ can only be expected when write invalidation is enabled, we only show results for an incremental update without result invalidation, with write invalidation, and with all strategies enabled (where component invalidation is reinforced by write invalidation).

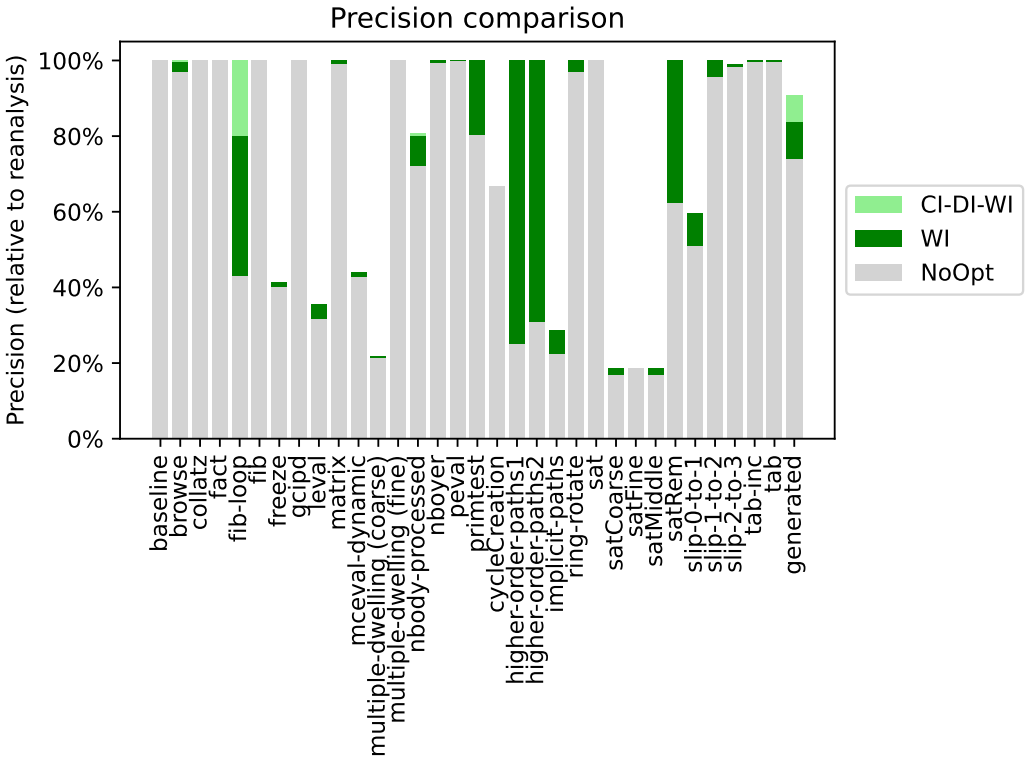


Figure 5.4: Precision of values in σ after an incremental update compared to a full reanalysis. Bars represent the percentage of addresses in σ of an incremental update whose values match a full reanalysis. In grey, precision of an incremental update without invalidation is shown. In dark green, the additional percentage of matching addresses due to write invalidation (WI) is shown. In light green, the further additional percentage of matching addresses using all strategies is shown. The rightmost bar shows the geometric mean of all benchmarks in the generated suite.

Configuration	Q1	Q2	Q3
No invalidation	73%	98%	100%
WI	97%	100%	100%
CI-DI-WI	100%	100%	100%

Table 5.3: Precision of values in σ after an incremental update compared to a full reanalysis. Percentages indicate the number of addresses in σ of an incremental update whose values match a full reanalysis. The table shows the quartiles of the distribution of these percentages among all programs in the generated suite, for an incremental analysis without invalidation, with write invalidation, and with all strategies.

Precision Improvements over Naive Incremental Analysis

For the curated suite, in some cases such as `higher-order-paths1`, we observe a big precision improvement. On other programs, the improvement remains minor. `fib-loop` shows that reinforcing component invalidation with write invalidation can lead to additional precision improvements. On benchmarks such as `browse` and `nbody-processed`, the benefit is smaller, though `browse` now reaches full precision. Unexpectedly, and only for `slip-0-to-1`, the reinforcement of component invalidation with write invalidation decreases the precision: when only write invalidation is enabled, 115 out of 193 values in σ (59.6%) are precise but when all strategies are enabled, just 112 out of 192 values (58.3%) are precise (this is not shown on the figure). The reason for this seems to be that, although sound, the obtained fixed-point depends on the analysis order of the components. On the generated suite, the number of imprecise values in the store is reduced by 15%pt.-20%pt. on average (geometric mean over all generated benchmarks): there is an improvement of about 10%pt. with write invalidation and an additional improvement of about 10%pt. when using all strategies.

Table 5.3 shows the quartiles of the distribution of the store's precision among all benchmarks in the generated suite for the same configurations. Without invalidation, more than 50% of all benchmark programs do not achieve full precision. However, using all strategies, the analysis reaches full precision on most benchmarks. The table shows the added benefit of reinforcing component invalidation with write invalidation.

Remaining Imprecision in the Analysis Result

Figure 5.4 also shows remaining possibilities for precision improvement. On 13 curated benchmarks for which the incremental update without invalidation did

not achieve full precision, the update with all strategies now does (indicated by a bar reaching 100%). However, on other benchmarks, more improvements remain possible.

The precision of σ influences the control flow explored by the analysis, and so the number of components and dependencies: precision gains due to write invalidation can lead to the invalidation of components and dependencies when all strategies are enabled. Of course, component invalidation and dependency invalidation can also be beneficial without write invalidation, though only write invalidation can propagate precision gains to other components through the store.

The imprecision in σ is worsened by our change representation: change expressions always require an old and new expression. For example, to introduce a new variable in a program, a placeholder value for the old program needs to be used, e.g., `#f` (false): `(define x (<change> #f 10))`. As this value will reside in σ and cannot be removed by the incremental update when write invalidation is not enabled, some values in σ may be artificially imprecise. However, imprecision still remains for some benchmarks when write invalidation is enabled. One reason we found is *cyclic reinforcement of lattice values* [130, 133], which arises when, due to the abstractions in the analysis, the computation of a value at an address is influenced by the value at that address itself, thereby influencing its own provenance.[5] Write invalidation cannot restore the precision of values in such a cycle. We also believe that this phenomenon causes the result to depend on the worklist order, e.g., when a value is refined before being introduced into a cycle, the cycle will be more precise than when refining would have taken place afterwards. This, however, does not have a negative impact on the soundness of our method.

Remark on Component Invalidation

For completeness' sake, we find it important to note that when component invalidation is used without write invalidation, the worklist order may affect the analysis result and thus the precision of the analysis. In this case, situations may arise where a component α that is to be deleted may cause a store change, depending on whether this component gets analysed first and thereby causes the store change or on whether another component β is analysed first and causes component invalidation to remove α. When write invalidation is enabled, this situation is avoided as the contributions of deleted components are removed. Since

[5] Some programs in our curated suite, such as `cycleCreation` and `implicit-paths`, have been created explicitly to contain this behaviour.

write invalidation has always been enabled for the results shown in Figure 5.4 and in Table 5.3, this situation thus could not arise and thus could not have affected our results.

As stated above, only in the presence of cyclic reinforcement of lattice values, there may be an influence of the worklist order in a limited number of cases. As we only noticed a minor effect for one benchmark program, we do not expect this matter to influence our results nor our answer for **RQ 5.1**.

> ***Answer RQ 5.1.*** Only write invalidation can improve the precision of the global value store, σ. Write invalidation significantly improves the precision of values for a limited number of curated benchmarks. Maximal precision is reached for 13 extra benchmarks when using all strategies, i.e., using reinforced component invalidation. For other curated benchmarks, a large percentage of addresses remains less precise. We also observe a large improvement on the generated suite, though several addresses still remain imprecise. Once again, the combination of component invalidation and write invalidation leads to a substantial additional precision improvement.

5.4.3 Performance w.r.t. No Invalidation (RQ 5.2)

Figure 5.5 shows the results of the performance evaluation for **RQ 5.2**. Times are shown relative to an incremental update without invalidation. Component invalidation and dependency invalidation do not cause a significant slowdown of the incremental analysis. A slowdown appears when using write invalidation, but, overall, the incremental update remains faster than a full reanalysis (see Section 5.4.4). This slowdown can be explained as follows. As write invalidation refines σ, updates may trigger the reanalysis of components, leading to further reanalyses and impacting performance. On the curated benchmarks, this increase in running time is more moderate for the combination of component invalidation and write invalidation.

Component invalidation and write invalidation combined reduce, in some cases, the analysis time as outdated components are not analysed anymore. Also, write invalidation may create more opportunities for component invalidation: when values become more refined, this may lead to more outdated components, which may in turn lead to an improvement of values in σ.

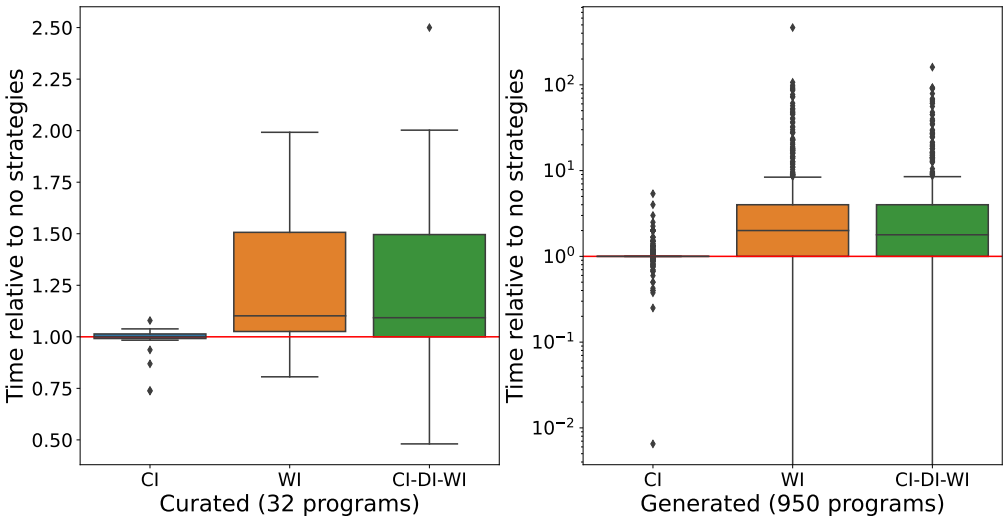


Figure 5.5: Analysis time of the incremental update relative to an incremental update without invalidation. Benchmarks for which the incremental update completed in 0ms have been omitted from the graphs because a relative time cannot be computed. The horizontal red line at 1 marks the normalised analysis time needed by an incremental update without invalidation. Data points below this line indicate execution times that are shorter than an incremental update without invalidation, whereas data points above this line indicate execution times that take longer than an incremental update without invalidation.

> ***Answer RQ 5.2.*** Component invalidation and dependency invalidation have no substantial negative impact on the running time of an incremental update. Only write invalidation causes a slowdown: as write invalidation regains precision, changes to σ may cause components to be scheduled for reanalysis.

5.4.4 Performance w.r.t. a Full Reanalysis (RQ 5.3)

Figure 5.6 shows the results of our performance evaluation for **RQ 5.3**. Times are shown relative to the time needed by a full reanalysis.

For the curated suite, overall, the incremental update is faster than a full reanalysis. The medians are consistently under 0.2, meaning that on more than half of the benchmark programs, the incremental update is more than 5 times faster. When both component invalidation and write invalidation are used, we see one outlier which corresponds to the `primtest` benchmark for which the running times are very low, meaning that there is no opportunity for the incremental analysis to gain time.

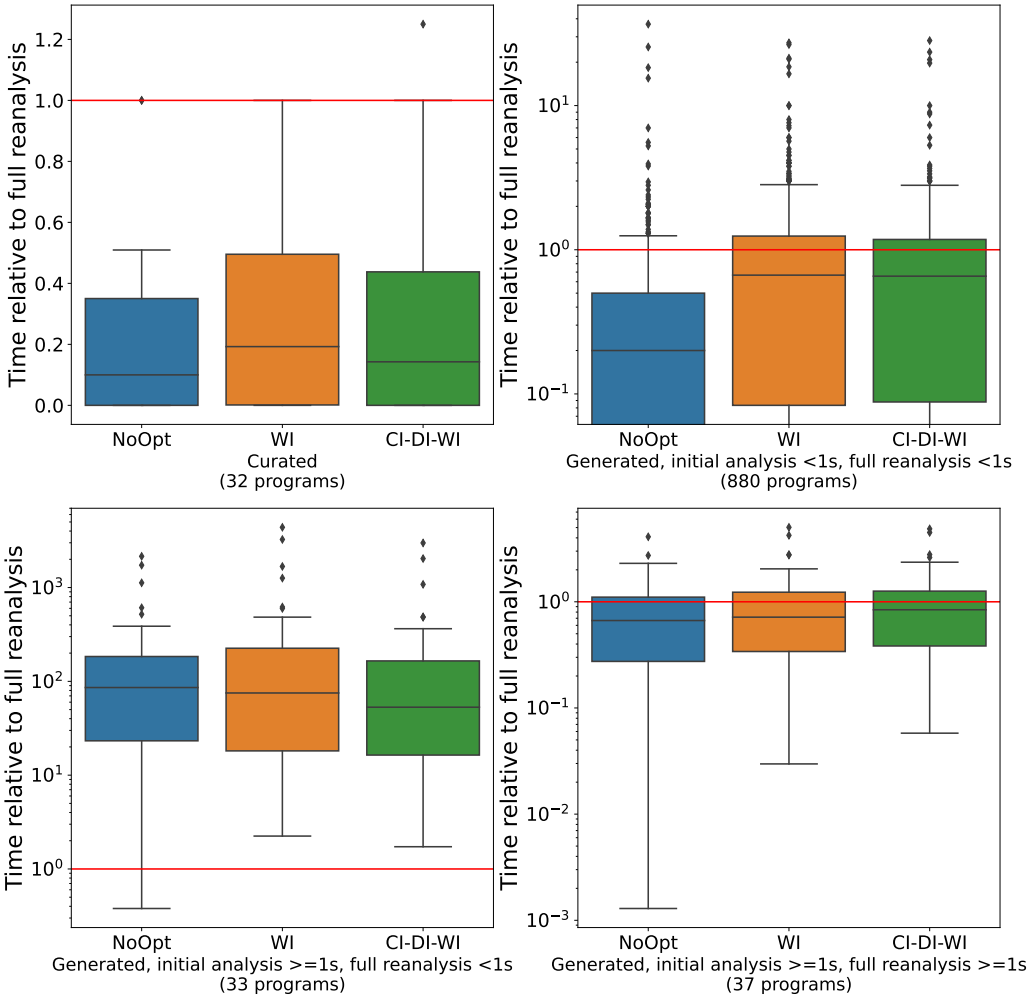


Figure 5.6: Analysis time of the incremental update relative to a full reanalysis. Benchmarks for which the full reanalysis completed in 0ms are counted but have been omitted from the graphs because a relative time cannot be computed. The horizontal red line at 1 marks the normalised analysis time needed by a full reanalysis of the updated program. Data points below this line indicate execution times that are shorter than a full reanalysis, whereas data points above this line indicate execution times that take longer than a full reanalysis.

The results of the generated suite are grouped based on the time taken by the initial analysis and the full reanalysis. The slowdown caused by write invalidation is most outspoken for short-running generated benchmark programs, where the overhead of the strategies may be relatively high. When both the initial analysis and full reanalysis complete in under a second, and when both analyses run a second or longer, overall, the incremental update remains faster than a full reanalysis.

Although write invalidation may cause minor slowdowns, the incremental update can be more than 10× faster compared to a full reanalysis. On programs that have an initial analysis taking a second or more but a shorter full reanalysis, the incremental update is slower: for almost all configurations, the incremental analysis takes at least as long as a full reanalysis for most benchmarks, with median slowdowns of up to 100 and outliers showing slowdowns larger than 1000. It is difficult to pinpoint the exact root cause for each performance difference. We list several possible reasons that may explain this behaviour:

- The change representation may cause less result reuse. In our implementation, change expressions cannot be placed at any program point. Some changes must be represented through coarse-grained change expressions. For example, to rename a function parameter, the change expression must wrap around the entire function definition, thus components corresponding to the function cannot be reused.
- The generated programs may contain too many changes, leading to a lot of impacted components: 25 programs have over 30 changes and 79 programs have over 20 changes. On almost half of the programs, more than 20% of the components is directly affected. This impacts the ability of the incremental analysis to efficiently update the result. As many components are affected, the incremental analysis has to reanalyse all these components. Moreover, lots of other components may be indirectly affected so that the analysis may not be able to benefit from its effect-driven modular design which bounds the impact of the changes.
- Changes may significantly alter program behaviour. 33 benchmarks had a long-running initial analysis and short-running full reanalysis. In these cases, the incremental analysis performs very poorly. It is possible that the randomly inserted changes prune away a lot of program functionality, leading to a very fast reanalysis, whereas an incremental update needs to propagate information deletion. Although we haven't verified the behaviour of all benchmarks individually, the reduced running time of the full analysis indicates that in these cases, an incremental update is inadequate due to the nature of the program changes.
- No dedicated worklist algorithm is used. Components may be scheduled for analysis due to newly inferred information or due to invalidation, but neither is prioritised. By intertwining recomputation by invalidation, information may be added or removed in an unspecified order; information may be removed that is later readded, or vice versa. We assume that the analysis of components in an unordered way may negatively impact the analysis performance.

To improve performance, future work should consider imposing an order on the worklist. It may also be useful to investigate heuristics to determine which changes would better be processed by a full reanalysis, e.g., when a program update leads to a big removal of program functionality.

> ***Answer RQ 5.3.*** On the curated suite, on the short-running generated benchmarks, and on the generated benchmarks with a long-running initial analysis and reanalysis, overall, the incremental update is faster than a full reanalysis. Yet, on the generated benchmarks with a long-running initial analysis but with a short-running full reanalysis, almost all incremental updates are slower. The nature of the changes may be to blame for this: a very fast reanalysis may indicate a serious reduction in program behaviour, in which case the incremental update has to invalidate many parts of the result, causing high relative runtimes.

5.5 Conclusion

This chapter presented three complementary invalidation strategies to improve the precision and performance of the method to incremental effect-driven static analysis presented in Chapter 4. These strategies are built on top of an invalidation principle that interleaves the reanalysis of components with invalidation. The first strategy, component invalidation, removes outdated components and their dependencies, and, when combined with write invalidation, can also improve the precision of the values in the store σ. The second strategy, dependency invalidation, removes outdated dependencies. Finally, the third strategy, write invalidation, uses provenance tracking to retract and replace outdated contributions from components to the global value store σ, enabling non-monotonic updates.

We tested our strategies for unsoundness and evaluated their precision and performance empirically on real-world programs using a small suite of 32 programs with possible developer edits and a large corpus 950 of programs with generated edits. Our strategies allow the incremental analysis to reach the same result as a full reanalysis on 13 more programs in the curated suite in comparison to when none of the proposed strategies is used. On other programs, the precision loss is reduced, yet the results did not match the precision of a full reanalysis. For the generated suite, using all strategies, on average, the number of less precise addresses in σ is reduced from 30% to about 10%. The best improvements were realised by the combination of write invalidation with component invalidation.

Performance-wise, overall, the incremental analysis scores well. We did find some benchmarks with particular program changes for which the incremental update proved to be slower than a full reanalysis, e.g., in 33 of the 950 programs in the generated suite where the changes removed a big part of a program's functionality. Future work should include handling cyclic reinforcement of lattice values, stratifying the worklist of the analyses, and investigating heuristics for triggering a full reanalysis rather than an incremental update.

Method to incrementalise an effect-driven analysis (part 2).

To improve the precision of an effect-driven analysis incrementalised according to the method presented in Chapter 4, take the following steps:

- Intercept calls to the global store of the analysis and keep track of all values written by the components. For each address, store the contribution of every component that has written to it.
- After every intra-component analysis, collect the set of generated effects and use them to determine the outdated parts of the analysis result.
  - Determine which components and dependencies are outdated and remove them. For every deleted component, remove its dependencies and retract its contributions to the global store.
  - For every address that is no longer written by the intra-component analysis, remove the contribution of the component.
  - For every address that is written by the intra-component analysis, compute the new value based on the new contribution of the component and the provenance.

  If the store is updated while executing the above steps, add the dependent components to the worklist of the analysis.
- Ensure that all caches are updated after the invalidation step.

6

Conclusion

Static analysis is a widely-used technique to ensure the quality of software and the absence of certain bug types prior to the execution of a program. However, complex static analyses on large code bases may take a long time to run, certainly when a high precision is required. This may make such analyses difficult to fit into the development process. To increase the usability of analyses and the fix rate of defects, developers must receive analysis feedback fast, preferably as part of their build or within their IDE.

To speed up static analysis in the presence of program updates, incremental analyses can be employed. Upon a program change, such analyses reuse and update the previous analysis result, thereby avoiding needless recomputations and aiming to produce the analysis result faster. In the literature, bespoke incremental analyses are tailored to a specific purpose and application domain. Incremental analysis frameworks provide incrementalisation for analyses specified within the framework. In this dissertation, we took a different approach and presented a method to incrementalise existing effect-driven static analyses.

This chapter summarises the dissertation, recapitulates our contributions, and presents possible directions for future research on this topic.

6.1 Summary of the Dissertation

Chapter 1 presented the context of our research and introduced the need for incremental static analysis.

In Chapter 2, we covered supporting material. First, we introduced the broad, general principles of static analysis and approximation. Second, we zoomed in on lattices as a means to represent abstract values in a static analysis. Third, we covered effect-driven static analysis, the analysis design for which we provide an incrementalisation method in this dissertation, and presented a formal specification of and a visual representation for this analysis design.

Chapter 3 surveyed the state of the art in incremental static program analysis. First, we introduced the concept of incremental static analysis as an analysis technique that reuses and updates results when changes are made to the analysed program. Next, we gave an overview of the change representations used in the literature. We divided the related work on incremental static analysis into two main categories – bespoke incremental analyses, which are designed for a specific purpose, and incremental analysis frameworks, which take care of the incrementalisation of analyses specified in the framework – and discussed the literature for both. We found that a gap in the literature on incremental static analysis is a step-by-step incrementalisation method, applicable to a broad class of existing analyses, which precludes the need to redevelop an analysis from scratch to render it incremental.

In Chapter 4, we first introduced change expressions, which we use to represent program changes. We then presented a monotonic approach to render effect-driven analyses incremental. The presented method consists of two steps. First, a change-impact calculation infers which components have been impacted directly by the changes and adds them to the worklist. Then, the analysis result can be updated by restarting the fixed-point computation of the inter-component analysis. Due to the effect-driven nature of the analysis, all directly and indirectly affected components are reanalysed, thereby propagating the changes through all parts of the analysis result. However, the result is updated monotonically, meaning that outdated parts of the result cannot be invalidated and, therefore, imprecision may arise in the result.

To improve the precision of the resulting incremental analysis, Chapter 5 introduced three complementary strategies for result invalidation: component invalidation, dependency invalidation, and write invalidation. Each strategy refines one part of the analysis result: the set of components, the dependencies, and the contents of the global value store σ. At the heart of these strategies lies the principle of interleaving the invalidation and the recomputation phases of the incremental analysis. This allows maintaining a black-box view of components and avoids over-approximation in the invalidation phase, ensuring the broad applicability of our approach and allowing as many parts of the analysis result to be reused by an incremental update as possible.

6.2 Recapitulation of the Contributions

In this dissertation, we made several contributions to the state of the art in incremental static program analysis.

In Chapter 4, we presented a novel method to incrementalise effect-driven analyses. Our approach is based on computational dependencies within the program, which are reified by the analysis. The division of the program into modules and of the analysis into components enables bounding the impact of changes to the parts of the analysis that are affected. We kept our approach as general as possible by treating the actual analysis and the components under analysis as black boxes. This allowed, for example, applying our method to both a function-modular and a thread-modular effect-driven analysis.

In Chapter 5, we presented three complementary strategies to improve the precision of the resulting incremental analysis. In contrast to many other incremental static analyses, the analysis interleaves the invalidation phase with the recomputation phase and thereby allows us to uphold the black-box view of components.

We presented a formal specification of our method and evaluated it for soundness, precision, and performance in different settings. To evaluate the resulting incremental analysis, we compiled two sets of programs with change expressions: a smaller curated suite containing programs with real-world changes and a bigger suite containing programs with generated changes. Our results show that our method is sound, and that, overall, the resulting incremental analysis outperforms a from-scratch analysis although, only in the presence of cyclic reinforcement of lattice values, imprecision may still occur in the analysis result.

6.2.1 Recipe for the Incrementalisation of Effect-driven Static Analyses

We now summarise the incrementalisation method for effect-driven static analyses presented in this dissertation. The following steps need to be taken to incrementalise an effect-driven analysis using our method:

- Add expression tracking to the intra-component analysis to track which expressions are encountered during the analysis of every component. For every expression, store the components for which the intra-component analysis encountered the expression (see Section 4.2.1[1]).
- Intercept write calls to the global analysis state, i.e., to the global store, during the intra-component analysis. For every written address, keep track of the join of all values written during the intra-component analysis.
- After every intra-component analysis, collect the set of generated effects and use them to determine the outdated parts of the analysis result as follows:
  - If some call/generation effects are no longer inferred, compute the set of live components by computing the transitive closure of the component graph starting from Main. Remove all components not in this set as well as their dependencies (see Section 5.2.2). In addition, retract the contributions from the deleted components to the global store (see Section 5.2.4, page 98). If the store is updated, add the dependent components to the worklist.
  - For every read effect no longer inferred, remove the corresponding dependency (see Section 5.2.3).
  - For every address that is no longer written by the intra-component analysis, remove the contribution of the component (see Section 5.2.4). If this updates the store, add the dependent components to the worklist.
  - For every address that is written by the intra-component analysis, compute the new value for the address based on the tracked written values and the provenance (see Section 5.2.4). Again, if this updates the store, add the dependent components to the worklist.
- Ensure that all caches are updated after the invalidation step.

Once the analysis has been incrementalised, the following steps are to be taken to perform an incremental update (see Sections 4.2.1 and 4.2.2):

1. Determine the set of expressions that are affected by the source code changes. Add all components for which the intra-component analysis encountered an affected expression to the worklist of the analysis. Depending on the contents of the analysis data structures, update these data structures first if necessary, e.g., to account for new source code locations. (In this dissertation, we avoided this necessity by the use of change expressions.)
2. Restart the fixed-point computation of the inter-component analysis.

[1] As mentioned in this section, for some types of effect-driven analysis, e.g., for a ModF analysis, the affected components can also be inferred lexically from the source code. In this case, the tracking of expressions is not needed.

6.3 Limitations and Future Work

In this section, we put forward some possible directions for future work on our incrementalisation method.

6.3.1 Applying the Method to Other Analyses

The method to incrementalisation which we introduced in this dissertation is focussed on effect-driven analyses. One of the key points of our method is its generality, as we kept a black-box view of components and ensured that the method does not rely on a specific client analysis, lattice, or context-sensitivity. In Chapter 4, we applied it to both a function-modular and a thread-modular analysis; in Chapter 5, we focused on a function-modular analysis alone. Consequently, we think a first interesting avenue for future work is the application of our method to multiple different effect-driven analyses, allowing for a comparative study on the impact of module granularities, for example.

Another possibility is to render the intra-component analysis incremental as well. In earlier work [144], we already incrementalised an AAM-based intra-component analysis that was used by a thread-modular effect-driven analysis. However, in this work, the inter-component analysis remained non-incremental. Future work could hence explore the influence of the use of incremental intra-component analyses in combination with the incremental inter-component analysis presented in this dissertation.

Finally, we think that it would also be interesting to see how well the method suits several other types of analyses. Applying (and adapting) our method to different types of analyses and in different settings can be useful to demonstrate its applicability and generality in practice, although the performance and precision of the resulting incremental analyses in these settings need to be investigated for every situation. Moreover, this also yields a concrete comparison between the incremental analyses resulting from the application of our method and other incremental analyses. In this sense, our incrementalisation method could present an alternative to the development of bespoke incremental analyses for a broader range of analyses, thereby reducing the development effort for incremental analyses. The remainder of this paragraph discusses two possible domains to which our method may be (adapted and) applied to demonstrate its generality.

The original motivation for the effect-driven analysis design is to avoid the presence of a visited set in machine-based analyses which needs to be traversed for every

state that is removed from the worklist [96]. However, in practice, effect-driven analyses require the traversal of the set of visited components to determine whether a discovered component, indicated by a call effect in ModF, needs to be added to the worklist. In this sense, effect-driven analyses can be seen as traditional machine-based analyses, such as e.g., *Abstracting Abstract Machines* (AAM) [52, 153, 154] and *Abstracting Abstract Control* (AAC) [51] analyses, with global-store widening, larger states, and where states are selectively added to the worklist upon a change of the global store. The latter contrasts with AAM and AAC analyses, where states may be revisited due to non-relevant changes in the store. Also, since states are larger in effect-driven analyses, the set of visited components is smaller than the visited set in AAM and AAC analyses. Thus, as effect-driven analyses are related to AAM and AAC analyses, we think it would be interesting to investigate (1) whether our method scales well to an effect-driven analysis with fine-grained states, like the states of an AAM analysis, and (2) whether our method can be applied directly to AAM and AAC analyses without global-store widening. In the latter case, we exclude global-store widening as AAM analyses do not reify the computational dependencies related to the use of the store; only the discovery of new states is explicit. This, however, creates a different setting for the application of our method, with many small states and no read or write effects. In this case, the techniques of dependency invalidation and write invalidation would therefore become obsolete.

In Section 2.3, we defined a modular static analysis as an analysis that is decomposed into the analysis of possibly inter-dependent parts called components, which are analysed separately. When dependencies between components are circular, the reanalysis of components may be needed. In their seminal paper on modular static analysis, however, Cousot and Cousot [30] adhere to a more strict definition of modularity, that is essentially compositionality, where circular dependencies between components must be broken and for which methods are presented in their work. For this reason, we think that an interesting direction for future work may be to investigate how well our incrementalisation method can be applied to (other) modular analyses, where no circular dependencies between components arise and where the structure of a possible global analysis state may differ.

6.3.2 More Flexible Change Representation

To represent changes, we introduced change expressions in Section 4.1. Although this change representation was sufficient for our study of incremental static analysis, we found that change expressions provided no flexible means to represent

changes. First, change expressions could not be placed everywhere. This hampered result reuse, as e.g., the addition of a parameter to a function would need to be encoded using an entirely new function definition, meaning that the corresponding component(s) could not be reused by an incremental function-modular analysis. Second, change expressions always require an old and a new expression. When no old or new expression was available, e.g., upon the addition of an expression, the expressions `#f` or `()` were used to indicate the absence of an expression.

Engineering wise, the benefit of change expressions is that both program versions are stored as one, meaning that the internal data structures of the analysis do not need updating. Although we argued that this does not have an impact on our results, we think that a different change representation would facilitate a more extensive evaluation of the presented method: another change representation may not only approach a real-life operation setting for incremental analyses (e.g., an IDE or continuous-integration system) more closely and, ideally, allow easy integration; it could also facilitate the curation of benchmarks to support the evaluation for different settings and allow the representation of some fine-grained changes that were not easily expressible using change expressions.

6.3.3 Handling Cyclic Reinforcement of Lattice Values

Even when applying the three strategies for precision recovery introduced in Chapter 5, incremental updates may still lose precision. Although the precision loss may be acceptable for a single incremental update, successive incremental updates may exacerbate the precision loss, leading to a degraded precision and usability.

In Section 5.4.2, we pinpointed the cause of this precision loss to be *cyclic reinforcement of lattice values* [130, 133], a situation in which the computation of an abstract value in the analysis is influenced by the abstract value itself, directly or indirectly. Such a situation can arise due to the abstractions made by the analysis, which allow circularities to arise.

Listing 6.1 shows a very simple program that causes cyclic reinforcement of lattice values in the analysis. It consists of a function that subtracts one from its argument, using assignment, and returns the new value. The fact that the function uses assignment is critical, as it affects the data flow (and thus the information flow [31, 147]) within the analysis as follows. When the function `f` is called, the abstract value of the literal `10` is bound to the variable `n` and therefore stored in the global store σ at the address of `n`. Upon the assignment of `n`, the value stored at the address of `n` is read from the store, 1 is subtracted, and the resulting value

```
1 (define (f n)
2   (set! n (- n 1))
3   n)
4 (f 10)
```

Listing 6.1: Simple example program causing cyclic reinforcement of lattice values in the analysis.

is written to the store at the address of `n`, where the original value was read from. Hence, a value flow is created from (the value at) the address of `n` to itself, that is, the flow is now cyclic: the value at the address of `n` is computed using the value at that address itself! Afterwards, the value at the address of `n` is assigned to the return value of `f` after which it gets assigned to the return value of `Main`.

Due to this cyclicity, write invalidation cannot remove reinforcing cycles as the intra-component analysis remains monotonic. Suppose a type lattice is used. Continuing the previous example, two components contributed to the value of `n`, that is, both Main and `f` wrote `Int` to the address. Assume the program is now updated and `f` is now called with `10.5`. In this case, the contribution of Main is updated in a non-monotonic manner as it becomes `Real`. However, although ideally we expect the value of `n` to become `Real`, this non-monotonic update results in write invalidation generating the value `{Int, Real}` for `n`, as the contribution of `f` remains `Int`. Even though the only original source of the value `Int` has disappeared, this value cannot be removed due to the cyclic information flow within the analysis.

We have, however, already started research on the invalidation of reinforcing cycles. Next, we briefly describe this preliminary, unpublished research, and its challenges, on a high level. Although our approach to handle reinforcing cycles has not been finalised, we have already obtained promising first results. Yet, the finalisation of this approach remains future work.

To detect where cyclic information flows are present within the analysis, the information flows need to be tracked. We track the flows of information between values in the store, to determine how the values at certain addresses in the store influence each other. This data constitutes a graph, the information flow graph, where the nodes are the addresses in the store and the edges (transitively) indicate that the value at an address depends on the value at another address. This graph can then be checked for cycles, or, more precisely, for strongly-connected components (SCCs), which indicate cyclic reinforcement of lattice values. After every intra-component analysis, the strongly-connected components in the information flow graph can be computed, and it can be established which SCCs need to be refined.

We refer to these SCCs of addresses as *strongly-connected addresses*, or SCAs for short: the value of every address in an SCA influences the value of every other address within the SCA.

Unfortunately, to accurately compute the information flow graph, the black-box view of components can no longer be held: to compute the information flow graph, the intra-component analysis may need to be modified to trace how values influence each other, if this information flow is not already computed as part of the analysis. In our preliminary work, we have managed to do this with minor modifications. For this, we rely on the labelling of values with the set of addresses they are influenced by. When a value is read from the store, it is labelled with the address it is read from. When a value is written to the store, its labels are removed and stored separately, indicating that the written address was influenced by the addresses in the labels. Labelling the values should not cause any updates to the actual semantics of the analysis. However, some changes are required, taking into account the differences between explicit and implicit information flow [31, 147]:

- *Explicit information flow* or data dependence can be traced by propagating the labels in the semantics during value computation. If an operation is executed, e.g., an addition of two numbers, the result carries the union of the labels of all arguments. If two values are joined by the analysis, then so are their labels.
- *Implicit information flow* or control dependence is caused by branching and dynamic function calls. For example, the execution of a conditional branch is dependent on the condition, although there is no explicit information flow. To this end, the labels of the condition need to be taken into account during the analysis of the branch and in the result of the conditional.
- In addition, it needs to be traced how literal values and data structures in the program influence the values in the store. Although literal values are lexical elements of the program text, their corresponding value may end up in the store in addresses that are part of an SCA.

In prior work, we already presented the computation of intra-component information flow along these lines but in a non-incremental setting [147].

An SCA needs to be refined if one of three conditions is met:

- an *externally incoming value*, i.e., a value at an address in the store that is not part of the SCA but from which there is a flow of information to an address in the SCA, is updated non-monotonically;
- a value in the store is no longer externally incoming or the SCA is (partially) broken;

- a literal value is no longer used to compute the value for an address in the SCA.

To refine an SCA, the values of the addresses within the SCA don't need to be set all the way to bottom. This would remove all information at the addresses in the SCA even though part of the information may come from outside the SCA. Moreover, this may cause a big invalidation pass if the intra-component analysis halts when bottom is encountered (as is the case in our implementation). A better approach is to remove information only if the information is influenced/affected by the SCA itself. The provenance can be used to infer which writes to the addresses in the SCA were not influenced by the values in the SCA itself, thereby allowing the reuse of more results. After all, for every address, the analysis knows (1) the contribution per component, and (2) the information flows inferred by the analysis of the component. To refine an SCA, contributions of components to addresses within the SCA can be removed if they were affected by information in the SCA itself and retained otherwise. In this case, not only dependent components need to be scheduled for reanalysis, but possibly also all components that wrote to the refined addresses (as their contribution may have been removed and needs to be replaced). Note that the computation and refinement of the SCAs needs to happen after every intra-component analysis: values flowing into the SCAs may be refined, and SCAs may be formed or broken up.

Our preliminary results show that this approach leads to full precision on almost all of the 982 benchmark programs from Chapter 5; on less than 10 benchmarks, the analysis remains imprecise. We identified that imprecision still occurs in programs in which the program changes introduce an infinite loop in the program under analysis. In this case, a cycle is created but not detected. The reason for this is to be investigated further. We hypothesise that non-termination sensitive control dependence [106] may be required to achieve full precision. Yet, our preliminary solution still needs further elaboration, as well as conceptual refinement and validation.

Although this preliminary solution significantly improves the precision of our incrementalisation, even when full precision is achieved, several questions remain to be investigated, such as the following:

- How does the detection and refinement of SCAs impact the performance of the analysis?
- Are there possibilities to mitigate an eventual performance hit?

The refinement of an SCA may cause multiple components to be scheduled for reanalysis, both reading and writing components. This may impact the performance of the incremental analysis. Moreover, components are analysed in an order

determined by the worklist algorithm, such as last in first out (LIFO), first in first out (FIFO), or a random order. However, a smart scheduling algorithm that takes into account the reason for which components have been added to the worklist as well as their dependencies may be more optimal.

In addition, the computations of SCAs themselves also needs to be efficient. For this, incremental graph algorithms may need to be applied, as it is computationally expensive to run e.g., Tarjan's SCC algorithm [138] in full after every intra-component analysis given the many information flows there are between the addresses in the store, even in small programs.

6.3.4 Improved Worklist Algorithm

Currently, an effect-driven analysis incrementalised using the method presented in this dissertation uses the worklist algorithm of the original, non-incrementalised analysis. We already hinted in the previous section that we assume that this may not always be optimal when cyclic reinforcement of lattice values is handled. The same may be true even without the handling of cyclic reinforcement of lattice values. We think it would be worthwhile to investigate whether a custom worklist algorithm is beneficial for derived incremental analyses. Such a "smart" scheduling algorithm could take into account the following, among other things:

- The reason for which a component has been added to the worklist, e.g., due to invalidation or recomputation. A smart worklist algorithm may e.g., prioritise components scheduled for reanalysis due to invalidation. This prioritises invalidation over recomputation and may avoid invalidations late during the incremental update.
- The number of dependencies of a component or the number of addresses a component writes to. For example, components with many dependencies could be scheduled later as they may be influenced by many other components, whereas components that write to many addresses could be scheduled early as they may influence many other components.
- An ordering of the components according to their dependencies. Components that depend on a certain address could be scheduled after components that write to that address.
- A fallback algorithm to order components that have the same priority according to the other criteria, such as the original worklist algorithm of the non-incremental effect-driven analysis, for example.

In previous work [127, 151], we have already performed a comparative study on worklist algorithms for a parallel effect-driven modular static analysis. However,

the setting of incremental updates performed by an incremental static analysis is very different from the non-incremental and parallel context in which our previous research was conducted, for which the results may not be transferable and additional research is needed on this topic in the setting of our method.

6.3.5 Heuristics for Performance Improvement

The goal of an incremental analysis is to outperform a full reanalysis of a program upon a program update. To this end, we have proposed a method that starts from the previous analysis result and tries to extend and/or refine it, according to the program changes. However, arguably, the performance gain that is to be expected does not only depend on the quality of the incrementalisation technique, but also heavily depends on the exact program and changes at hand; we believe that for some types of changes, an incremental analysis will always be slower unless it can detect these situations and act accordingly. For example, a program update may cause a significant alteration of program behaviour. In this case, it may not be possible for an incremental analysis to outperform a full reanalysis.

To handle situations in which the execution of an incremental update may not be beneficial, it may be expedient to develop heuristics that assess the potential benefits of an incremental update and suggest to perform a full program analysis in case an incremental update may not be advantageous. Such heuristics could take into account e.g., the possible impact of the changes based on which parts of the program have been updated, given a dependency graph, or based on the type of change. Underlying this is an idea that we also employed in earlier work (Wauters, Van der Plas *et al.* [158]), where particular program changes that match a specific pattern are treated separately: an entire incremental update may not always be the most efficient way to handle certain changes. This is also related to, but different from, work of Szabó *et al.* [134], who define a metric to empirically test the incrementalizability of Datalog computations, whereas we propose a heuristic to estimate the benefit of an incremental analysis for specific program changes.

6.4 Closing Remarks

It is difficult to overemphasise the importance of software in modern society. As a result, the consequences of software faults may be immense. To avoid such mistakes, static analyses support software developers during the software engineering process. However, to be of most use, analysis results need to be

produced timely. This can be challenging in large software ecosystems and when complex analyses are performed. To this end, incremental analyses, which update pre-existing analysis results upon a program change, can be employed.

In this dissertation, we presented a novel method to incrementalise existing effect-driven static analyses. Effect-driven analyses support the analysis of programs written in highly-dynamic programming languages with dynamic typing, polymorphism, and several forms of late binding and closures, in which control flow and data flow are intertwined. This complicates the efficient and precise updating of a previous analysis result. Our method makes use of the computational dependencies within the program under analysis, which are reified by the effect-driven analysis. The dependencies enable the propagation of updated analysis results and are also used to detect which parts of the result have become outdated and therefore can be removed. Moreover, the effect-driven design allows to easily bound the impact of program changes.

We introduced three complementary strategies to invalidate outdated parts of the analysis result. The invalidation of outdated information within the analysis result is built around the core idea of interleaving invalidation with recomputation, which enhances reuse of analysis results as it avoids excess invalidations. Moreover, this interleaving allows us to uphold a black-box view of components, which ensures a broad applicability by instating minimal requirements on the intra-component analysis.

We evaluated our method for soundness, precision, and performance on two benchmarking suites. We found that incrementalised effect-driven analyses produce sound results, although the analysis result may not always match the result of a full program analysis. Performance-wise, overall, the incremental analysis outperforms a from-scratch analysis of the updated program.

Bibliography

[1] Abella, J., Hernández, C., Quiñones, E., Cazorla, F. J., Conmy, P. R., Azkarate-askasua, M., Pérez, J., Mezzetti, E., and Vardanega, T. (2015). WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness. In *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, pages 39–48. IEEE.

[2] Abelson, H., Adams, N. I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., Steele, G. L., Sussman, G. J., and Wand, M. (1998). Revised^5 Report on the Algorithmic Language Scheme. R. Kelsey, W. D. Clinger, J. Rees, editors. Available online at `https://standards.scheme.org`.

[3] Acar, U. A. (2005). *Self-Adjusting Computation*. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA, USA.

[4] Acar, U. A. (2009). Self-Adjusting Computation: An Overview. In G. Puebla and G. Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 1–6, New York, NY, USA. ACM.

[5] Alaboudi, A. and LaToza, T. D. (2021). Edit-Run Behavior in Programming and Debugging. In K. J. Harms, J. Cunha, S. Oney, and C. Kelleher, editors, *Proceedings of the 2021 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2021, Saint Louis, MO, USA, October 10-13, 2021*, pages 1–10, Los Alamitos, CA, USA. IEEE.

[6] Alali, A., Kagdi, H. H., and Maletic, J. I. (2008). What's a Typical Commit? A Characterization of Open Source Software Repositories. In R. L. Krikhaar, R. Lämmel, and C. Verhoef, editors, *Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 182–191, Los Alamitos, CA, USA. IEEE.

[7] Albert, E., Correas, J., Puebla, G., and Román-Díez, G. (2012). Incremental Resource Usage Analysis. In O. Kiselyov and S. J. Thompson, editors, *Proceedings of*

the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, PA, USA, January 23-24, 2012, pages 25–34, New York, NY, USA. ACM.

[8] Albert, E., Correas, J., Puebla, G., and Román-Díez, G. (2015). A multi-domain incremental analysis engine and its application to incremental resource analysis. *Theoretical Computer Science*, **585**, 91–114.

[9] Andreasen, E. S., Møller, A., and Nielsen, B. B. (2017). Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP 2017, Barcelona, Spain, June 18, 2017*, pages 31—36, New York, NY, USA. ACM.

[10] Ariane 501 Inquiry Board (1996). Ariane 5: Flight 105 Failure. Retrieved from `https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report` on August 9, 2024.

[11] Arzt, S. and Bodden, E. (2013). Efficiently updating IDE-based data-flow analyses in response to incremental program changes. Technical Report TUD-CS-2013-0253, EC SPRIDE. `https://bodden.de/pubs/TUD-CS-2013-0253.pdf`.

[12] Arzt, S. and Bodden, E. (2014). Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, May 31-June 07, 2014*, pages 288–298, New York, NY, USA. ACM.

[13] Atiyah, M. F. and Macdonald, I. G. (1969). *Introduction to Commutative Algebra*. Addison-Wesley Series in Mathematics. Addison-Wesley, London, UK.

[14] Beizer, B. (1990). *Software Testing Techniques (2nd Ed.)*. International Thomson Computer Press, Boston, MA, USA.

[15] Binkley, D. W. (2007). Source Code Analysis: A Road Map. In L. C. Briand and A. L. Wolf, editors, *Proceedings of the 2007 Workshop on the Future of Software Engineering, FOSE 2007, Minneapolis, MN, USA, May 23-25, 2007*, pages 104–119, Los Alamitos, CA, USA. IEEE Computer Society.

[16] Birkhoff, G. (1967). *Lattice Theory (3rd Ed.)*. Colloquium Publications, volume XXV. American Mathematical Society, Providence, RI, USA.

[17] Bodden, E. (2018). The Secret Sauce in Efficient and Precise Static Analysis: The beauty of distributive, summary-based static analyses (and how to master them).

In J. Dolby, W. G. J. Halfond, and A. Mishra, editors, *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 85–93, New York, NY, USA. ACM.

[18] Bravenboer, M. and Smaragdakis, Y. (2009). Strictly Declarative Specification of Sophisticated Points-to Analyses. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, FL, USA*, pages 243–262, New York, NY, USA. ACM.

[19] Burke, M. G. (1990). An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. *ACM Transactions on Programming Languages and Systems*, **12**(3), 341–395.

[20] Bygde, S. (2013). *Parametric WCET Analysis*. Doctoral dissertation, Mälardalen University, Västerås, Sweden.

[21] Calcagno, C. and Distefano, D. (2011). Infer: An Automatic Program Verifier for Memory Safety of C Programs. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *Proceedings of the 3rd International NASA Formal Methods Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465, Berlin, Heidelberg, Germany. Springer.

[22] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P. W., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving Fast with Software Verification. In K. Havelund, G. J. Holzmann, and R. Joshi, editors, *Proceedings of the 7th International NASA Formal Methods Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11, Cham, Switzerland. Springer.

[23] Carroll, M. D. and Ryder, B. G. (1988). Incremental Data Flow Analysis via Dominator and Attribute Updates. In J. Ferrante and P. Mager, editors, *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, San Diego, CA, USA, January 10-13, 1988*, pages 274–284, New York, NY, USA. ACM.

[24] Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change Detection in Hierarchically Structured Information. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD 1996, Montreal, Quebec, Canada, June 4-6, 1996*, pages 493–504, New York, NY, USA. ACM.

[25] Collins, M. and Roark, B. (2004). Incremental Parsing with the Perceptron Algorithm. In D. Scott, W. Daelemans, and M. A. Walker, editors, *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics, Barcelona, Spain, 21-26 July, 2004*, pages 111–118, USA. ACL.

[26] Conway, C. L., Namjoshi, K. S., Dams, D., and Edwards, S. A. (2005). Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 449–461, Berlin, Heidelberg, Germany. Springer.

[27] Cousot, P. (2021). *Principles of Abstract Interpretation*. The MIT Press, Cambridge, MA, USA.

[28] Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, Los Angeles, CA, USA, January 17-19, 1977*, pages 238–252, New York, NY, USA. ACM.

[29] Cousot, P. and Cousot, R. (1992). Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming, PLILP 1992, Leuven, Belgium, August 26-28, 1992*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Berlin, Heidelberg, Germany. Springer.

[30] Cousot, P. and Cousot, R. (2002). Modular Static Program Analysis. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction, CC 2002, Grenoble, France, April 8-12, 2002*, pages 159–178, Berlin, Heidelberg, Germany. Springer.

[31] Denning, D. E. and Denning, P. J. (1977). Certification of Programs for Secure Information Flow. *Communications of the ACM*, **20**(7), 504–513.

[32] Di Pierro, A. and Wiklicky, H. (2001). Measuring the Precision of Abstract Interpretations. In K. Lau, editor, *Proceedings of the 10th International Workshop on Logic Based Program Synthesis and Transformation, LOPSTR 2000, London, UK, July 24-28, 2000*, volume 2042 of *Lecture Notes in Computer Science*, pages 147–164, Berlin, Heidelberg, Germany. Springer.

[33] Diwan, A., McKinley, K. S., and Moss, J. E. B. (1998). Type-Based Alias Analysis. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, Montreal, Canada, June 17-19, 1998*, pages 106–117, New York, NY, USA. ACM.

[34] Dowson, M. (1997). The ARIANE 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, **22**(2), 84.

[35] Dura, A., Reichenbach, C., and Söderberg, E. (2021). JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. *Proceedings of the ACM on Programming Languages*, **5**(OOPSLA), 1–31.

[36] Düsing, J. and Hermann, B. (2023). Persisting and Reusing Results of Static Program Analyses on a Large Scale. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, Luxembourg, September 11-15, 2023*, pages 888–900, Los Alamitos, CA, USA. IEEE.

[37] Eichberg, M. and Hermann, B. (2014). A Software Product Line for Static Analyses: The OPAL Framework. In S. Arzt and R. A. Santelices, editors, *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, June 12, 2014*, pages 2:1–2:6, New York, NY, USA. ACM.

[38] Eichberg, M., Kahl, M., Saha, D., Mezini, M., and Ostermann, K. (2007). Automatic Incrementalization of Prolog Based Static Analyses. In M. Hanus, editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 109–123, Berlin, Heidelberg, Germany. Springer.

[39] Erdweg, S., Szabó, T., and Pacak, A. (2021). Concise, Type-Safe, and Efficient Structural Diffing. In S. N. Freund and E. Yahav, editors, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Virtual Event, June 20-25, 2021*, pages 406–419, New York, NY, USA. ACM.

[40] Erhard, J., Saan, S., Tilscher, S., Schwarz, M., Holter, K., Vojdani, V., and Seidl, H. (2022). Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap. `https://arxiv.org/abs/2209.10445`.

[41] Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. In I. Crnkovic, M. Chechik,

and P. Grünbacher, editors, *Proceedings of the 19th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden, September 15-19, 2014*, pages 313–324, New York, NY, USA. ACM.

[42] Fan, W., Hu, C., and Tian, C. (2017). Incremental Graph Computations: Doable and Undoable. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD 2017, Chicago, IL, USA, May 14-19, 2017*, pages 155–169, New York, NY, USA. ACM.

[43] Firsov, D. and Jeltsch, W. (2016). Purely Functional Incremental Computing. In F. Castor and Y. D. Liu, editors, *Proceedings of the 20th Brazilian Symposium on Programming Languages, SBLP 2016, Maringá, Brazil, September 22-23, 2016*, volume 9889 of *Lecture Notes in Computer Science*, pages 62–77, Cham, Switzerland. Springer.

[44] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The Essence of Compiling with Continuations. In R. Cartwright, editor, *Proceedings of the 14th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1993, Albuquerque, NM, USA, June 23-25, 1993*, pages 237–247, New York, NY, USA. ACM.

[45] Fluri, B., Würsch, M., Pinzger, M., and Gall, H. C. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, **33**(11), 725–743.

[46] Gall, H. C., Fluri, B., and Pinzger, M. (2009). Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, **26**(1), 26–33.

[47] Garcia-Contreras, I., Caballero, J. F. M., and Hermenegildo, M. V. (2018). An Approach to Incremental and Modular Context-Sensitive Analysis. Technical Report CLIP-2/2018.0, ETSI_Informatica, Madrid, España. `https://oa.upm.es/53067`.

[48] Garcia-Contreras, I., Morales, J. F., and Hermenegildo, M. V. (2021). Incremental and Modular Context-sensitive Analysis. *Theory and Practice of Logic Programming*, **21**(2), 196–243.

[49] Ghime, V., Khadsare, A., Jana, A., and Chimdyalwar, B. (2022). IR Mapping: Intermediate Representation (IR) based Mapping to facilitate Incremental Static Analysis. In S. Tiwari, S. Chaudhary, C. K. Roy, M. D'Souza, R. Sharma, and L. Kumar, editors, *Proceedings of the 15th Innovations in Software Engineering Conference, ISEC 2022, Gandhinagar, India, February 24-26, 2022*, pages 23:1–23:5, New York, NY, USA. ACM.

[50] Giese, I. (2002). Software Reliability - Ariane 5 - 501. `https://www.sarahandrobin.com/ingo/swr/ariane5.html`. Accessed: August 9, 2024.

[51] Glaze, D. A. and Van Horn, D. (2014). Abstracting Abstract Control. *ACM SIGPLAN Notices*, **50**(2), 11–22.

[52] Glaze, D. A., Labich, N., Might, M., and Van Horn, D. (2013). Optimizing Abstract Abstract Machines. In G. Morrisett and T. Uustalu, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA, September 25-27, 2013*, pages 443–454, New York, NY, USA. ACM.

[53] Goubault, E., Putot, S., and Védrine, F. (2012). Modular Static Analysis with Zonotopes. In A. Miné and D. Schmidt, editors, *Proceedings of the 19th International Static Analysis Symposium, SAS 2012, Deauville, France, September 11-13, 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 24–40, Berlin, Heidelberg, Germany. Springer.

[54] Grätzer, G. (2009). *Lattice Theory: First Concepts and Distributive Lattices*. Dover Publications Inc., Mineola, NY, USA. Unabridged republication of the work originally published in 1971 by W. H. Freeman and Company, San Francisco, CA, USA.

[55] Grätzer, G. (2011). *Lattice Theory: Foundation*. Birkhäuser, Basel, Switzerland.

[56] Guarnieri, S. and Livshits, B. (2010). Gulfstream: Staged Static Analysis for Streaming JavaScript Applications. In J. K. Ousterhout, editor, *Proceedings of the 2010 USENIX Conference on Web Application Development, WebApps 2010, Boston, MA, USA, June 23-24, 2010*. USENIX Association.

[57] Gupta, A., Mumick, I. S., and Subrahmanian, V. S. (1993). Maintaining Views Incrementally. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, Washington, DC, USA, May 26-28, 1993*, pages 157–166, New York, NY, USA. ACM.

[58] Hammer, M. A., Khoo, Y. P., Hicks, M., and Foster, J. S. (2014). Adapton: Composable, Demand-Driven Incremental Computation. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, June 9-11, 2014*, pages 156–166, New York, NY, USA. ACM.

[59] Hammer, M. A., Dunfield, J., Headley, K., Labich, N., Foster, J. S., Hicks, M. W., and Van Horn, D. (2015). Incremental Computation with Names. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International*

Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 748–766, New York, NY, USA. ACM.

[60] Harman, M. and O'Hearn, P. W. (2018). From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*, pages 1–23, Los Alamitos, CA, USA. IEEE.

[61] Hattori, L. and Lanza, M. (2008). On the Nature of Commits. In *Workshop Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE Workshops 2008, L'Aquila, Italy, September 15-16, 2008*, pages 63–71, Los Alamitos, CA, USA. IEEE.

[62] Hattori, L. and Lanza, M. (2010). Syde: A Tool for Collaborative Software Development. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, May 1-8, 2010*, page 235–238, New York, NY, USA. ACM.

[63] Hedin, G. (1988). Incremental Attribute Evaluation with Side-effects. In D. K. Hammer, editor, *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation, CCHSC 1988, Berlin, German Democratic Republic, October 10-14, 1988*, volume 371 of *Lecture Notes in Computer Science*, pages 175–189, Berlin, Heidelberg, Germany. Springer.

[64] Hedin, G. (1991). Incremental Static Semantic Analysis for Object-Oriented Languages using Door Attribute Grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems, SAGA 1991, Prague, Czechoslovakia, June 4-13, 1991*, volume 545 of *Lecture Notes in Computer Science*, pages 374–379, Berlin, Heidelberg, Germany. Springer.

[65] Hedin, G. (1992). *Incremental Semantic Analysis*. Doctoral dissertation, Lund University, Lund, Sweden.

[66] Hermenegildo, M. V., Puebla, G., Marriott, K., and Stuckey, P. J. (2000). Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, **22**(2), 187–223.

[67] Hind, M. (2001). Pointer Analysis: Haven't We Solved This Problem Yet? In J. Field and G. Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2001, Snowbird, UT, USA, June 18-19, 2001*, pages 54–61, New York, NY, USA. ACM.

[68] Igarashi, A. and Kobayashi, N. (2005). Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, **27**(2), 264–313.

[69] Journault, M., Miné, A., and Ouadjaout, A. (2018). Modular Static Analysis of String Manipulations in C Programs. In *Proceedings of the 25th International Static Analysis Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018*, volume 11002 of *Lecture Notes in Computer Science*, pages 243–262, Cham, Switzerland. Springer.

[70] Keidel, S., Poulsen, C. B., and Erdweg, S. (2018). Compositional Soundness Proofs of Abstract Interpreters. *Proceedings of the ACM on Programming Languages*, **2**(ICFP), 72:1–72:26.

[71] Krainz, J. and Philippsen, M. (2017). Diff Graphs for a fast Incremental Pointer Analysis. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS@ECOOP 2017, Barcelona, Spain, June 19, 2017*, pages 4:1–4:10, New York, NY, USA. ACM.

[72] Krishnan, P., O'Donoghue, R., Allen, N., and Lu, Y. (2019). Commit-Time Incremental Analysis. In N. Grech and T. Lavoie, editors, *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 26–31, New York, NY, USA. ACM.

[73] Kursun, T. R., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2022). RacketLogger: Logging and Visualising Changes in DrRacket. In D. Verna, editor, *Proceedings of the 15th European Lisp Symposium, ELS 2022, Porto, Portugal, April 21-22, 2022*, pages 61–68. ELSAA.

[74] Ley-Wild, R., Acar, U. A., and Blelloch, G. E. (2012). Non-monotonic Self-Adjusting Computation. In H. Seidl, editor, *Programming Languages and Systems, Proceedings of the 21st European Symposium on Programming, ESOP 2012, Tallinn, Estonia, March 24 - April 1, 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 476–496, Berlin, Heidelberg, Germany. Springer.

[75] Li, Y. S., Malik, S., and Wolfe, A. (1999). Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems*, **4**(3), 257–279.

[76] Liu, B., Huang, J., and Rauchwerger, L. (2019). Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, **41**(1), 6:1–6:31.

[77] Liu, Y. A. (2024). Incremental Computation: What Is the Essence? (Invited Contribution). In G. Keller and M. Wang, editors, *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, January 16, 2024*, pages 39–52, New York, NY, USA. ACM.

[78] Lu, Y., Shang, L., Xie, X., and Xue, J. (2013). An Incremental Points-to Analysis with CFL-Reachability. In R. Jhala and K. De Bosschere, editors, *Proceedings of the 22nd International Conference on Compiler Construction, CC 2013, Rome, Italy, March 16-24, 2013*, pages 61–81, Berlin, Heidelberg, Germany. Springer.

[79] Marlowe, T. J. and Ryder, B. G. (1990). An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. In F. E. Allen, editor, *Proceedings of the 17th ACM Symposium on Principles of Programming Languages, POPL 1990, San Francisco, CA, USA, January, 1990*, pages 184–196, New York, NY, USA. ACM.

[80] McPeak, S., Gros, C., and Ramanathan, M. K. (2013). Scalable and Incremental Software Bug Detection. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russia, August 18-26, 2013*, pages 554–564, New York, NY, USA. ACM.

[81] Mey, J., Schöne, R., Hedin, G., Söderberg, E., Kühn, T., Fors, N., Öqvist, J., and Aßmann, U. (2018). Continuous Model Validation using Reference Attribute Grammars. In D. J. Pearce, T. Mayerhofer, and F. Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 5-6, 2018*, pages 70–82, New York, NY, USA. ACM.

[82] Mey, J., Schöne, R., Hedin, G., Söderberg, E., Kühn, T., Fors, N., Öqvist, J., and Aßmann, U. (2020). Relational reference attribute grammars: Improving continuous model validation. *Journal of Computer Languages*, **57**, 100940.

[83] Might, M. and Shivers, O. (2006). Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In J. H. Reppy and J. L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, OR, USA, September 16-21, 2006*, pages 13–25, New York, NY, USA. ACM.

[84] Might, M. and Van Horn, D. (2011). A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-Order Programs. In E. Yahav, editor, *Proceedings of the 18th International Static Analysis Symposium, SAS 2011, Venice, Italy, September 14-16, 2011*, pages 180–197, Berlin, Heidelberg, Germany. Springer.

[85] Miné, A. (2014). Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In K. L. McMillan and X. Rival, editors, *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014*, pages 39–58, Berlin, Heidelberg, Germany. Springer.

[86] Miraldo, V. C., Dagand, P.-É., and Swierstra, W. (2017). Type-Directed Diffing of Structured Data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2017, Oxford, UK, September 3-9, 2017*, page 2–15, New York, NY, USA. ACM.

[87] Møller, A. and Schwartzbach, M. I. (2024). Static Program Analysis. Revised version of June 19, 2024. Retrieved from `https://cs.au.dk/~{}amoeller/spa/` on August 13, 2024.

[88] Mudduluru, R. and Ramanathan, M. K. (2014). Efficient Incremental Static Analysis Using Path Abstraction. In S. Gnesi and A. Rensink, editors, *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering, FASE 2014, Grenoble, France, April 5-13, 2014*, volume 8411 of *Lecture Notes in Computer Science*, pages 125–139, Berlin, Heidelberg, Germany. Springer.

[89] Muylaert, W. and De Roover, C. (2018). Untangling Composite Commits Using Program Slicing. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*, pages 193–202, Los Alamitos, CA, USA. IEEE Computer Society.

[90] NASA Safety Center (2007). Powerless. *System Failure Case Studies*, **1**(10). Retrieved from `https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2008-03-01-northeastblackoutof2003.pdf` on August 9, 2024.

[91] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D. (2012). Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In J. Noble, editor, *Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP 2010, Beijing, China, June 11-16, 2012*, pages 79–103, Berlin, Heidelberg, Germany. Springer.

[92] Nichols, L., Emre, M., and Hardekopf, B. (2019a). Fixpoint Reuse for Incremental JavaScript Analysis. In N. Grech and T. Lavoie, editors, *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 2–7. ACM.

[93] Nichols, L., Emre, M., and Hardekopf, B. (2019b). Fixpoint Reuse for Incremental JavaScript Analysis (Extended Version). `https://cs.ucsb.edu/sites/default/files/documents/worklist-reuse.pdf`, visited on 2023-08-23.

[94] Nicolay, J., Noguera, C., De Roover, C., and De Meuter, W. (2013). Determining Dynamic Coupling in JavaScript Using Object Type Inference. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pages 126–135, Los Alamitos, CA, USA. IEEE.

[95] Nicolay, J., Stiévenart, Q., De Meuter, W., and De Roover, C. (2017). Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, **29**(12), e1889.

[96] Nicolay, J., Stiévenart, Q., De Meuter, W., and De Roover, C. (2019). Effect-Driven Flow Analysis. In C. Enea and R. Piskac, editors, *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019*, pages 247–274, Cham, Switzerland. Springer.

[97] North American Electric Reliability Council (2004). Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn? Retrieved from `https://www.nerc.com/pa/rrm/ea/August%2014%202003%20Blackout%20Investigation%20DL/NERC_Final_Blackout_Report_07_13_04.pdf` on August 9, 2024.

[98] Omori, T. and Maruyama, K. (2008). A Change-Aware Development Environment by Recording Editing Operations of Source Code. In A. E. Hassan, M. Lanza, and M. W. Godfrey, editors, *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008, Leipzig, Germany, May 10-11, 2008*, pages 31–34, New York, NY, USA. ACM.

[99] Pacak, A., Erdweg, S., and Szabó, T. (2020). A Systematic Approach to Deriving Incremental Type Checkers. *Proceedings of the ACM on Programming Languages*, **4**(OOPSLA), 127:1–127:28.

[100] Pacak, A., Szabó, T., and Erdweg, S. (2022). Incremental Processing of Structured Data in Datalog. In B. Scholz and Y. Kameyama, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 20–32, New York, NY, USA. ACM.

[101] Palikareva, H., Kuchta, T., and Cadar, C. (2016). Shadow of a Doubt: Testing for Divergences Between Software Versions. In L. K. Dillon, W. Visser, and

L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1181–1192, New York, NY, USA. ACM.

[102] Pollock, L. L. and Soffa, M. L. (1989). An Incremental Version of Iterative Data Flow Analysis. *IEEE Transactions on Software Engineering*, **15**(12), 1537–1549.

[103] Puebla, G. and Hermenegildo, M. V. (1996). Optimized Algorithms for Incremental Analysis of Logic Programs. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the 3rd International Symposium on Static Analysis, SAS 1996, Aachen, Germany, September 24-26, 1996*, volume 1145 of *Lecture Notes in Computer Science*, pages 270–284, Berlin, Heidelberg, Germany. Springer.

[104] Purushothaman, R. and Perry, D. E. (2005). Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering*, **31**(6), 511–526.

[105] Ramalingam, G. and Reps, T. W. (1993). A Categorized Bibliography on Incremental Computation. In M. S. Van Deusen and B. Lang, editors, *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993, Charleston, SC, USA, January, 1993*, pages 502–510, New York, NY, USA. ACM.

[106] Ranganath, V. P., Amtoft, T., Banerjee, A., Dwyer, M. B., and Hatcliff, J. (2005). A New Foundation for Control-Dependence and Slicing for Modern Program Structures. In M. Sagiv, editor, *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK, April 4-8, 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 77–93, Berlin, Heidelberg, Germany. Springer.

[107] Razafintsialonina, M., Bühler, D., Miné, A., Perrelle, V., and Signoles, J. (2024). Réutilisations de caches et d'invariants pour l'analyse statique incrémentale. In *35es Journées Francophones des Langages Applicatifs, JFLA 2024, Saint-Jacut-de-la-Mer, France, January 30 - February 2, 2024*, pages 64–84. HAL. hal-04406463.

[108] Reps, T. W., Teitelbaum, T., and Demers, A. J. (1983). Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems*, **5**(3), 449–477.

[109] Reps, T. W., Horwitz, S., and Sagiv, M. (1995). Precise Interprocedural Dataflow Analysis via Graph Reachability. In R. K. Cytron and P. Lee, editors, *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, San Francisco, CA, USA, January 23-25, 1995*, pages 49–61, New York, NY, USA. ACM.

[110] Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, **74**(2), 358–366.

[111] Rival, X. and Yi, K. (2020). *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, Cambridge, MA, USA.

[112] Sagiv, M., Reps, T., and Horwitz, S. (1996). Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, **167**(1), 131–170.

[113] Saha, D. and Ramakrishnan, C. R. (2005). Incremental and Demand-driven Points-To Analysis Using Logic Programming. In P. Barahona and A. P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2005, Lisbon, Portugal, July 11-13 2005*, pages 117–128, New York, NY, USA. ACM.

[114] Schubert, P. D., Hermann, B., and Bodden, E. (2019). PhASAR: An Interprocedural Static Analysis Framework for C/C++. In T. Vojnar and L. Zhang, editors, *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019, Prague, Czech Republic, April 6-11, 2019*, volume 11428 of *Lecture Notes in Computer Science*, pages 393–410, Cham, Switzerland. Springer.

[115] Schubert, P. D., Leer, R. L., Hermann, B., and Bodden, E. (2021a). Into the Woods: Experiences from Building a Dataflow Analysis Framework for C/C++. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Virtual Event, September 27-28, 2021*, pages 18–23, Los Alamitos, CA, USA. IEEE.

[116] Schubert, P. D., Sattler, F., Schiebel, F., Herman, B., and Bodden, E. (2021b). Modeling the Effects of Global Variables in Data-Flow Analysis for C/C++. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Virtual Event, September 27-28, 2021*, pages 12–17, Los Alamitos, CA, USA. IEEE.

[117] Seidl, H., Erhard, J., and Vogler, R. (2020). Incremental Abstract Interpretation. In A. D. Pierro, P. Malacaria, and R. Nagarajan, editors, *From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement*, pages 132–148, Cham, Switzerland. Springer.

[118] Shang, L., Lu, Y., and Xue, J. (2012). Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation: Preliminary Experience. In

Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, September 3-7, 2012, pages 270–273, New York, NY, USA. ACM.

[119] Söderberg, E. and Hedin, G. (2012). Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking. Technical Report LU-CS-TR:2012-249, vol. 98, Department of Computer Science, Lund University.

[120] Sotin, P. (2010). Quantifying the Precision of Numerical Abstract Domains. Research report, INRIA Grenoble – Rhône-Alpes. HAL. inria-00457324.

[121] Stein, B., Chang, B. E., and Sridharan, M. (2021a). Demanded Abstract Interpretation. In S. N. Freund and E. Yahav, editors, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Virtual Event, June 20-25, 2021*, pages 282–295, New York, NY, USA. ACM.

[122] Stein, B., Chang, B. E., and Sridharan, M. (2021b). Demanded Abstract Interpretation (Extended Version). *Computing Research Repository*, **abs/2104.01270**. `https://arxiv.org/abs/2104.01270v2`.

[123] Stein, B., Chang, B. E., and Sridharan, M. (2024). Interactive Abstract Interpretation with Demanded Summarization. *ACM Transactions on Programming Languages and Systems*, **46**(1), 4:1–4:40.

[124] Stevens, R. and De Roover, C. (2014). Querying the History of Software Projects using QwalKeko. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 585–588, Los Alamitos, CA, USA. IEEE Computer Society.

[125] Stevens, R., Molderez, T., and De Roover, C. (2019). Querying distilled code changes to extract executable transformations. *Empirical Software Engineering*, **24**(1), 491–535.

[126] Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2019). A general method for rendering static analyses for diverse concurrency models modular. *Journal of Systems and Software*, **147**, 17–45.

[127] Stiévenart, Q., Van Es, N., Van der Plas, J., and De Roover, C. (2021). A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, **181**, 111042.

[128] Stiévenart, Q. (2018). *Scalable Designs for Abstract Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading*. Doctoral dissertation, Vrije Universiteit Brussel, Brussels, Belgium.

[129] Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2015). Detecting Concurrency Bugs in Higher-Order Programs through Abstract Interpretation. In M. Falaschi and E. Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming - PPDP 2015, Siena, Italy, June 14-16, 2015*, pages 232–243, New York, NY, USA. ACM.

[130] Szabó, T. (2021). *Incrementalizing Static Analyses in Datalog*. Doctoral dissertation, Johannes Gutenberg-Universität Mainz, Mainz, Germany.

[131] Szabó, T. (2023). Incrementalizing Production CodeQL Analyses. In S. Chandra, K. Blincoe, and P. Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1716–1726, New York, NY, USA. ACM.

[132] Szabó, T., Kuci, E., Bijman, M., Mezini, M., and Erdweg, S. (2018a). Incremental Overload Resolution in Object-Oriented Programming Languages. In J. Dolby, W. G. J. Halfond, and A. Mishra, editors, *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 27–33, New York, NY, USA. ACM.

[133] Szabó, T., Bergmann, G., Erdweg, S., and Voelter, M. (2018b). Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages*, **2**(OOPSLA), 1–29.

[134] Szabó, T., Erdweg, S., and Bergmann, G. (2021). Incremental Whole-Program Analysis in Datalog with Lattices. In S. N. Freund and E. Yahav, editors, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Virtual Event, June 20-26, 2021*, pages 1–15, New York, NY, USA. ACM.

[135] Szabó, T., Erdweg, S., and Voelter, M. (2016). IncA: A DSL for the Definition of Incremental Program Analyses. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 320–331, New York, NY, USA. ACM.

[136] Szabó, T., Bergmann, G., and Erdweg, S. (2019). Incrementalizing interprocedural program analyses with recursive aggregation in Datalog. Presented at the Second Workshop on Incremental Computing, IC 2019, Athens, Greece, October 21, 2019.

[137] Taneja, J., Liu, Z., and Regehr, J. (2020). Testing Static Analyses for Precision and Soundness. In J. Mars, L. Tang, J. Xue, and P. Wu, editors, *Proceedings of*

the 18th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2020, San Diego, CA, USA, February 22-26, 2020, pages 81–93, New York, NY, USA. ACM.

[138] Tarjan, R. E. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, **1**(2), 146–160.

[139] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**(2), 285–309.

[140] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). TAJ: Effective Taint Analysis of Web Applications. In M. Hind and A. Diwan, editors, *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97, New York, NY, USA. ACM.

[141] Tripp, O., Pistoia, M., Cousot, P., Cousot, R., and Guarnieri, S. (2013). Andromeda: Accurate and Scalable Security Analysis of Web Applications. In V. Cortellessa and D. Varró, editors, *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Rome, Italy, March 16-24, 2013*, pages 210–225, Berlin, Heidelberg, Germany. Springer.

[142] U.S.-Canada Power System Outage Task Force (2004). Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations. Retrieved from `https://www3.epa.gov/region1/npdes/merrimackstation/pdfs/ar/AR-1165.pdf` on August 9, 2024.

[143] Vallée-Rai, R., Gagnon, E., Hendren, L. J., Lam, P., Pominville, P., and Sundaresan, V. (2000). Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In D. A. Watt, editor, *Proceedings of the 9th International Conference on Compiler Construction, CC 2000, Berlin, Germany, March 25 - April 2, 2000*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Heidelberg, Germany. Springer.

[144] Van der Plas, J. (2019). *Incremental Thread-Modular Static Analysis for Concurrent Programs with Futures and Atoms*. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium.

[145] Van der Plas, J., Stiévenart, Q., Van Es, N., and De Roover, C. (2020a). Incremental Flow Analysis through Computational Dependency Reification. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, pages 25–36, Los Alamitos, CA, USA. IEEE Computer Society.

[146] Van der Plas, J., Stiévenart, Q., Van Es, N., and De Roover, C. (2020b). SCAM 2020 Benchmark Programs. `https://github.com/jevdplas/SCAM2020-Benchmarks`. DOI: `10.5281/zenodo.10808700`.

[147] Van der Plas, J., Nicolay, J., De Meuter, W., and De Roover, C. (2023a). ModInF: Exploiting Reified Computational Dependencies for Information Flow Analysis. In H. Kaindl, M. Mannion, and L. A. Maciaszek, editors, *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023*, pages 420–427, Setúbal, Portugal. INSTICC, SciTePress.

[148] Van der Plas, J., Stiévenart, Q., and De Roover, C. (2023b). Result Invalidation for Incremental Modular Analyses. In C. Dragoi, M. Emmi, and J. Wang, editors, *Proceedings of the 24th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023, Boston, MA, USA, January 15-21, 2023*, volume 13881 of *Lecture Notes in Computer Science*, pages 296–319, Cham, Switzerland. Springer.

[149] Van Es, N., Vandercammen, M., and De Roover, C. (2017). Incrementalizing Abstract Interpretation. In S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, editors, *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, BENEVOL 2017, Antwerp, Belgium, December 4-5, 2017*, pages 31–35, Aachen, Germany. CEUR Workshop Proceedings.

[150] Van Es, N., Stiévenart, Q., and De Roover, C. (2019). Garbage-Free Abstract Interpretation Through Abstract Reference Counting. In A. F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming, ECOOP 2019, London, UK, July 15-19, 2019*, pages 10:1–10:33.

[151] Van Es, N., Stiévenart, Q., Van der Plas, J., and De Roover, C. (2020a). A Parallel Worklist Algorithm for Modular Analyses. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, pages 1–12, Los Alamitos, CA, USA. IEEE Computer Society.

[152] Van Es, N., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2020b). MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*, Los Alamitos, CA, USA. IEEE Computer Society.

[153] Van Horn, D. and Might, M. (2010). Abstracting Abstract Machines. In P. Hudak and S. Weirich, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 51–62, New York, NY, USA. ACM.

[154] Van Horn, D. and Might, M. (2012). Systematic abstraction of abstract machines. *Journal of Functional Programming*, **22**(4-5), 705–746.

[155] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., and Zaidman, A. (2020). How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, **25**(2), 1419–1457.

[156] Wagner, T. A. and Graham, S. L. (1997). Incremental Analysis of Real Programming Languages. In M. C. Chen, R. K. Cytron, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI 1997, Las Vegas, NV, USA, June 15-18, 1997*, pages 31–43, New York, NY, USA. ACM.

[157] Wang, L., Kong, X., Wang, J., and Li, B. (2022). An incremental software architecture recovery technique driven by code changes. *Frontiers of Information Technology & Electronic Engineering*, **23**(5), 664–677.

[158] Wauters, C., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2023). Change Pattern Detection for Optimising Incremental Static Analysis. In L. Moonen, C. D. Newman, and A. Gorla, editors, *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, pages 49–60, Los Alamitos, CA, USA. IEEE.

[159] Wegman, M. N. and Zadeck, F. K. (1991). Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, **13**(2), 181–210.

[160] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*, volume 6 of *The Kluwer International Series in Software Engineering*. Springer, New York, NY, USA.

[161] Yan, D., Xu, G., and Rountev, A. (2011). Demand-Driven Context-Sensitive Alias Analysis for Java. In M. B. Dwyer and F. Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 155–165, New York, NY, USA. ACM.

[162] Yoon, Y. and Myers, B. A. (2011). Capturing and Analyzing Low-Level Events from the Code Editor. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2011, Portland, OR, USA, October 24, 2011*, page 25–30, New York, NY, USA. ACM.

[163] Yu, Q., He, F., and Wang, B.-Y. (2020). Incremental Predicate Analysis for Regression Verification. *Proceedings of the ACM on Programming Languages*, **4**(OOPSLA).

[164] Yur, J., Ryder, B. G., and Landi, W. (1999). An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999, Los Angeles, CA, USA, May 16-22, 1999*, pages 442–451, New York, NY, USA. ACM.

[165] Zadeck, F. K. (1984). Incremental Data Flow Analysis in a Structured Program Editor. In M. S. V. Deusen and S. L. Graham, editors, *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, pages 132–143, New York, NY, USA. ACM.

[166] Zhai, Y., Hao, Y., Zhang, Z., Chen, W., Li, G., Qian, Z., Song, C., Sridharan, M., Krishnamurthy, S. V., Jaeger, T., and Yu, P. L. (2022). Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, CA, USA, April 24-28, 2022*, Reston, VA, USA. The Internet Society.

[167] Zhan, S. and Huang, J. (2016). ECHO: Instantaneous In Situ Race Detection in the IDE. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 775–786, New York, NY, USA. ACM.

[168] Zhang, J. (2019). *Incremental Static Analysis of Probabilistic Programs*. Doctoral dissertation, University of New South Wales, Sydney, Australia.

[169] Zhao, Z., Wang, X., Xu, Z., Tang, Z., Li, Y., and Di, P. (2023). Incremental Call Graph Construction in Industrial Practice. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 471–482, Los Alamitos, CA, USA. IEEE.

[170] Zheng, X. and Rugina, R. (2008). Demand-Driven Alias Analysis for C. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, CA, USA, January 7-12, 2008*, pages 197–208, New York, NY, USA. ACM.