Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

A scalable static analysis framework for reliable program development exploiting incrementality and modularity

Ph.D. Thesis

Isabel García Contreras
MSc. in Artificial Intelligence

Departamento de Inteligencia Artificial

Escuela Técnica Superior de Ingenieros Informáticos

A scalable static analysis framework for reliable program development exploiting incrementality and modularity

Submitted in partial fulfillment of the requirements for the degree of:
PhD. in Artificial Intelligence

Author: **Isabel García Contreras, MSc.**

Director: **Manuel V. Hermenegildo, PhD.**

Co-director: **José F. Morales, PhD.**

Defended on: July 21st, 2021

Isabel García Contreras: *A scalable static analysis framework for reliable program development exploiting incrementality and modularity.*

Abstract[1]

Automatic static analysis tools allow inferring properties about software without executing it and without the need for human interaction. When these tools are based on formal methods, the properties are guaranteed to hold and come with a mathematical proof. The usage of these tools during the coding, testing, and maintenance phases of the software development cycle helps reduce efforts in terms of time and cost, as they contribute to the early detection of bugs, automatic optimizations, or automatic documentation. The increasing importance of the reliability of evolving software is evidenced by the current number of tools and on-line platforms for continuous integration and deployment. In this setting, when changes happen fast, analysis tools are only useful if they are precise and, at the same time, scalable enough to provide results before the next change happens.

In this thesis we study scalable analyses in the context of abstract interpretation. Since a way to improve scalability is to perform coarser abstractions, we first inspect what effect this may have in effectively proving the absence of bugs. Second, we present a framework for scalable static analyses which is generic, that is, independent of the data abstraction of the program. We present several algorithms for incrementally reanalyzing whole programs in a context-sensitive manner, reusing as much as possible previous analysis results. A key novel aspect of the approach is to take advantage of the modular structure of programs, typically as defined by the programmer, while keeping a fine-grained relation between the analysis result and the source program. Additionally, we present a mechanism for the programmer to help the analyzer in terms of precision and performance by means of assertions. We show that these assertions together with incremental analysis are specially useful when analyzing generic code. All these algorithms have been implemented and evaluated for different abstract domains within the CiaoPP framework. Lastly, we present an application of the analysis framework to perform *on-the-fly* assertion checking, providing continuous and almost instantaneous feedback to the programmer as the code is written. Here the incrementality and modular nature of the presented algorithms, and the locality of the changes, are key to achieving fast response times.

1 This is the revised and updated version of the thesis after the defense.

Resumen

Las herramientas automáticas de análisis estático permiten inferir propiedades del software sin ejecutarlo y sin necesidad de intervención humana. Si dichas herramientas se basan en métodos formales, éstas proporcionan garantías matemáticas de que las propiedades se cumplen. El uso de estas herramientas durante las fases de codificación, prueba y mantenimiento del ciclo de desarrollo del software ayuda a reducir esfuerzo en términos de tiempo y coste, ya que contribuye a la detección temprana de errores, a la optimización automática y a la documentación automática. La creciente importancia de la fiabilidad de un software en constante evolución se ha puesto de manifiesto por el número de herramientas y plataformas disponibles on-line para la integración y despliegue continuos de software. En este contexto, en el que los cambios ocurren rápido, las herramientas de análisis son útiles sólo si son precisas y lo suficientemente escalables como para proporcionar resultados más rápidamente de lo que ocurren los cambios.

En esta tesis estudiamos análisis escalables en el ámbito de la interpretación abstracta. Dado que una forma de mejorar la escalabilidad es generalizar las abstracciones, primero estudiamos qué efecto tienen estas generalizaciones en la eficacia del dominio para demostrar la ausencia de errores. Segundo, presentamos un marco para el análisis estático que es genérico, es decir, independiente de la abstracción de los datos del programa. Presentamos diferentes algorithmos para reanalizar incrementalmente programas enteros, de forma sensible al contexto, reutilizando lo máximo posible un resultado anterior. Un aspecto novedoso y clave de nuestro enfoque es aprovechar la estructura modular de los programas, típicamente definida por la persona programadora, pero manteniendo una relación precisa entre el programa original y el resultado del análisis. Adicionalmente, presentamos un mecanismo para que la programadora pueda ayudar al analizador en términos de precisión y rendimiento mediante aserciones. Mostramos que estas aserciones junto con un análisis incremental son especialmente útiles para analizar código genérico. Todos los algoritmos han sido implementados y evaluados para diferentes dominios abstractos dentro de la herramienta CiaoPP. Por último, presentamos una aplicación de este marco de análisis para realizar comprobación de aserciones en tiempo de compilación *al vuelo*, de forma que se proporciona una retroalimentación continua a la programadora mientras escribe el código. Aquí, la naturaleza incremental y modular de los algoritmos, además de la localización de los cambios, son clave para lograr tiempos de respuesta rápidos.

Acknowledgements

These acknowledgements have been written in Spanish, to capture my gratitude in the best possible way. Their translation can be found below.

Gracias, en primer lugar, a mis directores Manuel y Jose sin los cuales este documento no existiría. Gracias por todo el apoyo, dedicación, enseñanzas y empujones finales que me habéis brindado a lo largo de estos años. Desde que llegué un día diciendo que quería hacer la tesis de máster sobre "eso de la interpretación abstracta" hasta ahora. Me habéis enseñado, además de investigación, paciencia y perseverancia. Gracias por no escatimar en nada y por todas las oportunidades que me habéis ofrecido para mejorar.

Gracias a los miembros del tribunal Patrick Cousot, John Gallagher, Roberto Giacobazzi, Arie Gurfinkel, Yahong Annie Liu, Manuel Carro y Pedro Lopez-Garcia. Gracias a todas las personas que desinteresadamente me han ayudado con este proyecto, incluyendo a los revisores anónimos (también al revisor 2). Gracias a Patrick Cousot y Arie Gurfinkel por la revisión de este documento y por vuestros inestimables comentarios. Gracias a Roberto Giacobazzi por enseñarme computabilidad y por las discusiones interesantes. Gracias a todos mis co-autores.

Gracias a todos los miembros del grupo CLIP, cada uno de vosotros me habéis aportado una visión diferente de la investigación: Nata, Max, Ignacio, Víctor, Miguel Ángel, Bish, Pedro, Manuel C, Joaquín y Umer. Especialmente quiero dar las gracias a Nata por compartir conmigo su sabiduría y por ayudarme desde el primer día que compartimos despacho. También por enseñarme qué esperar.

Gracias a Francisco Bueno (R.I.P.) por despertar mi interés en la programación lógica y la interpretación abstracta.

Gracias a todos los investigadores de IMDEA Software, en especial, a aquellos que comíamos juntos, por irme acercando un poco cada día al mundo de la investigación. Por nombrar algunos: Germán, Srdjan, Platon, Miriam, Miguel, Elena, Pedro, Borja, Alejandro, Chana e Ignacio.

Gracias también a todos los que trajeron tarta.

Gracias Ignacio por sentarte conmigo, por tu paciencia, tu amabilidad y tu experiencia sin las cuales esta tesis no sería posible.

Gracias a Bruno Dutertre, Ashish Gehani y especialmente a Jorge Navas por acogerme en SRI International y enseñarme otros puntos de vista en la investigación y a Arie Gurfinkel por su inestimable guía y consejos.

Gracias a todo equipo administrativo de IMDEA Software por su amabilidad y ayuda a lo largo de estos años. Gracias también a Irene y Pilar.

Gracias a todas las investigadoras que han venido antes de mi, allanando el camino y haciéndolo más amigable para las que venimos después. También a todas ellas con las que he compartido el camino del doctorado: Alba, Alejandra, Ana, Anaïs, Betxu, Chana, Elena, Kyveli, Marta, Marta, Marta, Miriam, Nata, Niki, y Silvia.

Gracias a Liss, Alba, Alejandra y Roi por las necesarias cervezas y los desahogos.

Gracias a mis amigos de la carrera Pablo, Mónica y Javi. Y al grupo HPCN, donde hice investigación por primera vez, en especial a Rafa, Juan, Rubén, Jose y Sergio.

Gracias a mis profesores del instituto Ricardo Fernández y Luis de Peña por despertar mi interés en entender cómo y por qué funcionan las cosas.

Gracias a mi familia por ayudarme a llegar hasta aquí. Gracias a mi madre, a mi padre, a mi hermano Ginés, a mi tía Luci, a mis abuelas Ana y María por acompañarme en los atascos del autobús y a mis abuelos aunque ya no estén. Gracias a Isa, Javier y Javi por vuestro apoyo y por el viaje a comprar turrones.

Gracias, Diego, por tu apoyo continuo, tu optimismo, tus hackeos matemáticos y tu todo.

English. I thank, in the first place, my advisors Manuel and Jose, without whom this document would not exist. Thank you for your support, commitment, teachings, and final pushes that you have provided me with all these years, from the first day that I told you that I wanted to do a master thesis "about that abstract interpretation thing" until now. You have taught me, aside from research skills, patience and perseverance. Thanks for being so generous and all the opportunities that you have given me.

I thank the thesis committee members Patrick Cousot, John Gallagher, Roberto Giacobazzi, Arie Gurfinkel, Yahong Annie Liu, Manuel Carro, and Pedro Lopez-Garcia. I thank all the people that contributed selflessly to this thesis project, also including the anonymous reviewers (and also reviewer 2). Thanks to Patrick Cousot and Arie Gurfinkel for reviewing this document and their invaluable comments. Thanks to Roberto Giacobazzi for teaching me computability and the very interesting discussions. Thanks to all my co-authors.

I thank the members of the CLIP research group, each of you have showed to me a different perspective to research: Nata, Max, Ignacio, Víctor, Miguel Ángel, Bish, Pedro, Manuel C, Joaquín y Umer. I want to thank specially Nata for sharing with me her wisdom and helping me from the very first day that we shared our office. Thank you also for showing me what to expect.

I thank Francisco Bueno (R.I.P.) for bringing logic programming and abstract interpretation to me.

I thank all researchers all researchers at IMDEA Software, specially those that shared lunch with me. Just to name a few: Germán, Srdjan, Platon, Miriam, Miguel, Elena, Pedro, Borja, Alejandro, Chana e Ignacio. Thanks as well to those that brought cake.

Thanks Ignacio for taking so much time to help me, for your patience, your kindness and for sharing your experience, without which this thesis would not have been possible.

I thank Bruno Dutertre, Ashish Gehani, and, specially, Jorge Navas for having me at SRI International and teaching me other ways of tackling research, and Arie Gurfinkel for his guidance and advice.

Thanks to the whole administrative team at IMDEA Software for their kindness and help provided all these years. Thanks also to Irene and Pilar.

Thanks to all the women in science that came before me, making the path easier and friendlier for those of us that come after. Thanks also to all of them that shared with me the road to the PhD: Alba, Alejandra, Ana, Anaïs, Betxu, Chana, Elena, Kyveli, Marta, Marta, Marta, Miriam, Nata, Niki, and Silvia.

Thanks to Liss, Alba, Alejandra, and Roi the beers and the reliefs.

Thanks to my friends in undergrad Pablo, Mónica, and Javi. And to the HPCN lab, where I first started doing research, especially Rafa, Juan, Rubén, Jose, and Sergio.

I thank my high school teachers Ricardo Fernández and Luis de Peña for arousing my interest in learning how and why things work.

I thank my family for helping me to get here. Thanks to my mother, my father, my brother Ginés, my aunt Luci, my grandmothers Ana and María for keeping me company during the bus traffic jams, and to my grandfathers even though they are not with us any more. Thanks, Isa, Javier, and Javi for your support and for the trip to buy turrón.

Thank You, Diego, for your continuous support, your optimism, your math hacks, and your everything.

This thesis has been partially funded by the FPU grant 16/04811 of the Spanish Ministerio de Educación y Ciencia, MINECO TIN2015-67522-C3-1-R *TRACES* project, MICINN PID2019-108528RB-C21 *ProCode* project, the Madrid M141047003 *N-GREENS* and P2018/TCS-4339 *BLOQUES-CM* programs.

Contents

List of Algorithms

List of Figures

List of Tables

1

Introduction

> *I had a running compiler and nobody would touch it. . . . they carefully told me, computers could only do arithmetic; they could not do programs.*
>
> — Grace Hopper

In the past few years, the impact of software in our daily lives has vastly increased. Software is now present everywhere: from the industries that first used it (banking, nuclear, aeronautics, . . .), to medical and home appliances, cars, phones, and even government administrations. This naturally induced the creation of on-line platforms for hosting software projects, where the amount of code and programmers is massive and growing. Remarkable examples include GitHub with 56M+ developers, 3M+ organizations, 100M+ repositories; GitLab with 100K+ organizations; and SourgeForce with 502K+ projects. Lastly, the Software Heritage Project is a recent effort "*to collect all publicly available software*", which, so far, has crawled almost 10B source files, 2.2B+ commits, and 154M+ projects, from more than ten software archives. Not only is the number of projects growing fast, but so is their size and dependencies with external code.

Regardless of how critical a software is, full functionality and absence of incorrect or unexpected behaviors is of course always desired. The increasing importance of code reliability and robustness in both industrial and academic software projects is evidenced by the recent appearance and use of multiple tools and frameworks for *continuous integration/continuous deployment* (CI/CD) and code review, such as GitLab CI/CD pipelines, GitHub Actions, Azure DevOps, Jenkins, Phabricator, among others.

The continuous integration (CI) workflow aims to consider always the whole software project while developing a new feature or fixing a bug, as opposed to independently developing or maintaining one of its components. The process typically consists on triggering a set of *(semi-)automatic processes* to guarantee that every change to a project meets some standards. These processes range from syntactic convention checking to regression testing (checking if some functionality was lost), and

are organized in different and possibly independent stages in a pipeline. Additionally, other developers may review the code manually, interacting with the author of the change, who typically addresses the issues and resubmits changes, triggering again the whole process. The reader can immediately see that the benefits of this workflow in an evolving software project are significant. However, a drawback to this approach is that it can slow down the development process, as a manual (human) review takes time. Moreover, running software analysis tools may be quite expensive. Therefore it is crucial that the tools are as *efficient* as possible and that they scale up to whole software projects. Another drawback is that the analysis tools need to be configured and specifications need to be written, so the tools must ease these tasks and *automate* them as much as possible.

Taking CI one step further is known as *continuous delivery* and later *continuous deployment*. The first one extends CI to automatically deploy all code changes to a testing and/or production environment. This consists on an automated release process, where the application is later deployed at any time by clicking a button. Continuous deployment goes one step even further, as every change that passes all checks of the production pipeline is released directly to the customers. There is no human intervention, and only a failed stage will prevent a new change from being deployed. The reliability of these tools is, hence, essential as failure to comply with a contract may result in economic/client loss. In these cases, having a formal (mathematical) background theory to support the tool can help convince ourselves (and possibly clients) that the process is indeed trustworthy.

Finding mistakes before introducing them in projects is great but can be further improved if error detection is brought forward to code writing. In the past few years, many code analyzers have appeared in different IDEs, the vast majority are syntactic checkers, e.g., in Eclipse's marketplace 96 plugins are listed as "*Source Code Analyzer*"[1]; Intellij IDEA includes plugins for finding probable bugs, locating dead code, detecting performance issues, improving code structure and maintainability, conforming to coding guidelines and standards, and conforming to specifications[2]; Netbeans developed Code Inspect[3], *"a tool for finding potential problems and detecting inconsistencies"* in Java programs; and built-in analyzers are now included in Visual Studio[4], to improve code style and quality for `.Net` and `C++` programs. In this setting, performance is even more critical, as continuously reporting back to the programmers *on-the-fly* means having to reanalyze, as precisely as possible, in a matter of a few seconds.

1 https://marketplace.eclipse.org/taxonomy/term/14%2C31/title
2 https://www.jetbrains.com/idea/docs/StaticCodeAnalysis.pdf
3 https://netbeans.apache.org//kb/docs/java/code-inspect.html
4 https://docs.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2019

The trend pattern is clear: automatic source code analysis and verification is of great importance both at the level of software development and software maintenance. A wide range of tools has appeared including syntactic checkers, software dependency management, testing, documentation generation, static analyzers, and deployment/publishing. Specifically, *automatic static analyzers* allow inferring properties about software without actually executing it and without the need for human interaction. The underlying techniques behind these tools are of different natures, distributed into three main areas: *software engineering*, *empirical methods* and *formal methods*. *Software engineering* tools study the code from the perspective of software quality measurement and assessment. *Empirical methods* are typically statistical methods, and use large code bases to infer coding rules, to then check if a program differs from the inferred coding rules. These two are typically based on syntactic analysis.

Formal methods, on the other hand, are based on rigorous mathematical techniques that provide formal guarantees about the software [35]. Unfortunately, the use of formal method-based tools is not extended across all kinds of companies in practice, but rather localized to the software that is required to interact with critical systems. A number of possible causes have been identified [36, 76], e.g., higher computational cost, having to train the developers on the underlying formalisms, and/or the need of extensive program annotations or specifications.

On top of that, Rice proved in 1953 [153] that an algorithm cannot be designed to decide for all programs if they meet some functional property that is non trivial. Namely, there does not exist a non trivial functional property about a program (i.e., from an input/output perspective) that is computable. Due to this well-known impossibility result, and aiming to be *sound*, i.e., always give a correct result, static analysis tools must choose between *always producing an answer* or to being *complete* (i.e., finding all bugs).

Abstract interpretation is a framework that allows safely approximating all possible behaviors of programs. The mathematical characterization of all these possible behaviors of a program when executed for all possible input data is called the program's *semantics*. Abstract interpretation guarantees that the static analysis is effective (always produces an answer) if the mathematical abstraction meets some conditions. Given the undecidability result of Rice, effective abstract interpretations give up completeness for soundness and termination, that is, it always produces a correct answer, but with the caveat that the answer may be 'I do not know'. This thesis focuses on such abstract interpretations, but complete abstractions can be designed although they are not guaranteed to terminate. The approach of using static analysis as a program development tool was pioneered within the abstract interpretation community, and in particular by the Ciao system [82, 145, 83], which the work in this thesis extends.

Taking a step back to observe the common ground of static analysis tools, one can see the static software analysis process of a program as a black box that, given a program, infers a set of properties about the program. Tools then give different uses to these properties. For example, a *verifier* uses the properties to prove or disprove some specification about the program; an *optimizer* uses them to transform the program to improve some run-time characteristics, e.g., execution time, memory usage or security guarantees. Meanwhile, a *parallelizer* takes advantage of the properties to automatically generate a version of the program that can be run in several processes; a *documenter* uses the properties to produce some documentation or manual about a tool. Needless to say, any of the mentioned tools can be built directly, not necessarily in two steps of analysis and later processing. But the analysis part is common to all of them. This thesis contributes to improving the scalability of the analysis part that is relevant for all of such applications. In addition, we also address its relevance for verification.

Some contributions and findings of this thesis are written in terms of the logic programming paradigm. This logic programming approach to computing investigates the use of logic as a programming language and explores computational models based on controlled deduction. Deduction can hence be viewed as a form of computation in which deducing a statement is interpreted procedurally as deducing its premise, typically a list of other statements. An execution of a program then consists on determining whether a logical statement is a consequence of the program. Logic programming provides a framework that is useful both for programming and for reasoning about other programs and program semantics. This is due to the fact that program specifications are usually written as logic formulae. If such formulae can be expressed in a logic program then the program is correct 'by construction' but also machine-checkable, since the specification can be executed as part of the program. In fact, the techniques developed for logic programs have expanded beyond logic programming to a large variety of other programming languages, including imperative, functional, object-oriented, and concurrent ones [137, 77, 123, 71, 74, 75, 47, 94, 116, 138, 59, 45].

1.1 Static analysis of large code bases: scalability and its related challenges

In this thesis we address the performance and scalability issues that arise when analyzing large software projects, which can be divided into four specific goals.

Taking advantage of localized changes. Large, real-life programs typically have a complex structure combining a number of modules with system libraries. Global

analysis of such large code bases is expensive, and this is specially problematic in interactive uses of analyzers. In this scenario, triggering an entire reanalysis for each change set is often too costly. A key observation is that very often changes in the program are small and isolated inside a few components. This characteristic can be taken advantage of to reduce the cost of re-analysis in two ways: reusing as much information as possible from previous analyses, and avoiding the maintenance of analysis information for unaffected components. Thus, our first goal is to:

> *Design and evaluate static analysis algorithms that leverage program modularity and localization of changes to recompute program properties reusing previous results as much as possible.* [G1]

User-guided approximations. Approximations during program analysis are a necessary evil, as they ensure essential properties, such as soundness and termination of the analysis, but they also imply not always producing useful results. Automatic techniques have been studied to prevent precision loss, typically at the expense of larger resource consumption. In both cases, that is, when analysis produces inaccurate results and when resource consumption is too high, it is necessary to have some means for users to provide information to guide analysis and thus improve precision and/or performance. We want to study:

> *Techniques for supporting within an abstract interpretation framework a rich set of assertions to improve the precision and performance of fixpoint computation processes.* [G2]

Dealing with (incomplete) generic code. In modern coding it is rarely necessary to write everything from scratch. Modules and interfaces allow dividing the program into manageable and interchangeable parts. *Generic* components are a further abstraction over the concept of modules, introducing dependencies on other (not necessarily available) components implementing specified interfaces. They have become a key concept in large and complex software applications [172, 100, 51, 88]. Despite undeniable advantages, generic code is also *anti-modular*. Precise analysis requires such code to be instantiated with concrete implementations, potentially leading to expensive combinatorial explosion. In such cases both whole program analysis and user annotations can help alleviate the anti-modularity nature of generic code. An initial specification of the generic component provided by the user that can be later refined when the code is available can be a good compromise to achieve a precise, and yet not so expensive analysis while the application is being developed. Thus we are interested in studying:

> *Techniques for a precise analysis of programs involving generic code that can be guided by programmer annotations.* [G3]

Coping with imprecision. Program analysis is concerned with intensional properties not just because semantically equivalent programs may exhibit different abstract properties, but also because semantically different programs may appear identical abstractly. This familiar phenomenon can be overcome by increasing the precision of program analysis; but the incurred costs make it into one of the main challenges in program analysis, and into one of the main obstacles to the scalability of general-purpose program analysis. Being aware of the precision flaws of an abstract interpretation is equivalent to studying its completeness. Completeness encodes the greatest achievable precision when abstracting the concrete behavior of a program on an abstract interpreter. Here, the only loss of precision is due to the *expressivity* of the abstract domain, and not the abstraction functions or the abstract interpreter. Incomplete abstract interpretations give rise to weaker properties than those encountered under direct inspections of effective computations of the program. This means that in debugging, abstract interpretations yield false alarms. Studying the classification of programs in terms of being complete/incomplete for an abstract interpretation would shed some light in the systematic removal or introduction of false alarms. Our goal is then twofold:

> *Adapt the concept extensional (functional) equivalence of programs to abstract equivalences induced by abstract interpretations as a new family of index sets for partial recursive functions. Use this equivalence to study the classes of complete and incomplete programs for a given abstract interpretation and how they are related.* [G4]

1.2 Contributions of the thesis

This thesis makes the following contributions, related to the aforementioned goals:

- We generalize the notion of extensional (functional) equivalence of programs to *abstract equivalences* induced by *abstract interpretations*. The generalized framework gives rise to interesting and important new properties, and allows refined, non-extensional analyses. In particular, since programs turn out to be extensionally equivalent if and *only if* they are equivalent just for the concrete interpretation, it follows that any non-trivial abstract interpretation uncovers some intensional aspect of programs (G4). In collaboration with R. Bruni, R. Gori, R. Giacobazzi, and D. Pavlovich [21].

- We introduce a novel technique for building *cliques* of extensionally equivalent yet abstractly distinguishable programs. The obtained results also shed a new light on the relation between the techniques of code obfuscation and precision in program analysis (G4). In collaboration with R. Bruni, R. Gori, R. Giacobazzi and D. Pavlovich [21].

- We present a new formalization of existing (incremental and modular) abstract interpretation fixpoint algorithms for logic programs. We extend the description by including explicitly the description of the extrapolation steps via widening, which was left implicit in previous work. Then we provide some new theoretical results about the correctness and precision of such algorithms and how incrementality may affect analysis results (G1). In collaboration with J. F. Morales and M. Hermenegildo [64].

- We propose an algorithm that performs goal-directed, top-down, multivariant, and incremental abstract interpretation of *modular* programs. The analyzer takes a program (target), a set of initial call states, and, optionally, analysis results of a previous version of the program, and information about the changes w.r.t. the target program. We also provide theoretical results about the correctness and precision (G1). In collaboration with J. F. Morales and M. Hermenegildo [64].

- We present techniques for supporting within an abstract interpretation framework a rich set of assertions that can deal with multivariance/context-sensitivity, and can handle different run-time semantics for those assertions that cannot be discharged at compile time. We show how the proposed approach can be applied to both improve precision and accelerate analysis. We also provide some formal results on the effects of such assertions on the analysis results (G2). In collaboration with J. F. Morales and M. Hermenegildo [61].

- We propose a simple encoding of generic programs as *traits*, for logic programming using *open* predicates and assertions (G3). In collaboration with J. F. Morales and M. Hermenegildo [62].

- We extend the modular, incremental analysis algorithm to react incrementally not only to changes in program clauses, but also to *changes in the assertions*, upon which large parts of the analysis graph may depend (G3). In collaboration with J. F. Morales and M. Hermenegildo [62].

- We show how the techniques presented for incremental analysis can be used to perform *on-the-fly* assertion checking to give continuous semantic feedback to the programmer as the program is edited (G1). In collaboration with

M. A. Sanchez-Ordaz, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M. Hermenegildo [160], contributing specially to the integration of incremental analysis and static verification.

- Finally, we have implemented and included all the algorithms in the CiaoPP framework and performed an extensive experimental evaluation (G1, G2, G3). In collaboration with J. F. Morales and M. Hermenegildo [64, 60].

1.3 Overview of the thesis

Chapter 2 briefly introduces the preliminaries relevant to this thesis. Chapter 3 tackles the problem of imprecision and completeness in abstract interpretation, studied from the point of view of computability theory, by extending the notion of functional equivalence of programs to abstract equivalence of programs. In Chapter 4 we provide a description of the existing baseline incremental and modular fixpoint computation algorithms which will serve as a basis for the rest of the algorithms presented in the thesis. This includes a new formalization of their correctness properties. Chapter 5 presents an algorithm for modular and incremental fixpoint algorithm together with its theoretical properties. Chapter 6 presents the proposed algorithm that includes guidance using assertions. Chapter 7 presents an algorithm that reacts incrementally to both changes in assertions and in program clauses. Chapter 8 presents the experimental evaluation of the proposed fixpoint algorithm, together with some case studies. Chapter 9 presents the application of incremental analysis to compile-time assertion checking, and shows the feasibility of the approach to give continuous verification feedback through an IDE. Chapter 10 concludes the thesis and describes lines of future work.

2

Background

This chapter presents the basic notions the thesis is based on.

2.1 (Constraint) Logic Programs

The logic programming approach to computing investigates the use of logic as a programming language and explores computational models based on controlled deduction. Deduction can be viewed as a form of computation in which deducing a statement is interpreted procedurally as deducing its premise, typically a list of other statements. Therefore, the execution of a program consists on determining whether a logical statement is a consequence of the program. Logic programming provides a framework that is useful both for programming and for reasoning about other programs and other program semantics. This is due to the fact that program specifications are usually written as logic formulae. If such formulae can be expressed in a logic program then the program is both correct "by construction" and machine-checkable, since the specification can be executed as part of the program.

A Constraint Logic Program (CLP) is a set of *clauses* of the form:

$$H \texttt{ :- } A_1, \ldots, A_n$$

where $A_1, \ldots, A_n$ are *literals* that form the *body* and H is an *atom* said to be the *head* of the clause. This is equivalent to a logic formula:

$$A_1 \wedge \ldots \wedge A_n \;\rightarrow\; H$$

with all variables in the clause universally quantified. An atom is *normalized* if it is of the form $p(x_1, \ldots, x_m)$ where p is an m-ary predicate symbol and $x_1, \ldots, x_m$ are distinct variables. A set of clauses with the same head is called a *predicate* (or *procedure*). To refer to predicates we use normalized atoms or `p/N` where `p` is the name of the predicate and `N` is its *arity*, i.e., its number of arguments. The *literals* in the body are either predicate calls or *primitive constraints* (also called *built-ins*).

A *primitive constraint* is defined by the underlying domain(s) and is of the form $c(e_1, \ldots, e_k)$ where c is a k-ary predicate symbol and the $e_1, \ldots, e_k$ are expressions. A *fact* is a clause whose body is true. When CLP programs are used for verification purposes only, i.e., they are not intended to be executed, the term *Constrained Horn Clauses* (CHCs) is often used in the literature. See [45] for a recent survey of how the two research communities are related. In the examples we use `Prolog` syntax, therefore variables are encoded using capital letters, and implication ($\leftarrow$) is denoted with the symbol "`:-`".

Example 2.1 (A logic program)**.**
The program on the right computes the parity of a list of 1s and 0s, i.e., the exclusive or of all the elements. The parameters in a call `par(L,P0,P)` are a list `L`, an initial value of the parity, `P0`, and the parity, `P`, of the elements in `L` together with `P0`. The predicate `xor(A,B,C)` succeeds if `C` is the XOR operation of `A` and `B`. Note that no primitive constraints are involved in this program.

```
1 par([], P, P).
2 par([C|Cs], P0, P) :-
3     xor(C, P0, P1),
4     par(Cs, P1, P).
5
6 xor(0,0,0).
7 xor(0,1,1).
8 xor(1,0,1).
9 xor(1,1,0).
```

The predicate `par(L,P0,P)` defines the parity in a recursive manner. The base case (line 1) is a fact, and states that if `L` is the empty list, the parity `P` is `P0`. In the recursive case (lines 2-4), that is, if the list has one or more elements, the `xor` of `P0` and `C`, the character currently being processed, is obtained in `P1` (line 3) and then used to compute the parity of the rest of the list, `P` (line 4). `xor/3` is defined by declaring all four cases as facts (lines 6-9).

For presentation purposes, the heads of the clauses of each predicate in the program are referred to with a unique subscript attached to their predicate name (the clause number), and the literals of their bodies with dual subscripts (clause number, body position), e.g., A_k `:-` $A_{k,1}, \ldots A_{k,n_k}$. For the program in Example 2.1, $\texttt{par/3}_1$ denotes the head of the first clause of `par/3`, i.e., `par([], P, P)` and $\texttt{par/3}_{2,2}$ denotes the second literal of the second clause of, i.e., the recursive call `par(Cs, P1, P)`.

A *query* to a logic program is a means of executing it, and is equivalent to asking if a relation is defined in the program. A query is a pair of an atom and a set of constraints, e.g., $Q = \langle$`par([0, 1], 0, P)`$, true\rangle$ (same as $Q = \langle$`par(L, 0, P)`$, \{$`L = [0, 1]`$\}\rangle$). For queries we also use the `Prolog` notation for conciseness: `:- par([0, 1], 0, P)`. Intuitively, the answers or solutions of a query are found by *executing* each of the clauses whose head matches with the atom in the query. The variables of the query are *renamed* to be expressed in terms of the variables of the clause. Then, this resulting constraint is used to find whether the body of the clause holds, which is equivalent to executing the body by executing each of its literals. If the literal is a

primitive constraint, it is interpreted in the corresponding theory. If it is a predicate, it is treated as a new query, and the whole process is repeated again.

Example 2.2 (Queries to a logic program)**.** The predicate `par/3` in Example 2.1 may be queried to:

1. compute the parity of a list, for example with the query `:- par([0,1],0,P)`, whose answer is (*true* if) `P = 1` ;
2. find lists of a given parity, e.g., `:- par(L,0,1)`, which has an unbounded number of answers, e.g., `L = [1]`, `L = [1,0]`, `L = [1,1,1,1,1]`, ...;
3. check that a list is of a certain parity, e.g., `:- par([0,1],0,0)`, whose answer is *false*.

2.1.1 *Concrete Semantics.*

We use a semantics of (Constraint) Logic Programs that is query-dependent, or "top-down", based on *Selective Linear Definite* (SLD) resolution [154], and its generalization to Constraint Logic Programming [89, 122], where the standard logic programming domain of Herbrand terms with unification is seen as a constraint system that solves equality constraints over these terms, and is generalized by allowing other constraint domains and constraint solving procedures. The results of [89] imply that, provided certain defined conditions are met by the constraint domains used, all the standard results for logic programming carry over to constraint logic programming. Based on these results, and without loss of generality, we often phrase the discussion in terms of substitutions, i.e., equality constraints over the Herbrand domain. However, the results also apply to other constraint domains. We reduce the discussion for simplicity to top-down, left-to-right execution of CLP, although it can be generalized easily to other computations and search rules.

The traditional description of the resolution procedure [112, 7, 89] builds a tree structure in which the nodes contain *resolvents*. However, when used as a basis for top-down program analyses, this construction is typically adorned so that nodes in the resolution tree include representations of the constraints both before and after completing the branch in which they appear. These are then called the *call* and *success* states for that node, also called *call* and *success substitutions* (of the states). This is because the aim of goal-directed, top-down program analysis is to obtain information on the constraints before and after each program point. This idea of splitting call and success states is present in the notion of generalized AND trees of [22], that we use as basis for the main semantic representation structure used throughout the thesis, *analysis graphs* [126]. In the following we recall some basic

notation and notions of the resolution procedure, as well as how to represent the execution of a CLP program first as AND trees as then as generalized AND trees. We follow [112, 7] and [22].

Resolution Basics. A *substitution* is a set $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ with V_i distinct variables and t_i terms. We usually denote (concrete) substitutions by θ or σ. We say that t_i is the *value* of V_i in θ. The set $\{V_1, \ldots, V_n\}$ is the *domain* of θ; the *range* is the set of variables appearing in $t_1, \ldots, t_n$. By $vars(t)$ we denote the set of variables occurring in t, by $vars(\theta)$ the union of the domain and the range of θ. The composition of two substitutions σ and θ is denoted by $\sigma\theta$. A *resolvent* is represented by $\leftarrow (A_1, \ldots, A_n)\sigma_i$ where $\sigma_i = \theta_1 \ldots \theta_i$ is the composition of the substitutions applied so far (the *accumulated* substitution). For the initial resolvent (the query) we have $\sigma_0 = \epsilon$, the empty substitution. To perform a logical inference, the computation rule selects the leftmost literal $A_1\sigma_i$. The search rule then selects a clause $H \leftarrow (B_1, \ldots, B_m)$, renames it, and unifies the head H with $A_1\sigma_i$. If the unification is successful with most general unifier θ_{i+1}, then the resolvent $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\sigma_{i+1}$ is derived with $\sigma_{i+1} = \sigma_i\theta_{i+1}$. The new state is the *immediate successor* of the previous one. Because a literal can match different clause heads, a resolvent can have several immediate successors.

AND *trees.* It is common to represent resolvents as proof trees, also referred to as AND trees, where the literals in the resolvent (calls) are the leaves of the tree, also showing the origin of the calls. A *sequence* $\ldots t_i, t_j \ldots$ of AND trees is such that t_j is the immediate successor of t_i. The set of all AND trees which can originate from a given set of queries specifies completely the procedural behavior of a program for that set of queries. Fig. 1 illustrates this representation of proof states (i.e., resolvents) as AND trees, and program execution as a sequence of such states.

Generalized AND *trees.* An AND tree contains more information than needed for analysis purposes; it shows the whole state of the computation (all variables). To characterize the procedural behavior, it suffices to know the instance of procedure calls immediately before their execution and immediately after their completion. This is why [22] suggests representing a sequence of successive AND trees using a *generalized AND tree*. These trees are built starting with an initial node, which is the query Q, adorned on the left with a substitution θ, and with the domain of θ a subset of $vars(Q)$. θ is then called the *call substitution* of Q, and the rest of the tree is built by expanding the leaf nodes in the following way:

- If L is a leaf adorned on the left with the call substitution θ_i, then:

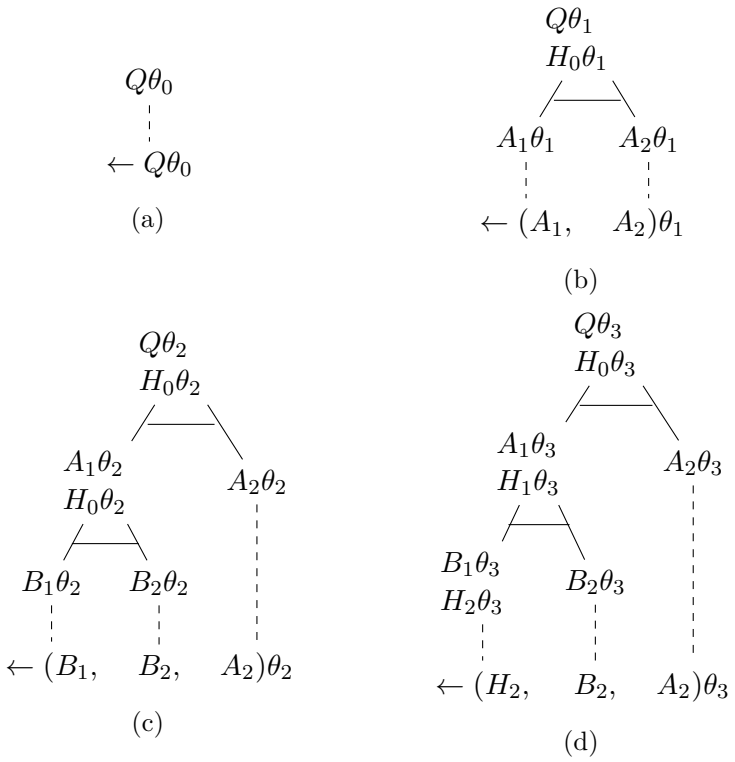


Fig. 1: Some successive AND trees. Figure adapted from [22].

  - If C is a properly renamed clause, $H \leftarrow B_1, \ldots, B_n$, defining L, then the tree is expanded by pairing L with H and adding the calls $B_1, \ldots, B_n$ as children of L. B_1 is adorned on the left with the call substitution θ_{i+1}. The domain of θ_{i+1} is $vars(H \leftarrow B_1, \ldots, B_n)$. If the clause C is a fact, θ_{i+1} adorns an empty body and is also the *success substitution* of the body.
  - If L is a built-in, then the tree is expanded by adorning L on the right with a substitution θ_{i+1}. The domain of θ_{i+1} is $vars$(clause of L). θ_{i+1} is the success substitution of L. With L the last call of its clause, θ_{i+1} is also the success substitution of the body; otherwise it is the call substitution of the next call.

- If a node (call) L is adorned on the left with a call substitution but not adorned with a substitution on the right, such that L is the parent of a clause body with success substitution θ_i. The tree is expanded by adorning L on the right with a substitution θ_{i+1}. The domain of θ_{i+1} is $vars$(clause of L). θ_{i+1} is the

success substitution of L. With L the last call of its clause (the query), θ_{i+1} is also the success substitution of the body (the query); otherwise it is the call substitution of the next call.

To obtain the whole tree of a successful SLD derivation of a program the previous steps must be repeated until the root node of the tree is adorned on the right with the success substitution. Note that for a given query a number of trees may exist, considering the different clauses that may be unifiable.

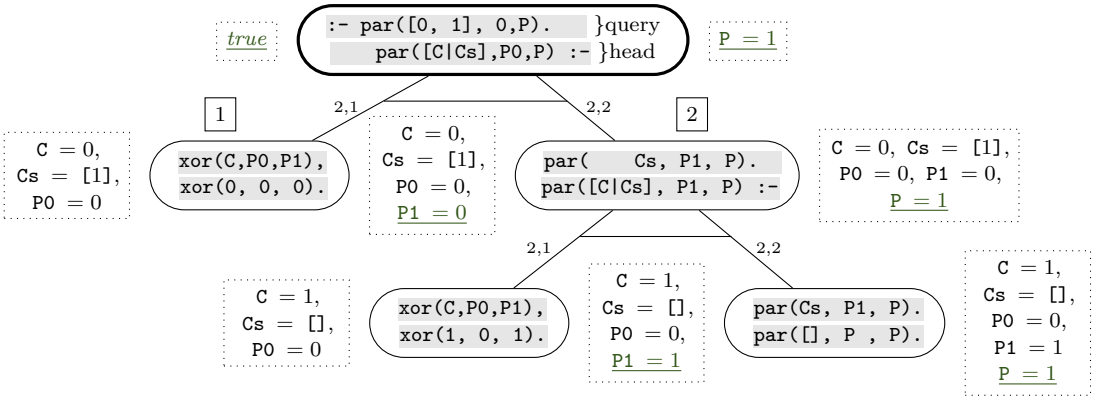
Generalized AND trees are specially convenient for analysis. Accumulated substitutions in a concrete domain, can result in having an unbounded number of variables. Also, it becomes easy to compare substitutions adorning different instances of the same clause, something which is essential for the treatment of recursive clauses. The only additional requirement is that procedure exit must recover a different restriction of the same accumulating substitution.

It is also often interesting to consider trees with also OR nodes, i.e., *AND-OR trees*, rather than considering sets of AND trees. *AND-OR trees* capture the semantics in a useful way for analyses such as determinacy [115, 99], cardinality [19] and non-failure [49], among others, but for simplicity we will limit the discussion herein to concrete semantics based on generalized AND trees.

Since generalized AND trees will form the basis of the concrete semantics used in the thesis we will sometimes refer to generalized AND trees simply as trees.

Concrete Semantics. The *concrete semantics* $[\![P]\!]_Q$ of a program P for a given set of queries Q is the set of trees that represent the execution of the queries in Q for P. We will represent queries (i.e., initial resolvents) as $Q = \langle A, \theta^c \rangle$, meaning the resolvent $\leftarrow (A_1, \ldots, A_n)\theta^c$, i.e., A is a normalized atom or conjunction of atoms and θ^c is the initial substitution. We refer to the nodes in the tree with $\langle p(V_1, \ldots, V_n), \theta^c, \theta^s \rangle$, where p is a predicate in P, and θ^c, θ^s are, respectively, the call and success substitutions over the variables $V_1, \ldots, V_n$. Nodes that are part of failing (or looping) branches have empty success fields, i.e., are of the form $\langle p(V_1, \ldots, V_n), \theta^c, \emptyset \rangle$.

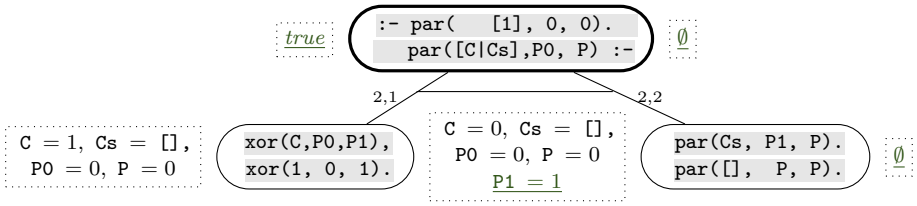
Example 2.3 (Generalized AND tree)**.** Let P be the program in Example 2.1. The following is the tree for the only successful SLD derivation of the query $Q = \{\langle$`par([0, 1], 0, P)`$, \mathit{true}\rangle\}$, i.e., the tree in $[\![P]\!]_Q$:



Nodes in this tree represent calls to predicates. Both the query and the head with which it is unified is depicted. The indices in the edges store the literal and clause to which the child corresponds. Dotted boxes describe the constraints at that point in the derivation in terms of the variables in the body. Constraints underlined and in green are the ones newly discovered by the execution of a predicate. When the query is performed, the second clause of `par/3` is selected, adding as children the two literals in the body. The first literal is adorned on the left with the substitution that resulted from renaming (`C` $= 0$, `Cs` $=$ `[1]`, `P0` $= 0$) and the expansion step is finished. Next, node $\boxed{1}$ is expanded, by unifying it with the only fact of `xor/3` that unifies, and this results in adorning it on the right with the substitution immediately after the execution of the literal (including `P1` $= 0$). Then, node $\boxed{2}$, the second literal of the clause, is processed, again selecting the second clause of `par/3`. This consists on expanding the node to have 2 child nodes, repeating an equivalent process. When this subtree is fully expanded and node $\boxed{2}$ is adorned on the right, the tree is finally obtained by adorning the initial query on the right after renaming.

Note that in general there are a number of unsuccessful trees, for all combinations of clauses whose head does not unify or leads to deriving false.

Example 2.4 (Generalized AND tree of a query that does not succeed)**.** Let P be the program in Example 2.1. The following is an execution tree for the query $Q = \{\langle$`par([1], 0, 0)`$, \mathit{true}\rangle\}$, which always fails (the answer is false):



Consulting the trees. The *calling context* of a predicate given by the predicate descriptor A defined in P for a set of queries Q is the set $\mathsf{calling\_context}(A, P, Q) = \{\theta^c \mid \exists T \in [\![P]\!]_Q$ s.t. $\exists \langle A', \theta^c, \theta^s\rangle$ in $T \wedge \exists \sigma A' = \sigma(A)\}$, where σ is a *renaming* substitution over variables in the program, i.e., a substitution that replaces each variable in the term it is applied to with distinct, fresh variables. In the following we use σ to denote such renaming substitutions. We denote by $\mathsf{answers}(P, Q)$ the set of answers (success constraints) computed by P for queries Q, i.e., $\mathsf{answers}(P, Q)$ is $\{\theta^s \mid s.t.\ \exists T \in [\![P]\!]_Q \wedge \langle A, \theta^c, \theta^s\rangle = root(T)\}$. We denote by $\mathsf{children}(P, Q, \langle A, \theta\rangle)$ the set of tuples of child nodes of $\langle A, \theta\rangle$ together with their position in the clauses (k) and literals (l) that define the predicate $(\langle B, \theta_B\rangle, k, l)$ of computed by P for queries Q. A sequence of calls of the form $\langle A_1, \theta_1\rangle \rightarrow_{k_1,l_1} \langle A_2, \theta_2\rangle \rightarrow_{k_2,l_2} \ldots \rightarrow_{k_{n-1},l_{n-1}} \langle A_n, \theta_n\rangle$ is a *path* in a tree iff for all $1 \leq i < n$ $\exists(\langle A_{i+1}, \theta_{i+1}\rangle, k_i, l_i) \in \mathsf{children}(P, Q, \langle A_i, \theta\rangle)$.

Example 2.5 (Calling context of a predicate)**.** Let P be the program in Example 2.1 and $Q = \{\langle$`par([0, 1], 0, P)`$, true\rangle\}$. The calling context of `xor/3`, i.e., how the predicate is called if the program is executed with Q, $\mathsf{calling\_context}($`xor(A,B,C)`$, P, Q)$ is the set of constraints: $\{(\mathtt{A} = 0, \mathtt{B} = 0), (\mathtt{A} = 1, \mathtt{B} = 0)\}$, one per node in the tree of Example 2.3.

Example 2.6 (Answers to a query)**.** Let P be the program in Example 2.1 and $Q = \{\langle$`par([0, 1], 0, P)`$, true\rangle\}$. The answers of the query in Q, $\mathsf{answers}(P, Q)$ is the set $\{(\mathtt{P} = 1)\}$.

Example 2.7 (Children of a node)**.** Let P be the program in Example 2.1 and $Q = \{\langle$`par([0, 1], 0, P)`$, true\rangle\}$. Let n be the node $\langle$`par(Cs, P1, P)`$, \{\mathtt{Cs} = [1], \mathtt{P1} = 0, \mathtt{P0} = 0\}\rangle$ in the tree of Example 2.3. The children of n, $\mathsf{children}(P, Q, n)$, is the set:

$$\{(\langle \texttt{xor(C, P0, P1)}, \{\mathtt{C} = 1, \mathtt{P0} = 0\}\rangle, \quad 2, \quad 1),$$
$$(\langle \texttt{par(Cs, P1, P)}, \{\mathtt{Cs} = [], \mathtt{P1} = 1, \mathtt{P} = 1\}\rangle, \quad 2, \quad 2)\}$$

Example 2.8 (A path in a generalized AND tree)**.** The following path is a valid path in the tree of Example 2.3:

$$\begin{array}{l}\langle \texttt{par([0,1],0,P)}, \mathit{true}\rangle \rightarrow_{2,2} \\ \quad \langle \texttt{par(Cs,P1,P)}, (\texttt{Cs} = [1], \texttt{P1} = 1)\rangle \rightarrow_{2,1} \\ \quad\quad \langle \texttt{xor(P,P0,P1)}, (\texttt{C} = 1, \texttt{P0} = 0)\rangle\end{array}$$

2.2 Abstract Interpretation

Abstract interpretation [37, 38] has been developed as an effective method for constructing sound-by-construction program analysis tools. The underlying idea is to *extract properties of a program by approximating its semantics*, which is implemented by interpreting program structures in an abstract domain. Let us first define some basic notation.

Sets and order. Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \setminus T$ the set-difference between S and T, with $S \subset T$ strict inclusion, with $S \subseteq T$ inclusion, with $\overline{S}$ set complementation, and with $|S|$ the cardinality of S. A set S is finite if $|S| < \omega$. A set L with ordering relation $\sqsubseteq$ is a *poset*, and it is usually denoted as $\langle L, \sqsubseteq\rangle$. A poset $\langle L, \sqsubseteq\rangle$ is a *lattice*, denoted $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot\rangle$, if $\forall x, y \in L$ we have that the *least upper bound (lub)* $x \sqcup y$, the *greatest lower bound (glb)* $x \sqcap y$, the *greatest element* (top) $\top$, and the *least element* (bottom) $\bot$ belong to L. It is *complete* when for every $X \subseteq L$ we have that $\bigsqcup X, \bigsqcap X \in L$. We use subscripts, like in $\bot_L$, $\top_L$, or $\bigsqcup_L X$ to disambiguate the underlying lattice when it is not evident from the context.

Functions. Given $f : S \to T$ and $g : T \to Q$ we denote with $g \circ f : S \to Q$ their composition, i.e., $(g \circ f)x = g(f(x))$. We let $\mathtt{id}_S : S \to S$ be the identity function over S, and omit the subscript S when clear from the context. A function $f : L \to D$ on complete lattices is *additive (resp. co-additive)* if for any non-empty $Y \subseteq L, f(\sqcup_L Y) = \sqcup_D f(Y)$ (resp. $f(\sqcap_L Y) = \sqcap_D f(Y)$). Continuity is kept when f preserves *lubs* of increasing (non-empty) chains. If $f : L \to D$ we overload the notation by writing $f : \wp(L) \to \wp(D)$ for the additive extension of f to sets of values (i.e., for any non-empty $S \in \wp(L)$ we have $f(S) = \{f(v) \mid v \in S\}$). For a continuous function f, the least fixed point $\mathit{lfp}(f) = \sqcap\{x \mid x = f(x)\} = \sqcup_{n\in\mathbb{N}} f^n(\bot)$ where $f^0(\bot) = \bot$ and $f^{n+1}(\bot) = f(f^n(\bot))$. Note that additive functions are continuous.

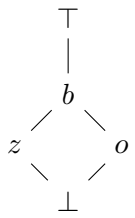


Fig. 2: Hasse diagram of the *Bits* lattice.

2.2.1 Abstract domains

Let C (*concrete*) and A (*abstract*) be complete lattices. Values in A and C are related via a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$. A pair (α, γ) forms an *adjunction* or a *Galois connection* between C and A if for any $c \in C$ and $a \in A$ we have $\alpha(c) \sqsubseteq_A a \Leftrightarrow c \sqsubseteq_C \gamma(a)$. The function α (resp. γ) is the *left-adjoint* (resp. *right-adjoint*) to γ (resp. α) and it is additive (resp. co-additive). A Galois connection such that $\alpha \circ \gamma = \mathtt{id}_A$ is called a *Galois insertion*.

Given a Galois connection, we call $\mathsf{A} = \langle A, \sqsubseteq, \sqcup, \alpha, \gamma \rangle$ an *abstract domain*, with join operator $\sqcup$. An abstract domain satisfies the *ascending chain condition* (*ACC*) if it has no infinite ascending chain. In such cases the fixpoint of any monotone function can be effectively computed in a finite number of steps, or by the use of a *widening* operator ∇ [37].

Example 2.9 (Abstracting constants 1 and 0)**.** Let the concrete domain of integers be $D_{\mathsf{Int}} = \langle \wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z} \rangle$, let $C_{Bits} = \{\bot, z, o, b, \top\}$, and $\gamma : C_{Bits} \to \wp(\mathbb{Z})$ and $\alpha : \wp(\mathbb{Z}) \to C_{Bits}$ be defined as

$$\gamma(x) = \begin{cases} \emptyset & \text{if } x = \bot \\ \{1\} & \text{if } x = o \\ \{0\} & \text{if } x = z \\ \{0,1\} & \text{if } x = b \\ \mathbb{Z} & \text{otherwise} \end{cases} \qquad \alpha(x) = \begin{cases} \bot & \text{if } x = \emptyset \\ o & \text{if } x = \{1\} \\ z & \text{if } x = \{0\} \\ b & \text{if } x = \{0,1\} \\ \top & \text{otherwise} \end{cases}$$

Let $\sqsubseteq_{Bits} \subseteq (C_{Bits} \times C_{Bits})$ be defined as $x \sqsubseteq_{Bits} y$ iff $\gamma(x) \subseteq \gamma(y)$. The Hasse diagram of the lattice induced by $\sqsubseteq_{Bits}$ is shown in Fig. 2. The lattice $\langle C_{Bits}, \sqsubseteq_{Bits} \rangle$, together with γ can be used to capture the property "*the program returns values* 1 *or* 0". For instance, assuming that a program can return any natural number, the output of the abstract interpretation process would be a value in C_{Bits} corresponding to an over approximation to the set of actual values of the execution.

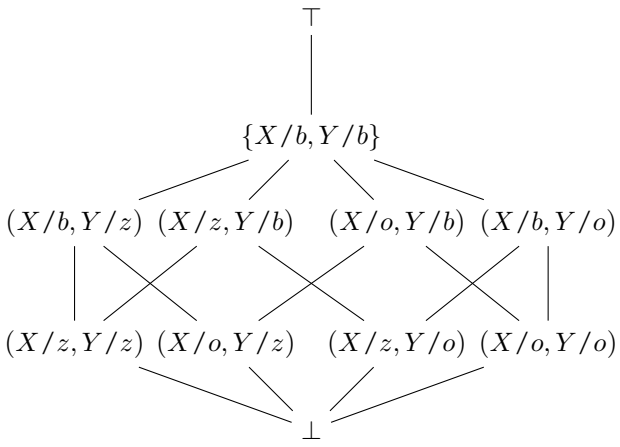


Fig. 3: An abstract domain using program variables X and Y, and the *Bits* lattice.

Being able to only represent input/output values of a program is very limiting, as, in general, abstract values are related to variables in the program. This is typically represented using sets of *Var/ AbstractVal* pairs.

Example 2.10 (Abstracting constants 1 and 0 for the variables in a program)**.** A domain to infer whether the variables in a program take values 1 or 0, D_{Bits}, is built using the lattice $\langle C_{Bits}, \sqsubseteq_{Bits} \rangle$. Given a program P, the set of values in the abstract domain is $A_P = \{X/v \mid X \in vars(P), v \in C_{Bits}\}$. That is, the Cartesian product of the program variables and the values in C_{Bits}. The ordering $\sqsubseteq_{D_{Bits}}$ is built extending $\sqsubseteq_{Bits}$ to the pairwise (i.e., variable per variable) comparison of the values assigned to each of the variables in $dom(\lambda), \lambda \in A_P$. Fig. 3 shows a lattice for an abstract domain for a program with two variables X, Y using C_{Bits}.

Note that, in general, in the *Var/ AbstractVal* pairs, abstract values can actually be any term, including variables, thus allowing the represention of *relational* properties, i.e., properties that represent relations between variables, e.g., "$x > y$". Also note that since the number of variables in the program is known (and finite), if we have a finite lattice, one can always build a finite lattice of *Var/ AbstractVal* pairs. Of course, this is only computable if the terms used as abstract values are finite.[1]

1 Using abstractions of this shape is not a requirement, but rather a general way of representing domains that abstract properties w.r.t. program variables. In fact, many implementations of abstract domains use other representations for efficiency.

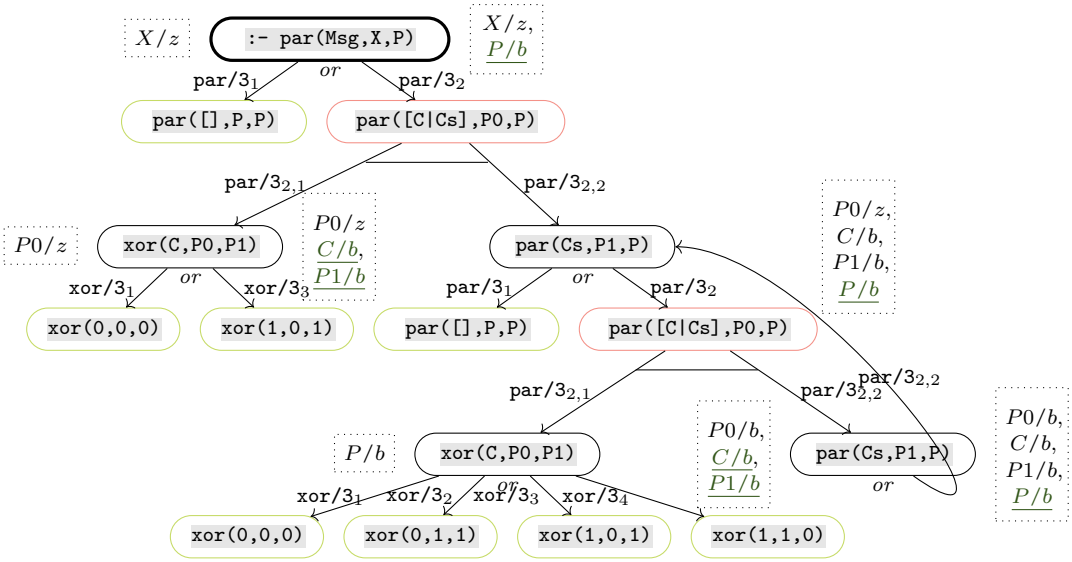


Fig. 4: AND-OR graph

2.3 Abstract Interpretation of (Constraint) Logic Programs

We perform *query-dependent abstract interpretation*, whose result is an abstraction of the generalized AND tree semantics. The purpose of this abstraction is to represent as a finite object the (possibly infinite) set of (possibly infinite) generalized AND trees in the execution of a CLP program. The input to the abstract interpretation process is the program P, an abstract domain D_α, and a set of initial *abstract queries* $Q_\alpha = \{\langle A_i, \lambda_i^c \rangle\}$, where each A_i is a normalized atom, and $\lambda_i^c \in D_\alpha$ (calls). Q_α defines the (typically infinite) set of concrete queries Q that the analysis must be correct for, $[\![P]\!]_Q = [\![P]\!]_{\gamma(Q_\alpha)}$. With some abuse of notation we represent the set $\gamma(Q_\alpha)$ as $Q = \{\langle A, \theta \rangle \mid \ \theta \in \gamma(\lambda) \wedge \langle A, \lambda \rangle \in Q_\alpha\}$.

To represent a set of generalized AND trees, Bruynooghe [22] proposes to encode choices among several clauses using OR nodes. Thus, the abstract semantics is called an *abstract AND-OR graph*, where (OR) nodes with cycles in the graph represent recursive calls, i.e., they abstract an infinite number of trees. When building such graphs, edges forming cycles are added whenever the same abstract call is inferred. If the domain is infinite, a generalization of the successive (increasing) abstract calls via a widening results in eventually finding the same abstract calls (and build the cycles).

Example 2.11 (AND-OR graph of Example 2.1)**.** Fig. 4 shows an AND-OR graph, using elements of D_{Bits} as abstract substitutions, for the abstract query $Q_\alpha = \{\langle$`par(Msg,X,P)`$, (X/z)\rangle\}$. Consequently, it abstracts the trees in Examples 2.3 and 2.4. Nodes in green (with no children) represent facts, nodes in pink are the *and* nodes, and they capture the execution of a *clause*. Nodes in black represent the *or* nodes, and represent the execution of a *predicate*. When an abstract call substitution is encountered twice, the call is tabulated and the abstract success substitution is reused. This is the case of the recursive call to `par/3` in the bottom right of the graph of Fig. 4.

The AND-OR graph example has been included for completeness, however, this work is based on a compact representation of the AND-OR graphs: *analysis graphs*. They were originally designed in [126, 127, 129] with performance in mind, and they store explicitly only the OR nodes, i.e., the *predicate calls*. Analysis graphs (Def. 2.1) are the elements in our abstract domain, and thus represent the abstract properties of programs.

An analysis graph is a (directed) call graph and a mapping function from predicate descriptors and call substitutions to answer substitutions, both elements of D_α. A *node* in an analysis graph represents that a call to a predicate ($\langle A, \lambda^c\rangle$) is possibly made, and it has an associated answer λ^s (if succeeds), through the mapping, $\langle A, \lambda^c\rangle \mapsto \lambda^s$, with $\lambda^c, \lambda^s \in D_\alpha$. This represents that *the answer pattern for calls to predicate A with calling pattern λ^c is λ^s*, and it implies that for any node in the concrete trees in $[\![P]\!]_Q$ of the form $\langle A, \theta^c, \theta^s\rangle$, there must exist a node $\langle A, \lambda^c\rangle \mapsto \lambda^s$ in the analysis graph such that $\theta^c \in \gamma(\lambda^c)$ and $\theta^s \in \gamma(\lambda^s)$. Therefore, analysis graphs must capture all the call–success pairs, i.e., all the nodes in the AND trees of the concrete semantics. For a given predicate A, the analysis graph may contain more than one node capturing different call situations. A call mapped to $\bot$ ($\langle A, \lambda^c\rangle \mapsto \bot$) indicates that all calls to predicate A with substitution $\theta \in \gamma(\lambda^c)$ either fail or loop, that is, they never succeed.

An *edge* in an analysis graph is of the form $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c\prime}\rangle$. This represents that *calling predicate A with calling pattern λ^c may cause predicate B to be called (via the literal $A_{k,i}$) with calling pattern $\lambda^{c\prime}$*. Correctness with respect to the concrete semantics requires that if in any concrete tree in $[\![P]\!]_Q$ the clause A_k is executed with a calling pattern θ^c that causes predicate B (the literal $A_{k,i}$) to be called with some calling pattern $\theta^{c'}$, then there must be an edge in the graph $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c\prime}\rangle$ and $\theta^c \in \gamma(\lambda^c)$, $\theta^{c'} \in \gamma(\lambda^{c'})$. These edges capture the dependencies between the immediate calls of a predicates, i.e., given a node in the tree, the immediately following nodes. For simplicity, we omit k, i when not relevant in the context.

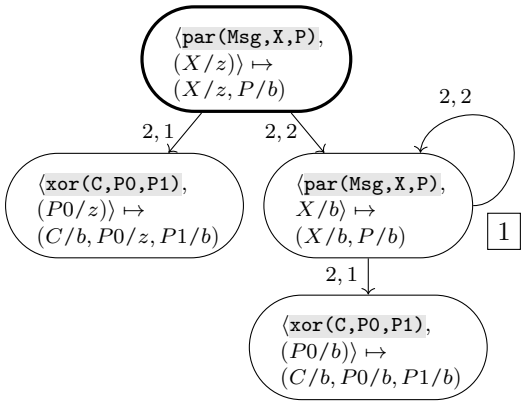


Fig. 5: The analysis graph that corresponds to the AND-OR graph of Fig. 4.

Example 2.12. Fig. 5 shows an analysis graph that corresponds to the AND-OR graph of Fig. 4. In the examples we mark with a bold outline the initial nodes (i.e., the calls in Q_α). *Node* $\boxed{1}$ $(\langle \texttt{par(Msg,X,P)}, (X/z)\rangle \mapsto (X/z, P/b))$ captures that `par/3` may be called with X bound to any in $\gamma(z) = \{0\}$ and, if it succeeds, the third argument P is bound to any of $\gamma(b) = \{1, 0\}$. Note that a different node (the one below) captures that there are other calls to `par` where X/b holds. The edges in the graph represent the $\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c\prime}\rangle$ relation. For example, two such edges exist starting at node $\boxed{1}$, denoting (right) that it may call `xor/3` and (below) that it may call itself with a different call substitution.

The information in the *AND-OR graph* can be reconstructed from the analysis graph by renaming and projecting abstract substitutions that appear in it, which keeps the information only at the predicate and literal level.

Note that if a predicate is called from different points of the program with the same abstract substitution there will be an edge from each of the clauses to that predicate. There may also be more than one edge between two nodes if the same predicate appears several times in the body of a clause or clauses. In these cases the edges necessarily differ by the tag *k,i*.

Multivariance (a.k.a., polyvariance, context- and path-sensitivity). As seen in the example, these analysis graphs allow representing the different call patterns encountered during the execution, separating the cases in which such calls differ, even if some of them subsume some others. This feature is traditionally referred to as *multivariance* in the context of analysis of logic programs, and here it serves two purposes:

```
1  % ⟨par(Msg, X, P), ⊤⟩ ↦ (X/z, P/b)
2  par([], P, P).
3  par([C|Cs], P0, P) :-
4      xor_1(C, P0, P1).
5      par_2(Cs, P1, P).
6
7  % ⟨par(Msg, X, P), (X/b)⟩ ↦ (X/b, P/b)
8  par_2([], P, P).
9  par_2([C|Cs], P0, P) :-
10     xor_2(C, P0, P1),
11     par_2(Cs, P1, P).
12
13 % ⟨xor(C, P0, P1), (P0/z)⟩ ↦ (C/b, P0/z, P1/b)
14 xor_1(0,0,0).
15 xor_1(0,1,1).            % After abstract partial eval.:
16 xor_1(1,0,1).            % xor_1(0,0,0).
17 xor_1(1,1,0).            % xor_1(1,0,1).
18
19 % ⟨xor(C, P0, P1), (P0/b)⟩ ↦ (C/b, P0/b, P1/b)
20 xor_2(0,0,0).
21 xor_2(0,1,1).
22 xor_2(1,0,1).
23 xor_2(1,1,0).
```

Fig. 6: Program specialization implicit in the analysis after *version materialization.*

1. *Precision:* Different calling patterns to the same predicate are stored depending from which exact clause and literal this predicate is called from and with which call pattern. This idea of storing multiple calling contexts in this way is used in recent implementations of context sensitivity in imperative program analyses (e.g., [97, 170]) where it is referred to as keeping *multiple value contexts.*

2. *Efficiency:* For the same literal and clause in the program, storing different calling patterns allows keeping the fixpoint computation localized to only those patterns that change.

While beyond the scope of this thesis, note also that multivariance is a form of *multiple specialization* of predicates. For example, the graph in Fig. 5 contains two *versions* of predicate `par/3` and another two of `xor/3`, and implies the specialization shown in Fig. 6. The tuples written as comments (lines 1, 7, 13, and 19) contain the corresponding inferred abstract substitutions (for call and success). This is referred to as *materializing* the versions in the analysis graph [129].

Reconstructing the paths of concrete executions. The analysis graph, through the *edges* ($\langle A, \lambda^c\rangle \rightarrow_{k,i} \langle B, \lambda^{c'}\rangle$) relation, provides an abstraction of the *paths* explored

in the concrete executions of the program, represented by the concrete trees. In particular, it is possible to reconstruct, for every node, all possible (and possibly infinite) execution trees that lead to the call pattern described by the node, by following the edges of the analysis graph. The analysis graph thus embodies two different abstractions (two different abstract domains): the graph itself is a *regular approximation* of the paths through the program, using a domain of regular structures. Separately, the abstract values (call and success patterns) contained in the graph nodes are finite representations of the states occurring at each point in the program paths, by means of the *data abstract domain*. Note that the path abstraction implicit in the graph is more powerful than the call stack representation in the well known call-strings method introduced in [163] (see, e.g., [97, 170] for two recent examples of use), as this method only keeps track of the *callers* of the abstracted call, and typically as a limited-length sequence [163], whereas the analysis graph approach captures, as a regular structure, all the arbitrarily large sequences of procedures *executed* before that call, i.e, not only its direct callers or a limited-depth sequence. As mentioned before, the analysis also includes the call patterns and paths leading to failure or non-termination in the concrete semantics, whose answer is $\bot$ (s.t. $\gamma(\bot) = \emptyset$).

Analysis graphs and paths. We introduce some basic notation for graphs. The following abstract transformers over an analysis result g allow us inspecting and manipulate analysis results.

$\boxed{\langle A, \lambda^c \rangle \in g}$: returns a boolean answer to whether there is a node in the call graph of g with key $\langle A, \lambda^c \rangle$.

$\boxed{\langle A, \lambda^c \rangle \mapsto \lambda^s \in g}$: returns a boolean answer to whether there is a node in g with key $\langle A, \lambda^c \rangle$ and the answer mapped to that call is λ^s.

$\boxed{n \rightarrow_{k,i} m \in g}$: returns a boolean answer to whether there are two nodes n, m in g and there is an edge from n to m with tag k, i.

$\boxed{\mathsf{del}(g, \{n_i\})}$: returns a graph after removing from g nodes n_i and its incoming and outgoing edges and unsets the element in the mapping function (it becomes undefined for all n_i).

$\boxed{\mathsf{upd}(g, n \mapsto \lambda^s)}$: returns a graph after overwriting in g the value of n in the mapping function. If a node with key n did not exist in g, it is added.

$\boxed{\mathsf{upd}(g, \{n \rightarrow_{k,i} m\})}$: returns a graph after adding to g an edge from node n to node m with tag k, i. If an edge already existed from n with the same tag, it is removed.

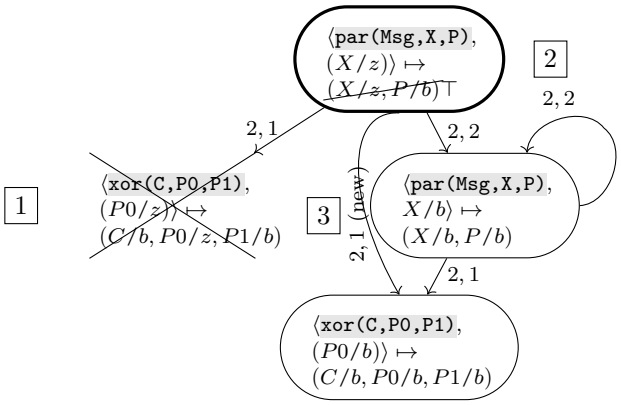


Fig. 7: Graph after the modification operations.

$\boxed{\mathsf{upd}(g, \{e_i\})}$: returns a graph after performing $\mathsf{upd}(g, e_i)$ for each element of $\{e_i\}$.

A sequence of calls of the form $\langle A_1, \lambda_1\rangle \rightarrow_{k_1,l_1} \langle A_2, \lambda_2\rangle \rightarrow_{k_2,l_2} \ldots \rightarrow_{k_{n-1},l_{n-1}} \langle A_n, \lambda_n\rangle$ is a *path* in a graph g iff for all $1 \leq i < n$ there exists an edge $\langle A_i, \lambda_i\rangle \rightarrow_{k_i,l_i} \langle A_{i+1}, \lambda_{i+1}\rangle$ in g.

Example 2.13. To illustrate the graph operations we show some examples of operations done to the analysis graph of Fig. 5, to which we refer with $\mathscr{A}$. The following operations do not modify the graph.

- Check if there is a call to `par/3` with the second argument as z $\langle$`par(Msg,X,P)`$, (X/z)\rangle \in \mathscr{A}$. This is true (node $\boxed{1}$).
- Check if there is a call to `par/3`, that, if it succeeds the second argument is a *bit*: $\langle$`par(Msg,X,P)`$, (X/z)\rangle \mapsto (X/z, P/b) \in \mathscr{A}$. This is true (entry node).
- Check if there is a literal with `xor/3` in any of the clauses of `par/3`: $\langle$`par(M,P0,P)`$, (P0/z)\rangle \rightarrow \langle$`xor(C,P0,P1)`$, (P0/b)\rangle \in \mathscr{A}$. This is false, there is a path from `par/3` to a node containing `xor/3` but there is not a direct call.

Example 2.14. The following examples perform graph modification operations over the analysis graph of Fig. 5, referred to again with $\mathscr{A}$.

- Remove the node for the abstract call $\langle$`xor(C,P0,P1)`$, (P0/z)\rangle$: $\mathsf{del}(\mathscr{A}, \{\langle$`xor(C,P0,P1)`$, (P0/z)\rangle\})$ (see $\boxed{1}$ in Fig. 7).
- Update the node for `par/3` with a more general success pattern: $\mathsf{upd}(\mathscr{A}, \langle$`par(Msg,X,P)`$, (X/z)\rangle \mapsto \top)$ (see $\boxed{2}$ in Fig. 7).

- Add an edge from node 1 to the remaining node for `xor/3`:
  $\mathsf{upd}(\mathscr{A}, \{\langle$`par(M,X,P)`$, (X/z)\rangle \to \langle$`xor(C,P0,P1)`$, (P0/b)\rangle\})$ (see 3 in Fig. 7).

After these operations, the state of the analysis graph is depicted in Fig. 7. Note that, since we have replaced all calling patterns with more general abstractions, this analysis graph approximates the behaviors of Fig. 5.

2.3.1 *Correctness*

We now introduce some definitions that are instrumental for determining correctness of analysis results. We first define a well-formed graph, which is valid for a given program and is restricted syntactically by it.

Definition 2.1 (A well-formed analysis graph for a program)**.** Let P be a program and $\mathscr{A}$ an analysis graph. $\mathscr{A}$ is *well formed* for P if:

- for every edge $\langle A, \lambda^c_A\rangle \to_{k,i} \langle B, \lambda^c_B\rangle \in \mathscr{A}$ there exists a clause k of predicate A, and B is the i-th literal of the clause, and
- it does not contain two different nodes with the same key $\langle A, \lambda^c\rangle$.

Note that the same graph may be well formed for several programs. E.g. if a graph is well formed for a program P and this program is extended with some clauses to form P', the graph is also well formed for P'. Also note that an empty graph is well formed for all programs. Henceforth, in the definitions we only consider well-formed analysis graphs. The following Definitions 2.2 to 2.5 determine how the concrete semantics of a program is correctly abstracted by a well-formed analysis graph.

Definition 2.2 (Correctly approximated calls)**.** Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ an analysis graph. We say that $\mathscr{A}$ *correctly approximates the calls in* $[\![P]\!]_Q$ if all encountered call patterns during the concrete execution are contained in $\mathscr{A}$. That is, for all predicates A in P:

$$\forall \theta^c \in \mathsf{calling\_context}(A, P, Q).\exists \langle A, \lambda^c\rangle \in \mathscr{A} \text{ s.t. } \theta^c \in \gamma(\lambda^c).$$

Definition 2.3 (Correctly approximated answers)**.** Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ an analysis graph. We say that *the answers in* $\mathscr{A}$ *correctly approximate the answers in* $[\![P]\!]_Q$ if they abstract all the answer patterns to the encountered call patterns. That is, for all predicates A of P:

$$\forall \langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}, \forall \theta^c \in \gamma(\lambda^c) \text{ if } \theta^s \in \mathsf{answers}(P, \{\langle A, \theta^c\rangle\}) \text{ then } \theta^s \in \gamma(\lambda^s).$$

Definition 2.4 (Correctly approximated call dependencies)**.** Given a program P and initial concrete queries Q, and an analysis graph $\mathscr{A}$. We say that $\mathscr{A}$ *correctly approximates the call dependencies in* $[\![P]\!]_Q$ if for every node $\langle A, \theta^c_A \rangle$ in $[\![P]\!]_Q$ for every child $(\langle B, \theta^c_B \rangle, k, i) \in \mathsf{children}(P, Q, \langle A, \theta^c_A \rangle)$, there is an edge:

$$\langle A, \lambda^c_A \rangle \rightarrow_{k,i} \langle B, \lambda^c_B \rangle \in \mathscr{A} \text{ s.t. } \theta^c_A \in \gamma(\lambda^c_A) \wedge \theta^c_B \in \gamma(\lambda^c_B).$$

Definition 2.5 (Correct global analysis)**.** Let P be a program, Q a set of initial concrete queries, and $\mathscr{A}$ a well-formed analysis graph for P. $\mathscr{A}$ *is correct for* P, Q if

a) $\mathscr{A}$ correctly approximates the calls for P, Q (Def. 2.2),

b) $\mathscr{A}$ correctly approximates the answers for P, Q (Def. 2.3), and

c) $\mathscr{A}$ correctly approximates the call dependencies for P, Q (Def. 2.4).

2.4 Analyzing other languages

The analysis graph approach can be applied to other programming languages outside of the logic programming paradigm, although depending on the characteristics of the language some parts of the approach (e.g., OR-nodes) may not be used. As an example, in [134] the base algorithms that we extend in this work were shown to be directly applicable to Java bytecode. However, rather than this direct application, it is often advantageous to use a separation of concerns approach and divide the process into two steps: first a translation from the original program to a set of Horn clauses (a CLP program), as an intermediate representation, so that the semantics of the Horn clauses captures correctly the semantics of the original program, and then application of the analysis techniques. In the rest of the thesis we will assume that input programs are either CLP programs or they are converted to this Horn clause-based intermediate representation, on a modular basis. The conversion itself is beyond the scope of this thesis (and obviously dependent on the source language). It is trivially direct in the case of (C)LP programs or (eager) functional programs. For imperative programs we refer the reader to, e.g., [77, 123, 5, 59, 116, 138, 45]. In fact, Horn clauses have since been used successfully as intermediate representations for many different programming languages and compilation levels (e.g., bytecode, llvm-IR, ISA, ...), in a good number of analysis and verification tools [12, 135, 73, 90, 3, 108, 116, 16, 46, 75, 15, 107, 117, 48, 92]. We note that some of these approaches use the *bottom-up* semantics on the Constrained Horn Clauses (CHC) side, and then typically the *small-step semantics* in the translation, while others, including ours, exploit the complementary approach of using the *top-down* semantics on the CHC side, and then typically the *big-step semantics* in the translation,

but some combine, e.g., big-step with bottom-up [75]. Big-step and small-step are nicknames often used to refer to, respectively, Kahn's natural semantics [93] and Plotkin's structural operational semantics [141]. In the big-step semantics approach, the clause-based encoding is equivalent to a block-based control flow graph, which is in turn a well-established intermediate representation for program analysis. Each block is represented by a clause, constraints or built-ins in a clause represent the primitives of the language (bytecodes, machine instructions, commands, etc.), literals represent calls to other blocks, and predicates with multiple clauses implement alternatives such as conditionals, case statements, dynamic dispatch, etc. (see, e.g., [123, 116]). This approach is particularly well-suited for programs with structured control flow. See [59, 45] for a more detailed discussion of this topic.

2.5 The Ciao System

Ciao [80] is a modern, multiparadigm programming language with an advanced programming environment. The main motivation behind the system is to develop a combination of programming language and development tools that together help programmers produce better code in less time and with less effort. This can be helped, concretely, by two approaches: *verification* and *testing*. The former uses formal methods to prove automatically or interactively some specification of the code, while the latter mainly consists in executing the code for concrete inputs or test cases and checking that the program input-output relations (and behavior, in general) are the expected ones. The Ciao language introduced a development workflow [145, 82, 84] that integrates the two approaches above. In this model, program *assertions* (see Sec. 2.5.2) are fully integrated in the language, and serve both as specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. Assertions are optional and the model admits from the start that some parts of assertions may not be checkable at compile-time and will then generate run-time tests for them when possible. This model represents an alternative approach for writing safe programs without relying on full static typing, which is specially useful for dynamic languages like Prolog. The intention is to combine the best elements from static and dynamic language approaches [79] and is an antecedent to the now popular *gradual-* and *hybrid-typing* approaches [57, 164, 151].

A high-level view of the Ciao System is shown in Fig. 8. Blue-colored boxes represent user-written code; green boxes represent different tools within the system: the compiler, LPdoc and the CiaoPP Program Processor; and the red box represents the execution environment of the system, i.e., its run-time abstract machine and libraries. In this thesis, only some of them are detailed, as not all of them are used.

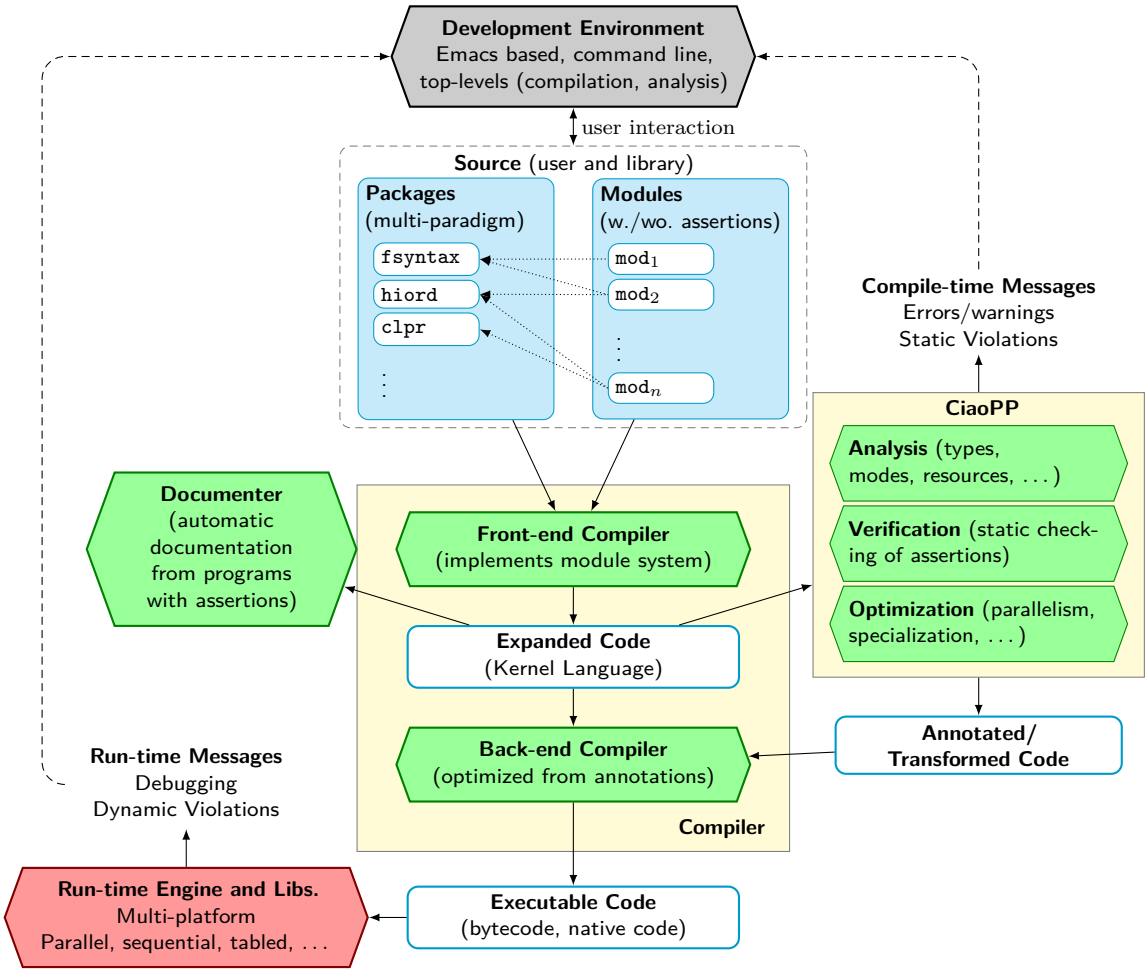


Fig. 8: A high-level view of the Ciao system [80].

2.5.1 *The CiaoPP Program Processor*

CiaoPP [127, 129, 25, 82, 145, 84] (see the right part of Fig. 8, and Fig. 9) is the abstract interpretation-based program (pre)processor of Ciao, and the most relevant part of the system for the thesis. CiaoPP performs a number of program debugging, analysis, and source-to-source transformation tasks. It can be applied to (Ciao) Prolog programs and, as mentioned in Sec. 2.4, also to many other high- and low-level languages. However, herein we will concentrate, without loss of generality, on programs represented in the core Horn clause-based intermediate representation used by the tool. The tasks performed by CiaoPP include:

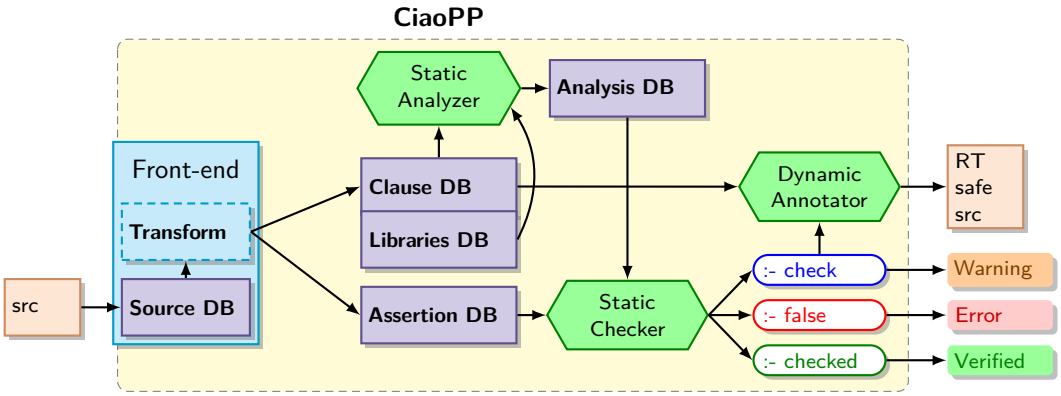


Fig. 9: Architecture of the `CiaoPP` verification framework.

- Inference of properties at the level of predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.
- Static debugging and verification. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program.
- Source to source program transformations such as partial evaluation, including slicing and specialization, and program parallelization (with granularity control). It also produces run-time test annotations for assertions which cannot be checked completely at compile-time, so that the program can be run safely by dynamically checking properties.
- Producing abstract models of programs that act as certificates of the correctness of the code. The system is used to certify that code is safe w.r.t. the given policy. I.e., an abstraction-carrying code approach to mobile code safety [6]. [2]

All the aforementioned features rely on the statically inferred properties based on fixpoint computation. Fig. 9 is an overview of the components in `CiaoPP`. `CiaoPP` has a *Front-end* that transforms programs (possibly written in a different language) to extract clauses and assertions (specification of the program). The *Static Analyzer* component has several fixpoint computation algorithms that are used to produce analysis graphs (stored in the *Analysis DB*). The information in the Analysis DB

2 The code and a certificate is provided so that the fixpoint result can be checked efficiently. For example providing the invariants of recursive predicates so that no iteration is necessary.

(`true` assertions) is used to statically check the assertions in the *Static Checker*. For each assertion originally with status `check`, the result of this process (boxes on the right of Fig. 25) can be: that it is verified (the new status is `checked`), that a violation is detected (the new status is `false`), or that it is not possible to decide either way, in which case the assertion status remains as `check`, as detailed in Sec. 2.5.2. In such cases, optionally, a warning may be displayed and/or a run-time test generated by the *Dynamic Annotator* component for the (the part of) the assertion that could not be discharged at compile-time, test cases generated, etc.

Using the Horn clause transformation approach (Sec. 2.4) `CiaoPP` has been applied to the analysis, verification, and optimization of a number of languages (besides `Ciao`) ranging from very high-level ones to bytecode and machine code, such as Java, XC (C like) [116], Java bytecode [133, 135], ISA [108], LLVM IR [107], Michelson [138], etc., and properties ranging from pointer aliasing and heap data structure shapes to execution time, energy, or smart contract "gas" consumption [124, 106].

2.5.2 Assertions

Assertions are linguistic constructions that allow stating properties of a program, such as, conditions on the state (current substitution or constraint) that hold or must hold at certain points of program execution. In this thesis we use the Ciao assertion language [82, 145, 144, 84]. These assertions are instrumental for many purposes, such as expressing the results of analysis, providing specifications, guiding the analysis, and documenting. Such assertions can express a wide range of properties, including functional (state) properties (e.g., shapes, modes, sharing, aliasing, ...) as well as non-functional (i.e., global, computational) properties such as resource usage (energy, time, memory, ...), determinacy, non-failure, or cardinality. The set of properties that can be used in assertions is extensible and new abstract domains can be defined as "plug-ins" to support them. Without loss of generality, we use for concreteness a subset of the syntax of the `pred` assertions of [23, 82, 144], which allows describing sets of *preconditions* and *conditional postconditions* on the state for a given predicate as well as global properties. A `pred` assertion is of the form:

`:- [` *Status* `] pred` *Head* `[:` *Pre* `] [=>` *Post* `] [+` *Comp* `].`

where *Head* is a predicate descriptor (i.e., a normalized atom) that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*, i.e., literals corresponding to predicates meeting certain conditions which make them amenable to checking [144]. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. *Comp* describes properties of the whole computation such as resource usage,

termination, determinism, non-failure, etc., and they apply to calls to the predicate that meet *Pre*. *Pre*, *Post*, and *Comp* can be empty conjunctions (meaning true), and in that case they can be omitted. *Status* is a qualifier of the meaning of the assertion. The following statuses are intended to be specified by the programmer:

- `check`: the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove (or else generate run-time checks for). `check` is the *default* status, and can be omitted.
- `trust`: the assertion represents an actual behavior of the predicate that the analyzer assumes to be correct although it may not be able to infer it automatically.

The following statuses are intended to be used as communication between the different components and providing information to the user, as part of the analysis/verification process (corresponding to the ovals in Fig. 9):

- `checked`: the analyzer proved that the property holds in all executions.
- `true`: the analyzer inferred the assertion.
- `false`: the analyzer proved that the property does not hold in some execution.

As mentioned before, *parts* of assertions that cannot be discharged statically will remain in `check` status and run-time tests will be generated for them if necessary.

Example 2.15. The following assertions describe different behaviors of the `pow` predicate that computes $\mathtt{P} = \mathtt{X}^{\mathtt{N}}$: `(1)` is stating that if the exponent of a power is an even number, the result (`P`) is non-negative, `(2)` states that if the base is a non-negative number and the exponent is a natural number the result `P` also is non-negative:

```
1  :- pred pow(X,N,P) : (int(X), even(N)) => P ≥ 0.   % (1)
2  :- pred pow(X,N,P) : ( X ≥ 0,  nat(N)) => P ≥ 0.   % (2)
3  pow(_, 0, 1).
4  pow(X, N, P) :-
5      N > 0,
6      N1 is N - 1,
7      pow(X, N1, P0),
8      P is X * P0.
9
10 :- prop even/1.
11 even(N) :-
12     0 is N mod 2.
```

Here, the `even/1` property is defined by the user, while `int/1` and `nat/1` are assumed to be understood by the abstract domain. The *predicate* defining the property is

analyzed using the abstract domain, thus inferring the abstract meaning of the user-defined property, and that meaning is used. Different treatment is required when the assertion is used for analysis or for verification, this is detailed in Chapter 6 and Chapter 9.

In addition to `pred` assertions we also consider *program-point assertions*. They are expressed as regular literals using as predicate name their *Status*, i.e., `trust`(*Cond*) and `check`(*Cond*). They imply that whenever the execution reaches a state originated at the program point in which the assertion appears, *Cond* (should) hold. Example 2.16 illustrates their use. Program-point assertions can be translated to `pred` assertions,[3] so without loss of generality we limit the discussion to `pred` assertions.

Definition 2.6 (Meaning of a Set of Assertions for a Predicate)**.** Given a predicate represented by a normalized atom *Head*, and a corresponding set of assertions $\{a_1 \ldots a_n\}$, with $a_i =$ "`:- pred` *Head* `:` Pre_i `=>` $Post_i$`.`" the set of *assertion conditions* for *Head* is $\{C_0, C_1, \ldots, C_n\}$, with:

$$C_i = \begin{cases} \mathsf{calls}(Head, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ \mathsf{success}(Head, Pre_i, Post_i) & i = 1 \ldots n \end{cases}$$

where $\mathsf{calls}(Head, Pre)$[4] states conditions on all concrete calls to the predicate described by *Head*, and $\mathsf{success}(Head, Pre_i, Post_i))$ describes conditions on the success constraints produced by calls to *Head* if Pre_i is satisfied. These allow representing behaviors for the same predicate for different call substitutions (multivariance). If the assertions a_i above, $i = 1, \ldots, n$, include a `+` *Comp* field, then the set of *assertion conditions* also include conditions of the form $\mathsf{comp}(Head, Pre_i, Comp_i)$, for $i = 1, \ldots, n$, that express properties of the whole computation for calls to *Head* if Pre_i is satisfied. In this thesis we will concentrate fundamentally on dealing with *Pre* and *Post* conditions.

The assertion conditions for the assertions in Example 2.15 are:

$$\left\{ \begin{array}{llll} \mathsf{calls}(& pow(X,N,P), & ((int(X), even(N)) \vee (X \geq 0, nat(N)))), & \\ \mathsf{success}(& pow(X,N,P), & (int(X), even(N)), & (P \geq 0)), \\ \mathsf{success}(& pow(X,N,P), & (X \geq 0, nat(N)), & (P \geq 0)) \end{array} \right\}$$

3 E.g., we can replace line 4 in Example 2.16, by "`assrt_aux(Z),`", and add a predicate to the program, `assrt_aux(_).`, with an assertion "`:- pred assrt_aux(Z) : Z = 2.`".

4 We denote the calling conditions with `calls` (plural) for historic reasons, and to avoid confusion with the higher order predicate in Prolog `call/2`.

2.5.3 Practical uses of assertions

Assertions are also a means for providing information that isn't available to the analyzer for different reasons. Here we show some examples.

Example 2.16 (Regaining precision during analysis)**.** The analysis of the following program with the (non-relational) intervals domain would infer for `Z` that it can be "any integer" (line 3). However, it can be seen that $Z = 2$ for any `X` and `Y`. The programmer can provide this information to the analyzer with an assertion (line 4). The analyzer *trusts* this information even if it cannot be inferred by composing each of the abstraction of the literals (being non-relational), even if the property is representable in the domain.

```
1 p(Y) :-            % (Y > 0)
2     X is Y + 2,    % (X > 2,  Y > 0)
3     Z is X - Y,    % (int(X), X > 2, Y > 0)
4     trust(Z = 2), % (Z = 2,  X > 2, Y > 0)
5     % implementation continues
```

Example 2.17 (Bridging external proofs with the analyzer)**.** This is equivalent to using interfaces to other languages in compilation. It is safe to reuse these properties during analysis because they have been proven to be safe in some external tool. For example with a proof assistant.

```
1 :- trust pred pow(X,N,P) : (int(X), even(N)) => (P > 0) + proof(1).
2 :- trust pred pow(X,N,P) : (X > 0, nat(N)) => (P > 0) + proof(2).
3 pow(_, 0, 1).
4 pow(X, N, P) :- N > 0,
5     N1 is N - 1,
6     pow(X, N1, P0),
7     P is X * P0.
```

Example 2.18 (Speeding up analysis)**.** Very precise domains suffer less from loss of precision and are useful for proving complex properties, but can be very costly. In some cases less precise information in enough. In this code extracted from `LPdoc` [78], the `Ciao` documentation generator, `html_escape/2` is a predicate that takes a string of characters and transforms it to html:

```
:- trust pred html_escape(S0, S) => (string(S0), string(S)).
html_escape("‘‘"||S0, "&ldquo;"||S) :- !, html_escape(S0, S).
html_escape("’’"||S0, "&rdquo;"||S) :- !, html_escape(S0, S).
html_escape([34|S0],  "&quot;"||S)  :- !, html_escape(S0, S).
html_escape([39|S0],  "&apos;"||S)  :- !, html_escape(S0, S).
% ...
html_escape([X|S0], [X|S])          :- !, character_code(X),
    html_escape(S0, S).
html_escape([],[]).

% string(Str) :- list(Str, int).
```

Analyses based on regular term languages, e.g. `eterms` [173] infer precise regular types with subtyping, which is often costly. In this example it would be equivalent to computing an accurate regular language that over-approximates the HTML text encoding. The `trust` assertion provides a general invariant that the analyzer uses instead of inferring a more complex type.

Example 2.19 (Defining abstract usage or specifications of libraries or dynamic predicates)**.** When sources are not available, or cannot be analyzed, assertions can provide the missing *abstract* semantics. The following assertions describe the behavior of predicates `receive` and `send` in a `sockets` library (e.g., written in C). The assertion in this case transcribes what is stated in natural language in the documentation of the library. Note that if no annotations were made, the analyzer would have to assume the most general abstraction ($\top$) for the library arguments.

```
:- trust pred receive(S, M) : (socket(S), var(M))        => list(M, utf8).
:- trust pred send(S, M)    : (socket(S), list(M, utf8)).
```

Example 2.20 ((Re)defining the language semantics for abstract domains)**.** Lastly, `trust` assertions are also a useful tool for defining the meaning (transfer function) of the basic operations of the language. In this example we define some basic properties of the product predicate in a simple types-style abstract domain:

```
:- trust pred '*'(A, B, C) : (int(A), int(B)) => int(C).
:- trust pred '*'(A, B, C) : (flt(A), int(B)) => flt(C).
:- trust pred '*'(A, B, C) : (int(A), flt(B)) => flt(C).
:- trust pred '*'(A, B, C) : (flt(A), flt(B)) => flt(C).
```

The semantics of bytecode or machine instructions can be specified for each domain after transformation into Horn Clauses.

Lastly, assertions are also a means to report the results in the nodes of the analysis graphs back to the programmer.

Example 2.21 (Reporting analysis results)**.** The following assertions report the analysis graph of Fig. 5.

```
:- true pred par(M,X,P) : (z(X)) => (z(X), b(P)). % abstract query
:- true pred par(M,X,P) : (b(X)) => (b(X), b(P)).
par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

:- true pred xor(C,P0,P1) : (z(P0)) => (b(C), z(P0), b(P1)).
:- true pred xor(C,P0,P1) : (b(P0)) => (b(C), b(P0), b(P1)).
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).

% properties
b(0).
b(1).

z(0).
```

The assertion language allows stating any property that can be expressed as a logic program. Thus, in general, these properties may not be representable in the abstract domain. To be able to use such properties they need to be approximated. This issue is addressed in Sections 6.1 and 9.1.

2.5.4 *Modular Logic Programming in Ciao*

Modularity is a basic notion in modern computer languages. Modules allow dividing programs into several parts, which have their own independent name spaces and a clear interface with the rest of the program. This isolated way of seeing the code has two main advantages. It allows a divide-and-conquer approach to program development and maintenance and, in terms of efficiency, tools which work with programs can be more efficient if they can process a single program at a time.

Defining Modules: The source of a Ciao module is typically contained in a single file. The fact that a file contains a module is flagged by the presence of a "`:- module(...)`" declaration at the beginning of the file. The predicates defined within a module are visible only if they are exported.

Imports and Exports: Predicates in a module are *exported*, i.e., made available outside the module, via explicit `:- export` declarations or in an export list in the `:- module(...)` declaration.

Also, it is possible to import a number of individual predicates or also all predicates from another module, by using `:- use_module` and `:- import` declarations. Those predicate must be previously exported by the concrete module.

Visibility Rules: The set of predicates which are visible in a module are predicates defined in that module plus the predicates imported from other modules. It is possible to refer to predicates with or without a *module qualification*. A module-qualified predicate name has the form *module:predicate*. An example of this form is the call `lists:append(A,B,C)`.

Modular Partitions of Programs. A partition of a program is said to be modular when its source code is distributed in several source units, each defining its interface with other such units of the program. We refer to these units as *modules*. The interface of a module contains the names of the predicates it exports and the names of the modules it imports. Modular partitions of programs may be synthesized, or specified by the programmer, for example, via a strict module system, i.e., a system in which modules can only communicate via their interface. We use M and M' to denote modules. Given a module M:

- $\mathsf{exports}(M)$ denotes the set of predicate names exported by module M,
- $\mathsf{imports}(M)$ is the set of modules which M imports, and
- $\mathsf{mod}(A)$ denotes the module in which the predicate corresponding to atom A is defined. We sometimes abuse notation and denote the module of a query as $\mathsf{mod}(Q)$, to refer to the module of the predicate called in the query, i.e., if $Q = \langle A, \lambda^c \rangle$ then $\mathsf{mod}(Q) = \mathsf{mod}(A)$.

2.5.5 *Modular generic logic programming:* traits

In this section we present a simple approach to modular generic programming for logic programs without static typing. The concept of *open* predicates can be reused to deal with generic code. We propose a simple syntactic extension for logic programs for writing and using generic code (*traits*) and its translation to plain clauses. Generic code offers many opportunities for the application of the new analysis techniques proposed in this thesis. For example: standalone analysis of trait-based code without particular implementations by using the (trust) assertions in the interfaces; refinement of standalone analysis for particular implementations; or reuse of analysis results when more implementations are made available.

Open vs. closed predicates. Traditionally, in a module system for logic programming, predicates are distributed in modules, each predicate name belongs to a particular module, and module dependencies are explicit in the program [27]. Closed predicates within a module are those whose complete definition is available in the module. In contrast, the definition of open predicates (traditionally declared as `multifile` in many Prolog systems) can be scattered across different modules, and thus not known until all the application modules are linked (note that programs still use the closed world assumption). Despite its flexibility, open predicates are "anti-modular" (in a similar way to typeclasses in Haskell). Here we only consider *static* predicates and modules. Predicates whose definition may change during execution, or modules that are dynamically loaded/unloaded at run time can also be dealt with, using various techniques, and in particular the incremental analysis proposed.

Open as "multifile." The following example shows an implementation of a generic password-checking algorithm in Prolog:

```
:- multifile dgst/3.

check_passwd(User) :-
    get_line(Plain),                % Read plain text password
    passwd(User,Hasher,Digest,Salt), % Consult password database
    append(Plain,Salt,Salted),       % Append salt
    dgst(Hasher,Salted,Digest).      % Compute and check digest
```

The code above is generic w.r.t. the selected hashing algorithm (`Hasher`). Note that there is no explicit dependency between `check_passwd/1` and the different hashing algorithms. The special *multifile* predicate `dgst/3` acts as an *interface* between implementations of hashing algorithms and `check_passwd/1`. While this type of encoding is widely used in practice, the use of multifile predicates is semantically obscure and error-prone. Instead we propose *traits* as a syntactic extension that captures the essential mechanisms necessary for writing generic code.[5]

Traits. A *trait* is defined as a collection of predicate specifications (as predicate assertions)[6]. For example:

```
:- trait hasher { :- pred dgst(Str, Digest) : string(Str) => int(Digest). }.
```

5 Here we only focus on traits as interfaces. The actual design in Ciao supports default implementations, which makes them closer to traits in Rust.

6 See http://ciao-lang.org/ciao/build/doc/ciao.html/traits_doc.html for the full documentation.

defines a trait `hasher`, which specifies a predicate `dgst/2`, which must be called with an instantiated string, and obtains an integer in `Digest`.

As a minimalistic syntactic extension, we introduce a new head and literal notation $(X$ `as` $T)$.`p`$(A_1,\ldots,A_n)$, which represents the predicate p for X implementing trait T. Basically, this is equivalent to $p(X,A_1,\ldots,A_n)$, where X is used to select the trait implementation. In literals, X is annotated with a trait, which can be different for each call due to dynamic typing and multiple trait implementations for the same data. Such annotations could be reduced in many cases by automatically inferring them locally with simple static typing rules. When X (the implementation) is unknown at compile-time, this is equivalent to dynamic dispatch. The `check_passwd/1` predicate using the trait above is:

```
check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).
```

The following translation rules transform code using traits into code with traditional predicates. We rely on the underlying module system to add module qualification to function and trait (predicate) symbols. Calls to trait predicates are done through the interface (open) predicate, which also carries the predicate assertions declared in the trait definition:

```
% open predicates and assertions for each p/n in the trait
:- multifile 'T.p'/(n+1).
:- pred 'T.p'(X,A_1,...,A_n) : ... => ... .
% call to p/n for X implementing T
... :- ..., 'T.p'(X,A_1,...,A_n), ... % (X as T).p(A_1,...,A_n)
```

A trait *implementation* is a collection of predicates that implements a given trait, indexed by a specified functor associated with that implementation. E.g.:

```
:- impl(hasher, xor8/0).
(xor8 as hasher).dgst(Str, Digest) :- xor8_dgst(Xs, 0, Digest).

xor8_dgst([], D, D).
xor8_dgst([X|Xs], D0, D) :- D1 is D0 # X, xor8_dgst(Xs, D1, D).
```

declares that `xor8` implements a `hasher`. In this case, `xor8` is an atom but trait syntax allows arbitrary functors. The implementation for the `dgst/2` predicate is provided by `(xor8 as hasher).dgst(Str, Digest)`.

The translation rules to plain predicates are as follows:

```
1 % the implementation of the trait is a closed predicate (head renamed)
2 '<f/k as T>.p'(f(...),A1,...,An) :- ... % (f(...) as T).p(A1,...,An)
3
4 % bridge from interface (open predicate) to the implementation
5 'T.p'(X,A1,...,An) :- X=f(...), '<f/k as T>.p'(X,A1,...,An).
```

This generates predicate names based on the original names of the traits and the implementations.

In line 1, `'<f/k as T>.p'` is the name generated for the predicate that implements the trait. Line 2 contains the clause that bridges the predicates in the trait with the predicates in the implementation. For example, `(xor8 as hasher).dgst` is expanded to the predicate name `'<xor8/0 as hasher>.dgst'` and then, this predicate is the one called from the predicate in the trait. This is done in line 5, our example would generate: `'hasher.dgst'(X,A1,A2) :- X=xor8, '<xor8/0 as hasher>.dgst'(X,A1,A2).`.

Adding new implementations is simple, e.g.:

```
1 :- impl(hasher, sha256/0).
2 (sha256 as hasher).dgst(Str, Digest) :- ...
```

This predicate would be renamed to `'<sha256/0 as hasher>.dgst'` and a clause would be added to the `'hasher.dgst'` predicate, `'hasher.dgst'(X,A1,A2) :- X=sha256, '<sha256/0 as hasher>.dgst'(X,A1,A2).`

This approach preserves some interesting modular features: trait names can be local to a module (and exported as other predicate/function symbols), and trait implementations (e.g., `sha256/0`) are just function symbols, which can also be made local to modules in the underlying module system.

```
1  % sha256.pl
2  :- impl(hasher, sha256/0).
3  (sha256 as hasher).dgst(Str, Digest) :- ...
4
5  % sha512.pl
6  :- impl(hasher, sha512/0).
7  (sha512 as hasher).dgst(Str, Digest) :- ...
8
9  % passwd.pl
10 passwd(don,xor8,0x6d,"eNfwuBhtN9CwogBugeUHxg==").  % password: 000000
11 passwd(pete,sha256,0xed8f...,"a2Fy+w+g4XNIAR31ot+3Sg==").  % password:
       123123
12 passwd(peggy,sha512,0xe653...,"9t68Vz9rcPqP1u4HMB5Hfg==").  % password:
       qwerty
```

This concludes the background required for the thesis.

3

Abstract Extensionality – *On the properties of incomplete abstract intepretations*

Program analysis is concerned with intensional properties not just because semantically equivalent programs may exhibit different abstract properties, but also because semantically different programs may appear identical abstractly. This familiar phenomenon can be overcome by increasing the precision of program analysis; but the incurred costs make it into one of the main challenges in program analysis, and into one of the main obstacles of general-purpose program analysis. Furthermore, scalable analyses are often achieved by making abstractions even coarser. Being aware of the precision flaws of an abstract interpretation is equivalent to studying its *completeness*. Completeness encodes the greatest achievable precision when abstracting the concrete behavior of a program on an abstract interpreter. If an interpretation of a program is complete, the only loss of precision is due to the *expressivity* of the abstract domain, and not the abstraction functions or the abstract interpreter. *Incomplete* abstract interpretations give rise to weaker properties than those encountered under direct inspections of effective computations of the program. This means that in debugging, incomplete abstract interpretations yield false alarms. The immediate consequence is that assertions that could be proved are now reported to be potentially violated. Our goal is to study how abstract interpreters partition the program space in equivalence classes induced by abstract program equivalence, and the intersection of these equivalence classes with the completeness and incompleteness properties of abstract interpretations.

In this chapter we study completeness of program analysis from a computability perspective. We generalize the notion of extensional (functional) equivalence of programs to *abstract equivalences* induced by *abstract interpretations*. The standard notion of extensional equivalence is recovered as the special case, induced by the concrete interpretation. Some properties of the extensional equivalence, such as the one spelled out in Rice's theorem, lift to the abstract equivalences in suitably generalised forms. On the other hand, the generalized framework gives rise

to interesting and important new properties, and allows refined, non-extensional analyses. In particular, since programs turn out to be extensionally equivalent *if and only if* they are equivalent just for the concrete interpretation, it follows that any non-trivial abstract interpretation uncovers some intensional aspect of programs. This striking result is also effective, in the sense that it allows constructing, for any non-trivial abstraction, a pair of programs that are extensionally equivalent, but have different abstract semantics. The construction is based on the fact that abstract interpretations are always sound, but that they can be made incomplete through suitable code transformations that inject code that cannot be understood by the domain [44]. A further consequence is that the class of incomplete programs for a non-trivial abstraction is Turing complete.

3.1 Introduction

The main tradition of theory of computation [155] has been concerned with extensional aspects of computation, i.e. with properties of programs reduced to the functions that they implement: two programs (often represented as indices in an enumeration of all programs) are extensionally equivalent if they produce the same outputs on the same inputs. From this point of view we cannot tell apart any pair of extensionally equivalent programs. In other words, it studies properties of computable functions. Much less is known about the intensional properties of programs, which distinguish different algorithms for computing the same function, or different descriptions in programming languages, or different executions [1]. The intensional side of computation includes everything that happens after the input data is read, and before the output data is written: all states and state changes, how many steps are made, how much memory is used, namely all possible side effects. But besides the widely studied properties such as program complexity, the intensional properties include relations of programs with programmers, such as understandability and quality; or with other programs, such as optimizing compilers, static analyzers, software debuggers; or with abstract interpreters. In any case, intensional properties are the central concern of software design, and maintenance, and they lie at the heart of software process in general.

Abstract interpretations usually approximate some *extensional* properties of the computations induced by programs. When abstract interpretations are guaranteed to terminate, reasoning about programs is reduced to total computable functions,

P :	Q :	R :
$x := 10;$	$x := 10;$	$x := -9;$
while @ $(x > 0)$	**while** @ $(x > 1)$	**while** @ $(x < 0)$
do $x := x - 1;$	**do** $x := x - 2;$	**do** $x := x + 2;$

Fig. 10: Some simple programs.

in contrast with the traditional, concrete interpretation, which associates with each program the *partial* function that it computes, and reason about such functions.[1]

We illustrate the sensitivity of program analysis to code structure with an example.

Example 3.1. Consider the abstract domain of intervals D^{Int}. Each element in the domain corresponds to an interval $[a, b]$, where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$ and $a \leq b$. This domain is an abstraction of properties of integer valued variables; i.e., D^{Int} abstracts the set $\wp(\mathbb{Z})$ of sets of integers. Consider the programs P and Q in Fig. 10, where the symbol @ is used to mark some program points that interest us. Suppose that we want to prove the Hoare triples $\{x \in \mathbb{Z}\}P\{x = 0\}$ and $\{x \in \mathbb{Z}\}Q\{x = 0\}$. It is easy to see that the programs are extensionally equivalent, and that they always output $x = 0$. However, interpreted in D^{Int}, the programs P and Q exhibit different abstract semantics. At the point @ of the program Q, the values are approximated by the increasing sequence of intervals: $[10, 10] \subset [8, 10] \subset [6, 10] \subset [4, 10] \subset [2, 10] \subset [0, 10]$. When the loop condition becomes false, and the execution exits the loop, the postcondition $x \leq 1 \ \wedge \ x \in [0, 10]$ is reached, which is true if and only if $x \in [0, 1]$. Intepreting Q in D^{Int} thus only allows proving $\{x \in \mathbb{Z}\}Q\{x \in [0, 1]\}$, whereas it is easy to see that interpreting P in D^{Int} we can prove the stronger postcondition $\{x \in \mathbb{Z}\}P\{x = 0\}$. This shows that the intervals domain D^{Int} is *incomplete* for abstract interpretation of the program Q, while it remains complete for the program P.

It is well known that the abstract semantics of *complete* abstract interpretations produces the same property approximations as the direct abstraction of the concrete semantics [38, 70]. On the other hand, *incomplete* abstract interpretations give rise to weaker properties than those encountered under direct inspections of effective computations of the program. This means that in debugging, abstract interpretations yield false alarms. And this is not a rare phenomenon, since incompleteness is in program analysis more common than completeness [67].

1 Here the tacit assumption is that the program is deterministic and effect-free. Otherwise, the reasoning is reduced to more general families of functions, often captured by computational monads. In any case, all such functions are extensional objects, which is the main point here.

Example 3.2. Now consider the programs Q and R in Fig. 10. These two satisfy the Hoare triples $\{x \in \mathbb{Z}\}Q\{x = 0\}$ and $\{x \in \mathbb{Z}\}R\{x = 1\}$, so their extensional semantics are different; yet their abstract semantics in D^{Int} are the same: $\{x \in \mathbb{Z}\}R\{x \in [0, 1]\}$.

This striking disparity between abstract semantics and concrete semantics gives rise to a basic question: *Can the extensional program equivalence be extended based on abstract semantics?* If this is possible, then it may be of great interest to explore the properties of this abstract notion of program equivalence, and its potential applications.

In the rest of the chapter we explore abstract interpretation from the perspective of computable functions. We introduce a new family Π^{A} of index sets for partial recursive functions which are parameterized by a given abstraction A. The indices represent programs, assumed to be enumerated in some way, as it is usually done in theory of computation. The family of index sets Π^{A} is obtained by replacing the usual concrete program equivalence with an equivalence based on the abstract semantics induced by A. The results are properties of programs captured by what an abstract interpreter computes rather than what the actual programs compute. We call these properties *abstract program properties*. The main contributions are:

(1) We prove that abstract program properties are the union of equivalence classes, where two programs are considered equivalent if they have the same abstract semantics. These classes are not in general extensional, i.e., they are not closed w.r.t. standard semantic equivalence of programs. Indeed, we prove that these equivalence classes are extensional if and only if the abstraction is trivial (see Theorem 3.18), i.e., it is the identity abstraction (no approximation) making the abstract semantics coincide with the concrete one, or it is the abstraction that is unable to distinguish any pair of programs (the greatest possible approximation).

(2) We introduce two classes of programs called respectively *completeness* and *incompleteness cliques*. The completeness (resp., incompleteness) clique $\mathbb{C}(P, \mathsf{A})$ (resp., $\overline{\mathbb{C}}(P, \mathsf{A})$) of a program P and an abstraction A is the set of all programs that are semantically equivalent to P and for which A is complete (resp., incomplete). These two classes have interesting properties: $\mathbb{C}(P, \mathsf{A})$ represents the class of all variants of P for which the analysis based on A is precise (no false alarms) while $\overline{\mathbb{C}}(P, \mathsf{A})$ is instead the class of all variants of P for which A is imprecise. The connections between $\mathbb{C}(P, \mathsf{A})$ and $\overline{\mathbb{C}}(P, \mathsf{A})$ are our main focus.

(2a) We prove that there is an infinity of non-trivial abstractions for which the systematic computable removal of false alarms for all programs is impossible, namely under mild assumptions on the abstraction A there is no many-to-one reducibility of $\overline{\mathbb{C}}(P, \mathsf{A})$ to $\mathbb{C}(P, \mathsf{A})$ (see Theorem 3.8).

(2b) For a wide class of abstract domains, called variable finite (see Def. 3.4), we provide a systematic reduction of $\mathbb{C}(P, \mathsf{A})$ into $\overline{\mathbb{C}}(P, \mathsf{A})$, namely an effective

transformation that maps any program P into another program $\tau(P)$ which is equivalent to P with respect to the concrete semantics but that is distinguished from P by the abstraction A (see Corollary 3.16). In this case, although τ preserves the concrete semantics, the abstract semantics distinguishes any complete program P from $\tau(P)$, because $\tau(P)$ produces false alarms.

(3) As a consequence of our construction we prove that the class of all programs that are incomplete for a given non-trivial variable finite abstraction A is Turing complete.

Our results give new insights about the impossibility to automatically remove false alarms from program analyses. In particular we expose the typical structure of incomplete programs. These include predicates that the abstract interpreter fails to evaluate. Their structure turns out to arise from the structure of the abstract domain from which the abstract interpreter is designed. We believe that this result, together with the proof system in [67], may suggest a practical path towards specific strategies for code transformations with the goal to improve the precision of program analysis. Moreover, the Turing completeness of the class of all programs that are incomplete for a given non-trivial variable finite abstraction A, suggests specific code protecting transformations against reverse engineering. Given any (non-trivial) abstract domain A, every computable function can be implemented by a program that is incomplete on A, i.e. every computable function can be intentionally obfuscated against a given non-trivial program analysis. This establishes a *possibility* result concerning code obfuscation when the attack model is any non-trivial abstract interpreter.

3.2 Related work

Interesting results concerning the possibility of widening standard recursion theory towards intensional aspects of code have been produced in the last decade. These include, among others, the results in computational complexity via programming languages in [14], the intentional contents of Rice theorem in the case of Blum's complexity in [10], the semantics of intensionality [95], the field of implicit computational complexity (see [43] for a short survey), until the more recent *Analyzing Program Analyses* in [67] and the comparison of the hardness of program analysis with respect to program verification in [42].

The very first recursive theoretic account of complete abstract interpretations is in [67], where the notion of completeness class for an abstract interpretation was introduced. Given an abstract interpretation A, the *completeness class* of A, denoted $\mathbb{C}(\mathsf{A})$, is the set of all programs for which A is complete, i.e., for which A produces no false alarms. The complement set $\overline{\mathbb{C}(\mathsf{A})}$ is instead the set of programs that are incomplete for A. These two classes have many interesting properties but they fail

to capture the extensional behaviour of programs, namely, within $\mathbb{C}(\mathsf{A})$ and $\overline{\mathbb{C}(\mathsf{A})}$ there may coexist programs that are semantically different, just because they have complete or incomplete abstract semantics respectively.

We consider here the notion of completeness clique which is a transposition of the complexity cliques introduced in [10] for Blum's complexity to abstract interpretations. We prove some recursion-theoretic properties of completeness and, in particular, incompleteness cliques. For the latter case we provide effective transformations that allow transforming any program for which a given abstract interpretation is complete (it belongs to a completeness clique) into an extensionally equivalent one (i.e., preserving its concrete semantics) for which the same abstract interpreter produces false alarms, i.e., it is in the corresponding incompleteness clique. As a consequence this allows us proving properties and limitations of control and data-flow semantics-preserving obfuscating transformations [31]. In particular we can go beyond [118, 52] and prove that semantics-preserving transformations cannot be extended to all abstract domains when applied for data-flow obfuscations, hence formally justifying the need of data-type complication in data-flow obfuscation of programs, as postulated in [52], while they can always be implemented when dealing with control-flow obfuscations.

The systematic construction of generic abstraction-agnostic semantics-preserving obfuscators for a given abstract interpreter or model checker was also considered in [66, 44, 65] and more recently in [20]. Here we generalize those approaches and consider the more general problem of correlating the completeness and incompleteness cliques, therefore providing a more general setting to reason about the precision of an abstract interpretation. In [68] and [69] the authors introduced model deformations making a semantics respectively complete and incomplete, with the aim of giving a measure of the strength of obfuscation. None of these approaches however considered extensional equivalence as requirement.

The possibility of establishing a sound representation of the extensional equivalence of two programs by abstract interpretation is studied in [136]. The authors introduce the notion of *correlating program*, i.e., a new program $P \bowtie Q$ obtained as the combination of P and Q, together with a correlating abstract domain such that the analysis of $P \bowtie Q$ gives information about the extensional equivalence of P and Q. In the light of our results, it is therefore always possible to transform $P \bowtie Q$ in order to break the completeness of the proof of equivalence, and hence to foil the equivalence analysis. We believe that this has reflections on the structure of both P and Q, i.e., when P and Q have some specific shape, and include the predicates that expose the incompleteness of the analysis, the differencing analysis is imprecise. The early detection of these predicates may help in driving the use of the correlating program method in [136] to achieve the best of the analysis.

3.3 Preliminaries

Programs We will consider a basic deterministic while-language `Imp` with arithmetic and Boolean expressions, as defined, e.g., in [175], whose syntax is as follows:

$$\begin{aligned}
\texttt{AExp} \ni a &::= v \in \mathbb{Z} \mid x \in \mathit{Var} \mid f(\widetilde{a}) \\
\texttt{BExp} \ni b &::= \mathbf{tt} \mid \mathbf{ff} \mid a = a \mid a > a \mid b \wedge b \mid \neg b \\
\texttt{Imp} \ni P &::= \mathbf{skip} \mid x := a \mid P; P \mid \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\ b\ \mathbf{do}\ P
\end{aligned}$$

where $\mathbb{Z}$ denotes the set of integers, Var is a denumerable set of program variables, f ranges over total recursive functions, and $\widetilde{a}$ denotes a list of arithmetic expressions. As usual we denote by $\mathit{var}(P)$ the finite set of variables that occur in the program P.

Abstract interpretation In the following we consider program analyzers as specified by Galois insertion-based abstract interpreters (see [37, 38] for details). Our results are based on some hypotheses that are often left implicit in the context of program analysis. Here, in order to emphasise their role, we prefer to name them explicitly as [H1], [H2] and [H3] in the text that follows.

We say that A is *strict* if $\gamma(\bot_A) = \bot_C$. We say that an element $c \in C$ is *exactly represented* in A if $(\gamma \circ \alpha)c = c$. We say that an abstract domain is *trivial* if it is isomorphic to the concrete domain, i.e., $\gamma \circ \alpha = \mathtt{id}_C$, or if it consists of only one element $A = \{\top_A\}$, i.e., for all $c \in C$, $(\gamma \circ \alpha)c = \top_C$. The former is called identity abstraction, and the latter top abstraction, in which case we denote $\mathsf{A} = \top\!\!\top$ and $\gamma \circ \alpha = \top$.

> *In the rest of the chapter we only consider Galois insertions over strict or trivial abstract domains.* [H1]

As a consequence of [H1], any non-trivial abstract domain is assumed strict ($\top\!\!\top$ is not strict).

Sound and complete abstractions Given an abstract domain $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ and a monotone operator $f : C \to C$, we say that a function $f^\sharp : A \to A$ is a *correct abstract interpretation* of f if $\alpha(f(c)) \leq f^\sharp(\alpha(c))$ for any $c \in C$. Note that if $f^\sharp$ is a correct abstract interpretation of f then we have also fixpoint correctness, i.e., $\alpha(\mathit{lfp}(f)) \leq \mathit{lfp}(f^\sharp)$. An abstract function f^α is the *best correct approximation* of f in A iff $f^\alpha \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \to A$, because for any correct abstract interpretation $f^\sharp$ it holds that $f^\alpha(a) \leq f^\sharp(a)$ for any $a \in A$. Note that this does not imply the best fixpoint.

We say that $f^\sharp$ is *complete* if $\alpha \circ f = f^\sharp \circ \alpha$. We say that A is a complete abstraction for f if there exists a complete abstract interpretation $f^\sharp$ of f. Completeness of $f^\sharp$

intuitively encodes the greatest achievable precision when abstracting the concrete behaviour of f on A. In complete abstractions the only loss of precision is due to the abstract domain and not to the abstract functions $f^\sharp$. If $f^\sharp$ is complete for f then we have fixpoint completeness (also called fixpoint transfer): $\alpha(\mathit{lfp}(f)) = \mathit{lfp}(f^\sharp)$, which follows from $\alpha \circ f \leq f^\sharp \circ \alpha$. It turns out that completeness $\alpha \circ f = f^\sharp \circ \alpha$ holds iff $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$. Thus, the possibility of defining a complete approximation $f^\sharp$ of f on some abstract domain A only depends upon the best correct approximation of f in A, i.e., completeness is a property of the concrete semantics and of the abstract domain only [70]. We will say both "A is complete for f" and "f is complete on A" to refer to completeness. Note that the identity and the top abstractions, $\mathtt{id}$ and $\top\!\!\top$, are both complete for any f.

It is also worth noting that by replacing the abstract join operator $\sqcup$, defined in the abstract domain A, with a widening operator $\nabla : A \times A \to A$, such that for any $a, b \in A$: $a \sqcup b \leq a \nabla b$ and ∇ extrapolates the possibly infinite chain of the iterates of $f^\sharp$ to a finite chain, see [37], we always reduce the precision of the fixpoint computed in A, i.e., $\mathit{lfp}(f^\sharp) \leq \nabla_{n<\omega} f^{\sharp^n}(\bot_A)$. This means that if a widening-based program analysis on the abstract domain A is complete for $f^\sharp$ then A is complete for f. Because we are interested in the recursive properties of the class of programs for which an abstract interpreter defined on an abstract domain A is incomplete, in the following we do not consider widening-based fixpoint extrapolating operators to force termination in program analysis, and consider the tighter condition of incompleteness caused by the Galois insertion specifying A.

3.3.1 Program Semantics

Our denotations are *stores*, i.e., partial functions m in $\mathsf{S} \stackrel{\text{def}}{=} \mathit{Var} \to \mathbb{Z}$ that assign values only to a finite set of variables. We will often represent a store $\mathsf{m} \in \mathsf{S}$ as a tuple $\langle x_1/v_1, \ldots, x_n/v_n \rangle$ of its defined values, i.e., such that $\mathsf{m}(y) = \$$ if $y \notin \{x_1, \ldots, x_n\}$ and $\mathsf{m}(x_i) = v_i$ for all $i \in [1, n]$. As usual we let $\mathit{var}(\mathsf{m}) = \left\{ \, x \in \mathit{Var} \, \middle| \, \mathsf{m}(x) \neq \$ \, \right\}$. Without loss of generality, in the following instead of dealing with values in the set $\mathbb{Z} \cup \{\$\}$ we assume that the default value $\$$ is just a chosen element of $\mathbb{Z}$. In this sense a store m is seen as a total function where $\$$ is the default value for unused variables. For S a set of stores, we denote by $\mathit{var}(S)$ the set $\bigcup_{\mathsf{m} \in S} \mathit{var}(\mathsf{m}) = \left\{ \, x \in \mathit{Var} \, \middle| \, \exists \mathsf{m} \in S \wedge \mathsf{m}(x) \neq \$ \, \right\}$.

Programs represent partial recursive functions from input stores to output stores. Store update is written $\mathsf{m}[x \mapsto v]$ with

$$\mathsf{m}[x \mapsto v](y) = \begin{cases} v & \text{if } y = x \\ \mathsf{m}(y) & \text{otherwise} \end{cases}$$

As a matter of notation, for a set of values V, we write $\mathsf{m}[x \mapsto V]$ for the set of stores where x is updated with values in V: $\mathsf{m}[x \mapsto V] = \{\mathsf{m}[x \mapsto v] \mid v \in V\}$. Similarly, for a set of stores S we let $S[x \mapsto V] = \{\mathsf{m}[x \mapsto v] \mid \mathsf{m} \in S \wedge v \in V\}$. The semantics of arithmetic and boolean expressions are the functions $(\!|a|\!) : \mathbb{S} \to \mathbb{Z}$ and $(\!|b|\!) : \mathbb{S} \to \{\mathbf{tt}, \mathbf{ff}\}$ defined as usual.

Abstract interpretations consider *collecting semantics* of programs, which are the additive extension of the standard semantics, defined as partial recursive functions on stores, to functions on properties of stores. The collecting semantics of arithmetic expressions $a \in \mathtt{AExp}$, $[\![a]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{Z})$, is defined as $[\![a]\!]S \stackrel{\text{def}}{=} \{(\!|a|\!)\mathsf{m} \mid \mathsf{m} \in S\}$ for any set of stores S. Similarly, for boolean expressions $b \in \mathtt{BExp}$, $[\![b]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ is defined as $[\![b]\!]S \stackrel{\text{def}}{=} \{\mathsf{m} \in S \mid (\!|b|\!)\mathsf{m} = \mathbf{tt}\}$, i.e., $[\![b]\!]S$ filters the stores of S which make b true. The collecting semantics of a program P is the function $[\![P]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$, defined inductively as follows:

$$
\begin{aligned}
[\![\mathbf{skip}]\!]S &\stackrel{\text{def}}{=} S \\
[\![x := a]\!]S &\stackrel{\text{def}}{=} \{\ \mathsf{m}[x \mapsto (\!|a|\!)\mathsf{m}] \mid \mathsf{m} \in S\ \} \\
[\![P_1; P_2]\!]S &\stackrel{\text{def}}{=} [\![P_2]\!]([\![P_1]\!]S) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]S &\stackrel{\text{def}}{=} [\![P_1]\!]([\![b]\!]S) \cup [\![P_2]\!]([\![\neg b]\!]S) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ P]\!]S &\stackrel{\text{def}}{=} [\![\neg b]\!]\Big(\mathit{lfp}(\lambda T.\, S \cup [\![P]\!]([\![b]\!]T))\Big).
\end{aligned}
$$

In this case $\lambda \mathsf{m} \in \mathbb{S}\,.\,[\![P]\!]\{\mathsf{m}\}$ is the partial recursive function computed by P, where $[\![P]\!]\{\mathsf{m}\} = \varnothing$ means non-termination of program P when evaluated in the store m. Conversely, when a program terminates on a store m we have $[\![P]\!]\{\mathsf{m}\} = \{\mathsf{m}'\}$ for a suitable store m'. Note that, when P does not contain any assignment to a variable x, if $[\![P]\!]\{\mathsf{m}\} = \{\mathsf{m}'\}$ then $\mathsf{m}'(x) = \mathsf{m}(x)$.

Despite the definition of collecting semantics applies to any subset of $\mathbb{S}$, in the following we consider only sets of stores that predicate over a finite (possibly unbounded) set of variables, as is always the case in abstract interpretation. This still allows assigning the same variable x infinitely many different values by the stores in a set $S \subseteq \mathbb{S}$.

Definition 3.1 (Variable finite sets of stores)**.** We say a set of stores $S \subseteq \mathbb{S}$ is *variable finite* if $|\mathit{var}(S)| < \omega$. We denote by $\widehat{\wp}(\mathbb{S})$ the powerset of variable finite sets of stores, together with the top element $\mathbb{S}$, i.e.

$$
\widehat{\wp}(\mathbb{S}) \stackrel{\text{def}}{=} \Big\{\ S \subseteq \mathbb{S} \ \Big|\ |\mathit{var}(S)| < \omega \vee S = \mathbb{S}\ \Big\}.
$$

Lemma 3.1. *$\langle \widehat{\wp}(\mathbb{S}), \subseteq, \cup, \cap, \mathbb{S}, \varnothing\rangle$ is a complete lattice.*

Proof. Obviously the bottom element is $\varnothing$ and the top element is $\mathbb{S}$. The fact that $\widehat{\wp}(\mathbb{S})$ is closed under intersection follows from the fact that countable intersection of variable finite sets is also variable finite. For countable union, if the ordinary union is not variable finite, then we can always take $\mathbb{S}$ as the lub. □

Because programs always manipulate a finite set of variables, the concrete collecting semantics $[\![\cdot]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ defined above can be seen as restricted to $\widehat{\wp}(\mathbb{S})$, that is $[\![\cdot]\!] : \widehat{\wp}(\mathbb{S}) \to \widehat{\wp}(\mathbb{S})$. We will often abuse notation and represent with $[\![P]\!]$ both the above mentioned collecting semantics (i.e., a total function from set of stores to set of stores) and the ordinary denotational semantics of P (i.e., a partial function $\lambda x.\, [\![P]\!](\{x\})$ from stores to stores).

Some points of the concrete domain are particularly important because their membership can be effectively tested with computable predicates. We call such sets of stores recursive.

Definition 3.2 (Recursive set of stores)**.** Let $\mathsf{m}_{|V} : \mathit{Var} \to \mathbb{Z}$ be the restriction of m to the set $V \subseteq \mathit{Var}$ defined as

$$\mathsf{m}_{|V}(x) \stackrel{\text{def}}{=} \begin{cases} \mathsf{m}(x) & \text{if } x \in V \\ \$ & \text{otherwise} \end{cases}$$

and similarly let $S_{|V} \stackrel{\text{def}}{=} \{\mathsf{m}_{|V} \mid \mathsf{m} \in S\}$.

We say that $S \in \widehat{\wp}(\mathbb{S})$ is *recursive* iff there exists a total recursive function f_S that decides membership in $\left\{ \mathsf{m} \in \mathbb{S} \;\middle|\; \mathsf{m}_{|\mathit{var}(S)} \in S_{|\mathit{var}(S)} \right\}$.

Note that for any finite set of variables V the set of stores $\mathbb{S}_{|V}$ belongs to $\widehat{\wp}(\mathbb{S})$ and for any $S \in \widehat{\wp}(\mathbb{S})$ we have $S \subseteq \mathbb{S}_{|\mathit{var}(S)}$.

3.3.2 Abstract Semantics

We now define the *abstract semantics* for a generic abstract domain $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$. As usual in abstract interpretation, we are interested in abstract domains whose elements represent recursive properties (sets) of stores. This because membership, i.e, whether any concrete computed store satisfies the property detected by the analyzer at a given program point, must be decidable in order for the analysis to be useful.

Definition 3.3 (Recursive abstract domain)**.** Given an abstract domain A, an abstract store $S^\sharp \in \mathsf{A}$ is *recursive* if $\gamma(S^\sharp)$ is recursive (according to Def. 3.2). A is *recursive* if all its elements are recursive.

In the following we only consider recursive or trivial abstractions of $\widehat{\wp}(\mathbb{S})$. [H2]

Note that $\top\!\!\top$ is recursive while `id` is not.

It is well known that abstract interpretation is not compositional, namely the composition of two best correct abstract semantics may not be the best correct abstraction of the semantics of the two components. This is indeed the main source of imprecision in program analysis, in addition to the abstract domain, e.g. $\top\!\!\top$. In the following, for the sake of simplicity, we will focus only on loss of precision induced by the inductive composition of commands. Therefore, for arithmetic or boolean expressions $e \in \texttt{AExp} \cup \texttt{BExp}$, we consider as abstract semantics their best correct approximating semantics in A, i.e., we consider the ideal situation where the function $[\![e]\!]^{\mathsf{A}} \stackrel{\text{def}}{=} \alpha \circ [\![e]\!] \circ \gamma$ is computable and introduces no loss of information in our language. This way the abstract semantics of expressions is computed independently from their syntax, and we focus on the composition of commands as the main source of imprecision in our presentation. This assumption can be of course weakened by composing the abstract semantics of sub-expressions, at the price of further loss of precision which is not relevant to our purposes.

The best correct abstract semantics $[\![P]\!]^{\mathsf{A}} : A \to A$ of a program $P \in \texttt{Imp}$ is therefore inductively defined on the syntax of commands as composition of the best correct approximations of their concrete semantics. As above, we assume that the best correct approximation of assignments are computable in our language. In particular in any Galois insertion, the abstract join $\sqcup$ is by definition the best correct approximation of the concrete join $\cup$. If $S^\sharp \in A$ is an abstract store:

$$
\begin{aligned}
[\![\textbf{skip}]\!]^{\mathsf{A}} S^\sharp &\stackrel{\text{def}}{=} S^\sharp \\
[\![x := a]\!]^{\mathsf{A}} S^\sharp &\stackrel{\text{def}}{=} \alpha(\{\ \mathsf{m}[x \mapsto (\!|a|\!)\mathsf{m}] \mid \mathsf{m} \in \gamma(S^\sharp)\ \}) \\
[\![P_1; P_2]\!]^{\mathsf{A}} S^\sharp &\stackrel{\text{def}}{=} [\![P_2]\!]^{\mathsf{A}}([\![P_1]\!]^{\mathsf{A}} S^\sharp) \\
[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]^{\mathsf{A}} S^\sharp &\stackrel{\text{def}}{=} [\![P_1]\!]^{\mathsf{A}}([\![b]\!]^{\mathsf{A}} S^\sharp) \sqcup [\![P_2]\!]^{\mathsf{A}}([\![\neg b]\!]^{\mathsf{A}} S^\sharp) \\
[\![\textbf{while } b \textbf{ do } P]\!]^{\mathsf{A}} S^\sharp &\stackrel{\text{def}}{=} [\![\neg b]\!]^{\mathsf{A}}\Big(\mathit{lfp}(\lambda T^\sharp.\, S^\sharp \sqcup [\![P]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{A}} T^\sharp)\Big)
\end{aligned}
$$

Note that the best correct approximation of an assignment $[\![x := a]\!]^{\mathsf{A}}$ does not rely on the best correct abstract semantics $[\![a]\!]^{\mathsf{A}}$ of the arithmetic expression a. Moreover in the least fixpoint definition for $[\![\textbf{while } b \textbf{ do } P]\!]^{\mathsf{A}}$, the abstract function $\lambda T^\sharp.\, S^\sharp \sqcup [\![P]\!]^{\mathsf{A}}([\![b]\!]^{\mathsf{A}} T^\sharp)$ turns out to be the best correct approximation on A of the concrete function $\lambda T.\, S \cup [\![P]\!][\![b]\!]T$. Because the abstract semantics $[\![\cdot]\!]^{\mathsf{A}}$ above is fully determined by the abstract domain A, we often abuse notation and indicate A as abstract semantics or abstract interpretation.

In the following we assume that the best correct approximation in A of expressions and assignments are computable in our language. [H3]

Example 3.3. As a running example of an abstract interpretation, consider the abstract domain of intervals Int on the integers $\mathbb{Z}$, already mentioned in the Introduction. Elements of Int are finite intervals $[a, b]$ with $a \leq b$, or infinite intervals of the form $[-\infty, b]$ or $[a, \infty]$, together with the empty interval $\varnothing$ (the bottom element). The top element is $[-\infty, \infty]$. Intervals are ordered by inclusion. The concretization function γ is defined as expected, while the abstraction function α maps a set of integers to the smallest interval that contains it [37].

In this context, let the default value \$ be 0, so that in the abstract store the default value is the interval $[0, 0]$. Moreover, since the abstraction is non-relational, an abstract store $S^\sharp$ maps a finite number of variables to (non-default) intervals of the form $[a, b]$ with

$$\gamma(S^\sharp) = \{\ \mathsf{m}\ \mid\ \forall y, a, b.\ S^\sharp(y) = [a, b] \Rightarrow a \leq \mathsf{m}(y) \leq b\ \}$$

and for any set of stores S we let

$$\alpha(S)(y) = [\min_{\mathsf{m} \in S} \mathsf{m}(y), \max_{\mathsf{m} \in S} \mathsf{m}(y)]$$

Let us consider the program

$$\begin{aligned} P \ \stackrel{def}{=}\ & \textbf{\textit{if}}\ x = 1\ \textbf{\textit{then}}\ y := 1 \\ & \textbf{\textit{else}}\ y := 3 \end{aligned}$$

and the concrete set of stores $S = \{\langle x/1\rangle, \langle x/2\rangle\}$. In the collecting semantics we have of course $[\![P]\!]S = \{\langle x/1, y/1\rangle, \langle x/2, y/3\rangle\}$. Let $S^\sharp = \alpha(S)(x)$. Clearly $S^\sharp(x) = [1, 2]$. Then, we have

$$[\![P]\!]^{\mathsf{A}} S^\sharp \quad = \quad [\![y := 1]\!]^{\mathsf{A}}([\![x = 1]\!]^{\mathsf{A}} S^\sharp) \sqcup [\![y := 3]\!]^{\mathsf{A}}([\![x \neq 1]\!]^{\mathsf{A}} S^\sharp)$$

Now let $[\![x = 1]\!]^{\mathsf{A}} S^\sharp = S_1^\sharp$ and $[\![x \neq 1]\!]^{\mathsf{A}} S^\sharp = S_2^\sharp$, then $S_1^\sharp(x) = [1, 1]$ and $S_2^\sharp(x) = [2, 2]$. Therefore

$$[\![P]\!]^{\mathsf{A}} S^\sharp = ([\![y := 1]\!]^{\mathsf{A}} S_1^\sharp) \sqcup ([\![y := 3]\!]^{\mathsf{A}} S_2^\sharp) = T^\sharp$$

with $T^\sharp(x) = [1, 2]$ and $T^\sharp(y) = [1, 3]$. Thus, in this particular case we have $\alpha([\![P]\!]S) = [\![P]\!]^{\mathsf{A}} \alpha(S)$.

Below we list some technical lemmas about the abstract semantics of programs that will be used in the proofs of our main results. The proofs of these results are quite standard since they are derived directly from the definitions.

Lemma 3.2. *For any arithmetic expression $a \in \texttt{AExp}$, boolean expression $b \in \texttt{BExp}$, and program $P \in \texttt{Imp}$, we have that $[\![a]\!]^{\mathsf{A}}$, $[\![b]\!]^{\mathsf{A}}$ and $[\![P]\!]^{\mathsf{A}}$ are monotone.*

Lemma 3.3. *For any boolean expression $b \in \texttt{BExp}$ and any abstract store $S^\sharp \in A$,*

$$[\![b]\!]^{\mathsf{A}}([\![b]\!]^{\mathsf{A}} S^\sharp) = [\![b]\!]^{\mathsf{A}} S^\sharp \ .$$

Lemma 3.4. *For any program $P \in \texttt{Imp}$ and any $S^\sharp, T^\sharp \in A$, if $S^\sharp \leq T^\sharp$ we have*

$$[\![P]\!]^{\mathsf{A}} S^\sharp \sqcup [\![P]\!]^{\mathsf{A}} T^\sharp = [\![P]\!]^{\mathsf{A}} T^\sharp \ .$$

Lemma 3.4 follows by monotonicity of $[\![P]\!]^{\mathsf{A}}$ (Lemma 3.2).

Some abstract domains preserve, in the abstraction, the variable finiteness condition of the concrete domain. We will exploit this condition to prove some important properties of the abstraction.

Definition 3.4 (Variable finite abstract domains)**.** We say that an abstract domain A is *variable finite* if for any $S \in \widehat{\wp}(\mathbb{S})$ we have $var((\gamma \circ \alpha)S) = var(S)$.

The condition of variable finiteness amounts to require that the abstraction does not introduce information about unused variables and therefore preserves the variable finiteness of any concrete set $S \neq \mathbb{S}$. It is worth noting that except for the top abstraction, most standard abstract domains in abstract interpretation are variable finite. This is because programs manipulate a finite set of variables and it is always possible to increase the number of variables of interest by building a new program. Abstract domains are therefore usually defined having V as parameter.

A simple consequence of Def. 3.4 is that for any variable finite abstract domain A we have that $\alpha(S) = \alpha(\mathbb{S})$ implies $S = \mathbb{S}$. This because any $S \neq \mathbb{S}$ satisfies that $S \subseteq \mathbb{S}_{|var(S)}$ and therefore we have $\alpha(S) \leq \alpha(\mathbb{S}_{|var(S)}) < \alpha(\mathbb{S})$. The following lemma says that when A is variable finite, then $[\![P]\!]^{\mathsf{A}}$ cannot map to top an abstract element which is different from top.

Lemma 3.5. *If A is variable finite, for any $P \in \texttt{Imp}$ and $S^\sharp \neq \alpha(\mathbb{S})$, we have that $[\![P]\!]^{\mathsf{A}} S^\sharp \neq \alpha(\mathbb{S})$.*

The proof of Lemma 3.5 is by structural induction on the program P, exploiting the variable finiteness condition on the abstract domain and the fact that P can only manipulate a finite number of variables.

It is worth noting that, as a consequence of [H2], if a domain is variable finite and recursive then it is not trivial. In fact the identity abstraction is not recursive and the top abstraction is not variable finite.

Completeness classes The notion of completeness class of programs has been introduced in [67] as the set of all programs for which an abstract interpretation is complete:

$$\mathbb{C}(\mathsf{A}) \stackrel{\text{def}}{=} \left\{ P \in \texttt{Imp} \;\middle|\; \alpha \circ [\![P]\!] = [\![P]\!]^{\mathsf{A}} \circ \alpha \right\}.$$

Roughly speaking, the completeness class $\mathbb{C}(\mathsf{A})$ is defined to be the set of all programs whose static analysis on a given abstraction A will never produce false alarms. By complement notation, we denote by $\overline{\mathbb{C}(\mathscr{A})}$ the set of all programs whose abstract analysis can produce false alarms. This is a property of programs with respect to a fixed abstraction. It is worth noting that the number of programs that meet this property is always infinite. For any abstract domain A whose abstraction function is computable we have that $|\mathbb{C}(\mathsf{A})| = \omega$. This is shown by a straightforward padding argument by observing that $\mathbf{skip} \in \mathbb{C}(\mathsf{A})$ for any A and because the composition of two complete functions is complete, therefore the sequential composition of complete commands is still complete, i.e., if $P \in \mathbb{C}(\mathsf{A})$ then $\mathbf{skip}; P \in \mathbb{C}(\mathsf{A})$. In [67] the authors proved that $\mathbb{C}(\mathsf{A}) = \texttt{Imp}$ if and only if A is trivial, moreover $\mathbb{C}(\mathsf{A})$ is a non-recursive enumerable set for non-trivial abstractions.

3.4 Abstract extensionality

A set (i.e., a property) of programs $\Pi \subseteq \texttt{Imp}$, or an index set for partial recursive functions, is Rice-extensional when

$$P \in \Pi \;\wedge\; [\![P]\!] = [\![Q]\!] \;\Longrightarrow\; Q \in \Pi \tag{3.1}$$

In this case $P \in \Pi$ is the index of the partial recursive function $[\![P]\!]$. We recall that an index set Π is recursive if and only if it is trivial, i.e., $\Pi = \varnothing$ or $\Pi = \texttt{Imp}$. This is the well known Rice theorem [153]. In the following we generalise this notion by replacing the concrete semantics $[\![\cdot]\!]$ with a generic abstract semantics $[\![\cdot]\!]^{\mathsf{A}}$ for an abstract domain A. This allows us introducing a parametric notion of extensionality, called *abstract extensionality* which depends upon the abstraction A.

Definition 3.5. Let A be an abstract domain. An (abstract) A-index set for partial recursive functions or abstract program property is any $\Pi^{\mathsf{A}} \subseteq \texttt{Imp}$ such that

$$P \in \Pi^{\mathsf{A}} \;\wedge\; [\![P]\!]^{\mathsf{A}} = [\![Q]\!]^{\mathsf{A}} \;\Longrightarrow\; Q \in \Pi^{\mathsf{A}}$$

Abstract program properties are properties of programs that are closed by abstract semantics. All properties in program analysis as formalized by abstract interpretation are abstract semantic properties, or equivalently induce an abstract index set for partial recursive functions. This models precisely program analysis as an equivalence relation on programs.

Theorem 3.6. *If A is trivial then Π^{A} is Rice-extensional.*

Proof. If $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ is trivial then $\gamma \circ \alpha = \mathtt{id}$ or $\gamma \circ \alpha = \top\!\!\top$. In the first case any abstract program property $\Pi^{\mathtt{id}}$ coincides precisely with the notion of Rice-extensionality, as $[\![P]\!]^{\mathtt{id}} = [\![Q]\!]^{\mathtt{id}}$ iff $[\![P]\!] = [\![Q]\!]$. In the second case, if $\Pi^{\top\!\!\top} = \varnothing$ then $\Pi^{\top\!\!\top}$ is vacuously Rice-extensional. If instead $\Pi^{\top\!\!\top} \neq \varnothing$ then $\Pi^{\top\!\!\top} = \mathtt{Imp}$, because for any $P, Q \in \mathtt{Imp}$: $[\![P]\!]^{\top\!\!\top} = [\![Q]\!]^{\top\!\!\top} = \top_A$, and thus $\Pi^{\top\!\!\top}$ is Rice-extensional. □

We know that, in general, an abstract program property Π^{A} can be non Rice-extensional and that a Rice-extensional property may not be an abstract program property, so that the two notions of Rice-extensional property of programs and abstract program property are in general incomparable (e.g., see Examples 3.1 and 3.2).

Given an abstraction A, we consider the following similarity relation on programs based on the equivalence of the analysis performed by the abstract interpreter induced by A:

$$P \approx^{\mathsf{A}} Q \;\text{ iff }\; [\![P]\!]^{\mathsf{A}} = [\![Q]\!]^{\mathsf{A}}$$

In the same way that Rice-extensional properties of programs are the union of equivalence classes of programs, where two programs are equivalent if they represent the same partial recursive function, abstract program properties are union of similar classes, where two programs are considered equivalent if they have the same abstract semantics, i.e., the same analysis. The following proposition us straightforward from the definitions.

Proposition 3.7. *For any abstraction $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$:*

- *$\approx^{\mathsf{A}}$ is an equivalence relation.*
- *For any Π^{A} there exists $F \subseteq A \to A$ such that*

$$\Pi^{\mathsf{A}} = \bigcup_{f \in F} \left\{ \; P \in \mathtt{Imp} \;\middle|\; [\![P]\!]^{\mathsf{A}} = f \; \right\}.$$

- $\Pi^{\mathsf{A}} = \bigcup_{P \in \Pi^{\mathsf{A}}} [P]_{\approx^{\mathsf{A}}}$.

In order to understand the structure of abstract program properties as sets of indices of partial recursive functions we need to study the structure of the equivalence classes of programs as induced by an abstract semantics $[\![\cdot]\!]^{\mathsf{A}}$. In particular we are interested in determining whether such equivalence classes are Rice-extensional, namely, if they are closed under the Rice-extensional equivalence. We will prove that a partition of programs into Rice-extensional equivalence classes induced by an abstract semantics is possible if and only if the abstraction is trivial, i.e., *all the*

equivalence classes of programs induced by meaningful abstract interpretations are not Rice-extensional. This formalizes the widely accepted folklore for which in program analysis it is always possible to transform a program into a semantically equivalent one for which the abstract semantics (i.e., the analysis) of the source is different from the one of the transformed program [105, 65]. In order to prove this result we need to find a program Q such that for a given $P \in \Pi^{\mathsf{A}}$: $[\![P]\!] = [\![Q]\!]$ but $[\![Q]\!]^{\mathsf{A}} \neq [\![P]\!]^{\mathsf{A}}$.

In the following we prove that for any non-trivial (recursive) abstraction there exist such P, Q. Moreover, for a large class of abstractions, those based on variable finite abstract domains (see Def. 3.4), we prove the stronger result that *for any complete program* P we can always find a program Q and an abstract store $S^\sharp$ such that $[\![P]\!] = [\![Q]\!]$ and $[\![P]\!]^{\mathsf{A}} S^\sharp < [\![Q]\!]^{\mathsf{A}} S^\sharp$ (Theorem 3.15). This can be formalized in terms of incompleteness of Q, i.e., Q is an incomplete version of P with respect to a non-trivial program analysis A (Corollary 3.16). In the case of non variable finite domains we just provide two sample programs P and Q such that $P \in \Pi^{\mathsf{A}}$: $[\![P]\!] = [\![Q]\!]$ but $[\![Q]\!]^{\mathsf{A}} < [\![P]\!]^{\mathsf{A}}$ (see Theorem 3.17). This means that Q is semantically equivalent to P but its abstract semantics produces more false alarms.

3.5 Completeness and incompleteness cliques

Given an abstract interpretation A, the incompleteness (resp. completeness) clique of a program $P \in \texttt{Imp}$ is the set of all programs that compute the same partial function as P and whose abstract semantics is incomplete (resp. complete). Hence in a clique every two distinct programs are semantically equivalent on the concrete domain of stores. In an incompleteness clique these programs produce, when analysed, false alarms, while in a completeness clique they produce no false alarms.

Definition 3.6 (Completeness clique)**.** The *completeness clique* of $P \in \texttt{Imp}$ w.r.t. A is

$$\mathbb{C}(P, \mathsf{A}) \stackrel{\text{def}}{=} \left\{ \, Q \, \middle| \, [\![P]\!] = [\![Q]\!] \, \right\} \cap \mathbb{C}(\mathsf{A}).$$

Definition 3.7 (Incompleteness clique)**.** The *incompleteness clique* of $P \in \texttt{Imp}$ w.r.t. A is

$$\overline{\mathbb{C}}(P, \mathsf{A}) \stackrel{\text{def}}{=} \left\{ \, Q \, \middle| \, [\![P]\!] = [\![Q]\!] \, \right\} \cap \overline{\mathbb{C}(\mathsf{A})}.$$

Completeness (incompleteness) cliques model precisely the notion of semantically equivalent programs for which completeness (incompleteness) holds with respect to a given abstract semantics A. Note that $\overline{\mathbb{C}}(P, \mathsf{A}) \neq \overline{\mathbb{C}(P, \mathsf{A})}$, because $\overline{\mathbb{C}}(P, \mathsf{A})$ only contains programs that compute the same partial function as P, while this is not the case for $\overline{\mathbb{C}(P, \mathsf{A})}$. The cliques $\mathbb{C}(P, \mathsf{A})$ and $\overline{\mathbb{C}}(P, \mathsf{A})$ form a partition of the set $[P]_{\approx^{\mathtt{id}}}$ of extensively equivalent programs to P, therefore $\mathbb{C}(P, \mathsf{A})$ and $\overline{\mathbb{C}}(P, \mathsf{A})$ cannot be

both empty. Whenever A is trivial we have that $\mathbb{C}(\mathsf{A}) = \mathtt{Imp}$ and thus $\overline{\mathbb{C}}(P, \mathsf{A}) = \varnothing$ and $\mathbb{C}(P, \mathsf{A}) = [P]_{\approx^{\mathtt{id}}}$. Also note that it may well happen that $P \notin \mathbb{C}(P, \mathsf{A})$ or that $P \notin \overline{\mathbb{C}}(P, \mathsf{A})$, because A can be either complete or incomplete for P. Moreover for any $P \in \mathtt{Imp}$: (1) if $P \in \mathbb{C}(\mathsf{A})$ then $|\mathbb{C}(P, \mathsf{A})| = \omega$ and (2) if $P \in \overline{\mathbb{C}(\mathsf{A})}$ then $|\overline{\mathbb{C}}(P, \mathsf{A})| = \omega$. This is indeed obvious in both cases by padding P with **skip**.

As well as complexity cliques [10], completeness and incompleteness cliques are not in general extensional and enjoy interesting recursive properties that show the nature of false alarms in program analysis and the limit of their systematic removal. In particular, there are infinitely many abstract domains for which it is impossible to systematically remove false alarms by effective code transformations. This is proved by the following theorem, where we denote by $X \preceq_m Y$ the *many-to-one reducibility*, i.e., the existence of a total recursive function f such that $x \in X \;\Leftrightarrow\; f(x) \in Y$.

Theorem 3.8. *For any strict ACC abstract domain A, there exists $P \in \mathtt{Imp}$ such that if $\overline{\mathbb{C}}(P, \mathsf{A}) \neq \varnothing$ then $\overline{\mathbb{C}}(P, \mathsf{A}) \not\preceq_m \mathbb{C}(P, \mathsf{A})$.*

Proof. Note that any strict ACC abstract domain A is non-trivial: The identity abstraction is not ACC and the topmost abstraction is such that $\gamma(\bot_A) = \mathbb{S}$ contradicting the strictness condition $\gamma(\bot_A) = \varnothing$. Assume by contradiction that for all $P \in \mathtt{Imp}$: $\overline{\mathbb{C}}(P, \mathsf{A}) \neq \varnothing$ and $\overline{\mathbb{C}}(P, \mathsf{A}) \preceq_m \mathbb{C}(P, \mathsf{A})$. $\overline{\mathbb{C}}(P, \mathsf{A}) \preceq_m \mathbb{C}(P, \mathsf{A})$ implies that if $\mathbb{C}(P, \mathsf{A}) = \varnothing$ then also $\overline{\mathbb{C}}(P, \mathsf{A}) = \varnothing$, which contradicts the hypothesis. Then we can conclude that $\mathbb{C}(P, \mathsf{A}) \neq \varnothing$. By the above assumption, for any $P \in \mathtt{Imp}$ there exists $f_P : \mathtt{Imp} \to \mathtt{Imp}$ which is total recursive and $Q \in \overline{\mathbb{C}}(P, \mathsf{A}) \Leftrightarrow f_P(Q) \in \mathbb{C}(P, \mathsf{A})$. Hence for any $P \in \mathtt{Imp}$, any program $Q \in \overline{\mathbb{C}}(P, \mathsf{A})$ is such that $[\![P]\!] = [\![f_P(Q)]\!]$ and $f_P(Q) \in \mathbb{C}(\mathsf{A})$. Because A is strict then $\alpha(S) = \bot_A$ iff $S = \varnothing$, therefore for any S:

$$[\![P]\!]S = [\![f_P(Q)]\!]S = \varnothing \;\Leftrightarrow\; \alpha([\![f_P(Q)]\!]S) = \bot_A \;\Leftrightarrow\; [\![f_P(Q)]\!]^{\mathsf{A}}\alpha(S) = \bot_A$$

because $f_P(Q) \in \mathbb{C}(\mathsf{A})$. We know that the abstract domain A is ACC therefore it is decidable whether $[\![f_P(Q)]\!]^{\mathsf{A}}\alpha(S) = \bot_A$, namely whether $[\![P]\!]S = \varnothing$, for any $S \in \widehat{\wp}(\mathbb{S})$. It would therefore be possible to effectively transform any program P into an equivalent one $f_P(Q)$ for which termination is decidable, which is impossible. □

Our goal now is to show that, when A is variable finite and not trivial, for any program $P \in \mathtt{Imp}$ we can always effectively generate a program $\tau(P) \in \overline{\mathbb{C}}(P, \mathsf{A})$, therefore for any $P \in \mathtt{Imp}$: $\overline{\mathbb{C}}(P, \mathsf{A}) \neq \varnothing$. This has relevant consequences on the structure of the class of incomplete programs $\overline{\mathbb{C}(\mathsf{A})}$. In the above case, $\overline{\mathbb{C}(\mathsf{A})}$ includes at least a program for all computable functions, i.e., it is a Turing complete language.

3.6 Reducing completeness to incompleteness

In this section we investigate the code transformation aiming at making the analysis of a program incomplete for a given non-trivial abstract domain, still preserving the concrete semantics. As observed above, trivial abstract domains A, being complete for any program, cause $\overline{\mathbb{C}}(P, \mathsf{A})$ to be empty, making the definition of a transformation in these cases unfeasible. For non-trivial abstract domains, our goal is to define a transformation function $\tau : \mathbb{C}(P, \mathsf{A}) \to \overline{\mathbb{C}}(P, \mathsf{A})$ that must satisfy the following conditions:

1. *Semantics preservation*: For each $P \in \texttt{Imp} : \ [\![P]\!] = [\![\tau(P)]\!]$.
2. *Incompleteness*: There exists $S \in \wp(\mathsf{S}) : \ \alpha([\![\tau(P)]\!]S) < [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S)$.

The first condition states that transformed programs must have the same functional behaviour on every variable. The second condition requires that the abstract analysis is incomplete for at least one set of stores.

We introduce a simple control-flow transformation that injects in the program some dead code that is not recognized by the abstract semantics. The main idea is to exploit the dead code assigning fixed values to unused variables of the program so that the abstract semantics of the transformed program would report that change. We use a conditional statement with an opaque predicate [31] in the abstraction so that the abstract control-flow has to take both branches into account, even if only one of them is actually taken in any concrete execution.

In order to define our control-flow transformation we just require that the abstract domain is *variable finite* (see Def. 3.4) and *recursive* (see Def. 3.3). Equivalently, these are the variable finite abstract domains that are not trivial (because of our hypothesis [H2]).

The next lemma guarantees that recursive abstract domains cannot be the identical abstraction on recursive sets, namely there exists a concrete recursive set S that is not exactly represented in the abstract domain. The main results in this section (namely, Theorems 3.14 and 3.15 and Corollary 3.16) additionally require that the abstract element $\alpha(S)$ is different than top, which is a consequence of the variable finiteness condition (see Corollary 3.10). We will exploit this S to inject incompleteness in arbitrary programs in `Imp`.

Lemma 3.9. *For any recursive abstract domain A there exists a recursive set $S \in \wp(\mathsf{S})$ that is not exactly represented in A.*

Proof. Towards a contradiction, assume that all recursive sets in $\wp(\mathsf{S})$ are exactly represented in A, i.e., $(\gamma \circ \alpha)S = S$ for any recursive set $S \in \wp(\mathsf{S})$. Let U_k be the set

of indices of Turing machines that after k steps on input 0 do not terminate. For any $k \in \mathbb{N}$: U_k is a recursive set. With each U_k we can associate a variable finite set of stores $S_k = \left\{ \langle x/v \rangle \;\middle|\; v \in U_k \right\}$. Clearly the set $U = \bigcap_k U_k$ is the set of indices of Turing machines that on input 0 do not terminate, which is not recursive. Let $S = \left\{ \langle x/v \rangle \;\middle|\; v \in U \right\}$. Because the pair $\langle \alpha, \gamma \rangle$ forms a Galois insertion: $\bigwedge_k \alpha(S_k) \in \mathsf{A}$. Being A recursive, the set $T = \gamma(\bigwedge_k \alpha(S_k))$ is also recursive. Since γ is co-additive we have $T = \gamma(\bigwedge_k \alpha(S_k)) = \bigcap_k \gamma(\alpha(S_k))$. Since we have assumed that all recursive sets are exactly represented, we have $T = \bigcap_k \gamma(\alpha(S_k)) = \bigcap_k S_k = S$, which yields a contradiction, because T is recursive while S is not. □

Corollary 3.10. *If A is variable finite and recursive, then there exists a recursive $S \in \widehat{\wp}(\mathbb{S})$ such that*

$$S \subset (\gamma \circ \alpha)S \subset \mathbb{S}\,.$$

Proof. The existence of a recursive set S such that $S \subset (\gamma \circ \alpha)S$ is guaranteed by Lemma 3.9. Since A is variable finite it follows that for any $V \subset Var$:

$$|V| < \omega \implies \alpha(\mathbb{S}_{|V}) < \alpha(\mathbb{S}).$$

This means that, for any finite set of variables V, the abstract domain has to represent the set $\mathbb{S}_{|V}$ of all stores defined at most on V, and this abstract object $\alpha(\mathbb{S}_{|V})$ cannot represent the set of all stores. Clearly, $S \subseteq \mathbb{S}_{|var(S)}$, and thus $(\gamma \circ \alpha)S \leq (\gamma \circ \alpha)(\mathbb{S}_{|var(S)}) < \mathbb{S}$, by the properties of Galois insertions and because the abstract domain is variable finite. □

Let S be a set satisfying the hypothesis of Corollary 3.10. Let $V = var((\gamma \circ \alpha)S) = var(S)$, by the variable finiteness condition. Since S is recursive, there exists a total recursive function f such that

$$f(\mathsf{m}) = \begin{cases} 1 \text{ if } \mathsf{m} \in \left\{ \mathsf{m} \;\middle|\; \mathsf{m}_{|V} \in S_{|V} \right\} \\ 0 \text{ otherwise} \end{cases}$$

Since f is at most concerned with only a finite list of variables $\widetilde{x} \subseteq V$ it can be expressed as an arithmetic expression $f(\widetilde{x})$ such that $(\!|f(\widetilde{x})|\!)\mathsf{m} = 1$ if $\mathsf{m} \in \left\{ \mathsf{m} \;\middle|\; \mathsf{m}_{|V} \in S_{|V} \right\}$ and $(\!|f(\widetilde{x})|\!)\mathsf{m} = 0$ otherwise. We let the predicate $\mathtt{In}(S)$ be defined as $f(\widetilde{x}) = 1$. We prove that $\mathtt{In}(S)$ is an opaque predicate for the abstract interpreter, i.e., the abstract interpreter cannot decide whether $\mathtt{In}(S)$ is true or false on some input property of memories. In this way $\mathtt{In}(S)$ may drive the injection in the abstract semantics of values that are not computed in the concrete semantics. We design this by an assignment that will produce some store outside the scope

of P. Since $(\gamma \circ \alpha)S \subset \mathbb{S}$, then there exists at least one store $\mathsf{m}' \notin (\gamma \circ \alpha)S$. Moreover, for any program P, we have $[\![P]\!]^{\mathsf{A}}\alpha(S) < \alpha(\mathbb{S})$ by Lemma 3.5, because $(\gamma \circ \alpha)S \subset \mathbb{S}$. Therefore, without loss of generality, we can find a store $\mathsf{m}' \notin (\gamma \circ \alpha)S$ such that $var(\mathsf{m}')$ is disjoint from $var((\gamma \circ [\![P]\!]^{\mathsf{A}} \circ \alpha)S)$, because the (non-top) points of our concrete domain $\widehat{\wp}(\mathbb{S})$ predicate over finite set of variables. Suppose $\mathsf{m}' = \langle x_1/v_1, \ldots, x_n/v_n \rangle$ with $var((\gamma \circ \alpha)S) \setminus var(\mathsf{m}') = \{x_{n+1}, ..., x_{n+k}\}$ we let $\mathtt{Set}(\mathsf{m}', S)$ be the program

$$x_1 := v_1; \ldots; x_n := v_n; x_{n+1} := \$; \ldots; x_{n+k} := \$$$

We transform P into:

$$\begin{array}{rcl} \tau_{\mathsf{m}',S}(P) & \stackrel{\text{def}}{=} & \textbf{if } \mathtt{In}(S) \textbf{ then} \\ & & \quad \textbf{if } \neg\mathtt{In}(S) \textbf{ then } \mathtt{Set}(\mathsf{m}', S) \\ & & \quad \textbf{else } P \\ & & \textbf{else } P \end{array}$$

The intuition of the transformation is the following: since $S \subset (\gamma \circ \alpha)S$ there will be at least one set of stores that can go through both branches in the abstract evaluation. As a consequence $\mathtt{Set}(\mathsf{m}', S)$ is dead code for the concrete semantics but it is not dead code for the abstract one, marking $\tau_{\mathsf{m}',S}(P)$ incomplete for A.

Note that in $\tau_{\mathsf{m}',S}(P)$, the choice of S may be independent of P, while that of m' is not. For brevity we denote by $\tau(P)$ the code injection transformation $\tau_{\mathsf{m}',S}(P)$ for suitable m' and S.

Proposition 3.11. *For any $P \in \mathtt{Imp}$, with τ defined as above, $[\![\tau(P)]\!] = [\![P]\!]$.*

Proof. In the following we let

$$\begin{array}{rcl} Q & \stackrel{\text{def}}{=} & \textbf{if } \neg\mathtt{In}(S) \textbf{ then } \mathtt{Set}(\mathsf{m}', S) \\ & & \textbf{else } P \end{array}$$

such that $\tau(P) = \textbf{if } \mathtt{In}(S) \textbf{ then } Q \textbf{ else } P$. Then, for any $\mathsf{m} \in \mathbb{S}$ we have:

$$\begin{array}{rcl} [\![\tau(P)]\!]\mathsf{m} & = & [\![Q]\!]([\![\mathtt{In}(S)]\!]\mathsf{m}) \cup [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathsf{m}) \\ & = & [\![\mathtt{Set}(\mathsf{m}', S)]\!]([\![\neg\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathsf{m})) \\ & & \cup\, [\![P]\!]([\![\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathsf{m})) \cup\ [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathsf{m}) \\ & = & [\![P]\!]([\![\mathtt{In}(S)]\!]\mathsf{m}) \cup [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathsf{m}) \\ & = & [\![P]\!]\mathsf{m} \end{array}$$

because $[\![\neg\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathsf{m}) = \varnothing$ and either $[\![\mathtt{In}(S)]\!]\mathsf{m} = \{\mathsf{m}\}$ and $[\![\neg\mathtt{In}(S)]\!]\mathsf{m} = \varnothing$ or $[\![\mathtt{In}(S)]\!]\mathsf{m} = \varnothing$ and $[\![\neg\mathtt{In}(S)]\!]\mathsf{m} = \{\mathsf{m}\}$. □

We show in Theorem 3.14 that the above control-flow transformation makes any program incomplete. In the proof we exploit the following two technical lemmas.

Lemma 3.12. *Let A be variable finite and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$. Then $[\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) = \alpha(S)$.*

Proof. By definition, $[\![\mathtt{In}(S)]\!]^{\mathsf{A}} = \alpha \circ [\![\mathtt{In}(S)]\!] \circ \gamma$. Thus $[\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) = (\alpha \circ [\![\mathtt{In}(S)]\!] \circ \gamma \circ \alpha)S = \alpha(\{(\!|\mathtt{In}(S)|\!)\mathsf{m} \mid \mathsf{m} \in (\gamma \circ \alpha)S\})$. Let $V = \mathit{var}((\gamma \circ \alpha)S) = \mathit{var}(S)$. We recall that $(\!|\mathtt{In}(S)|\!)\mathsf{m} = \mathbf{tt}$ iff $\mathsf{m} \in \{\mathsf{m} \mid \mathsf{m}_{|V} \in S_{|V}\}$. Now, for $\mathsf{m} \in (\gamma \circ \alpha)S$, we have $\mathsf{m}_{|V} \in S_{|V}$ iff $\mathsf{m} \in S$. Therefore we have that $\{(\!|\mathtt{In}(S)|\!)\mathsf{m} \mid \mathsf{m} \in (\gamma \circ \alpha)S\} = S$ and $[\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) = \alpha(S)$. □

Lemma 3.13. *Let A be variable finite and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set s.t. $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$. Then $\bot < [\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) \leq \alpha(S)$.*

Proof. By definition, $[\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}} = \alpha \circ [\![\neg\mathtt{In}(S)]\!] \circ \gamma$. Thus

$$\begin{array}{rcl} [\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) & = & (\alpha \circ [\![\neg\mathtt{In}(S)]\!] \circ \gamma \circ \alpha)S \\ & = & \alpha(\{(\!|\neg\mathtt{In}(S)|\!)\mathsf{m} \mid \mathsf{m} \in (\gamma \circ \alpha)S\}) \end{array}$$

Let $S' = \{(\!|\neg\mathtt{In}(S)|\!)\mathsf{m} \mid \mathsf{m} \in (\gamma \circ \alpha)S\} \subseteq (\gamma \circ \alpha)S$. By monotonicity $\alpha(S') \leq (\alpha \circ \gamma \circ \alpha)S = \alpha(S)$ (because the abstract domain defines a Galois insertion). Let $V = \mathit{var}((\gamma \circ \alpha)S) = \mathit{var}(S)$. We recall that $(\!|\neg\mathtt{In}(S)|\!)\mathsf{m} = \mathbf{tt}$ iff $\mathsf{m} \notin \{\mathsf{m} \mid \mathsf{m}_{|V} \in S_{|V}\}$. Since $(\gamma \circ \alpha)S \supset S$, there exists some $\mathsf{m}'' \in (\gamma \circ \alpha)S$ and $\mathsf{m}'' \notin \{\mathsf{m} \mid \mathsf{m}_{|V} \in S_{|V}\}$. It follows that $\mathsf{m}'' \in S' \neq \varnothing$. Since A is strict we have $\alpha(S') > \alpha(\varnothing) = \bot$. □

Theorem 3.14. *Let A be variable finite and recursive. Let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$, and τ be defined as above. Then for any $P \in \mathtt{Imp}$: $\alpha([\![\tau(P)]\!]S) < [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S)$.*

Proof. We have:

$$\begin{array}{rcl} [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) & = & [\![\mathtt{Set}(\mathsf{m}', S)]\!]^{\mathsf{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathsf{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \end{array}$$

By Lemmas 3.3 and 3.12: $[\![\mathtt{In}(S)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) = [\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) = \alpha(S)$. Moreover, since

$$[\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \leq \alpha(S),$$

by Lemma 3.4: $[\![P]\!]^{\mathsf{A}}([\![\neg \mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \sqcup [\![P]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) = [\![P]\!]^{\mathsf{A}}\alpha(S)$. Thus

$$\begin{array}{rcl} [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) & = & [\![\mathtt{Set}(\mathsf{m}', S)]\!]^{\mathsf{A}}([\![\neg \mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \\ & & \sqcup\, [\![P]\!]^{\mathsf{A}}\alpha(S) \end{array}$$

By Lemma 3.13 there exists S' with $\bot < \alpha(S') \leq \alpha(S)$ such that $[\![\neg \mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) = \alpha(S')$ and thus $[\![\mathtt{Set}(\mathsf{m}', S)]\!]^{\mathsf{A}}\alpha(S') = \alpha(\{\mathsf{m}'\})$, because $\mathtt{Set}(\mathsf{m}', S)$ sets to \$ all variables in $var((\gamma \circ \alpha)S) \setminus var(\mathsf{m}')$ and $\alpha(S') \leq \alpha(S)$. Hence $[\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) = \alpha(\{\mathsf{m}'\}) \sqcup [\![P]\!]^{\mathsf{A}}\alpha(S)$. By hypothesis $var(\mathsf{m}')$ is disjoint from the set of variables manipulated by $[\![P]\!]^{\mathsf{A}}\alpha(S)$, therefore we have:

$$[\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) = \alpha(\{\mathsf{m}'\}) \sqcup [\![P]\!]^{\mathsf{A}}\alpha(S) > [\![P]\!]^{\mathsf{A}}\alpha(S) \geq \alpha([\![P]\!]S) = \alpha([\![\tau(P)]\!]S)$$

where the last equality follows from Prop. 3.11. □

Theorem 3.15. *Let A be variable finite and recursive. Let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset \gamma \circ \alpha(S) \subset \mathbb{S}$, and τ be defined as above. Then for any $P \in \mathbb{C}(P, \mathsf{A})$: $[\![P]\!]^{\mathsf{A}}\alpha(S) < [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S)$.*

Proof. We have

$$\begin{array}{rcll} [\![P]\!]^{\mathsf{A}}\alpha(S) & = & \alpha([\![P]\!]S) & \text{(because } P \in \mathbb{C}(P, \mathsf{A})) \\ & = & \alpha([\![\tau(P)]\!])S & \text{(by Prop. 3.11)} \\ & < & [\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) & \text{(by Theorem 3.14)} \end{array}$$

□

As a straightforward consequence, the semantics-preserving transformation described above can be used to define an effective transformation $\tau : \mathbb{C}(P, \mathsf{A}) \to \overline{\mathbb{C}}(P, \mathsf{A})$.

Corollary 3.16. *If A is variable finite and recursive and $\mathbb{C}(P, \mathsf{A}) \neq \varnothing$ then there exists a computable code transformation $\tau : \mathbb{C}(P, \mathsf{A}) \to \overline{\mathbb{C}}(P, \mathsf{A})$.*

It follows immediately that if A is variable finite and recursive then for any program $P \in \mathtt{Imp}$: $|\overline{\mathbb{C}}(P, \mathsf{A})| = \omega$. This is because the control-flow transformation introduced above is semantics-preserving. Therefore for any $P \in \mathtt{Imp}$, $\tau(P) \in \overline{\mathbb{C}}(P, \mathsf{A})$. Then by padding with **skip**, or by applying any number of further transformation steps, we obtain infinitely many programs that are equivalent to P but incomplete for A.

Example 3.4. Consider the abstract domain of intervals Int on integer numbers $\mathbb{Z}$ already discussed in the Example 3.3. Clearly Int is strict, recursive, variable finite

and of course non-trivial. A simple Imp program belonging to $\mathbb{C}(\mathsf{Int})$ is for example $x := x + n$ for any $n \in \mathbb{Z}$. Indeed for any store m we have $[\![x := x + n]\!]\mathsf{m} = \mathsf{m}[x \mapsto \mathsf{m}(x) + n]$ and for any set of stores S

$$\alpha([\![x := x + n]\!]S)(y) \quad = \quad \begin{cases} [\min_{\mathsf{m}\in S} \mathsf{m}(y), \max_{\mathsf{m}\in S} \mathsf{m}(y)] & \text{if } y \neq x \\ [\min_{\mathsf{m}\in S}(\mathsf{m}(x) + n), \max_{\mathsf{m}\in S}(\mathsf{m}(x) + n)] & \text{if } y = x \end{cases}$$

while

$$\begin{array}{rcl} [\![x := x + n]\!]^{\mathsf{A}}\alpha(S) & = & \alpha(\{\ \mathsf{m}[x \mapsto \mathsf{m}(x) + n] \mid \mathsf{m} \in \gamma(\alpha(S))\ \}) \\ & = & \alpha(\{\ \mathsf{m}[x \mapsto \mathsf{m}(x) + n] \mid \forall y.\ \mathsf{m}(y) \in \gamma([\min_{\mathsf{m}'\in S} \mathsf{m}'(y), \max_{\mathsf{m}'\in S} \mathsf{m}'(y)])\ \}) \\ & = & \alpha(\{\ \mathsf{m}[x \mapsto \mathsf{m}(x) + n] \mid \forall y.\ \min_{\mathsf{m}'\in S} \mathsf{m}'(y) \leq \mathsf{m}(y) \leq \max_{\mathsf{m}'\in S} \mathsf{m}'(y)\ \}). \end{array}$$

Thus:

$$([\![x := x + n]\!]^{\mathsf{A}}\alpha(S))(y) \quad = \quad \begin{cases} [\min_{\mathsf{m}\in S} \mathsf{m}(y), \max_{\mathsf{m}\in S} \mathsf{m}(y)] & \text{if } y \neq x \\ [\min_{\mathsf{m}\in S}(\mathsf{m}(x) + n), \max_{\mathsf{m}\in S}(\mathsf{m}(x) + n)] & \text{if } y = x \end{cases}$$

Therefore $x := x + n \in \mathbb{C}(\mathsf{Int})$ and therefore $\mathbb{C}(x := x + n, \mathsf{Int})$ is not empty. Let $P = x := x + 1 \in \mathbb{C}(\mathsf{Int})$. We plan to apply our control-flow transformation to derive a program $\tau(P)$ that is equivalent to P but such that $\tau(P) \in \overline{\mathbb{C}(\mathsf{Int})}$ or, in other terms, $\tau(P) \in \overline{\mathbb{C}}(x := x + 1, \mathsf{Int})$. To this aim, let us take the set of stores $S = \{\langle x/1\rangle, \langle x/3\rangle\}$ and $\mathsf{m}' = \langle y/2\rangle$. We have in this case $\mathtt{In}(S) = (x = 1) \vee (x = 3)$. P is then transformed into (@ is an annotation of the program point to ease the presentation):

$$\begin{array}{rcl} \tau(P) & \stackrel{def}{=} & \textbf{\textit{if}}\ \mathtt{In}(S)\ \textbf{\textit{then}} \\ & & \quad \textbf{\textit{if}}\ \neg\mathtt{In}(S)\ \textbf{\textit{then}}\ (y := 2; x := 0) \\ & & \quad \textbf{\textit{else}}\ x := x + 1\ \ (@1) \\ & & \textbf{\textit{else}}\ x := x + 1\ (@2) \end{array}$$

We show that program $\tau(P)$ is incomplete for the abstract domain Int. We have $\alpha([\![\tau(P)]\!]S) = \alpha([\![x := x+1]\!]S) = \alpha(S[x \mapsto \{2,4\}]) = \alpha(S)[x \mapsto [2,4]]$. Moreover because $\alpha(S)(x) = [1,3]$ and $\forall z \neq x, \alpha(S)(z) = [0,0]$, we have:

$$\begin{array}{rcl}
[\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) & = & \alpha(S) \\
[\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S) & = & \alpha(S)[x \mapsto [2,2]] \\
[\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) & = & [\![y := 2; x := 0]\!]^{\mathsf{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S))) \\
 & & \sqcup\, [\![x := x+1\ (@1)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S))) \\
 & & \sqcup\, [\![x := x+1\ (@2)]\!]^{\mathsf{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \\
 & = & [\![y := 2; x := 0]\!]^{\mathsf{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S)) \\
 & & \sqcup\, [\![x := x+1\ (@1)]\!]^{\mathsf{A}}\alpha(S) \\
 & & \sqcup\, [\![x := x+1\ (@2)]\!]^{\mathsf{A}}(\alpha(S)[x \mapsto [2,2]]) \\
 & = & [\![y := 2; x := 0]\!]^{\mathsf{A}}(\alpha(S)[x \mapsto [2,2]]) \sqcup\ \alpha(S)[x \mapsto [2,4]] \sqcup \alpha(S)[x \mapsto [3,3]] \\
 & = & \alpha(S)[y \mapsto [2,2], x \mapsto [0,0]] \sqcup \alpha(S)[x \mapsto [2,4]] = \alpha(S)[x \mapsto [0,4], y \mapsto [0,2]])
\end{array}$$

This shows that the interval abstraction Int is incomplete for the program $\tau(P)$, because e.g.

$$\alpha([\![\tau(P)]\!]S)(y) = (\alpha(S)[x \mapsto [2,4]])(y) = [0,0] < [0,2] = ([\![\mathtt{In}(S)]\!]^{\mathsf{A}}\alpha(S))(y).$$

Example 3.5. As a second example, we consider a simple relational domain formed by two-variables inequalities: $A = \{\bot, x-y < 0, x-y > 0, x-y = 0, x-y \gtreqless 0\}$, where $\bot$ is the bottom element, $x - y \gtreqless 0$ is the top element, and the other three elements are pairwise incomparable. The concrete denotation of an abstract element $S^\sharp$ is the set of stores satisfying all the relationships in $S^\sharp$. For example, given the singleton $S = \{\langle x/1, y/2\rangle\}$ we have:

$$\alpha(S) = \{\forall z \neq y.\ y - z > 0, \quad \forall z \neq x, y.\ x - z > 0, \quad \forall z_1, z_2 \neq x, y.\ z_1 - z_2 = 0\ \}\ .$$

Then we take $\mathsf{m}' = \langle w/3\rangle$ and consider the (complete) program $P \stackrel{\mathsf{def}}{=} u := x$. We have:

$$\begin{array}{rcl}
\tau(P) & \stackrel{def}{=} & \textbf{\textit{if}}\ (x = 1 \wedge y = 2)\ \textbf{\textit{then}} \\
 & & \quad \textbf{\textit{if}}\ (x \neq 1 \vee y \neq 2)\ \textbf{\textit{then}}\ (w := 3; x := 0; y := 0) \\
 & & \quad \textbf{\textit{else}}\ P \\
 & & \textbf{\textit{else}}\ P
\end{array}$$

By some simple calculation we get

$$
\begin{array}{rcl}
[\![P]\!] & = & [\![\tau(P)]\!] \\
[\![x = 1 \wedge y = 2]\!]^{\mathsf{A}}\alpha(S) & = & \alpha(S) \\
[\![x \neq 1 \vee y \neq 2]\!]^{\mathsf{A}}\alpha(S) & = & \alpha(S) \\
\alpha([\![P]\!]S) = [\![P]\!]^{\mathsf{A}}\alpha(S) & = & \{\ \forall z \neq y.\ y - z > 0, \quad \forall z \neq x, y, u.\ x - z > 0, \\
 & & \quad \forall z \neq x, y, u.\ u - z > 0, \quad \forall z_1, z_2 \neq y.\ z_1 - z_2 = 0\ \} \\
[\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) & = & [\![w := 3; x := 0; y := 0]\!]^{\mathsf{A}}([\![x \neq 1 \vee y \neq 2]\!]^{\mathsf{A}}([\![x = 1 \wedge y = 2]\!]^{\mathsf{A}}\alpha(S))) \\
 & & \sqcup\ [\![P]\!]^{\mathsf{A}}([\![x = 1 \wedge y = 2]\!]^{\mathsf{A}}([\![x = 1 \wedge y = 2]\!]^{\mathsf{A}}\alpha(S))) \\
 & & \sqcup\ [\![P]\!]^{\mathsf{A}}([\![x \neq 1 \vee y \neq 2]\!]^{\mathsf{A}}\alpha(S)) \\
 & = & [\![w := 3; x := 0; y := 0]\!]^{\mathsf{A}}\alpha(S) \\
 & & \sqcup\ [\![P]\!]^{\mathsf{A}}\alpha(S) \\
 & = & \{\ \forall z \neq w.\ w - z > 0, \quad \forall z_1, z_2 \neq w.\ z_1 - z_2 = 0\ \} \sqcup\ [\![P]\!]^{\mathsf{A}}\alpha(S) \\
 & = & \{\ \forall z \neq y, w.\ y - z \gtreqless 0, \quad \forall z \neq x, y, u, w.\ x - z \gtreqless 0, \\
 & & \quad \forall z \neq x, y, u, w.\ u - z \gtreqless 0, \quad \forall z \neq w.\ w - z \gtreqless 0, \\
 & & \quad \forall z_1, z_2 \neq y, w.\ z_1 - z_2 = 0\ \}
\end{array}
$$

This shows that the abstraction is incomplete for $\tau(P)$ because $[\![\tau(P)]\!]^{\mathsf{A}}\alpha(S) \neq \alpha([\![P]\!]S) = \alpha([\![\tau(P)]\!]S)$.

3.7 Rice extensionality of the abstract semantics

The top trivial abstraction is an example of non variable finite abstraction. The next theorem proves that whenever A is neither variable finite trivial nor trivial, there exists a pair of programs P and Q such that they have the same concrete semantics $[\![P]\!] = [\![Q]\!]$, but different abstract semantics $[\![P]\!]^{\mathsf{A}} < [\![Q]\!]^{\mathsf{A}}$.

Theorem 3.17. *If A is neither variable finite nor trivial then there exist $P, Q \in \mathtt{Imp}$ such that $[\![P]\!] = [\![Q]\!]$, and $[\![P]\!]^{\mathsf{A}} < [\![Q]\!]^{\mathsf{A}}$.*

Proof. Since $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ is not trivial, it must be strict, and since A is not variable finite then there exists a set of stores $S \in \widehat{\wp}(\mathbb{S})$ such that $var(S) \subset var((\gamma \circ \alpha)S)$. Thus, there exist a store $\mathsf{m} \in (\gamma \circ \alpha)S$ and a variable $x \in var((\gamma \circ \alpha)S) \setminus var(S)$

with $\mathsf{m}(x) = v \neq \$$. Moreover, we have that $\gamma(\alpha(\varnothing)) = \varnothing$, because A is strict. We can now build P and Q such that $[\![P]\!] = [\![Q]\!]$, and $[\![P]\!]^{\mathsf{A}} < [\![Q]\!]^{\mathsf{A}}$ in the following way.

$$P \stackrel{\text{def}}{=} \textbf{while } \textbf{tt } \textbf{do skip}$$

$$\begin{aligned} Q \stackrel{\text{def}}{=} \ & \textbf{if } x \neq v \textbf{ then} \\ & \quad \textbf{if } x = v \ \textbf{ then skip} \\ & \quad \textbf{else } P \\ & \textbf{else } P \end{aligned}$$

Clearly $[\![P]\!] = [\![Q]\!]$. Next we prove that $[\![P]\!]^{\mathsf{A}}\alpha(S) < [\![Q]\!]^{\mathsf{A}}\alpha(S)$. In fact, for any abstract store $S^\sharp$ (including $\alpha(S)$): $[\![P]\!]^{\mathsf{A}}S^\sharp = \bot \leq [\![Q]\!]^{\mathsf{A}}S^\sharp$. Moreover, as far as $\alpha(S)$ is concerned we have:

$$\begin{aligned} [\![Q]\!]^{\mathsf{A}}\alpha(S) &= [\![\textbf{skip}]\!]^{\mathsf{A}}([\![x = v]\!]^{\mathsf{A}}([\![x \neq v]\!]^{\mathsf{A}}\alpha(S))) \\ &\quad \sqcup [\![P]\!]^{\mathsf{A}}([\![x \neq v]\!]^{\mathsf{A}}([\![x \neq v]\!]^{\mathsf{A}}\alpha(S))) \\ &\quad \sqcup [\![P]\!]^{\mathsf{A}}([\![x = v]\!]^{\mathsf{A}}\alpha(S)) \\ &= [\![\textbf{skip}]\!]^{\mathsf{A}}([\![x = v]\!]^{\mathsf{A}}([\![x \neq v]\!]^{\mathsf{A}}\alpha(S))) \\ &\quad \sqcup \bot \\ &= [\![\textbf{skip}]\!]^{\mathsf{A}}([\![x = v]\!]^{\mathsf{A}}([\![x \neq v]\!]^{\mathsf{A}}\alpha(S))) \end{aligned}$$

Note that all the memories in S are such that $\mathsf{m}(x) \neq v$, thus $[\![x \neq v]\!]^{\mathsf{A}}\alpha(S) = \alpha(S)$. Thus

$$\begin{aligned} [\![Q]\!]^{\mathsf{A}}\alpha(S) &= [\![\textbf{skip}]\!]^{\mathsf{A}}([\![x = v]\!]^{\mathsf{A}}\alpha(S)) \\ &= [\![\textbf{skip}]\!]^{\mathsf{A}}\alpha(S') \\ &= \alpha(S') \end{aligned}$$

for some $S' \supset \{\mathsf{m}\} \supset \varnothing$. Since A is strict, $\alpha(S') > \bot$. Hence, we have that $[\![P]\!]^{\mathsf{A}}\alpha(S) < [\![Q]\!]^{\mathsf{A}}\alpha(S)$ and therefore $[\![P]\!]^{\mathsf{A}} < [\![Q]\!]^{\mathsf{A}}$. □

We are now in the position of proving that the equivalence classes $[P]_{\approx^{\mathsf{A}}}$, e.g., those forming a non-trivial abstract program property Π^{A}, are Rice-extensional if and only if A is trivial.

Theorem 3.18. *An abstract domain A is trivial iff $[P]_{\approx^{\mathsf{A}}}$ is Rice-extensional, for any $P \in \texttt{Imp}$.*

Proof. Let $\mathsf{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$. If A is trivial then by Theorem 3.6 for any $P \in \texttt{Imp}$: $[P]_{\approx^{\mathsf{A}}}$ is Rice-extensional. For the converse implication, assume A is non-trivial. If A is variable finite then consider $P \in [P]_{\approx^{\mathsf{A}}}$. By Prop. 3.11: $[\![P]\!] = [\![\tau(P)]\!]$ and Theorem 3.14 $[\![P]\!]^{\mathsf{A}} \neq [\![\tau(P)]\!]^{\mathsf{A}}$, which proves that $[P]_{\approx^{\mathsf{A}}}$ is not Rice-extensional. If instead A is not variable finite then by Theorem 3.17 we have that there exist $P, Q \in \texttt{Imp}$ such that $P \in [P]_{\approx^{\mathsf{A}}}$, $[\![P]\!] = [\![Q]\!]$ and $[\![P]\!]^{\mathsf{A}} \neq [\![Q]\!]^{\mathsf{A}}$, which implies that $[P]_{\approx^{\mathsf{A}}}$ is not Rice extensional. □

3.8 Conclusion

In this chapter we proved that the equivalence induced by abstract semantics on programs is an index set of partial recursive functions if and only if the abstraction is trivial. We considered the strongest possible scenario in order to establish when incompleteness can be injected. In particular the assumption [H1] ensures the existence of the best correct approximation, so that making this approximation incomplete would make incomplete any other weaker approximations, i.e., we proved that incompleteness can be injected in every program also when the abstraction is designed to be the most precise one. This result has important consequences in program analysis and abstract interpretation:

(1) It shows that any non-trivial abstraction of extensional (functional) properties of programs is susceptible to their intensional structure. This means that any non-trivial abstract interpretation always unveils implicitly also properties concerning the way programs are written. While true alarms only concern the extensional (functional) behaviour of the program, false alarms always concern their intensional structure. Stated in a different way: we can look at the log of alarms generated by an abstract interpreter to classify programs according to extensional—what they compute, and intensional—how they are implemented, similarity. This log is a footprint of the code analysed which, to the best of our knowledge, has never been used for program analysis, e.g., in the context of program similarity by encompassing both semantic and implementation similarity.

(2) Program analysis behaves precisely as other well known intensional properties of programs, like computational complexity [10]. This relates program analysis with computational complexity in an unexpected and remarkable way. The question of whether these two fields can be unified under a unique formal setting and what properties and structures are shared by both is still an open question. This is the very first step towards this ambitious goal: *What is left of the standard model of recursive functions when intensional aspects of computation are considered?*

(3) We concentrated our attention on the class $\overline{\mathbb{C}}(P, \mathsf{A})$. This is the space of action of any code protecting transformations whose aim is to foil program analysis and

therefore foil any tool supporting reverse engineering. We proved that the set of all programs that are incomplete for any non-trivial abstraction A, i.e., the set $\overline{\mathbb{C}(\mathsf{A})}$, is a very rich structure: a Turing complete language! This means that it is possible to build a compiler that compiles any program P into an equivalent program in $\overline{\mathbb{C}}(P, \mathsf{A})$, therefore justifying code transformations that protect code against program analysis. On the other side, the expressivity of the class $\mathbb{C}(\mathsf{A})$ of all programs that are complete for a non-trivial abstraction A is still obscure. We know by Theorem 3.8 that for terminating non-trivial program analyses we cannot find a many-to-one reduction of $\overline{\mathbb{C}}(P, \mathsf{A})$ into $\mathbb{C}(P, \mathsf{A})$. This implies that $\mathbb{C}(\mathsf{A})$ cannot be always Turing complete, otherwise by the first Futamura projection (e.g., see [58, 91]) we could build inside $\mathbb{C}(\mathsf{A})$ a compiler τ mapping any program in $\overline{\mathbb{C}}(P, \mathsf{A})$ into $\mathbb{C}(P, \mathsf{A})$, and conversely any program outside $\overline{\mathbb{C}}(P, \mathsf{A})$ into a program outside $\mathbb{C}(P, \mathsf{A})$. The question: *Given a non-trivial abstraction A, what are the functions that we can program in $\mathbb{C}(\mathsf{A})$?* is still open. This question may have relevant applications in automating systematic false alarm removal by refactoring code snippets.

(4) The proofs of our results show that effective program transformations can be derived under a very weak hypothesis, what we called variable finiteness of an abstraction. The connection with code obfuscation is particularly interesting here. Being $\overline{\mathbb{C}}(P, \mathsf{A})$ Turing complete, we believe that code obfuscation, which is nowadays mostly considered a cryptographic concept [13], can be fully reconciled with recursion theory and programming languages.

4

A Framework for Fixpoint Computation in Abstract Interpretation

We now turn to the abstract interpretation procedure per se and present a series of abstract interpretation-based *analysis algorithms* for logic programs. These are the fixpoint-calculating procedures that infer analysis graphs (see Sec. 2.3). *Incremental* algorithms are those that can recompute such analysis graphs after program changes, without having to start the process from scratch. *Modular* algorithms (in contrast to *monolithic* algorithms) are those that are capable of analyzing subsets (of modular partitions) of programs without having to load or treat the whole program at any given step. In this chapter we present the two existing fixpoint algorithms [86, 146] that constitute our baseline. We also extend them by including explicitly the description of the generalization steps via widening, which was left implicit in previous work, and providing a new, unified view of the correctness and precision results of those algorithms.

Abstract Domain Operations. All these abstract interpretation-based algorithms are *parametric on the abstract domain*, i.e., they are independent of the data abstractions used. Each abstract domain is defined by providing the basic domain operations ($\sqsubseteq, \sqcap, \sqcup$ and, optionally, the widening ∇ operator); the abstract semantics of the primitive constraints (representing the *built-ins*, or basic operations of the source language) via *transfer functions* (f^α); and the following additional instrumental operations over abstract substitutions:

- `Aproj`(λ, Vs): restricts λ to the set of variables Vs.
- `Aextend`$(A_{k,n}, \lambda^p, \lambda^s)$: propagates the success λ^s, defined over the variables of the literal $A_{k,n}$, to λ^p, that includes all the variables of the clause k of A.
- `Acall`(A, λ, A_k): performs the *abstract call*. That is, the unification (conjunction) of a call in a literal $\langle A, \lambda \rangle$ with the head of a clause, A_k. The result is a new substitution in terms of the variables of clause k of A.

- `Aproceed`(A_k, λ_k^s, A): performs the *abstract proceed*. That is, the reverse operation of `Acall`. It unifies head of the clause (A_k) and the abstract substitution at the end of the clause (λ_k^s) with the original call A to produce the success substitution over the variables of A.
- `Ageneralize`$(\lambda, \{\lambda_i\})$: joins λ with the set of abstract substitution $\{\lambda_i\}$, all of them over the same variables. The result is an abstract substitution that is greater than or equal to λ. It either returns λ, when no generalization is needed, performs the least upper bound ($\sqcup$), or performs the widening (∇) of λ together with $\{\lambda_i\}$, depending on termination or performance needs.

In the presentation of the algorithms for simplicity we not consider narrowing (Δ). However, we believe that narrowing would also benefit from the incremental algorithms, since the strategies used on the incremental algorithms can also be used to propagate the improvements in precision, as already suggested in [86], which already included some initial ideas in this direction.

4.1 The monolithic and incremental fixpoint algorithm

As baseline for the thesis, we first present the *monolithic incremental* analysis algorithm of [85, 86], but, as mentioned before, including explicitly the widening steps (Algorithm 1). The discussion is based on the PLAI algorithm [127, 129], using the presentation of [86]. $\mathscr{A} =$ IncAnalyze$(P, Q_\alpha, \Delta, \mathscr{A}_0)$ takes as input a program P, a set of initial abstract queries Q_α, the differences Δ of P with respect to a previous version P', and an analysis graph $\mathscr{A}_0$ that is *well formed* for P'. It returns an analysis graph $\mathscr{A}$ that is correct for P and Q_α. Note that, if the algorithm is called with $\mathscr{A}_0$ empty, i.e, *from scratch*, then it behaves as the traditional monolithic PLAI algorithm [127, 129]. As mentioned before, we refer to this algorithm as *monolithic* because it assumes that all the predicates executed in the target program P are provided to the analyzer, i.e., this algorithm treats only whole programs. That is, if no code is available for a predicate, they assume it never succeeds and infer $\bot$. For efficiency, in this algorithm the edges of the analysis graph are annotated with additional information, denoted by λ^p, i.e., of the form $\langle A, \lambda_A^c\rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda_B^c\rangle$. λ^p is the abstract substitution right before the execution of the i-th literal of the k-th clause of predicate A and describes the variables of that clause.

Intuitively, generating an analysis graph consists of the execution of the program by building the generalized AND tree with two main differences: (i) instead of the concrete operations for the substitutions, the operations from the abstract domain D_α are used, (ii) in the construction of the graph, call descriptions are tabulated so that, if the abstract call constraint of a node is not equal to (or, optionally, subsumed

Algorithm 1 IncAnalyze: monolithic, context-sensitive, incremental fixpoint algorithm.

IncAnalyze $(P, Q_\alpha, \Delta, \mathscr{A})$
1: **for all** $\langle A, \lambda^c\rangle \in Q_\alpha$ **do**
2: add-event$(newcall(\langle A, \lambda^c\rangle))$
3: `deleteClauses`(Δ)
4: `addClauses`(Δ)
5: **while** events$() \neq \emptyset$ **do**
6: $E :=$ next-event$()$
7: `process`(E)
8: `removeUnreachable`$(\mathscr{A}, Q_\alpha)$
9: **return** $\mathscr{A}$

proc `addClauses`(Cls)
10: **for all** A_k :- $A_{k,1}, \ldots, A_{k,n_k} \in Cls$ **do**
11: **for all** $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
12: $\lambda^p :=$ `Acall`(A, λ^c, A_k)
13: $\lambda^c_1 :=$ `Aproj`$(\lambda^p, vars(A_{k,1}))$
14: add-event$(arc(\langle A, \lambda^c\rangle \xrightarrow{\lambda^p}_{k,1} \langle A_{k,1}, \lambda^c_1\rangle))$

proc `deleteClauses`(Cls)
15: $Calls :=$ $\{\langle A, \lambda^c\rangle | \langle A, \lambda^c\rangle \in \mathscr{A},$
16: $(A$:- $\ldots) \in Cls\}$
17: $Ns :=$ $\{n \in \mathscr{A} | n \rightsquigarrow c \in \mathscr{A}, c \in Calls\}$
18: del$(\mathscr{A}, Ns)$

func `lookupAnswer`$(\langle A, \lambda^c\rangle)$
19: **if** $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ **then**
20: **return** λ^s
21: **else**
22: add-event$(newcall(\langle A, \lambda^c\rangle))$
23: **return** $\bot$

proc `removeUnreachable`$(\mathscr{A}, Q_\alpha)$
24: $U :=$ $\{n \in \mathscr{A} |$ $\nexists q \rightsquigarrow n \in \mathscr{A}, q \in Q_\alpha\}$
25: del$(\mathscr{A}, U)$

proc `process`$(newcall(\langle A, \lambda^c\rangle))$
26: **for all** A_k :- $A_{k,1}, \ldots, A_{k,n_k} \in P$ **do**
27: $\lambda^p :=$ `Acall`(A, λ^c, A_k)
28: $\lambda^c_1 :=$ `Aproj`$(\lambda^p, vars(A_{k,1}))$
29: add-event$(arc(\langle A, \lambda^c\rangle \xrightarrow{\lambda^p}_{k,1} \langle A_{k,1}, \lambda^c_1\rangle))$

proc `process`$(arc(\langle A, \lambda^c_0\rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda^c_1\rangle))$
30: $Calls :=$ $\{\lambda \mid \langle A, \_\rangle \rightarrow_{k,i} \langle B, \lambda\rangle \in \mathscr{A}\}$
31: $\lambda^c :=$ `Ageneralize`$(\lambda^c_1, Calls)$
32: **if** B is a *built-in* **then**
33: $\lambda^s_0 :=$ $f^\alpha(\langle B, \lambda^c\rangle)$
34: **else**
35: $\lambda^s_0 :=$ `lookupAnswer`$(\langle B, \lambda^c\rangle)$
36: upd$(\mathscr{A}, \langle A, \lambda^c_0\rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda^c\rangle)$
37: $\lambda^r :=$ `Aextend`$(A_{k,i}, \lambda^p, \lambda^s_0)$
38: **if** $\lambda^r \neq \bot$ **and** $i \neq n_k$ **then**
39: $\lambda^c_2 :=$ `Aproj`$(\lambda^r, vars(A_{k,i+1}))$
40: add-event$(arc(\langle H, \lambda^c_0\rangle \xrightarrow{\lambda^r}_{k,i+1} \langle B, \lambda^c_2\rangle))$
41: **else if** $i = n_k$ **then**
42: $\lambda^s_k :=$ `Aproj`$(\lambda^r, vars(A_k))$
43: $\lambda^s :=$ `Aproceed`(A, λ^s_k, A_k)
44: `insertAnswerInfo`$(\langle A, \lambda^c_0\rangle, \lambda^s)$

proc `insertAnswerInfo`$(\langle A, \lambda^c\rangle, \lambda^s)$
45: **if** $\langle A, \lambda^c\rangle \mapsto \lambda^s_0 \in \mathscr{A}$ **then**
46: $\lambda^s_1 :=$ `Ageneralize`$(\lambda^s, \{\lambda^s_0\})$
47: **else** $\lambda^s_0 :=$ $\bot$, $\lambda^s_1 :=$ λ^s
48: upd$(\mathscr{A}, \langle A, \lambda^c\rangle \mapsto \lambda^s_1)$
49: **if** $\lambda^s_0 \neq \lambda^s_1$ **then**
50: `reanalyzeUpdated`$(\langle A, \lambda^c\rangle)$

proc `reanalyzeUpdated`$(\langle A, \lambda^c\rangle)$
51: **for all** $E :=$ $\langle B, \lambda^c_0\rangle \xrightarrow{\lambda^p}_{k,i} \langle A, \lambda^c\rangle \in \mathscr{A}$ **do**
52: add-event$(arc(E))$

by) that of a node already present, the graph is not extended, and instead, a back edge is introduced pointing to that node, and (iii) dependencies are tracked, as mentioned before, to speed up convergence.

For presentation purposes and without loss of generality in the following we assume that all atoms are normalized. However, in the examples we also use non normalized atoms for brevity.

4.1.1 Operation of the algorithm

Algorithm 1 is centered around processing two kinds of events. The *newcall* events control which predicate calls need reanalysis, while the *arc* events abstractly execute one literal of the body of a clause for an abstract call. The algorithm starts by queueing a *newcall* event for each of the abstract calls that need to be (re)computed. This triggers `process`$(newcall(\langle A, \lambda^c \rangle))$, which schedules the analysis of each of the clauses of the predicate. This is done by performing the abstract call using `Acall`(which includes the renaming) and adding an *arc* event for the first literal (lines 26-29). The procedure `process`$(arc(\langle A, \lambda_0^c \rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda_1^c \rangle))$ performs a single step of the left-to-right traversal of a clause body. Since the algorithm is multivariant, an infinite number of different call patterns may be encountered, even if the domain has finite height. Therefore, the calls are generalized in the `Ageneralize` operation (lines 30-31) via a widening. Then, if the literal $A_{k,i}$ is a *built-in*, its transfer function is applied (line 33); otherwise, it is a call to a predicate, and the answer λ^s is looked up (line 35). This is done by `lookupAnswer` (lines 19-23). It returns the answer in the analysis graph if available, or creates a *newcall* event for $\langle A, \lambda^c \rangle$ if not (to schedule its analysis, because it has not been analyzed yet). Then, in line 36, an edge is added to capture this call dependency; it is at this point where cycles may be added to the graph, if the same call pattern is encountered twice, either directly by abstracting the call or after generalizing it via widening (in line 31). In line 37, the answer is combined with λ^p, the substitution immediately before the execution of the literal $A_{k,i}$ to obtain λ^r, the substitution immediately after the execution of the literal. This is used either to process the next literal, scheduling an *arc* event (lines 38-40), or to update the answer of the predicate (lines 41-44) if it is the last literal in the body. The function `insertAnswerInfo` combines the new answer of a clause with the semantics of the previous answers and previously analyzed clauses. To ensure termination when analyzing with abstract domains not satisfying the *ACC*, the answer is generalized (lines 45-46). Lastly, the new answer is propagated if it differs from the existing one, meaning that fixpoint has not been reached yet (lines 49-50). The procedure `reanalyzeUpdated` propagates the information of newly computed answers across the analysis graph by creating *arc* events for the literals from which the analysis has to be restarted (lines 51-52). After the fixpoint is reached, in line 8, the procedure `removeUnreachable` deletes from the analysis graph nodes or subgraphs that are not reachable from the initial queries. As, for instance, they may have existed in a previous analysis but are not useful now. This allows improving precision or removing redundant (temporary) nodes, e.g., the ones produced as intermediate results before the fixpoint is reached.

The procedure `addClauses` schedules the analysis of new clauses. It suffices, for every clause (line 10), to queue *arc* events for each of encountered calls to the predicates that they define (lines 11-14). These trigger the analysis of each clause and the later update of the callers to these predicates $\mathscr{A}$ by following the edges in the graph (done in `insertAnswerInfo`).

The `deleteClauses` procedure deletes the information that is potentially imprecise. That is, the nodes representing calls to the predicates defined by the deleted clauses (line 16), because the success information is potentially inaccurate, and all their callers (line 17). The scheduling of events is not necessary, because events for the queries are always scheduled, and this will trigger the reanalysis of the necessary parts.

At the end of the chapter, Example 4.3 shows a detailed execution of the algorithm.

4.1.2 *Differences w.r.t. the original monolithic incremental algorithm*

Algorithm 1, IncAnalyze, extends the algorithm first described in 1995 [85], the latter is henceforth consequently called IncAnalyze95. As mentioned before, widening was considered in [86] but here we incorporate it explicitly in Algorithm 1. In particular, lines 30 and 31, which perform the widening of the encountered calls for the cases in which the abstract domain has infinite width/height; and in line 46, that performs the widening on the success for the cases in which the abstract domain is of infinite height. IncAnalyze95 is then obtained by:

- replacing lines 30-31 by "$\lambda^c := \lambda_1^c$",
- and replacing line 46 by "$\lambda_1^s := \lambda^s \sqcup \lambda_0^s$".

4.1.3 *Correctness and precision of* IncAnalyze95

We now present the fundamental results of the algorithms. Lemmas 4.2 to 4.4 and Theorem 4.5 were formulated and proved in [86]. However, we present a more generalized view and justification of such results. To this end, we propose a new ordering for well-formed analysis graphs that will be instrumental to understand how the algorithm computes the supremum of Kleene sequences of functions that manipulate such graphs. Note that we are only interested in comparing graphs for the same program, since the program is not going to be modified while computing the fixpoint. Therefore, in the following, we assume that we only compare graphs for the same program P.

Partial order of analysis graphs. Henceforth we denote by AG the set of all possible analysis graphs that has as call and success values elements in a domain D_α. The following pre-orders encapsulate, among other things, the notion that an analysis graph g_2 is *finer* than another analysis g_1. Moreover, observe that both definitions below are parametric on D_α.

Definition 4.1. Given two nodes $n = \langle A, \lambda_n \rangle, m = \langle B, \lambda_m \rangle$ we say $n \sqsubseteq_{node} m$ if

$$\exists \text{ a renaming } \sigma \text{ such that } A = B\sigma \wedge \lambda_n \sqsubseteq_{D_\alpha} \lambda_m\sigma$$

Definition 4.2. Given two well-formed analysis graphs g_1, g_2 for a program P we say $g_1 \sqsubseteq_{AG} g_2$ if

1) $\forall n_1 \in g_1.\exists n_2 \in g_2.n_1 \sqsubseteq_{node} n_2$,
2) $\forall n \mapsto \lambda_1^s \in g_1.\exists n \mapsto \lambda_2^s \in g_2 \ \wedge \lambda_1^s \sqsubseteq_{D_\alpha} \lambda_2^s$, and
3) $\forall (n_1 \rightarrow_{k,i} m_1) \in g_1.\exists (n_2 \rightarrow_{k,i} m_2) \in g_2.(n_1 \sqsubseteq_{node} n_2) \wedge (m_1 \sqsubseteq_{node} m_2)$.

In particular, note that two nodes n, m are only $\sqsubseteq_{node}$-comparable if they describe calls to the same predicate. In addition, if $g_1 \sqsubseteq_{AG} g_2$ then g_2 abstracts all the nodes in g_1, i.e., the analysis graph g_2 is finer than the analysis graph g_1.

Remark. Observe that $\sqsubseteq_{AG}$ is *not* a partial order in the set AG, as one may have analysis graphs g_1 and g_2 such that $g_1 \sqsubseteq_{AG} g_2$ and $g_2 \sqsubseteq_{AG} g_1$, but $g_1 \neq g_2$ as graphs. However, one can consider the equivalence relation $g_1 \sim_{\sqsubseteq_{AG}} g_2$ given by $g_1 \sqsubseteq_{AG} g_2$ and $g_2 \sqsubseteq_{AG} g_1$. In this setting, it is routine to show that $\sqsubseteq_{AG}$ defines a partial order in the $\sim_{\sqsubseteq_{AG}}$-equivalence classes. We hence slightly abuse notation, and say that $\sqsubseteq_{AG}$ defines a partial order in AG. Likewise, we denote $\sqsubseteq_{AG}$ simply by $\sqsubseteq$ for notational simplicity.
Remark. Observe that there exists a $\sqsubseteq$-minimal correct analysis graph, which we call the *least correct analysis graph*. This is the smallest (well-formed) analysis graph, with the most precise (smallest) calls or answers, that correctly over-approximates the behavior of a program (Def. 2.5).
Lastly, note that λ^p is not relevant in the partial order because it is included in the algorithm only for efficiency.

Let us first state an interesting property of the `removeUnreachable` procedure.

Lemma 4.1. *Let $\mathscr{A}$ be an analysis graph, Q_α a set of queries, and let $\mathscr{A}' =$ `removeUnreachable`$(\mathscr{A}, Q_\alpha)$. Then $\mathscr{A}'$ abstracts the same trees as $\mathscr{A}$ w.r.t. Q_α.*

Proof. By definition, `removeUnreachable` does not remove edges in paths starting from the query in $\mathscr{A}$. The nodes in $\mathscr{A}$ that are not reachable from Q_α, i.e., for which

there is not a path from any $q \in Q_\alpha$, do not belong in any of the trees that $\mathscr{A}$ represents. Therefore, if they are removed, the trees abstracted by the nodes starting in Q_α remain the same. □

This implies that if $\mathscr{A}$ is correct for some program P and queries Q_α, then $\mathscr{A}'$ is correct as well. Note that this implies that if $\mathscr{A}$ is in the class of least program analysis graphs then $\mathscr{A}'$ remains in the class of least program analysis graphs as well.

Let us now recall the correctness and precision results of IncAnalyze95 from [86]. We simply reformulate those guarantees and give an intuition as to why they hold in terms of obtaining the fixpoint by computing the supremum of Kleene sequences of functions manipulating analysis graphs, provided that they are continuous, additive, or join preserving. The *least fixed point (lfp)* of such functions is the least correct analysis graph of the program being analyzed.

Given a program P and a set of abstract queries Q_α, let $f_P : AG \to AG$ be the function that processes an *arc* event in Algorithm 1. According to the graph ordering the processing of these events always causes the analysis graphs to become greater because the insert (greater) answers (`insertAnswerInfo`) or insert new nodes with edges (line 36). Thus, processing events is a monotone increasing process, and ACC is guaranteed because we only consider finite domains. In the explanations we omit Q_α for conciseness, as it does not change during the fixpoint computation process.

Lemma 4.2 (Correctness and precision of IncAnalyze95 from scratch)**.** *Let P be a program, and Q_α a set of abstract queries. The analysis result $\mathscr{A} =$ IncAnalyze95$(P, Q_\alpha, \emptyset, \emptyset)$ is the least correct analysis graph for P and $\gamma(Q_\alpha)$.*

In abstract interpretation, the fixpoint of a set of (monotonic) clauses is computed by repeatedly composing the abstract semantics of each of the clauses iterating in a chaotic manner. If the iteration is started at $\bot$, it is guaranteed that the least fixed point of the system is found. When starting from an empty analysis, IncAnalyze computes the *lfp*, which, in this case is the least analysis graph by applying $f_P(X)$ a number of times:

$$\bot \sqsubseteq f_P(\bot) \sqsubseteq f_P(f_P(\bot)) \sqsubseteq f_P^3(\bot) \sqsubseteq \ldots \sqsubseteq f_P^k(\bot) = \ldots = f_P^{k+n}(\bot) = lfp(P)$$

In the sequence above, the fixpoint value is found in the k-th step of the iteration. However, this value is not yet known to be the fixpoint. The chaotic iteration process needs to continue until all the clauses have been exhaustively applied and the value of the fixpoint is kept, this is represented by the n steps exectued after f_P^k. Note that the number of steps k and n depends highly on the strategy for the chaotic iteration. In our case, we safely reduce them by keeping the dependencies between clauses. Also note that the order in which the intermediate steps (f_P^i) of the Kleene chain are

computed does not affect the analysis results (since $\sqcup$ is commutative). Therefore, the order in which events are processed in the algorithm does not affect the final analysis graph. Lastly, by Lemma 4.1, the result is guaranteed to be safe and the *lfp*.

Lemma 4.3 (Correctness and precision of IncAnalyze95 adding clauses)**.** *Let P and P' be two programs such that $\Delta = (C_{add}, \emptyset)$, $P = (P' \cup C_{add})$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze95}(P', Q_\alpha, \emptyset, \emptyset)$, then*

$$\text{IncAnalyze95}(P, Q_\alpha, \Delta, \mathscr{A}_0) = \text{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset)$$

As mentioned before, $\mathscr{A}_0$ is well formed for P and P'. Therefore it is comparable when reusing it to analyze P. For both P and P' there exists a valid sequence to compute their fixpoints that consists in processing first all the clauses in P':

$$\begin{array}{l} \bot \sqsubseteq f_{P'}(\bot) \sqsubseteq f_{P'}(f_{P'}(\bot)) \sqsubseteq \ldots \sqsubseteq f^k_{P'}(\bot) \;= \mathscr{A}_{P'}\ (\mathit{lfp}(P')) \\ \quad \text{equivalent computation steps} \quad \downarrow \text{reuse} \\ \underbrace{\bot \sqsubseteq f_P(\bot) \sqsubseteq f_P(f_P(\bot)) \sqsubseteq \ldots}_{\text{avoided steps}} \;\sqsubseteq f^k_P(\bot) \sqsubseteq \ldots \sqsubseteq f^n_P(\bot) = \mathscr{A}_P\ (\mathit{lfp}(P)) \end{array}$$

Therefore restarting the analysis of P with $\mathscr{A}_0 = \mathit{lfp}(P')$ produces a safe and accurate result.

Lemma 4.4 (Correctness and precision of IncAnalyze95 deleting clauses)**.** *Let P and P' be two programs such that $\Delta = (\emptyset, C_{del})$, $P = P' \setminus C_{del}$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze95}(P', Q_\alpha, \emptyset, \emptyset)$, then*

$$\text{IncAnalyze95}(P, Q_\alpha, \Delta, \mathscr{A}_0) = \text{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset)$$

That is, if the program P' is analyzed for entries Q_α, obtaining $\mathscr{A}_0$, and $\mathscr{A}$ is incrementally recomputed deleting the clauses C_{del}, the analysis result of P will be the same as when analyzing it from scratch.

A first approach would consist in taking advantage of the existence of a sequence of applications of $f_{P'}$ s.t. by going back to the *k-th* step (backwards) of the Kleene sequence supremum computation we would have exactly the *lfp* of f_P.

$$\begin{array}{l} \bot \sqsubseteq f_{P'}(\bot) \sqsubseteq f_{P'}(f_{P'}(\bot)) \sqsubseteq \ldots \sqsubseteq f^k_{P'}(\bot) \;\sqsubseteq \ldots \sqsubseteq f^n_{P'}(\bot) = \mathscr{A}_{P'} \\ \qquad\qquad\qquad\qquad\qquad\qquad \downarrow \text{reuse} \quad \leftarrow \quad \text{go back} \\ \underbrace{\bot \sqsubseteq f_P(\bot) \sqsubseteq f_P(f_P(\bot)) \sqsubseteq \ldots}_{\text{avoided steps}} \;\sqsubseteq f^k_P(\bot) \;= \mathscr{A}_P \end{array}$$

Note that this is infeasible in practice. It would imply storing each of the intermediate steps of fixpoint computation, and, most importantly, this intermediate state k only exists if we specifically analyze in a sequence that leaves processing to the semantics of C_i to the end. It is therefore desirable to be able remove any clause(s) from the program, not only the last processed by the algorithm.

To this end, the algorithm deletes subgraphs to obtain a state of the analysis that corresponds to an earlier state of analysis sequence in which the processing of clauses that depend on the deleted clauses is left for the end.

$$\begin{array}{lllll} \bot \sqsubseteq f_{P'}(\bot) \sqsubseteq \ldots & \sqsubseteq f^i_{P'}(\bot) & \sqsubseteq \ldots & \sqsubseteq \ldots & \sqsubseteq f^n_{P'}(\bot) = \mathscr{A}_{P'} \\ & \text{accurate reuse} \downarrow & \leftarrow & \leftarrow & \text{delete subgraph} \\ \underbrace{\bot \sqsubseteq f_P(\bot) \sqsubseteq \ldots}_{\text{avoided steps}} & \sqsubseteq f^j_P(\bot) & \sqsubseteq \ldots \sqsubseteq f^m_P(\bot) & = \mathscr{A}_P & \end{array}$$

Theorem 4.5 (Correctness and precision of IncAnalyze95)**.** *Let P, P' be programs, such that P differs from P' by Δ, let Q_α a set of abstract queries, and $\mathscr{A}_0 = \text{IncAnalyze95}(P', Q_\alpha, \emptyset, \emptyset)$ be the least analysis graph. The following hold:*

- *If $\mathscr{A} = \text{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset)$, then $\mathscr{A}$ is the least analysis graph for P and $\gamma(Q_\alpha)$, and*
- $\text{IncAnalyze95}(P, Q_\alpha, \Delta, \mathscr{A}_0) = \text{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset)$.

That is, when analyzing from scratch, the most precise result is always produced, and when reusing a least program analysis graph in the incremental analysis, the new result is the least program analysis graph as well.

4.1.4 *Correctness of* IncAnalyze

We now formulate the correctness results of Algorithm 1, i.e., the algorithm of [86] extended *with generalization* (extrapolating using widening). The abstract interpretation technique guarantees that generalization with a widening operation preserves soundness, and guarantees termination at the expense of losing precision and monotonicity. Given the definitions of Sec. 2.3, the following Lemmas 4.6 to 4.8 and Theorem 4.9 of [86] hold, because, as stated earlier, generalization via a widening guarantees correctness:

Lemma 4.6 (Correctness of IncAnalyze from scratch)**.** *Let P be a program, and Q_α a set of abstract queries. The analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$ for P with Q_α is correct for P and $\gamma(Q_\alpha)$.*

Lemma 4.7 (Correctness of IncAnalyze adding clauses)**.** *Let P and P' be two programs such that $\Delta = (C_{add}, \emptyset)$, $P = (P' \cup C_{add})$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ for P with Q_α is correct for P and $\gamma(Q_\alpha)$.*

Lemma 4.8 (Correctness of IncAnalyze deleting clauses)**.** *Let P and P' be two programs such that $\Delta = (\emptyset, C_{del})$, $P = P' \setminus C_{del}$, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ for P with Q_α is correct for P and $\gamma(Q_\alpha)$.*

Theorem 4.9 (Correctness of IncAnalyze)**.** *Let P and P' be two programs that differ by Δ, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{IncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ for P with Q_α is correct for P and $\gamma(Q_\alpha)$.*

Note that if we have a different program P'' and $\mathscr{A}'' = \text{IncAnalyze}(P'', Q_\alpha, \emptyset, \emptyset)$, $\mathscr{A}_2 = \text{IncAnalyze}(P, Q_\alpha, \Delta_{P''}, \mathscr{A}'')$, $\mathscr{A}$ and $\mathscr{A}_2$ are both correct for P and $\gamma(Q_\alpha)$ but may not be comparable because ∇ is not associative and not monotone.

4.1.5 *Starting from partial analyses*

Theorems 4.5 and 4.9 show that, if $\mathscr{A}_0$ is a (precise and) correct analysis, then the incremental analysis result is (precise and) correct. However, the conditions on $\mathscr{A}_0$ can be relaxed if the conditions on the queries are strengthened and still guarantee the same correctness and precision results. In the following we state conditions to guarantee precision and correctness when (re)starting from a partial analysis result.

Lemma 4.10 (Correctness of IncAnalyze starting from a correct partial analysis)**.** *Let P be a program, Q_α be a set of abstract queries, and fix $q \in Q_\alpha$. Suppose that $\mathscr{A}_0$ is the analysis result $\mathscr{A}_0 = \text{IncAnalyze}(P, Q_\alpha \setminus \{q\}, \emptyset, \emptyset)$. Then the analysis result $\mathscr{A} = \text{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$ is correct for P and $\gamma(Q_\alpha)$.*

Proof. The proof goes case by case, and there are three cases. If $q \in \mathscr{A}$ the analysis is correct by assumption. If q is independent from $\mathscr{A}_0$ then $\mathscr{A}$ is correct by Lemma 4.6. Otherwise, q depends on some nodes of $\mathscr{A}_0$, and since the algorithm triggers the reanalysis of all dependent calls, it analyzes these nodes. This procedure eventually leads to the former two cases. □

Theorem 4.11 generalizes Theorem 4.9 and Lemmas 4.6 to 4.8.

Theorem 4.11 (Correctness of IncAnalyze starting from a partial analysis)**.** *Let P be a program, Q_α a set of abstract queries, and $\mathscr{A}_0$ a well-formed analysis graph*

for P. Suppose for all concrete queries $q \in \gamma(Q_\alpha)$, for all nodes n from which there is a path in the concrete execution $q \rightsquigarrow n$ in $[\![P]\!]_Q$, and for all $n_\alpha \in \mathscr{A}_0$ such that $n \in \gamma(n_\alpha)$ either:

a) $n_\alpha \in Q_\alpha$, or

b) the subgraph with root n_α is correct for P and $\{\gamma(n_\alpha)\}$.

Then $\mathscr{A} = \textsc{IncAnalyze}(P, Q_\alpha, \emptyset, \mathscr{A}_0)$ is correct for P and $\gamma(Q_\alpha)$.

The subgraph of $\mathscr{A}_0$ with root n_α is the result of `removeUnreachable`$(\mathscr{A}_0, \{n_\alpha\})$.

Proof. The proof goes case by case. For a fixed query $q \in Q_\alpha$ either:

- $\mathscr{A}_0$ does not abstract any calls in the execution of $\gamma(\{q\})$, whence $\mathscr{A}_q = \textsc{IncAnalyze}(P, \{q\}, \emptyset, \mathscr{A}_0)$ is correct by Lemma 4.6.
- $\mathscr{A}_0$ is itself correct for $\gamma(\{q\})$, and hence so is $\mathscr{A}_q = \textsc{IncAnalyze}(P, \{q\}, \emptyset, \mathscr{A}_0)$ correct for $\gamma(\{q\})$ by Lemma 4.10.
- $\mathscr{A}_0$ is not correct for P, $\gamma(\{q\})$ but $\mathscr{A}_q = \textsc{IncAnalyze}(P, \{q\}, \emptyset, \mathscr{A}_0)$ is correct for P, $\gamma(\{q\})$.
- Otherwise, if $\mathscr{A}_q = \textsc{IncAnalyze}(P, \{q\}, \emptyset, \mathscr{A}_0)$ is not correct for P, $\gamma(\{q\})$, then there exists ${Q_\alpha}' \subset Q_\alpha$, $q \notin {Q_\alpha}'$ such that $\mathscr{A}_q = \textsc{IncAnalyze}(P, \{q\} \cup {Q_\alpha}', \emptyset, \textsc{IncAnalyze}(P, {Q_\alpha}', \emptyset, \mathscr{A}_0))$ is correct for P, $\gamma(\{q\})$. Indeed, note that if the reused graph is not correct, then condition (b) in the statement of the theorem is not met, and by condition (a) the root of the subgraph is included in Q_α.

That is, for any query $q \in Q_\alpha$ there is ${Q_\alpha}' \subseteq Q_\alpha$ with $q \in {Q_\alpha}'$ such that the analysis $\mathscr{A}_{{Q_\alpha}'} = \textsc{IncAnalyze}(P, {Q_\alpha}', \emptyset, \mathscr{A}_0)$ is correct for P and $\gamma(q)$. Therefore, the analysis $\mathscr{A}$ is correct for P and $\gamma(Q_\alpha)$ by iteratively applying Lemma 4.10. □

Note that $\mathscr{A}_0$ is not assumed to be the (correct) output of a previous analysis, it can be any analysis (below, above, or incomparable with the fixpoint). Also note that if all nodes in the analysis graph are included (together with the original queries), in Q_α the result is guaranteed to be correct.

If $\mathscr{A}_0$ already contains information about q, it needs to be rechecked by recomputing the analysis of all the nodes in which q depends by including them in Q_α. Theorem 4.11 is a generalization because, implicitly, procedures `addClauses` and `deleteClauses` are doing this. Either removing subgraphs, adding the respective queries so that they are computed from scratch, or adding the necessary queries (by directly creating the corresponding *newcall* events) for the subgraphs that are not yet correct.

The following theorem generalizes Theorem 4.5 and Lemmas 4.2 to 4.4, i.e., about the algorithm not using widening.

Theorem 4.12 (Correctness and precision of IncAnalyze95 starting from a partial analysis)**.** *Under the same conditions as Theorem 4.11, if* $\mathscr{A}_0 \sqsubseteq \mathscr{A}$*, then:*

$$\textsc{IncAnalyze95}(P, Q_\alpha, \emptyset, \emptyset) = \textsc{IncAnalyze95}(P, Q_\alpha, \emptyset, \mathscr{A}_0).$$

Proof. Starting from a partial analysis is equivalent to computing the Kleene fixpoint of the original program together with a new "abstract clause", which is a constant representing the initial results. Let us call this clause $\mathscr{A}_0$. Our goal is to prove that chaotic iteration of f_P with $\mathscr{A}_0$ also results in the $lfp(P)$ if $\mathscr{A}_0 \sqsubseteq lfp(P)$.

By definition, for any k-th step of the iteration $f_P^k(\bot) \sqsubseteq lfp(P)$, also, by hypothesis, $\mathscr{A}_0 \sqsubseteq lfp(P)$. Therefore, for any k and applying any random clause, $\mathscr{A}_0 \sqcup f_P^k(\bot) \sqsubseteq lfp(P)$. So, if we "plug in" the initial analysis $\mathscr{A}_0$ at any point of the chaotic iteration over f_P, because the semantics of the clauses of P are monotonic, for any k, $f_P(\mathscr{A}_0 \sqcup f_P^k(\bot)) \sqsubseteq f_P(lfp(P))$, and precision is preserved. Concretely, this also implies that precision is preserved if we start from $f_P(\mathscr{A}_0)$.

The condition imposed on the set of queries guarantees that the chaotic iteration includes all the clauses that the iteration needs to be rerun with (see Theorem 4.11). This justifies not reprocessing the clauses that are not affected by the changes in the algorithm, since the corresponding steps can be skipped safely. □

4.2 The intermodular fixpoint algorithm

We now present the reference algorithm for analyzing modular programs, originally described in [24, 146]. As expected, the approach consists in analyzing partitions of programs making assumptions about the code that is external to each partition. Several possibilities were proposed in that work for making such assumptions, e.g., assuming that nothing is known about the answer ($\top$), computing the "topmost" abstraction of the call (as before but taking into account any local information available), or strategies with better precision but, in general, more costly, such as assuming $\bot$ temporarily for the unknown answers and later reanalyzing whenever the correct abstraction of the answer is available. Note that these abstractions help in achieving scalability, as one can disable precise abstractions for some components, and still obtain results that are meaningful. In this work we do not want to give up precision to obtain scalability and therefore fix the strategy to the latter one to obtain the best precision. Additionally, note that module analysis order may affect the speed at which the fixpoint computation converges. Some scheduling policies were studied in [33] but this is out of the scope of this thesis.

In this chapter we provide a new pseudocode for the algorithm of [146], specialized for the case in which the maximum precision is aimed for. Then, we provide new formal results about correctness and precision of this algorithm. Also, both for generality

```
:- module(main, [main/1]).

:- use_module(bitops).
main(Msg, P) :-
    par(Msg, 0, P).

par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).
```

```
:- module(bitops, [xor/3]).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

Fig. 11: A modular version of the program in Example 2.1.

and reusability, although not required for our results, we propose a formulation of the algorithm that is parametric on the analysis used within each modular partition, which in our case is instantiated to IncAnalyze. Let us begin by looking at an example of a modular program.

Example 4.1 (A modular program)**.** Fig. 11 shows a modular version of the program in Example 2.1. The program contains two modules, as declared by the `Ciao` system module annotations (see Sec. 2.5 for the detailed description of the syntax). The program contains two modules: `main` and `bitops`. The module `main` imports (all the predicates in) `bitops` and exports the predicate `main/1` that computes the parity of a list. This means that `par/3` is not accessible to any module outside of `main`. The module `bitops` exports the predicate `xor/3`.

4.2.1 *Modular analysis results*

To store the overall analysis result of the program and keep track of fine-grain dependencies between modules, we propose to use also an analysis graph structure at the inter-modular level. One can see this as a "projection" of the *monolithic* analysis graph, described in Sec. 2.3, where only the predicate calls across module boundaries are kept. As before, nodes represent calls to predicates and edges capture the relations between the predicates in the boundaries of the partitions (exported/imported predicates) with arcs $\langle A, \lambda^c \rangle \rightarrow \langle B, \lambda^{c\prime} \rangle$ meaning *a call to* A *in* $\mathsf{mod}(A)$ *with description* λ^c *may cause a call to* B *with description* $\lambda^{c\prime}$ *and* $\mathsf{mod}(B) \in \mathsf{imports}(\mathsf{mod}(A))$. The precision of the inferred calls and successes of the predicates is independent of the analysis graph being modular or not. From this point on, we use $\mathcal{G}$ to denote the modular (global) analysis graph. In contrast, *local* analysis graphs, denoted by $\mathscr{L}$, contain the abstraction of the single module being analyzed.

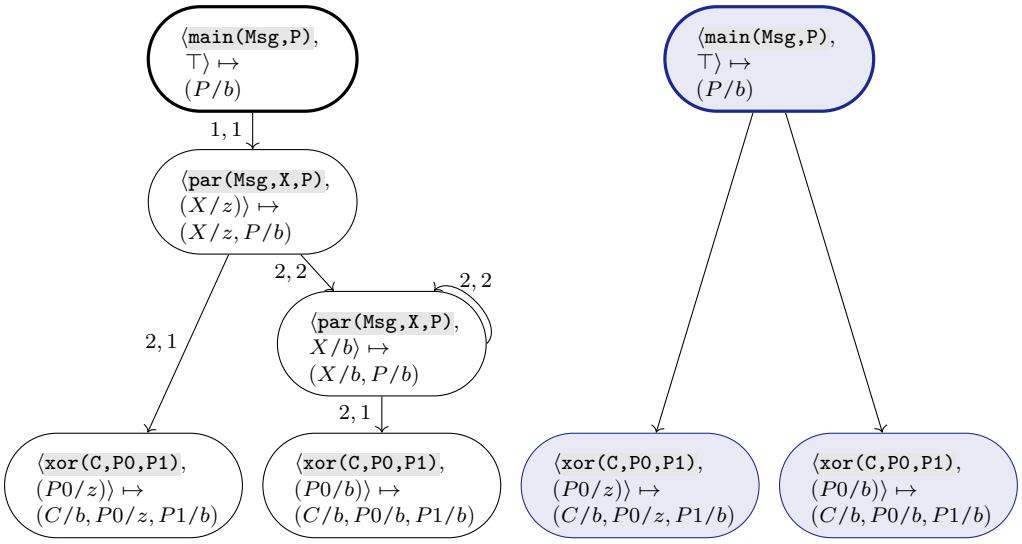


Fig. 12: A monolithic (left) and a modular (right) analysis result of the program in Fig. 11.

Example 4.2 (Modular analysis result)**.** Fig. 12 shows a comparison between a monolithic and a modular analysis result for the program in Fig. 11. The nodes of this (global) analysis graph encode that calling the exported predicate `main/1` of module `main` may cause a call to `xor/3` exported by module `bitops` with two different call descriptions (two edges).

We now formalize the notion of *correct modular analysis*. To distinguish between the queries defined by the user and the intermediate queries done internally by the modular analysis algorithm, the latter are called *entries* and referred to with E.

Definition 4.3 (Intermodular calls)**.** Given a modular program P, a set of concrete queries Q, and E a set of calls to predicates exported by any module in P. The set, $\mathsf{int\text{-}calls}(E, [\![P]\!]_Q)$, of intermodular calls from any $e \in E$ is the set of c to which there is a path $e \rightsquigarrow c_n$ in any tree of $[\![P]\!]_Q$ with $e = c_0$ of the form $(c_0 \to c_1 \to \ldots \to c_n)$ for all $0 \leq i < n$, $\mathsf{mod}(c_i) = \mathsf{mod}(c_0) \wedge \mathsf{mod}(c_n) \in \mathsf{imports}(\mathsf{mod}(c_0))$.

Note that we are not interested in which clause and literal generates the call. We abuse notation and use int-calls to refer to the intermodular calls in analysis graphs.

Definition 4.4 (Correctly approximated intermodular calls)**.** Let P be a program and Q a set of concrete queries, $\mathcal{G}$ an analysis graph, and E a set of entries, and let I

be the transitive closure of $\mathsf{int\text{-}calls}(Q, [\![P]\!]_Q)$. We say that $\mathcal{G}$ *correctly approximates the intermodular calls of* $[\![P]\!]_Q$ if it abstracts all the calls in I. That is:

$$\forall \langle A, \theta^c \rangle \in I. \exists \langle A, \lambda^c \rangle \in \mathcal{G} \wedge \theta^c \in \gamma(\lambda^c).$$

That is, $\mathcal{G}$ contains all the calls of the exported predicates that were originated from a different module in which they are defined, and that are reachable from Q. Note that this set in the concrete execution may be infinite, e.g., in the case in which an imported predicate is called inside a loop.

Definition 4.5 (Correctly approximated intermodular dependencies)**.** Given P, Q, $\mathcal{G}$, E, and I as in Def. 4.4. We say that $\mathcal{G}$ *correctly approximates the intermodular dependencies of* $[\![P]\!]_Q$ if for every two $c_1, c_2 \in I$, if $\exists c_2 \in \mathsf{int\text{-}calls}(\{c_1\}, [\![P]\!]_Q)$, then there is an edge $c_1 \to c_2$ in $\mathcal{G}$.

Definition 4.6 (Correct modular analysis)**.** Given a modular program P, and a set of concrete queries Q, we say a modular analysis graph $\mathcal{G}$ *is correct for* P, Q if:

a) it approximates the intermodular calls correctly (see Def. 4.4) and

b) it approximates the answers correctly (see Def. 2.3).

c) it approximates the intermodular dependencies correctly (see Def. 4.5).

Note again that this correctness definition does not require changes to the concrete semantics of the program. The concrete semantics are not modular.

4.2.2 *Operation of the algorithm*

The algorithm ModAnalyze(P, Q_α) takes as input a (partitioned) program $P = \{M_i\}$, some initial queries Q_α to any exported predicate of the program, i.e., any $\langle A, \lambda^c \rangle \in Q_\alpha, A \in \mathsf{exports}(\mathsf{mod}(A))$. If there are mutually-recursive dependencies between modules, the modules in each clique is grouped and analyzed as a whole module (after doing the necessary renamings). Each of the modules in the program is analyzed independently, and possibly several times.

Algorithm 2 shows the pseudocode. For each module, the algorithm keeps a set of all the calls that need to be (re)analyzed. The queue is initialized with an entry for each of the abstract queries (line 1). Modular analysis is controlled by this queue that contains the calls with possibly incomplete answers (added with procedure add-entries). At each iteration of the loop a module is reanalyzed independently for its set of annotated entries (E) extracted from the queue (line 3). This is done by the procedure next-entries which extracts from the queue entries that are reachable from the initial Q_α in $\mathcal{G}$. At every iteration one module is analyzed *from scratch*.

Algorithm 2 ModAnalyze: Modular fixpoint algorithm.

ModAnalyze($P = \{M_i\}, Q_\alpha$)

1: add-entries($\{k \in Q_\alpha \mid k \notin \mathcal{G}\}$), upd($\mathcal{G}, \{k \mapsto \bot \mid k \in Q_\alpha\}$)
2: **while** entries($\mathcal{G}, Q_\alpha$) $\neq \emptyset$ **do**
3: $(M, E) :=$ next-entries($\mathcal{G}, Q_\alpha$)
4: $\mathscr{L} := \emptyset$
5: upd($\mathscr{L}, \{\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{G} \mid \exists e \in E \wedge e \to \langle A, \lambda^c\rangle \in \mathcal{G}$) ▷ **PreloadImported**
6: $\mathscr{L} :=$ IncAnalyze($M, E, \emptyset, \mathscr{L}$)
7: **for** $\langle A, \lambda^c\rangle \mapsto \lambda^s_l \in \mathscr{L}.\langle A, \lambda^c\rangle \mapsto \lambda^s_g \in \mathcal{G} \Rightarrow \lambda^s_l \neq \lambda^s_g$ **do**
8: $\lambda^s :=$ `Ageneralize`($\lambda^s_l, \{\lambda^s_g\}$)
9: upd($\mathcal{G}, \langle A, \lambda^c\rangle \mapsto \lambda^s$) ▷ **StoreAnswers**
10: add-entries($\{k \mid k \to \langle A, \lambda^c\rangle \in \mathcal{G}\}$)
11: del($\mathcal{G}, \{e \to k \mid e \in E, e \to k \in \mathcal{G}\}$)
12: $R =$ int-calls($E, \mathscr{L}$)
13: **for** $k \to \langle B, \lambda^c{}_t\rangle \in R$ **do**
14: $Calls := \{\lambda \mid \langle B, \lambda\rangle \in \mathcal{G}\}$
15: $\lambda^c :=$ `Ageneralize`($\lambda^c_t, Calls$)
16: upd($\mathcal{G}, \{k \to \langle B, \lambda^c\rangle\}$) ▷ **StoreDependencies**
17: **if** $\langle B, \lambda^c\rangle \notin \mathcal{G}$ **then**
18: add-entries($\{\langle B, \lambda^c\rangle\}$) ▷ **ScheduleNewCalls**
19: `removeUnreachable`($\mathcal{G}, Q_\alpha$)
20: **return** $\mathcal{G}$

This means that, in principle, the analysis of module M with entries E should be performed by $\mathscr{L} =$ IncAnalyze$(M, E, \emptyset, \emptyset)$. However, IncAnalyze assumes that all code is available for analysis. Since this is not so in this modular case, IncAnalyze needs to be provided with an abstraction of the predicates imported by M. To this end, in line 5 (**PreloadImported**), the nodes and answers of the global graph $\mathcal{G}$ of predicates imported by M are added to $\mathscr{L}$. After a local analysis of the module (line 6), $\mathcal{G}$ is updated, by storing the newly computed answers, provided that a generalization is made before to ensure termination (lines 8-9). Then the dependencies of the predicates in the boundary of the modules are updated. For this, first the old dependencies of the entries are deleted (line 11). The new dependencies, obtained in line 12, are included in the graph (lines 13-18). First they are generalized if necessary (lines 14-15), and updated in the graph (line 16). Newly encountered calls are scheduled for reanalysis (line 18). Finally, `removeUnreachable` has the same purpose as in Algorithm 1.

4.2.3 *Correctness of* ModAnalyze

As mentioned earlier, IncAnalyze assumes that either the procedures executed by a program are defined in the clauses provided to the analyzer, or they are basic, built-in operations of the language, i.e., they are interpreted applying their corresponding transfer function. This is not the case when analyzing programs module by module, and assumptions need to be made about the imported code. The following lemma states that the analysis graph inferred by IncAnalyze is correct assuming the answers of $\mathscr{L}_0$ if it only contains abstractions of the imported predicates. In other words, if $\mathscr{L}_0$ correctly over-approximates the behavior of the imported predicates, then the analysis of the module is correct.

Lemma 4.13 (Correctness of IncAnalyze modulo imported predicates)**.** *Let M be a module of program P, E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall\langle A, \lambda^c\rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze}(M, E, \emptyset, \mathscr{L}_0)$$

is correct (see Def. 2.5) for M and $\gamma(E)$ assuming $\mathscr{L}_0$.

Proof. The argument is similar to that of Lemma 4.10. Since, $\mathscr{L}_0$ contains only information about the imported predicates, $\mathscr{L}$ is correct for M and $\gamma(E)$, assuming that the original information in $\mathscr{L}_0$ is correct. □

Theorem 4.14 (Correctness of ModAnalyze)**.** *Let P be a modular program, and Q_α a set of abstract queries. The modular analysis graph:*

$$\mathcal{G} = \textsc{ModAnalyze}(P, Q_\alpha)$$

is correct (Def. 4.6) for P and $\gamma(Q_\alpha)$.

Proof. By induction on the number of modular partitions. If there is only one partition, the conditions in Def. 4.4 hold trivially because the only intermodular call patterns are the Q_α (added in line 1). Since $\mathscr{L}$ is correct by Lemma 4.6 and the results are updated in line 9 the conditions in Def. 2.3 hold. And no further iteration is required.

If the program P is partitioned into n modules, we need to prove that if analyzing $n-1$ modules finishes, then analyzing all n modules also finishes. Assuming that the analysis of the first $n-1$ modules finishes and is correct, the result of these $n-1$ modules could be seen as one module, reducing this general case to the case of 2 modules. Hence, without loss of generality, we restrict ourselves to the case of 2 modules. To prove this the following invariant of the algorithm is used:

Before extracting from the queue via next-entries *(line 2), either the results in $\mathcal{G}$ are correct, or the queue is not empty.*

This invariant trivially holds immediately after initializing the queue with the queries in line 1. Then, at each iteration of the while loop, since there are only 2 modules, when one is extracted from the queue, the queue is empty. After analyzing (line 4), we know $\mathscr{L}$ is correct if $\mathcal{G}$ was correct. If no answers changed w.r.t. $\mathcal{G}$, no modules are added and the fixed point was reached. If the results change, every answer that changed is generalized and updated in $\mathcal{G}$, which results in adding an entry to it (line 9). Then, since there are only two modules, there can be at most one module in the queue, since the one being processed is extracted. If after processing one module the nodes and answers (excluding the answers to Q_α) stay the same, no new events will be added to the queue. In this case, then the analysis is already correct, by Lemma 4.13, because IncAnalyze was performed assuming already correct information. Otherwise, if new answers were encountered it means that the previous information was incomplete, these answers are stored (line 9), and the entries that depend on these answers are added to the queue, so the invariant holds. If new call patterns were encountered, then it means that the analysis was not completed yet. The algorithm, after generalization, schedules them to be reanalyzed (line 18), and therefore the invariant holds as well.

Intermodular dependencies are correctly approximated because the $\mathscr{L}$ is an over-approximation and we use int-calls$(E, \mathscr{L})$ to compute them.

Lastly, by Lemma 4.1, since `removeUnreachable` does not remove any inferred behavior and the result is guaranteed to be correct. □

As mentioned earlier, this algorithm was not designed to perform incremental analysis but rather to reduce the working set of the basic (monolithic) analyzer. In fact, in [146], the authors neither provide a clear strategy to tackle the problem of reusing the analysis result after modifying the program nor performed experiments.

4.2.4 *Correctness and precision of* ModAnalyzeI95

We now show the precision guarantees when analyzing with finite abstract domains, only if the generalization step is removed (no ∇ is performed). Henceforth we refer to this algorithm by ModAnalyzeI95, which is ModAnalyze modified by:

- replacing lines 15-14 by "$\lambda^c := \lambda_l^c$",
- replacing line 8 by "$\lambda^s := \lambda_l^s \sqcup \lambda_g^s$", and
- replacing the call to IncAnalyze by a call to IncAnalyze95.

Lemma 4.15 (Correctness and precision of IncAnalyze95 modulo imported predicates)**.** *Let M be a module of program P, E a set of abstract queries. Let $\mathscr{L}_0$ be an analysis graph such that $\forall\langle A, \lambda^c\rangle \in \mathscr{L}_0.\mathsf{mod}(A) \in \mathsf{imports}(M)$ correctly and precisely approximates the behavior of the imported predicates. The analysis result*

$$\mathscr{L} = \textsc{IncAnalyze95}(M, E, \emptyset, \mathscr{L}_0)$$

is the least analysis graph for M and $\gamma(E)$ assuming $\mathscr{L}_0$.

Proof. By hypothesis, since all values reused are the least fixed point, no imprecision is introduced by $\mathscr{L}_0$. Correctness follows from Lemma 4.13. □

Theorem 4.16 (Correctness and precision of ModAnalyzeI95)**.** *Let P be a modular program and Q_α a set of abstract queries. The modular analysis result*

$$\mathcal{G} = \textsc{ModAnalyzeI95}(P, Q_\alpha)$$

is the least modular analysis graph for P and $\gamma(Q_\alpha)$.

Proof. Since no imprecision is introduced during the modular processing, and all answers are started assuming $\bot$ (line 1), each of the calls to IncAnalyze95 produces results that are below or exactly the least fixed point. Correctness follows from Theorem 4.14. □

4.3 Running example of IncAnalyze

Example 4.3. (Analysis of Example 2.1, Fig. 5) In this example we show how events are queued and processed to produce the analysis graph of the program in Example 2.1 with initial query $\langle$`par(Msg,X,P)`$, (X/z)\rangle$ and the abstract domain of Figs. 2 and 3, i.e., the graph in Fig. 5. The algorithm starts by adding a *newcall* event for the initial query.

Algorithm processing	**Pending**
Add a *newcall* event for each initial query q: $newcall(q)$ (E1)	**E1**
Process E1: $newcall(\langle$`par(Msg,X,P)`$, (X/z)\rangle)$ Analyze `par/3`$_{1,1}$: l.27 $\lambda^p =$ `Acall(par(Msg,X,P), (X/z), par([],P,P))` $= (P/z)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars(true)) = \top$ l.29 $q = \langle$`par(Msg,X,P)`$, (X/z)\rangle$, `add-event`$(arc(q \to_{1,1} \langle true, \top\rangle))$ (E2).	**E2,E3**
Analyze `par/3`$_{2,1}$: l.27 $\lambda^p =$ `Acall(par(Msg,X,P), (X/z), par([C|Cs],P0,P))` $= (P0/z)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars($`xor(C,P0,P1)`$)) = (P0/z)$ l.29 $n_1 = \langle$`xor(C,P0,P1)`$, (P0/z)\rangle$, `add-event`$(arc(q \to_{1,1} n_1))$ (E3).	
Process E2: $arc(\langle$`par(Msg,X,P)`$, (X/z)\rangle \to_{1,1} \langle true, \top\rangle)$ l.31 $\lambda^c = \top$ l.33 $\lambda^s_0 = \top$ (*true* is a *built-in*) l.37 $\lambda^r =$ `Aextend`$(true, \lambda^p, \top) = (P0/z)$ l.42 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`par`$([],$`P,P`$))) = (P0/z)$ (*true* is the last literal) l.43 $\lambda^s =$ `Aproceed`$($`par`$([],$`P,P`$), \lambda^s_k,$ `par(Msg,X,P)`$) = (X/z, P/z)$ l.48 `upd`$(\mathscr{A}, \langle$`par(Msg,X,P)`$, (X/z)\rangle \mapsto (X/z, P/z))$ (in `insertAnswerInfo`, no generalization)	**E3**
Process E3: $arc(\langle$`par(Msg,X,P)`$, (X/z)\rangle \to_{1,1} \langle$`xor(C,P0,P1)`$, (P0/z)\rangle)$ l.31 $\lambda^c = (P0/z)$ l.22 `add-event`$(newcall($`xor(C,P0,P1)`$(P0/z)))$ (E4) l.35 $\lambda^s_0 = \bot$ (not analyzed) l.36 $n_2 = \langle$`xor(C,P0,P1)`$, (P0/z)\rangle$ `upd`$(\mathscr{A}, arc(n_1 \to_{2,1} n_2)$ l.37 $\lambda^r =$ `Aextend(xor(C,P0,P1)`$, \lambda^p, \bot) = \bot$ l.42 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`par([C|Cs],P0,P)`$)) = \bot$ l.38 Do not analyze `par/3`$_{2,2}$.	**E4**
Process E4: $newcall($`xor(C,P0,P1)`$(P0/z))$ Analyze `xor/3`$_{1,1}$: l.27 $\lambda^p =$ `Acall(xor(C,P0,P1), (P0/z), xor(0,0,0))` $= (C/z, P0/z, P1/z)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars(true)) = \top$ l.29 $n_3 = \langle$`xor(C,P0,P1)`$, (P0/z)\rangle$, `add-event`$(arc(n_3 \to_{1,1} \langle true, \top\rangle))$ (E5). Analyze `xor/3`$_{2,1}$: l.27 $\lambda^p =$ `Acall(xor(C,P0,P1), (P0/z), xor(0,1,1))` $= \bot$ abstract substitution and the head do not unify and the clause is not analyzed[a].	
Analyze `xor/3`$_{3,1}$: l.27 $\lambda^p =$ `Acall(xor(C,P0,P1), (P0/z), xor(1,0,1))` $= (C/o, P0/z, P1/o)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars(true)) = \top$ l.29 `add-event`$(arc(n_3 \to_{1,1} \langle true, \top\rangle))$ (E6).	**E5, E6**
Analyze `xor/3`$_{4,1}$: l.27 $\lambda^p =$ `Acall(xor(C,P0,P1), (P0/z), xor(1,1,1))` $= \bot$ abstract substitution and the head do not unify and the clause is not analyzed.	

a This happens because the example that is not normalized. Actually, analysis of facts is optimized in the implementation.

Process E5: $arc(\langle$`xor(C,P0,P1)`$, (P0/z)\rangle \to_{1,1} \langle true, \top\rangle)$ l.31 $\lambda^c = \top$ l.33 $\lambda^s_0 = \top$ (*true* is a *built-in*) l.37 $\lambda^r =$ `Aextend`$(true, \lambda^p, \top) = (P0/z)$ l.42 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`xor`$(0,0,0))) = \top$ (no variables, last literal) l.43 $\lambda^s =$ `Aproceed`$($`xor`$(0,0,0)), \lambda^s_k,$ `xor(C,P0,P1)`$) = (C/z, P0/z, P1/z)$ l.48 `upd`$(\mathscr{A}, \langle$`xor(C,P0,P1)`$, (P0/z)\rangle \mapsto (C/z, P0/z, P1/z))$ (in `insertAnswerInfo`, no generalization) l.52 `add-event`$(arc(q \to_{2,1} n_1))$ in `reanalyzeUpdated` (E6)	**E6, E7**
Process E6: $arc(\langle$`xor(C,P0,P1)`$, (P0/z)\rangle \to_{3,1} \langle true, \top\rangle)$ l.31 $\lambda^c = \top$ l.33 $\lambda^s_0 = \top$ (*true* is a *built-in*) l.37 $\lambda^r =$ `Aextend`$(true, \lambda^p, \top) = (P0/z)$ l.42 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`xor`$(1,0,1))) = \top$ (no variables, last literal) l.43 $\lambda^s =$ `Aproceed`$($`xor`$(1,0,1)), \lambda^s_k,$ `xor(C,P0,P1)`$) = (C/o, P0/z, P1/o)$ l.46 $\lambda^s_1 = (C/z, P0/z, P1/z) \sqcup (C/o, P0/z, P1/o) = (C/b, P0/z, P1/b)$ l.48 `upd`$(\mathscr{A}, \langle$`xor(C,P0,P1)`$, (P0/z)\rangle \mapsto \lambda^s_1)$ l.52 `add-event`$(arc(q \to_{2,1} n_1))$ in `reanalyzeUpdated` (already added)	**E7**
Process E7: $arc(\langle$`par(Msg,X,P)`$, (X/z)\rangle \to_{1,1} \langle$`xor(C,P0,P1)`$, (P0/z)\rangle)$ l.31 $\lambda^c = (P0/z))$ l.35 $\lambda^s_0 = (C/b, P0/z, P1/b)$ l.37 $\lambda^r =$ `Aextend`$($`xor(C,P0,P1)`$, \lambda^p, \lambda^s_0) = (C/b, P0/z, P1/b)$ l.38 Analyze then next literal, `par/3`$_{2,2}$ l.39 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`par`$($`Cs`$,$`P1`$,$`P`$))) = (P1/b)$ l.40 $n_3 = \langle$`par(Cs,P1,P)`$, (P1/b)\rangle$, `add-event`$(arc(n_1 \to_{2,2} n_3))$ (E8)	**E8**
Process E8: $arc(\langle$`par(Msg,X,P)`$, (X/z)\rangle \to_{1,1} \langle$`par(Cs,P1,P)`$, (P1/b)\rangle)$ l.31 $\lambda^c = (P0/b))$ l.22 `add-event`$(newcall($`par(Cs,P1,P)`$(P1/b)))$ (E9) l.35 $\lambda^s_0 = \bot$ (not analyzed) l.36 $n_4 = \langle$`par(Cs,P1,P)`$, (P1/b)\rangle$ `upd`$(\mathscr{A}, arc(n_2 \to_{2,2} n_4))$ l.37 $\lambda^r =$ `Aextend`$($`xor(C,P0,P1)`$, \lambda^p, \bot) = \bot$ l.42 $\lambda^s_k =$ `Aproj`$(\lambda^r, vars($`par`$([$`C`$|$`Cs`$],$`P0`$,$`P`$))) = \bot$	**E9**
Process E9: $newcall($`par(Cs,P1,P)`$(P1/b))$: Analyze `par/3`$_{1,1}$: l.27 $\lambda^p =$ `Acall`$($`par(Cs,P1,P)`$, (X/b),$ `par`$([],$`P`$,$`P`$)) = (P/b)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars(true)) = \top$ l.29 `add-event`$(arc(n_4 \to_{1,1} \langle true, \top\rangle))$ (E10). Analyze `par/3`$_{2,1}$: l.27 $\lambda^p =$ `Acall`$($`par(Cs,P1,P)`$, (P1/b),$ `par`$([$`C`$|$`Cs`$],$`P0`$,$`P`$)) = (P0/b)$ l.28 $\lambda^c_1 =$ `Aproj`$(\lambda^p, vars($`xor`$($`C`$,$`P0`$,$`P1`$))) = (P0/b)$ l.29 $n_5 = \langle$`xor(C,P0,P1)`$, (P0/b)\rangle$, `add-event`$(arc(n_4 \to_{1,1} n_5))$ (E11).	**E10, E11**

Process E10: $arc(\langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle \rightarrow_{1,1} \langle true, \top\rangle)$ Similar to processing E2, replacing X/z by X/b.	**E11**
Process E11: $arc(\langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle \rightarrow_{1,1} \langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle)$ Similar to processing E3, replacing X/z by X/b. In l.22 add-event$(newcall(\texttt{xor(C,P0,P1)}(P0/b)))$ (E12)	**E12**
Process E12: $newcall(\langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle)$ $\texttt{xor/3}_1$: l.27 $\lambda^p = \texttt{Acall}(\texttt{xor(C,P0,P1)}, (P0/b), \texttt{xor(0,0,0)}) = (C/z, P0/b, P1/z)$ l.28 $\lambda^c_1 = \texttt{Aproj}(\lambda^p, vars(true)) = \top$ l.29 add-event$(arc(n_5 \rightarrow_{1,1} \langle true, \top\rangle))$ (E13). $\texttt{xor/3}_2$: l.27 $\lambda^p = \texttt{Acall}(\texttt{xor(C,P0,P1)}, (P0/z), \texttt{xor(0,1,1)}) = (C/z, P0/b, P1/o)$ l.28 $\lambda^c_1 = \texttt{Aproj}(\lambda^p, vars(true)) = \top$ l.29 add-event$(arc(n_5 \rightarrow_{2,1} \langle true, \top\rangle))$ (E14). $\texttt{xor/3}_3$: l.27 $\lambda^p = \texttt{Acall}(\texttt{xor(C,P0,P1)}, (P0/z), \texttt{xor(1,0,1)}) = (C/o, P0/b, P1/o)$ l.28 $\lambda^c_1 = \texttt{Aproj}(\lambda^p, vars(true)) = \top$ l.29 add-event$(arc(n_5 \rightarrow_{3,1} \langle true, \top\rangle))$ (E15). $\texttt{xor/3}_4$: l.27 $\lambda^p = \texttt{Acall}(\texttt{xor(C,P0,P1)}, (P0/z), \texttt{xor(1,1,1)}) = (C/o, P0/b, P1/o)$ l.28 $\lambda^c_1 = \texttt{Aproj}(\lambda^p, vars(true)) = \top$ l.29 add-event$(arc(n_5 \rightarrow_{4,1} \langle true, \top\rangle))$ (E16).	**E13, E14, E15, E16**
Process E13, E14, E15, E16: Similar to E5 In l.52 add-event$(arc(n_4 \rightarrow_{2,2} n_5))$ in `reanalyzeUpdated` (E17)	**E17**
Process E17: $arc(\langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle \rightarrow_{2,2} \langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle)$ l.31 $\lambda^c = (P0/b))$ l.35 $\lambda^s_0 = (C/b, P0/b, P1/b)$ l.37 $\lambda^r = \texttt{Aextend}(\texttt{xor(C,P0,P1)}, (P0/z), \lambda^s_0) = (C/b, P0/b, P1/b)$ l.38 Analyze then next literal, $\texttt{par/3}_{2,2}$ l.39 $\lambda^s_k = \texttt{Aproj}(\lambda^r, vars(\texttt{par(Cs,P1,P)})) = (P1/b)$ l.40 $n_3 = \langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle$, add-event$(arc(n_1 \rightarrow_{2,2} n_3))$ (E18)	**E18**
Process E18: $arc(\langle \texttt{par(Cs,P1,P)}, (X/b)\rangle \rightarrow_{2,2} \langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle)$ l.31 $\lambda^c = (P1/b))$ l.35 $\lambda^s_0 = (C/b, P0/b, P1/b)$ l.37 $\lambda^r = \texttt{Aextend}(\texttt{xor(C,P0,P1)}, (C/b, P0/b, P1/b), \lambda^s_0) = (C/b, P0/b, P1/b, P/b)$ l.42 $\lambda^s_k = \texttt{Aproj}(\lambda^r, vars(\texttt{par([C\|Cs],P0,P)})) = (P0/b, P/b)$ (last literal) l.43 $\lambda^s = \texttt{Aproceed}(\texttt{par([C\|Cs],P0,P)}, \lambda^s_k, \texttt{par(Msg,X,P)}) = (X/b, P/b)$ l.48 upd$(\mathscr{A}, \langle \texttt{par(Cs,P0,P1)}, (P0/b)\rangle \mapsto (P0/b, P1/b))$ (`insertAnswerInfo`) l.40 add-event$(arc(q \rightarrow_{2,2} n_1))$ (E19) (`reanalyzeUpdated`)	**E19**
Process E19: $arc(\langle \texttt{par(Msg,X,P)}, (X/z)\rangle \rightarrow_{2,2} \langle \texttt{par(Cs,P1,P)}, (P1/b)\rangle)$ l.31 $\lambda^c = (P1/b))$ l.35 $\lambda^s_0 = (P1/b, P/b)$ l.37 $\lambda^r = \texttt{Aextend}(\texttt{par(Cs,P1,P)}, (C/b, P0/z, P1/b), \lambda^s_0) = (C/b, P0/z, P1/b, P/b)$ l.42 $\lambda^s_k = \texttt{Aproj}(\lambda^r, vars(\texttt{par([C\|Cs],P0,P)})) = (C/b, P0/z, P/b)$ l.43 $\lambda^s = \texttt{Aproceed}(\texttt{par([C\|Cs],P0,P)}, \lambda^s_k, \texttt{par(Msg,X,P)}) = (X/z, P/b)$ l.48 upd$(\mathscr{A}, \langle \texttt{par(Msg,X,P)}, (X/z)\rangle \mapsto (X/z, P/b))$ (`insertAnswerInfo`)	**fixpoint reached**

There are no nodes that are unreachable nodes from the initial query so the graph remains the same.

This concludes the chapter. A practical evaluation of the algorithms is presented in Chapter 8. In the following chapters we present extensions and uses of these algorithms aiming to improve performance and thus scalability of fixpoint algorithms.

5

Incremental and Modular Context-sensitive Analysis

As mentioned in the thesis introduction, our overall objective is to reduce the response times and increase the scalability of abstract interpretation-based analysis and verification of real-life programs, specially for interactive scenarios. Such real-life programs typically have a complex structure combining a number of modules with system libraries. At the same time, very often changes in the program are small and isolated inside a few components. To be able exploit these two aspects simultaneously, we turn our attention in this chapter to the combination of the two techniques studied in the previous chapter: incrementality and modularity. We provide a new analysis algorithm that performs a goal-directed, top-down, multivariant, incremental abstract interpretation of modular logic programs. This can be seen as the equivalent in analysis of modular recompilation. However, note that, unlike in compilation, this process requires iterating and possibly analyzing the same module several times, until an intermodular fixpoint is found.

5.1 Towards combining incrementality and modularity

As already introduced in the previous chapter, in the field of abstract interpretation, there have been proposals to deal with the following two cases: a) context-sensitive fine-grain incremental fixpoint algorithms [147, 96, 86, 4, 8, 169], which reuse information but still need to work with the program as a whole (incremental but *monolithic* analyzers), such as Algorithm 1 ; and b) *modular* algorithms, aimed at reducing the memory consumption or working set size [30, 24, 39, 146, 33, 41, 55], which work on a module at a time but do not support changes in the program, such as Algorithm 2. These help in scalability of analysis because they allow different levels of abstraction for each software component, and thus being more efficient at the expense of precision, e.g., for some components. Surprisingly, the combination of both techniques has not been explored to date. The monolithic incremental analyzers are

not directly applicable in the modular setting due to two issues: first, these analyzers do not deal with code that is partially available, i.e., they have no provisions to make assumptions about code that is external. Even though one could see builtin operations of the language as external calls, as they are obviously not defined in the module, the semantics of these are typically "hardwired" in the analyzer as *transfer functions*. This leads to the second issue: even though the monolithic analyzers can make assumptions using this mechanism, these algorithms are not prepared to deal in a correct and precise way with updates to these assumptions.

In order to bridge this gap, using the monolithic incremental analysis algorithm as a a starting point, we develop a modular, incremental analyzer capable of performing fine grain incremental analysis across modular program partitions. Our algorithm is based on (re)computing local fixpoints on one module at a time; identifying, invalidating, and recomputing only those parts of the analysis results that are affected by these fine-grain program changes; and propagating the fine-grained analysis information across module boundaries.

5.2 Analysis graphs for modular and incremental analysis

We propose to keep, in addition to $\mathcal{G}$, a local analysis graph per modular partition M, referred to with $\mathscr{L}_M$. This helps us processing partitions of programs modularly but, at the same time, being able to update localized information. The analysis result then consists on a set of graphs $\{\mathcal{G}, \{\mathscr{L}_i\}\}$. An example of an analysis result of this shape is depicted in Fig. 13. The information of the local analysis graphs is drawn in black. The left box corresponds to the `main` module, $\mathscr{L}_{\texttt{main}}$, and the box on the left to the `bitops` module, $\mathscr{L}_{\texttt{bitops}}$. The nodes in blue show the information in the global analysis graph $\mathcal{G}$, which coincides with Fig. 12.

5.3 Operation of the algorithm

The algorithm ModIncAnalyze$(P = \{M_i\}, Q_\alpha, \mathcal{G}, \{\mathscr{L}_i\}, \Delta)$ takes as input a (partitioned) program $P = \{M_i\}$, some abstract queries Q_α, a previous correct analysis result $\{\mathcal{G}, \{\mathscr{L}_i\}\}$, and a set of program edits in the form of additions and deletions (Δ_{M_i}), which collect the differences w.r.t. the previous state for each module. The pseudocode of the algorithm is detailed in Algorithm 3. Before starting the analysis process, the entries of edited modules and new queries are marked to be (re)analyzed. Each of the scheduled modules will be analyzed independently, and possibly several times. Modular analysis is, again, controlled by a queue to which entries with possibly incomplete answer substitutions are added (with the procedure add-entries). At each

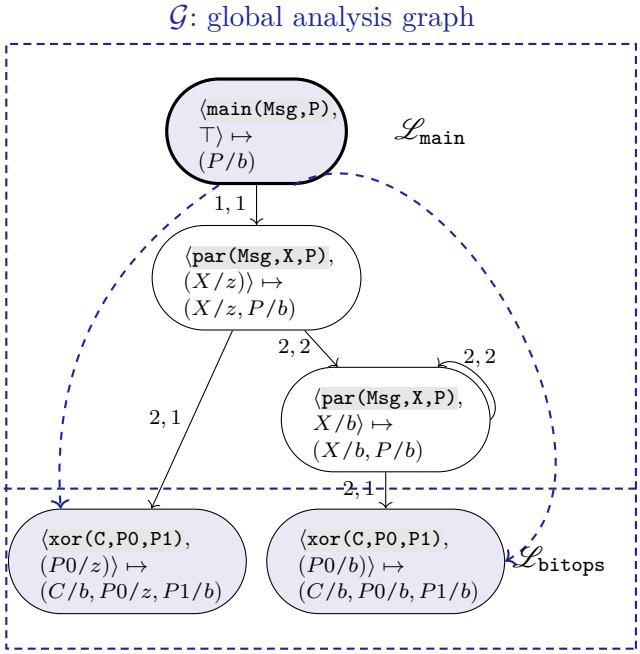


Fig. 13: Analysis result for the program in Fig. 11, keeping a local analysis graph per module.

iteration of the loop a module is reanalyzed independently for its set of annotated entries (E) extracted from the queue. This is done by procedure next-entries which extracts from the queue entries that are reachable from the initial Q_α in $\mathcal{G}$. Incrementally analyzing a module consists of updating the information about the calls to imported predicates in $\mathscr{L}_M$, by removing possibly inaccurate results and adding the newly computed ones, and calling IncAnalyze. Finally, $\mathcal{G}$ is updated, which includes updating the newly computed answers, updating the dependencies of the predicates in the boundary of the modules, and adding to the queue to reanalyze the dependent predicates and call patterns. The description of each set of operations is:

AnalyzeOutdated Adds to the analysis queue the entries of modules that changed, i.e, those whose diff (Δ) is not empty.

AnalyzeNew Adds to the queue the entries that have not been analyzed yet.

Imported Collects the current approximations made about the predicates imported by the module to be analyzed.

IncorrectImported Collects in I_c the nodes of the $\mathscr{L}_M$ that are incorrect (below the fixpoint), i.e., the ones whose approximation in the $\mathscr{L}_M$ was smaller than in the $\mathcal{G}$ to reanalyze them later.

Algorithm 3 ModIncAnalyze: Incremental and modular fixpoint algorithm.

ModIncAnalyze$(P = \{M_i\}, Q_\alpha, \mathcal{G}, \{\mathscr{L}_i\}, \Delta)$

1: add-entries$(\{k = \langle A, \lambda^c\rangle \mid k \in \mathcal{G}, \Delta_{\mathsf{mod}(A)} \neq \emptyset\})$ ▷ **AnalyzeOutdated**
2: add-entries$(\{k \in Q_\alpha \mid k \notin \mathcal{G}\})$, upd$(\mathcal{G}, \{k \mapsto \bot \mid k \in Q_\alpha\})$ ▷ **AnalyzeNew**
3: **while** entries$(\mathcal{G}, Q_\alpha) \neq \emptyset$ **do**
4: $(M, E) :=$ next-entries$(\mathcal{G}, Q_\alpha)$
5: $I := \{\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{L}_M \mid \mathsf{mod}(A) \in \mathsf{imports}(M)\}$ ▷ **Imported**
6: $I_p := \{k \mid k \mapsto \lambda^s \in I, n \mapsto \lambda^{s\prime} \in \mathcal{G}, \lambda^s \not\sqsubseteq \lambda^{s\prime}\}$ ▷ **ImpreciseImported**
7: $I_c := \{k' \mid k' \rightsquigarrow k \in \mathscr{L}_M,\ n \mapsto \lambda^s \in I, n \mapsto \lambda^{s\prime} \in \mathcal{G}, \lambda^{s\prime} \sqsubset \lambda^s\})$
8: ▷ **IncorrectImported**
9: del$(\mathscr{L}_M, \{k \mid k_c \in I_p, k \rightsquigarrow k_c \in \mathscr{L}_M$ or $(n_a \rightsquigarrow k_c \in \mathscr{L}_M \wedge k_a \rightsquigarrow k \in \mathscr{L}_M)\})$
10: ▷ **DelImprecise**
11: upd$(\mathscr{L}_M, \{\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathcal{G} \mid \mathsf{mod}(A) \in \mathsf{imports}(M)\})$ ▷ **PreloadImported**
12: $\mathscr{L}_M :=$ IncAnalyze$(M, E \cup I_c, \Delta_M, \mathscr{L})$, $\Delta_M \leftarrow \emptyset$
13: **for** $\langle A, \lambda^c\rangle \mapsto \lambda^s_l \in \mathscr{L}_M . \langle A, \lambda^c\rangle \mapsto \lambda^s_g \in \mathcal{G} \Rightarrow \lambda^s_l \neq \lambda^s_g$, $\mathsf{mod}(A) \neq M$ **do**
14: $\lambda^s :=$ `Ageneralize`$(\lambda^s_l, \{\lambda^s_g\})$
15: upd$(\mathcal{G}, \langle A, \lambda^c\rangle \mapsto \lambda^s)$ ▷ **StoreAnswers**
16: add-entries$(\{k \mid k \to \langle A, \lambda^c\rangle \in \mathcal{G}\})$
17: del$(\mathcal{G}, \{\langle A, \lambda^c\rangle \to k' \in \mathcal{G}\})$ ▷ **UpdateDependencies**
18: $R =$ int-calls$(E, \mathcal{G})$
19: **for** $k \to \langle B, \lambda^c_t\rangle \in R$ **do**
20: $Calls := \{\lambda \mid \langle A, \lambda\rangle \in \mathcal{G}\}$
21: $\lambda^c :=$ `Ageneralize`$(\lambda^s_t, Calls)$
22: **if** $\langle B, \lambda^c\rangle \notin \mathcal{G}$ **then** ▷ **ScheduleNewCalls**
23: add-entries$(\langle B, \lambda^c\rangle\})$
24: upd$(\mathcal{G}, \{k \to \langle B, \lambda^c\rangle\})$
25: `removeUnreachable`$(\mathcal{G}, Q_\alpha)$
26: **return** $\mathcal{G}, \{\mathscr{L}_i\}$

ImpreciseImported Collects in I_p all the imported nodes that are potentially imprecise, i.e., those in which in the abstraction $\mathscr{L}_M$ that are bigger than the current stored in $\mathcal{G}$.

DelImprecise Deletes from $\mathscr{L}_M$ the nodes that relied on assumptions that depend on I_p, because they are potentially imprecise.

PreloadImported *Preloads* the inferred behaviors of the imported predicates in $\mathscr{L}_M$ to avoid unnecessary module swaps. the unused ones are removed immediately after analyzing.

Analyze The IncAnalyze function is called with entries for: the calls scheduled by the modular analyzer (E), the nodes that depended on imported information

that may be below the fixpoint (I_c). Note that no entries will be added for the nodes that were imprecise as its information will be removed up to the entries that they were triggered by, which guarantees that the analysis will be correct and precise for those entries.

StoreAnswers Updates the answer substitutions of the global analysis graph, after generalizing them, and adding the dependent call substitutions to be reanalyzed. This process is the same as in the original modular analysis shown in Algorithm 2.

UpdateDependencies Updates the dependencies of exported-imported predicates by traversing the local analysis graph to find which exported predicate called imported predicates and with which call substitutions.

ScheduleNewCalls Adds to the analysis queue any newly encountered call substitutions, before generalizing them if needed. This process is the same as in the original modular analysis shown in Algorithm 2.

5.3.1 Enhancing the deletion strategy

The proposed deletion strategy is quite pessimistic. Updating imprecise information about imported predicates most of the times means reusing only a few answers that did not depend on the changes per module. However, it may occur that the analysis does not change after these changes occur, or that some nodes/edges are still correct and precise. A solution is to partially reanalyze the program without removing these potentially useful results. Our proposed algorithm allows performing such a partial reanalysis, by partitioning the desired module into smaller partitions, for example, using information on *strongly connected components*. This can be achieved within the algorithm by replacing line 9 (**DelImprecise**) with Algorithm 4. This runs the algorithm with a partition of the current module as input program, which is split using the (static) SCCs of the clauses (split-sources-scc). This includes also partitioning the results (split-in-scc) to initialize $\mathcal{G}$ using $\mathscr{L}_M$, and setting as Q_α the initial E of this modular analysis. The reanalysis of this partitioned module will be given in a modular form, so it has to be *flattened* back for it to be compatible with the rest of the analysis results. This process consists in merging all the graphs in which the analysis was performed into one graph that contains all the nodes and edges except the edges in $\mathcal{G}''$.

Algorithm 4 Enhanced modular deletion strategy for ModIncAnalyze.

DelImprecise

1: $Calls = \{k \mid k \mapsto \lambda^s \in \mathscr{L}_M, k \in I_p\}$
2: $\{M'_i\}, I_{M'_i} = \text{split-sources-scc}(M, I_p)$
3: $\{\mathcal{G}', \{\mathscr{L}'_i\}\} = \text{split-in-scc}(\mathscr{L}_M)$
4: $\{\mathcal{G}''\{\mathscr{L}'_i\}\} := \text{ModIncAnalyze}(\{M'_i\}, I_{M'_i}, \{\mathcal{G}', \{\mathscr{L}'_i\}\}, \emptyset)$
5: $\mathscr{L}_M := \text{flatten}(\{\mathcal{G}''\{\mathscr{L}'_i\}\})$
6: $\Delta_M \leftarrow \emptyset$

M: (unchanged) `main` module

```
1  :- module(main, [main/1]).
2
3  :- use_module(bitops).
4  main(Msg, P) :-
5      par(Msg, 0, P).
6
7  par([], P, P).
8  par([C|Cs], P0, P) :-
9      xor(C, P0, P1),
10     par(Cs, P1, P).
```

B_0: initial state of `bitops`

```
1  :- module(bitops, [xor/3]).
2  xor(0,0,0).
```

B_1: clauses added to `bitops`

```
1  :- module(bitops, [xor/3]).
2  xor(0,0,0). % also in B0
3  xor(0,1,1).
4  xor(1,0,1).
5  xor(1,1,0).
```

B_2: a clause is deleted from `bitops`

```
1  :- module(bitops, [xor/3]).
2  xor(0,0,0).
3  xor(0,1,1).
4  xor(1,0,1).
5  % xor(1,1,0). %%% commented
```

Fig. 14: Different program states.

5.3.2 *Precision using* IncAnalyze95

If generalization is removed from the algorithm by:

- replacing lines 21-20 by "$\lambda^c := \lambda^c_t$",
- replacing line 14 by "$\lambda^s := \lambda^s_l \sqcup \lambda^s_g$", and
- replacing the call to IncAnalyze by a call to IncAnalyze95.

We obtain, again, an algorithm, henceforth called ModIncAnalyzeI95 that is precise for finite abstract domains.

5.3.3 *Running examples of the algorithm*

To show the algorithm in action we now analyze incrementally different versions of the program that computes the parity (some of which are incomplete). The different

states of the sources are shown in Fig. 14. Initially we have the analysis result of $P_0 = \{M, B_0\}$, see $\mathscr{A}_0$ in Fig. 15. This was the result of running the algorithm from scratch $\mathscr{A}_0 = \textsc{ModIncAnalyze}(P_0, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, with initial query $Q_\alpha = \{\langle \texttt{main}(M, P), (\top)\rangle\}$. In this version it was inferred that if `main(M, P)` succeeded then `P` is 0 ($\gamma(z)$).

Example 5.1 (Adding clauses)**.** If some clauses are added to `bitops` resulting in B_1, the program to be (re)analyzed becomes $P_1 = \{M, B_1\}$. Incremental analysis by running $\textsc{ModIncAnalyze}(P, Q_\alpha, \mathscr{A}_0, (\{\texttt{xor}_2, \texttt{xor}_3, \texttt{xor}_4\}, \emptyset))$ proceeds as follows. The entries of `bitops` are added to the queue and it is analyzed with $E = \{\langle \texttt{xor(C,P0,P1)}, P0/z\rangle\}$ and the analysis result changes to $(C/b, P0/z, P1/b)$ (shown in $\mathscr{A}'_0$). This change needs to be propagated to module `main`, which is analyzed next in the queue. Following the steps of the algorithm:

AnalyzeOutdated The entries to the module `main` are added.

AnalyzeNew No entries are added because there are no new queries.

Imported $I = \{\langle \texttt{xor(C,P0,P1)}, P0/z\rangle\}$

IncorrectImported $I_c = \{\langle \texttt{xor(C,P0,P1)}, P0/z\rangle\}$

ImpreciseImported $I_p = \emptyset$ since the only imported node was below the fixed point.

Analyze The analyzer is called with $E = \{\langle \texttt{main(M,P)}, \top\rangle\}$ and I_c as described.

StoreAnswers $\langle \texttt{main(M, P)}, \top\rangle \mapsto (P/z)$ is updated in $\mathcal{G}$, no (parent) entries need to be added to the queue because it is the initial query.

UpdateDependencies All the edges of $\mathcal{G}$ from nodes of `main` to `bitops` are removed. $R = \{\langle \texttt{main(M,P)}, \top\rangle \to \langle \texttt{xor(C,P0,P1)}, (P0/z)\rangle,$
$\langle \texttt{main(M,P)}, \top\rangle \to \langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle\}$

ScheduleNewCalls A newly encountered call substitution is added in add-entries, $\langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle$, and all the edges in R are added to $\mathcal{G}$.

Next, module `bitops` needs to be analyzed again, only for the pending call substitution $\langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle$, the new answer $(C/b, P0/b, P1/b)$ will be updated in $\mathcal{G}$, adding again an entry for predicate `main`. The next iteration of the analysis loop, the answer will be updated but it will not imply any changes in the analysis result of the module, therefore the algorithm reached a fixed point ($\mathscr{A}_1$ in Fig. 15).

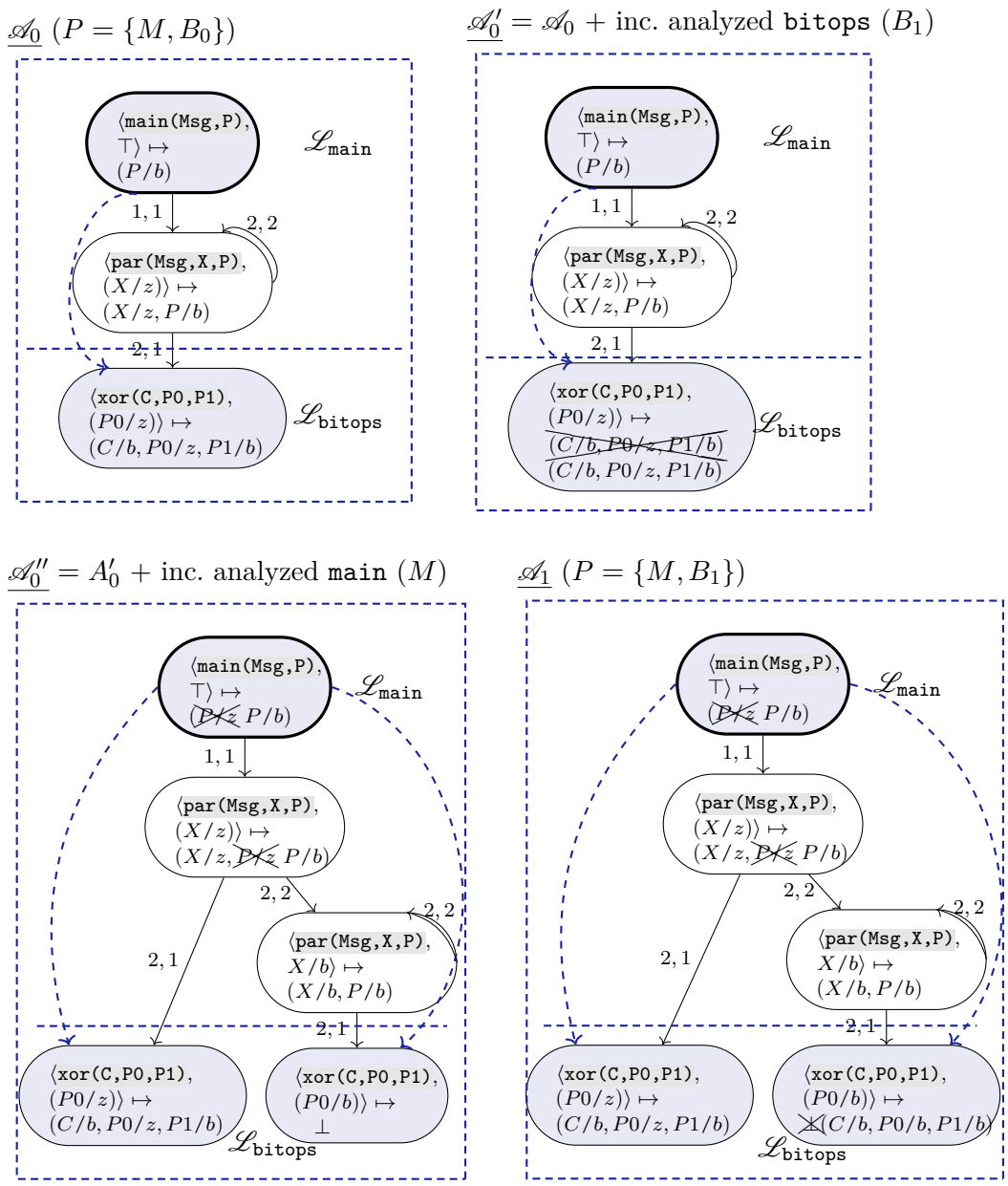


Fig. 15: Analysis results in several reanalysis steps.

Example 5.2 (Deleting clauses)**.** The `bitops` module is edited from B_1 to B_2, and the program to be analyzed is $P_2 = \{M, B_2\}$. Incremental analysis by ModIncAnalyze$(P_2, Q_\alpha, \mathscr{A}_1, (\emptyset, \{\texttt{xor}_4\}))$ proceeds as follows. Module `bitops` was changed, so it is analyzed with $E = \{\langle \texttt{xor(C,P0,P1)}, (P0/z)\rangle, \langle \texttt{xor(C,P0,P1)}, (P0/b)\rangle\}$. The answers are recomputed from scratch, however, the overall result of the module does not change, so nothing needs to be done in $\mathcal{G}$, and it is not necessary to recompute the analysis graph of module `main`, and $\mathscr{A}_2 = \mathscr{A}_1$.

5.4 Fundamental results of the algorithm

In this section we provide the correctness and precision guarantees of the proposed algorithm. We use the same notation as in Sec. 4.1.3. The incremental analysis of a module within the algorithm (in the body of the while loop in the pseudocode lines 5 to 12) is denoted with the function:

$$\mathscr{L}_{M'} = \text{LocIncAnalyze}(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M),$$

where M' is a module, $\mathscr{L}_M$ is the analysis result of M for E, Δ_M are the differences between M' and M, $\mathscr{L}_{M'}$ is the analysis result of M', and $\mathcal{G}$ contains the (possibly temporary) information for the predicates imported by M'.

Lastly, we represent performing an iteration of the while loop (lines 5 to 24) as the high-level operation of updating the newly computed information in $\mathcal{G}$:

$$MA(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M) = \texttt{upd}(\mathcal{G}, \text{LocIncAnalyze}(M', E, \mathcal{G}, \mathscr{L}_M, \Delta_M))$$

Note that, after a number of (chaotic) iterations, MA is monotonic, and ultimately stationary due to the use of the widening operator.

Let $MAI95$ and LocIncAnalyzeI95 the counterparts of the functions MA and LocIncAnalyze when referred to in ModIncAnalyzeI95 (Sec. 5.3.2).

5.4.1 *Correctness of* ModIncAnalyze

The following lemma shows that if a module M is analyzed for entries E assuming some $\mathcal{G}$ obtaining $\mathscr{L}_M$, if the assumptions change to $\mathcal{G}'$, incrementally updating these assumptions produces an analysis graph $\mathscr{L}'_M$ that is correct assuming $\mathcal{G}'$.

Lemma 5.1 (Correctness updating $\mathscr{L}$ modulo $\mathcal{G}$)**.** *Let M be a module of program P and E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M$ is correct for M and $\gamma(E)$ assuming $\mathcal{G}$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result*

$$\mathscr{L}'_M = \text{LocIncAnalyze}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset)$$

is correct (see Def. 2.5) for M and $\gamma(E)$ assuming $\mathcal{G}$.

Proof. To prove this we need to show that all the answers that differ from $\mathcal{G}$ to $\mathcal{G}'$ for the calls to predicates imported by M are included in E. Since these are the requisites in Theorem 4.11 to guarantee that the result is correct. The **ImpreciseImported** are collected and removed, therefore it is guaranteed that all the entries in E that depended on these will be correct (the analysis is empty). When collecting the **IncorrectImported** only those nodes are added to the entries. However, because $\mathscr{L}_M$ was assumed to be correct, it is guaranteed that adding these entries is enough, because $\mathscr{L}_M$ correctly over-approximates the parts of $[\![P]\!]_Q$ that were already in $\mathscr{L}_M$ and the ones that are missing are guaranteed to be correct by Lemma 4.10. □

The following Prop. 5.2 captures the correctness of the algorithm when starting from an empty analysis result, i.e., starting with an empty $\mathcal{G}$ and $\mathscr{L}_{M_i}$. Note that this is not the same as running the traditional modular analysis, as information is reused when iterating between modules, whereas in ModAnalyze every iteration the $\mathscr{L}$'s are clean.

Proposition 5.2 (Correctness of ModIncAnalyze from scratch)**.** *Let P be a modular program, and Q_α a set of abstract queries. Then, if:*

$$\{\mathcal{G}, \{\mathscr{L}_{M_i}\}\} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, \emptyset)$$

$\mathcal{G}$ is correct (see Def. 4.6) for P and $\gamma(Q_\alpha)$.

Proof. Correctness follows using the same argument as in Theorem 4.14, with the difference that instead of applying Lemma 4.13 to IncAnalyze, we apply Lemma 5.1 to LocIncAnalyze. □

Theorem 5.3 (Correctness of ModIncAnalyze)**.** *Let P, P' be modular programs that differ by Δ, Q_α a set of abstract queries, and $\mathscr{A} = \textsc{ModIncAnalyze}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then if:*

$$\{\mathcal{G}', \{\mathscr{L}'_{M_i}\}\} = \textsc{ModIncAnalyze}(P', Q_\alpha, \mathscr{A}, \Delta)$$

$\mathcal{G}'$ is correct (see Def. 4.6) for P and $\gamma(Q_\alpha)$.

Proof. We proceed by induction on the number of modular partitions. By assumption, the partitions of program P in n modules have no recursive dependencies on predicates between modules. This condition ensures that if removing some clause in $\mathscr{L}_M$ is needed *all* the dependent information for recomputing is indeed removed (nothing imprecise is reused from some other $\mathscr{L}_{M'}$).

- If program P has one module it is the case of the monolithic algorithm, and correctness follows from Theorem 4.9.
- As in the proof of Theorem 4.14, if program P is partitioned into n modules, we need to prove that if we finish analyzing $n-1$ modules, then we finish analyzing all n modules. Assuming that the analysis of the first $n-1$ modules finishes and it is correct, this $n-1$ result could be seen as one module, reducing this general case to the case of 2 modules.
- If the program $P = \{M_a, M_b\}$ is partitioned into 2 modules let us further assume that M_a imports M_b. Let us assume that we reanalyze M_b first. We study the reanalysis cases of $\mathcal{G}' = MA(M_b, E, \mathcal{G}, \mathscr{L}_{M_b}, \Delta_{M_b})$:
  1. If $\mathcal{G}' = \mathcal{G}$ the procedure is equivalent to the program P having one module.
  2. If $\mathcal{G} \sqsubset \mathcal{G}'$, then analysis results need to be propagated to A. Once the results of A are updated, the analysis iterations of A and B will be equivalent as when analyzing from scratch, only new call patterns may appear.
  3. If $\mathcal{G}' \sqsubset \mathcal{G}$ these analysis results need to be propagated to the analysis of A, which will be reanalyzed. Once A and B have updated their incomparable information the further (re)analyses can only become smaller, but since MA is monotonic, and there are no recursive dependencies between modules, the imprecise information is eventually removed, a fixpoint is reached, which is correct for P, since the computation of each of the modules is correct.
  4. Else, the information is incomparable. This can only happen if there were additions and deletions. This information needs to be propagated to M_a and the reanalysis of M_a will only lead to cases 1, 2, or 3.

□

Note that the correctness of the proposed enhanced deletion strategy follows from Theorem 5.3.

5.4.2 *Correctness and precision of* ModIncAnalyzeI95

We now show the precision guarantees of the algorithm when analyzing with finite abstract domains with ModIncAnalyzeI95. First note that for the *MA* function, since the *lfp* is monotonic w.r.t. the initial assumptions and `upd` is monotonic, if generalization is disabled then $\mathcal{G}$ will be the *least program analysis graph*, as the *lfp* of each of the individual modules was computed.

The following lemma shows that if a module M is analyzed for E assuming some $\mathcal{G}$ obtaining $\mathscr{L}_M$, then if the assumptions change to $\mathcal{G}'$, incrementally updating these assumptions will produce an analysis graph $\mathscr{L}'_M$ that is the same as analyzing M with assumptions $\mathcal{G}$ from scratch. That is, the least analysis graph for module M.

Lemma 5.4 (Correctness and precision updating $\mathscr{L}$ modulo $\mathcal{G}$)**.** *Let M be a module contained in program P, E a set of entries. Let $\mathcal{G}$ be a previous state of the global analysis graph, if $\mathscr{L}_M = \textsc{LocIncAnalyzeI95}(M, E, \mathcal{G}, \emptyset, \emptyset)$. If $\mathcal{G}$ changes to $\mathcal{G}'$ the analysis result:*

$$\textsc{LocIncAnalyzeI95}(M, E, \mathcal{G}', \mathscr{L}_M, \emptyset) = \textsc{LocIncAnalyzeI95}(M, E, \mathcal{G}', \emptyset, \emptyset)$$

is the same as analyzing from scratch, i.e., the least correct analysis graph of M, E.

Proof. The proof of this lemma follows from the proof of Lemma 5.1 and the guarantee that $\mathscr{L}_M$ is the least analysis graph when using IncAnalyze (Theorem 4.12). □

Proposition 5.5 (Correctness and precision of ModIncAnalyzeI95 from scratch)**.** *Let P be a modular program and Q_α a set of abstract queries. The analysis result*

$$\mathscr{A} = \textsc{ModIncAnalyzeI95}(P, Q_\alpha, \emptyset, \emptyset) = \textsc{ModAnalyzeI95}(P, Q_\alpha)$$

such that $\mathscr{A} = \{\mathcal{G}, \{\mathscr{L}_{M_i}\}\}$, then $\mathcal{G} = \mathcal{G}'$.

Proof. Since the *lfp* is monotonic w.r.t. the initial assumptions and upd is monotonic, *MA* is monotonic. Therefore, chaotic iteration of *MA* with the different modules of a program will reach a fixpoint which is the least analysis graph, because the separated *lfp* of each of the modules is computed. Chaotic iteration is guaranteed in the same way as correctness in Prop. 5.2. Termination is guaranteed because *MA* is monotonic and D_α is finite. □

If P is changed to P' by edits Δ and it is reanalyzed incrementally, the algorithm will return a $\mathcal{G}$ that encodes the same global analysis result as if P' is analyzed from scratch, i.e., the least program analysis graph.

Theorem 5.6 (Correctness and precision of ModIncAnalyzeI95)**.** *Let P and P' be modular programs that differ by Δ, Q_α a set of abstract queries, and $\mathscr{A} = \textsc{ModIncAnalyzeI95}(P, Q_\alpha, \emptyset, (\emptyset, \emptyset))$, then*

$$\textsc{ModIncAnalyzeI95}(P', Q_\alpha, \emptyset, (\emptyset, \emptyset)) = \textsc{ModIncAnalyzeI95}(P', Q_\alpha, \mathscr{A}, \Delta).$$

Proof. This is proved by following the same strategy as in Theorem 5.3, replacing the termination condition that relied on the widening operator with the guarantees that the abstract domain is finite and that *MA* is monotonic, and the guarantee of Lemma 5.4 that no imprecision is introduced analyzing each individual module. □

5.5 Analyzers amenable to incrementalizing

Even though we proposed a modular algorithm that reuses [86], the monolithic analyzer presented so far (represented by IncAnalyze) can be replaced by another one that meets the following conditions:

1. **Conditions on the analysis result representation.** There is a *relation between program points and analysis results*. This is vital to be able to partially reuse or discard information. This condition is met by the proposed analysis graph, as analysis results are related with program point via predicate names. Namely, we can assign any node of the analysis graph to the predicate and literals to which it corresponds.

2. **Conditions on the analysis output.** There is a *partial order* defined over the results of the analysis. This is necessary to detect convergence, and which parts need to be recomputed.

3. **Conditions on the analysis input.** The analyzer has as optional input *initial guesses* ($\mathscr{A}_0$) of the results. This serves two purposes, first, since our algorithm analyzes modules independently, this is a means for providing results for code that is external to the module (not analyzable at that point). And, second, to provide the analyzer with already computed results, thus avoiding unnecessary recomputation.

5.6 Related work

Modular analysis [39] is based on splitting large programs into smaller parts (e.g., based on the source code structure). Exploiting modularity has proved essential in industrial-scale analyzers [41, 55]. Despite the fact that separate analysis provides only coarse-grained incrementality, there have been surprisingly few results studying its combination with fine-grained incremental analysis.

Classical data-flow analysis. Since the first algorithm for incremental analysis was proposed in [156], there has been considerable research and proposals in this topic (see the bibliography of [150]). Depending on how data flow equations are solved, these algorithms can be separated into those based on variable elimination, which include [26], [28], and [158]; and those based on iteration methods which include [32] and [142]. A hybrid approach is described in [119]. Our algorithms are most closely related to those using iteration. Early incremental approaches such as [32] were based on *restarting iteration*. That is, the fixpoint of the new program's data flow

equations is found by starting iteration from the fixpoint of the old program's data flow equations. This is always safe, but may lead to unnecessary imprecision if the old fixpoint is not below the *lfp* of the new equations [159]. *Reinitialization approaches* such as [142] improve the accuracy of this technique by reinitializing nodes in the data flow graph to bottom if they are potentially affected by the program change. Thus, they are as precise as if the new equations had been analyzed from scratch. These algorithms are generally not based on abstract interpretation. REVISER [8] extends the more generic IFDS [152] framework to support incremental program changes. However IFDS is limited to distributive flow functions (related to *condensing* domains) while our approach does not impose any restriction on the domains.

Constraint Logic Programs. Apart from the work that we extend [86, 147], incremental analysis was proposed (just for incremental addition) in the Vienna abstract machine model [102, 103]. It was studied also in compositional analysis of modules in (constraint) logic programs [30, 17], but it did not consider incremental analysis at the level of clauses.

Datalog and tabled logic programming. In a related line to the previous one, other approaches are based on datalog and tabled logic programming. FLIX [117] uses a bottom-up semi-naive strategy to solve Datalog programs extended with lattices and monotone transfer functions. This approach is similar to CLP analysis via bottom-up abstract interpretation. However it has not been extended to support incremental updates. Incremental tabling [167] offers a straightforward method to design incremental analyses [54], when they can be expressed as tabled logic programs. While these methods are much closer to our incremental algorithm, they may suffer similar problems to generic incremental computation, as they may be difficult to control.

Generic incremental computation frameworks. Obviously, the possibility exists of using a general incrementalized execution algorithm. Incremental algorithms compute an updated output from a previous output and a difference on the input data, with the hope that the process is (computationally) cheaper than computing from scratch a new output for the new input. The approach of [169] takes advantage of an underlying incremental evaluator, IncQuery, and implements modules via the monolithic approach. There exist other frameworks for incremental computation [111, 177, 110, 2, 109], which greatly simplify writing incremental algorithms, but in return it is difficult to control the costs of the additional data structures.

This concludes the chapter. An experimental evaluation of this algorithm, and also the algorithms in Chapter 4, is presented in Chapter 8.

6

Assertion-guided Analysis

After developing a combined incremental and modular analysis, in order to address scalability and response-time concerns, we now turn our attention to the issue of precision. Approximations during program analysis are a necessary evil, as they ensure essential properties, such as soundness, termination, and performance, but they also imply not always producing useful results. If approximations are not carefully designed, the information reported by the analyzer may not be accurate enough for the intended application, such as, performing optimizations or verifying properties. Much work has been done towards improving both the accuracy and efficiency of analyzers through the design of automatic analysis techniques that include clever abstract domains, widening and narrowing techniques [11, 173, 34, 176], and sophisticated fixpoint algorithms [18, 40, 120, 126, 147, 96, 4, 8, 169]. Despite these advances, the impossibility results shown in Chapter 3 show that there will be always cases where it is necessary for the user to provide input to the analyzer to guide the process in order to regain accuracy, prevent imprecision from propagating, and improve analyzer performance [23, 50].

In this chapter we focus on techniques that provide a means for the programmer to be able to optionally annotate program parts in which precision needs to be recovered. Examples are the *entry* declarations and *trust* assertions (see Sec. 2.5.2) of CiaoPP [23, 144] and the *known facts* of Astrée [40, 50]. Such user annotations help dealing with program constructs for which the analysis is not complete or the source is only partially available. A number of additional analyzer-related roles for assertions, beyond the usual of providing program specifications, were proposed as part of the Ciao assertion language design [82, 144]. These included, in addition to guiding the analysis as mentioned before, serving as a means for expressing the analysis output in a user-friendly way or for example for communication of analysis results between modules during modular analysis. Surprisingly, there is little information in the literature on these assertions beyond a sentence or two in the user manuals or some examples of use in demo sessions. In particular, no precise descriptions exist on how these assertions affect the fixpoint computation process and its results.

We propose a user-guided multivariant fixpoint algorithm that makes use of information contained in different kinds of assertions, and provide formal results on the influence of such assertions on the analysis. We also extend the semantics of the assertions to control if precision can be relaxed, and also to deal with both the cases in which the program execution will and will not incorporate run-time tests for unverified assertions. Note that almost all current abstract interpretation systems assume in their semantics that the run-time checks will be run. In languages like C, this is to ensure that the semantics is well defined. Soundness is then that the analysis is valid at runtime up to the first time the semantics is undefined (whether there is a runtime check or not), so the analysis can assume that the test is passed. However, due to efficiency considerations, assertion checking is often turned off in production code, specially for complex properties [101].

6.1 Run-time semantics of assertions

As stated in Sec. 2.5.2, assertions provide a means for the programmer to be able to optionally annotate program parts in which precision needs to be recovered. Such user annotations allow dealing with program constructs for which the analysis is not complete or the source is only partially available. Most systems make assumptions during analysis with respect to the run-time semantics of assertions. For example, Astrée assumes that they are always taken into account, while CiaoPP assumes conservatively that they may not be (because in general they may in fact be disabled by the user, e.g., in production code).

In the Ciao system, the run-time behavior intended by the programmer is given by qualifying the assertion with a status. For `trust` assertions, their information is used by the analyzer but they are never checked at run time. For `check` assertions (provided that they have not been discharged statically) run-time checks must always be performed. When an assertion is checked, the execution will not pass beyond that point if the conditions are not met.[1] This means that `check` assertions can also be "*trusted*," in a similar way to `trust` assertions, because execution only proceeds beyond them if they hold. For some interesting usages of assertions we refer the reader to Sec. 2.5.3.

Recall that the calls and success conditions in the assertions are sets of property literals. The following definitions (adapted from [145]) are instrumental to correctly approximate the properties of the assertions. The following shows the set of calls for which a property formula trivially succeeds.

1 This strict run-time semantics for `check` assertions is described in [166].

Definition 6.1 (Trivial Success Set of a Property Formula)**.** Given a conjunction L of property literals and the definitions for each of these properties in P, we define the *trivial success set* of L in P as:

$$TS(L,P) = \{\theta | vars(L) \ s.t. \ \exists \theta' \in \mathsf{answers}(P, \{\langle L, \theta \rangle\}), \theta \models \theta'\}$$

where $\theta|vars(L)$ above denotes the projection of θ onto the variables of L, and $\models$ denotes that θ' is a more general constraint than θ (entailment). Intuitively, $TS(L,P)$ is the set of constraints θ for which the literal L succeeds without adding new constraints to θ (i.e., without constraining it further). For example, given the following program P:

```
1 list([]).
2 list([_|T]) :- list(T).
```

and L is $\texttt{list}(X)$, both $\theta_1 = \{X = [1,2]\}$ and $\theta_2 = \{X = [1,A]\}$ are in the trivial success set of L in P, since calling $(X = [1,2], \texttt{list}(X))$ returns $X = [1,2]$ and calling $(X = [1,A], \texttt{list}(X))$ returns $X = [1,A]$. However, $\theta_3 = \{X = [1|\_]\}$ is not, since a call to $(X = [1|Y], \texttt{list}(X))$ would further constrain the term $[1|Y]$, returning $X = [1|Y], Y = []$.

6.2 Abstract semantics of assertions.

We recall abstract counterparts for Def. 6.1, as proposed in [145, 140]. These abstractions come useful when the properties expressed in the assertions cannot be represented exactly in the abstract domain.

Definition 6.2 (Abstract Trivial Success Subset of a Property Formula)**.** Under the same conditions of Def. 6.1, given an abstract domain D_α, $\lambda^-_{TS(L,P)} \in D_\alpha$ is an *abstract trivial success subset* of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L,P)$.

Definition 6.3 (Abstract Trivial Success Superset of a Property Formula)**.** Under the same conditions of Def. 6.2, an abstract substitution $\lambda^+_{TS(L,P)}$ is an *abstract trivial success superset* of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L,P)$.

That is, $\lambda^-_{TS(L,P)}$ and $\lambda^+_{TS(L,P)}$ are, respectively, safe under- and over-approximations of $TS(L,P)$. Note that they are always computable by choosing the closest element, if it exists, or otherwise a close element in the abstract domain. At the limit $\bot$ is a trivial success subset of any property formula and $\top$ is a trivial success superset of any property formula. In the following, when P is a fixed program, given a property d from an assertion condition (*Pre* or *Post*), let $\lambda^-_d = \lambda^-_{TS(d,P)}, \lambda^+_d = \lambda^+_{TS(d,P)}$ be the abstract values for the under- and over-approximations of the property d.

Algorithm 5 GuidedIncAnalyze: monolithic, context-sensitive, incremental fixpoint algorithm using (not changing) assertion conditions.

global flag: `speed-up`

GuidedIncAnalyze $(P, Q_\alpha, \Delta, \mathscr{A})$

1: **for all** $\langle A, \lambda^c\rangle \in Q_\alpha$ **do**
2: add-event$(newcall(\langle A, \lambda^c\rangle))$
3: `deleteClauses`(Δ)
4: `addClauses`(Δ)
5: **while** events$() \neq \emptyset$ **do**
6: $E :=$ next-event$()$
7: `process`(E)
8: `removeUnreachable`$(\mathscr{A}, Q_\alpha)$
9: **return** $\mathscr{A}$

func `applyCall`(A, λ^c)

10: **if** $\exists\sigma.\lambda^t = \lambda^+_{Pre\ \sigma} \wedge$ calls$(H, Pre) \in P \wedge H\sigma = A$ **then**
11: **if** `speed-up` **then return** λ^t
12: **else return** $\lambda^c \sqcap \lambda^t$
13: **else return** λ^c

func `applySucc`$(A, \lambda^c, \lambda^s_0)$

14: $app = \{\lambda \mid \exists\ \sigma.$success$(H, Pre, Post) \in P \wedge H\sigma = A,\ \ \lambda = \lambda^+_{Post\ \sigma} \wedge \lambda^-_{Pre\ \sigma} \sqsupseteq \lambda^c\}$
15: **if** $app \neq \emptyset$ **then**
16: $\lambda^t = \bigsqcap app$
17: **if** `speed-up` **then**
18: **return** λ^t
19: **else return** $\lambda^t \sqcap \lambda^s_0$
20: **else return** λ^s_0

proc `process`$(arc(\langle A, \lambda^c_0\rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda^c_1\rangle))$

21: $Calls :=\ \{\lambda \mid \langle A, \_\rangle \rightarrow_{k,i} \langle B, \lambda\rangle \in \mathscr{A}\}$
22: $\lambda^c :=$ `Ageneralize`$(\lambda^c_1, Calls)$
23: $\lambda^a :=$ `applyCall`(B, λ^c, As)
24: **if** B is a *built-in* **then**
25: $\lambda^s_0 :=\ f^\alpha(\langle B, \lambda^a\rangle)$
26: **else**
27: $\lambda^s_0 :=$ `lookupAnswer`$(\langle B, \lambda^a\rangle)$
28: `upd`$(\mathscr{A}, \langle A, \lambda^c_0\rangle \xrightarrow{\lambda^p}_{k,i} \langle B, \lambda^a\rangle)$
29: $\lambda^r :=$ `Aextend`(λ^p, λ^s_0)
30: **if** $\lambda^r \neq \bot$ **and** $i \neq n_k$ **then**
31: $\lambda^c_2 :=$ `Aproj`$(\lambda^r, vars(A_{k,i+1}))$
32: add-event$(arc(\langle H, \lambda^c_0\rangle \xrightarrow{\lambda^r}_{k,i+1} \langle B, \lambda^c_2\rangle))$
33: **else if** $i = n_k$ **then**
34: $\lambda^s_k :=$ `Aproj`$(\lambda^r, vars(A_k))$
35: $\lambda^s :=$ `Aproceed`(A, λ^s_k, A_k)
36: `insertAnswerInfo`$(\langle A, \lambda^c_0\rangle, \lambda^s)$

proc `insertAnswerInfo`$(\langle A, \lambda^c\rangle, \lambda^s)$

37: **if** $\langle A, \lambda^c\rangle \mapsto \lambda^s_0 \in \mathscr{A}$ **then**
38: $\lambda^s_1 :=$ `Ageneralize`$(\lambda^a, \{\lambda^s_0\})$
39: **else** $\lambda^s_0 :=\ \bot,\ \lambda^s_1 :=\ \lambda^s$
40: $\lambda^a :=$ `applySucc`$(A, \lambda^c, \lambda^s_1, As)$
41: `upd`$(\mathscr{A}, \langle A, \lambda^c\rangle \mapsto \lambda^a)$
42: **if** $\lambda^s_0 \neq \lambda^a$ **then**
43: `reanalyzeUpdated`$(\langle A, \lambda^c\rangle)$

6.3 Operation of the algorithm

Algorithm 5 presents GuidedIncAnalyze, an extension of IncAnalyze (Algorithm 1) to apply assertions during analysis. The omitted procedures do not require any modification, i.e., procedures `addClause`, `deleteClauses`, `lookupAnswer`, `reanalyzeUpdated`, and `process`$(newcall(\langle A, \lambda^c\rangle))$ are as described in Algorithm 1. The algorithm shows in **orange** the operations and abstract values that comprise the extension. Assertions can be used to either refine the analysis results or to try to speed up convergence of the fixpoint. This is controlled by the boolean global flag `speed-up`.

The initialization and processing of events (lines 1-9) also remain as in Algorithm 1. Call conditions are used in line 23, that is, whenever a call to a literal is prepared to be abstractly executed. The assertion condition is used after generalization so that

some precision is regained after possibly performing a widening (line 22). The refined abstract call is later used to either compute the success of a primitive constraint, if the literal is a *built-in*, or to look for the answer if it is a predicate call. Last, a new edge and node are added in the graph for the call, if it did not exist (line 28). Success conditions are used in line 40, that is, whenever the success to an abstract call is computed. The refinement is done after generalization so that some precision is recovered after widening (line 38). The refined success is later included in the analysis graph and used to update the results of all literals that are calls of the form being processed (line 41).

Functions `applyCall`(A, λ^c, As) and `applySucc`$(A, \lambda^c, \lambda^s, As)$ abstract the meaning of the calls and success assertion conditions. The implementations of the operations of Definitions 6.2 and 6.3 are assumed to be monotonic. Note that if the properties can be expressed in the abstract domain, no over or under-approximations are needed.

Safely including calls conditions. In `applyCall`(A, λ^c, As), if there is a call condition for A (line 10), an over-approximation is performed. Depending on the `speed-up` flag, this is used directly as an over-approximation of the call (line 11) or to refine the inferred call (line 12). If no conditions are available, the original λ^c is returned (line 13).

Safely including success conditions. In `applySucc`$(A, \lambda^c, \lambda^s, As)$, given an atom A, an abstract call λ^c and its corresponding abstract success λ^s, all success conditions whose *precondition applies* ($\lambda^c \sqsubseteq \lambda^-_{Pre}$) are collected in *app* (line 14). Making an under-approximation of *Pre* is necessary to only apply the conditions that can be ensured by the abstract domain to be in the inferred abstract call, i.e., only if it would be applied in the concrete executions of the program. An over-approximation of *Post* needs to be performed since otherwise success states that actually happen in the concrete execution of the program may be removed. If no conditions are applicable (i.e., *app* is empty), the original success is returned (line 20). Then, if the `speed-up` flag is true, this is used directly as an over-approximation of the success (line 18) or to refine the inferred call (line 19).

Finally, note that the existence of guidance assertions for a predicate does not save having to analyze the code of the corresponding predicate if it is available, since otherwise any calls generated within that predicate would be omitted and not analyzed for, resulting in an incorrect analysis result.

6.4 Fundamental properties of GuidedIncAnalyze

We claim the following properties for analysis of a program P applying assertions as described in the previous sections. The inferred abstract execution is covered by the calls and (applicable) success assertion conditions. This is, of course, limited by how well they can be represented in the abstract domain D_α chosen to analyze with. The following hold both when the value of the `speed-up` flag is true and when it is false.

Lemma 6.1 (Applied calls condition)**.** *Let P be a program with an assertion condition* calls(H, Pre)*, and Q_α a set of abstract queries. Let $\mathscr{A} =$* GuidedIncAnalyze$(P, Q_\alpha, \emptyset, \emptyset)$*. For any call $\langle A, \lambda^c\rangle \in \mathscr{A}$, if $A = H\sigma$ for some renaming σ then $\lambda^c \sqsubseteq \lambda^+_{Pre\ \sigma}$.*

Proof. Function `applyCall` obtains in λ^t the trusted value for the call. It restricts the encountered call λ^c or uses it as is, in any case $\lambda^c \sqsubseteq \lambda^t = \lambda^+_{Pre}$. Hence if this function is applied whenever inferred call patterns are introduced in the analysis results, the lemma holds.

Since, by hypothesis, the initial analysis graph is empty, the lemma holds. Let us reason about how the algorithm changes the results. Call nodes are only inserted in $\mathscr{A}$ at line 28 (adding an edge includes adding the corresponding nodes if not present already). The procedure adds nodes to the analysis whenever new calls are found, it is called right after `applyCall`, and therefore it only inserts call patterns taking into account calls conditions, and all calls are added taking into account the call conditions. □

Lemma 6.2 (Applied success condition)**.** *Let P be a program with an assertion condition* success$(H, Pre, Post)$*, and Q_α a set of abstract queries. Let $\mathscr{A} =$* GuidedIncAnalyze$(P, Q_\alpha, \emptyset, \emptyset)$*. For any $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$, if $A = H\sigma$ for some renaming σ then $\lambda^c \sqsubseteq \lambda^-_{Pre\ \sigma} \Rightarrow \lambda^s \sqsubseteq \lambda^+_{Post\ \sigma}$.*

Proof. Function `applySucc` computes the $\sqcup$ of all applicable assertion conditions (checking $\lambda^c \sqsubseteq \lambda^-_{Pre}$), if existing. Since the $\sqcup$ of all applied conditions is obtained, $\lambda^s \sqsubseteq \bigsqcap \lambda^+_{Post_i} \sqsubseteq \lambda^+_{Post}$ for any $Post$. Hence if all results inserted in the analysis result have been previously processed by `applySucc`, the lemma holds. Since, by hypothesis, the initial analysis graph is empty, the lemma holds. In the rest of the algorithm, answers are only updated in $\mathscr{A}$ at line 41. □

Correctness in a guided analysis Lemmas 6.1 and 6.2 guarantee that the analysis indeed uses the information in the assertions to improve the precision. However, to guarantee analysis correctness, the conditions must correctly describe the behavior of the program. The following definitions formalize this.

Definition 6.4 (Correct calls condition)**.** Let P be a program with an assertion condition $C = \mathsf{calls}(H, Pre)$. C is correct for a query Q to P if for any predicate A, s.t. $A = H\sigma$ for some renaming σ:

$$\forall \theta^c \in \mathsf{calling\_context}(A, P, Q).\theta^c \in \gamma(\lambda^+_{Pre\ \sigma}).$$

Definition 6.5 (Correct success condition)**.** Let P be a program with an assertion condition $C = \mathsf{success}(H, Pre, Post)$. C is correct for P if for any predicate A, s.t. $A = H\sigma$ for some renaming σ:

$$\theta^c \in \gamma(\lambda^-_{Pre\ \sigma}) \wedge \theta^s \in \mathsf{answers}(P, \{\langle A, \theta^c \rangle\}) \Rightarrow \theta^s \in \gamma(\lambda^+_{Post\ \sigma}).$$

If all assertion conditions of a program are correct, then the analysis results correctly over-approximate all concrete executions.

Theorem 6.3 (Correctness of GUIDEDINCANALYZE from scratch)**.** *Let P be a program with correct assertion conditions and Q_α a set of abstract queries. The analysis result $\mathscr{A}$ = GUIDEDINCANALYZE$(P, Q_\alpha, \emptyset, \emptyset)$ is correct (Def. 2.5) for P and $\gamma(Q_\alpha)$.*

Proof. If there are no assertion conditions, the theorem trivially holds (Lemma 4.6). If assertion conditions are used to generalize, the theorem also holds because $\lambda^c = \lambda^+_{Pre}$ and $\lambda^s = \lambda^+_{Post}$ are by definition (Definitions 6.4 and 6.5, respectively) correct over-approximations. If then assertion conditions are used to regain precision then all the conditions of Def. 2.5 are satisfied:

- (correct calls): following the argument of Lemma 6.1, the only point in the analysis in which possible concrete calls are *pruned* from the analysis result is in line 28. Since an over approximation of the concrete property is used, and $\sqcap$ is assumed to correctly over-approximate the concrete operation, by hypothesis (the call assertion conditions are correct) no actual behaviors are removed from the analysis result by this process, and correctness follows from Lemma 4.6.
- (correct successes): following the argument of Lemma 6.2, as before, the only point in the analysis in which possible concrete successes are *pruned* from the analysis result is in line 41. The function `applySucc` conservatively applies the success assertions whose *Pre* is representable in the abstract domain, and with an over-approximation of *Post*. Again, by hypothesis, (the success assertion conditions are correct) no actual behaviors are removed from the analysis result by this process, and correctness follows from Lemma 4.6.
- (correct dependencies) the extensions of Algorithm 5 do not affect how call dependencies among nodes of the analysis graph are computed, only the

precision of the calls. Therefore correctness follows from Theorem 4.9 (for Algorithm 1).

□

In other words, Theorem 6.3 and Lemmas 6.1 and 6.2 ensure that correct assertion conditions bound imprecision in the result, without affecting correctness. By applying the assertion conditions no actual concrete states are removed from the abstractions. These are actually independent from the mechanisms of the algorithm to incrementally recompute the analysis results.

Theorem 6.4 (Correctness of GuidedIncAnalyze)**.** *Let P and P' be programs that differ by clauses Δ, with correct assertion conditions, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{GuidedIncAnalyze}(P', Q_\alpha, \emptyset, \emptyset)$, then the analysis result $\mathscr{A} = \text{GuidedIncAnalyze}(P, Q_\alpha, \Delta, \mathscr{A}_0)$ is correct for P and $\gamma(Q_\alpha)$ (Def. 2.5).*

Proof. Since all operations used during fixpoint computation are assumed to be monotonic (to guarantee termination), and applying correct assertion conditions preserves that approximations of concrete executions are safe by Theorem 6.3, correctness then follows from Theorem 4.9. □

Detecting potentially incorrect trusted conditions. Although checking that the conditions of Definitions 6.4 and 6.5 hold is not computable in general, the fixpoint algorithm can be instrumented with checks at the points where assertion semantics are incorporated in the results, i.e., when calling `applyCall` or `applySucc`. At that point, let λ^a be the correct approximation of a condition and λ be an inferred abstract state. If $\lambda \sqcap \lambda^a = \bot$ the inferred information is incompatible with that in the condition, therefore it is likely that the assertion is *erroneous* or that the program has a bug. If $\lambda \not\sqsubseteq \lambda^a$ then the algorithm inferred more concrete execution states than described in the assertion and the analysis results may also be wrong. In both cases it is trivial to instrument the algorithm to warn the user when the situations appear.

At the same time, it is possible to perform program analysis without taking into account assertions (using IncAnalyze), and then check the assertion conditions offline, by comparing their properties against the results in $\mathscr{A}$. A full description of this checking procedure is described in [145, 166], and in Chapter 9.

6.5 Related work

The inference of arbitrary semantic properties of programs is known to be both undecidable and potentially very expensive. Thus, most realistic and industrial implementations require user interaction. That interaction comes in two forms: parameters

or user annotations. In **static verification and theorem proving approaches**, for classical examples such as type checking in statically typed languages, as types become more expressive and closer to System-F, annotations become mandatory. In theorem provers such as ACL2 the user is required to complete the proofs by identifying necessary lemmas. Additionally in Coq and Isabelle/HOL, it is possible to guide the proof search via *tactics* (in their own domain-specific language). These approaches often require advanced knowledge of the type or proof system. Finally, verifiers that use SMT-based techniques are guided by the assertions to be proven (e.g., as *assert* statements in C programs). This means that they take the assumption that run-time checks always run. Abstract interpreters allow the **selection of different domains and parameters for such domains** (e.g., polyhedra, octagons, regtypes with depth-k, etc.), as well as their widening operations (e.g., type shortening, structural widening, etc.). Other parameters include policies for partial evaluation and other transformations (loop unrolling, inlining, slicing, etc.). These parameters are orthogonal or complementary to the issues discussed here. To the extent of our knowledge the use of **program-level annotations** (such as assertions) to guide abstract interpretation has not been widely studied in the literature, contrary to their (necessary) use in verification and theorem proving approaches. The **Cibai** [113] system includes *trust-style* annotations, while sources are processed to encode some predefined runtime semantics. In [72] the analysis is guided by modifying the analyzed program to restrict some of its behaviors. However, this guidance affects the *order of program state exploration*, rather the analysis results, as in our case. As mentioned in the introduction, the closest to our approach is **Astrée**, that allows *assert*-like statements, where correctness of the analysis is ensured by the presence of compulsory runtime checks, and trusted (*known facts*) asserts. These refine and guide analysis operations at program points. Like in CiaoPP, the analyzer shows errors if a known fact can be falsified statically. However, as with the corresponding Ciao assertions, while there has been some examples of use [50], there has been no detailed description of how such assertions are handled in the fixpoint algorithm.

7

Incremental analysis of programs with (changing) assertions

As mentioned in the thesis introduction, *generic* components are a further abstraction over the concept of modules, introducing dependencies on other (not necessarily available) components implementing specified interfaces. They have become a key concept in large and complex software applications and in modern coding it is rarely necessary to write everything from scratch. *Interfaces* are needed to connect software components with each other. They typically consist in a set of specifications, which are used as a place holder for (possibly different) implementations meeting such requirements.

However, despite its undeniable advantages, generic code is known to be in fact *anti-modular*, and its analysis poses several challenges: parts of the code are unavailable, and the interface specifications may not be descriptive enough to allow verifying the specifications for the whole application. Several approaches are possible in order to balance separate compilation with precise analysis and optimization. First, it is possible to analyze generic code by *trusting* its interface specifications, i.e., analyzing the client code and the interface implementations independently, flattening the analysis information inferred at the boundaries to that of the interface descriptions. This technique can reduce global analysis cost significantly at the expense of some loss of precision. Some of it may be regained by, e.g., enriching specifications manually for the application at hand. Alternatively, for a closed set of interface implementations, it may be desirable to analyze the whole application together with these implementations, keeping different specialized versions of the analysis across the interfaces. This allows getting the most precise information, specializations, compiler optimizations, etc., but at a higher cost. Instantiating the code with concrete implementations potentially leads to a prohibitive number of possible combinations of implementations, as seen when compiling C++ templates [172].

Generic code offers many opportunities for the application of the new analysis techniques proposed in this thesis. For example: standalone analysis of trait-based

code without particular implementations by using the (trust) assertions in the interfaces; refinement of standalone analysis for particular implementations; or reuse of analysis results when more implementations are made available. Note that fine-grained incremental analysis seems even more interesting when using generic code, where the scope of a program change may implicitly be scattered across many modules, as described in the `traits` for logic programming proposed in Sec. 2.5.5.

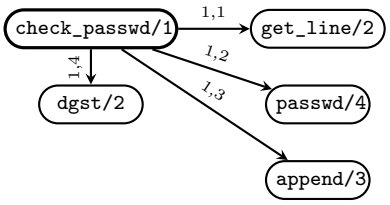
However note that reacting to assertion changes is also useful in more applications using abstract interpretation. For example, when using it interactively as a theory for verification; in the setting of frequent changes in the specification, e.g., when the application of the analysis is inferring energy consumption of a program and the physical models vary; or when the code depends on some libraries being developed.

7.1 Motivating examples

In the following examples we assume that we analyze with a shape domain in which the properties in the assertions can be exactly represented. Also, for simplicity, we only represent the nodes in the graph and not their abstract calls and successes.

Example 7.1 (Reusing a pre-analyzed generic program)**.** Consider a slightly modified version of the program that checks a password, as shown in Sec. 2.5.5, but only allows the user to write passwords with *lowercase* letters. Until we have a concrete implementation for the hasher it is not possible to analyze precisely this program. However, we can pre-analyze it by using the information in the assertion in the trait to obtain the following simplified analysis graph:

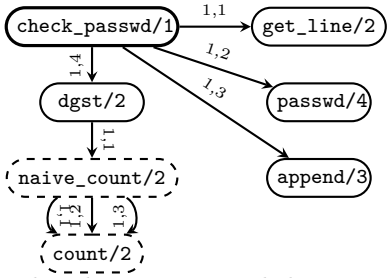
```
1  :- trait hasher { :- pred dgst(Str, Digest)
2           : lowercase(Str) => int(Digest). }.
3
4  check_passwd(User) :-
5      get_line(Plain),
6      passwd(User,Hasher,Digest,Salt),
7      append(Plain,Salt,Salted),
8      (Hasher as hasher).dgst(Salted,Digest).
9
10 passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```



The node for `dgst/2` represents the call $\langle$`dgst(S,D)`$, (S/\texttt{lowercase}, D/\texttt{num})\rangle \mapsto (S/\texttt{lowercase}, D/\texttt{int})$. In this case `D` was inferred to be a *number* because of the success of `passwd/4`. If we add a very naive implementation that consists in counting the number of times some letters appear in the password, reanalyzing causes the addition to the graph of some new nodes, which are shown with dashed borders:

```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



We detect that none of the previous nodes need to be recomputed due to tracking dependencies for each literal. The analysis was performed by going directly to the program point of `dgst/2` and inspecting the new clause (that was generated automatically by the translation) that calls `naive_count/2`. By analyzing `naive_count/2` we obtain nodes $\langle$`naive_count(S,D)`$, (S/\texttt{lowercase}, D/\texttt{num})\rangle \mapsto (S/\texttt{lowercase}, D/\texttt{int})$, and $\langle$`count(L,C,N)`$, (S/\texttt{lowercase}, C/\texttt{char})\rangle \mapsto (S/\texttt{lowercase}, C/\texttt{char}, N/\texttt{int})$. As no information needs to be propagated, because the head does not contain any of the variables of the call to digest, we are done, and we avoided reanalyzing any caller to `check_passwd/2`, if any existed.

Example 7.2 (Weakening assertion properties)**.** Consider the program and analysis result of Example 7.1. We realize that allowing the user to write only passwords with lowercase letters is not very secure. We can change the assertion of the trait to allow any string as a valid password.

```
:- trait hasher { :- pred dgst(Str, Digest) : string(Str) => int(Digest). }.
```

When reanalyzing, node $\langle$`dgst(S,D)`$, (S/\texttt{lowercase}, D/\texttt{num})\rangle$ becomes $\langle$`dgst(S,D)`$, (S/\texttt{string}, D/\texttt{num})\rangle$, and the same for `naive_count/3`. A new call pattern appears for `count/3` $\langle$`count(L,C,N)`$, (S/\texttt{string}, C/\texttt{char})\rangle \mapsto (S/\texttt{string}, C/\texttt{char}, N/\texttt{int})$, leading to the same result for `dgst/2`, i.e., we only had to partially analyze the library, instead of the whole program.

Our objective in this chapter it to modify GUIDEDINCANALYZE (Algorithm 5) so that a call to it produces results that are correct and precise after assertion changes, by inspecting and updating appropriately the analysis graph. We call this new analyzer GIAWAC, short for GUIDEDINCANALYZE-W/ASSERTIONCHANGES (Algorithm 6). We also reuse `applyCall` and `applySucc` from Algorithm 5. Additionally, we extend the annotations of the edges of the analysis graphs to store, for efficiency, the substitution *upon return* of calls in literals, namely λ^r, and the resulting edge is: $\langle A, \lambda\rangle \xrightarrow[\lambda^r]{\lambda^p}_{k,n} \langle B, \lambda^c\rangle$. Moreover, λ^r can always be constructed by obtaining the answer λ^s of $\langle B, \lambda^c\rangle$ via the mapping function in the analysis result and performing $\lambda^s_l = \texttt{Aproceed}(B, \lambda^s, A_{k,n})$ and $\lambda^r = \texttt{Aextend}(A_{k,n}, \lambda^p, \lambda^s_l)$.

Algorithm 6 GIAwAC: Incremental analysis of programs with assertions.

GIAwAC $(P = (Cls, As), \Delta_{Cls}, \Delta_{As}, Q_\alpha, \mathscr{A})$

1: $R := \emptyset$
2: **for each** predicate A in P **do**
3: **if** $\Delta_{As}[P] \neq \emptyset$ **then**
4: `updCallsPred`$(A, As, \mathscr{A}, R)$
5: `updSuccsPred`$(A, As, \mathscr{A}, R)$
6: $R := Q_\alpha \cup R$
7: $\mathscr{A} :=$ GuidedIncAnalyze$(P, R, \Delta_{Cls}, \mathscr{A})$
8: `removeUnreachable`$(\mathscr{A}, Q_\alpha)$
9: **return** $\mathscr{A}$

func `treatChange`$(\langle A, \lambda\rangle \xrightarrow[\lambda^r]{\lambda^p} {}_{k,n}\langle B, \lambda^c\rangle, \lambda^s, \mathscr{A})$

10: $\lambda^r_{new} :=$ `Aextend`(λ^p, λ^s)
11: **if** $\lambda^r = \lambda^r_{new}$ **then return** $\emptyset$
12: del$(\mathscr{A}, \langle A, \lambda\rangle \rightarrow_{k,n} _)$
13: upd$(\mathscr{A}, \langle A, \lambda\rangle \xrightarrow[\lambda^r_{new}]{\lambda^p} {}_{k,n}\langle B, \lambda^c\rangle)$
14: **if** $\lambda^r \sqsubset \lambda^r_{new}$ **then return** $\{\langle A, \lambda\rangle\}$
15: **else if** $\lambda^r \not\sqsubseteq \lambda^r_{new}$ **then**
16: $Lits := \{e \mid e = (\langle A, \lambda\rangle \rightarrow_{k,i} \_) \in \mathscr{A} \wedge i > n\}$
17: $I := \{e \mid e \rightsquigarrow l \in \mathscr{A} \wedge l \in Lits\}$
18: del$(\mathscr{A}, I)$
19: **return** I

proc `updCallsPred`$(A, As, \mathscr{A}, Q)$

20: **for each** $\langle B, \lambda\rangle \xrightarrow[\lambda^r]{\lambda^p} {}_{k,l}\langle A, \lambda^c{}_{old}\rangle \in \mathscr{A}$ **do**
21: $\lambda^c := \texttt{Aproj}(\lambda^p, vars(B_{k,l}))$
22: $\lambda^c{}_{new} :=$ `applyCall`(A, λ^c, As)
23: **if** $\exists\langle A, \lambda^c{}_{new}\rangle \mapsto \lambda^s \in \mathscr{A}$ **then**
24: $\lambda^s{}_{new} := \lambda^s$
25: **else** $\lambda^s{}_{new} := \bot$
26: $Ch = \langle B, \lambda\rangle \xrightarrow[\lambda^r]{\lambda^p} {}_{k,l}\langle A, \lambda^c_{new}\rangle$
27: $Q := Q \cup$ `treatChange`$(Ch, \lambda^s{}_{new}, \mathscr{A})$

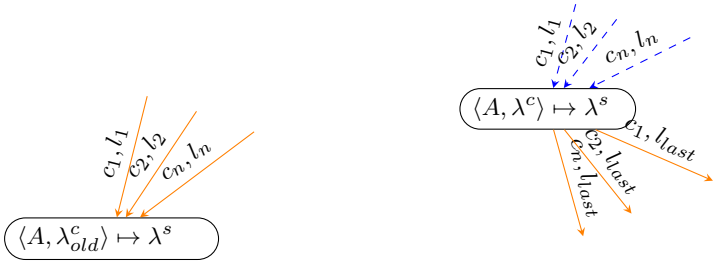
proc `updSuccsPred`$(A, As, \mathscr{A}, Q)$

28: **for each** $\langle A, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
29: $\lambda := \bot$
30: **for each** $\langle A, \lambda^c\rangle \xrightarrow[\lambda^r]{} {}_{k,last}\langle B, \lambda\rangle \in \mathscr{A}$ **do**
31: $\lambda^s := \texttt{Aproj}(\lambda^r, vars(A_k))$
32: $\lambda := \lambda \sqcup$ `applySucc`$(A, \lambda^c, \lambda^s, As)$
33: **for each** $e = (n \rightarrow \langle A, \lambda^c\rangle) \in \mathscr{A}$ **do**
34: $Q := Q \cup$ `treatChange` $(e, \lambda, \mathscr{A})$

7.2 Operation of the algorithm

Algorithm 6 starts with a preprocessing phase (lines 1-5) that inspects all the literals affected by the changes in the assertions. It updates the analysis graph, and obtains which calls need to be reanalyzed, and which successes need to be updated by the incremental analyzer. This is done by collecting additional abstract queries that need to be included together with the queries defined by the user to guarantee that the analysis is correct, taking advantage of Theorem 4.11.

Procedures `updCallsPred` and `updSuccsPred` compute, respectively, if changes in conditions affect the inferred calls or successes of predicates. In both cases, the overall idea is to obtain a substitution encoding the semantics of the previous version of the program if no assertions were present. Then functions `applyCall` and `applySuccess` obtain the abstraction with the meaning of the new assertions. Finally, `treatChange`, processes the potential changes.



(a) Edges affected by a calls(A, Pre) condition change.

(b) Edges affected by a success$(A, Pre, Post)$ condition change.

Fig. 16: Edges of nodes potentially affected by changes in assertion conditions.

In the case of calls conditions (`updCallsPred`), all the literals from which it is called are reviewed (line 20), by checking the incoming edges of the nodes of that predicate. For each of them the substitution of the clause (λ^p) is projected to the variables of the literal to obtain the call as if no assertions would be specified (line 21). Then, the new conditions are applied (line 22), and this is used to check if the call is already in the analysis graph, i.e., it was already analyzed (line 23). If so, the answer for that call is reused (line 24), if not, $\bot$ is assumed (line 25). Last, the new answer is treated by `treatChange`. Fig. 16a shows, intuitively, which edges need to be reviewed after a change in a calls condition, namely all incoming edges to a node that has a call to the predicate that the assertion describes.

In the case of success conditions (`updSuccsPred`), for each of the calls in the graph (line 28) the success needs to be recomputed. This is done by obtaining the success of each of the clauses defining the predicate, by using λ^r of the edges of the last literals (line 30), projecting them to the variables in the head of the clause (line 31), and finally applying the new assertions (line 32), and joining it with the meaning of the previous clauses. Then, each of the callers of this predicate (line 33) is treated (line 34). Fig. 16b gives an intuition of the affected edges. The orange edges contain the success substitution of each of the clauses defining the predicate, therefore they can be used to obtain the substitution without the assertion. Once this is computed, the changes in the success need to be propagated to the callers of the predicate, in the Figure these are the edges drawn in blue and dashed.

Amending the analysis results. The procedure `treatChange` processes an edge that points to a literal whose success potentially changed. It first recomputes the answer of that clause by extending the success to the analysis result (line 10). If the answer did not change, it returns an empty set as reanalysis is required for that edge (line 11).

If reanalysis is required, the edge is updated in the analysis graph (line 13). Then, if the new substitution (λ^r_{new}) is more general than the previous one (λ^r), the previous set of assertions were pruning more concrete states than the new one. Thus, there are potentially missing concrete executions and this call needs to be reanalyzed, and the query is returned (line 14). Else, if $\lambda^r \not\sqsubseteq \lambda^r_{new}$, i.e., the new abstract substitution is more concrete or incompatible, some parts of the analysis graph may not be accurate, these are collected in *Lits* (line 16). Therefore, they are removed from the graph, and returned for reanalysis (lines 17-19).

Lastly, `removeUnreachable` removes from the graph nodes that may have been included in R but were not actually necessary to produce a correct analysis for Q_α.

7.3 Correctness of GIAwAC

In the following let $(\mathscr{A}', R) = \textsc{PreProc}(P, \Delta_{As}, \mathscr{A})$ be the preprocessing phase of the algorithm (lines 1-5), with $\mathscr{A}'$ the analysis graph after updating the changes in the assertions and R the set of abstract queries, containing the user-defined queries and the additional ones to guarantee correctness of reanalysis.

Lemma 7.1 (Correctness of GIAwAC from scratch)**.** *Let P be a program with correct assertion conditions (Definitions 6.4 and 6.5), Q_α a set of abstract queries. The analysis $\mathscr{A} = \text{GIAwAC}(P, Q_\alpha, \emptyset, \emptyset, \emptyset)$ for P with Q_α is correct (Def. 2.5) for P and $\gamma(Q_\alpha)$.*

Proof. Since PreProc only modifies information that is already in the initial analysis and it is empty, correctness follows from Theorem 6.4. □

In terms of precision, we want to ensure that the meaning of the new assertions is precisely included in the analysis result. Recall that this is up to the point that the conditions can be effectively expressed in the abstract domain, since we are using Definitions 6.2 and 6.3.

Lemma 7.2 (Precision of calls after PreProc)**.** *Let P, P' be programs that differ by assertions Δ_{As} and Q_α a set of abstract queries. Let $\mathscr{A}_0 = \text{GIAwAC}(P, Q_\alpha, \emptyset, \emptyset, \emptyset)$ and $(\mathscr{A}', \_) = \textsc{PreProc}(P, \Delta_{As}, \mathscr{A}_0)$. Then:*

$$\forall \langle A, \lambda^c \rangle \in \mathscr{A}'. \exists\ \mathsf{calls}(A, Pre) \in P \Rightarrow \lambda^c \sqsubseteq \lambda^+_{Pre}.$$

Proof. First note that if the call depends on some callee whose abstraction is imprecise, after all predicates have been processed the node is removed, thus not being considered in this Lemma. For every other predicate A there are two possibilities.

1. The calls condition for this predicate did not change, in this case, nothing is done, and the lemma follows from Lemma 6.1.

2. If the condition changed, for every caller, the original call (i.e., without assertion) is recovered in line 21. Then, the new condition is applied by `applyCall`, λ^c_{new}, which, by definition, meets the conditions in the lemma. This is later inserted in the analysis graph by `treatChange` in line 13, after removing the node in line 12. Thus the Lemma holds.

□

Lemma 7.3 (Precision of successes after PREPROC)**.** *Under the conditions of Lemma 7.2. If* $(\mathscr{A}', \_) = \text{PREPROC}(P, \Delta_{As}, \mathscr{A}_0)$*, then*

$$\forall \langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}'. \forall\ \mathsf{success}(A, Pre, Post) \in P. \lambda^c \sqsubseteq \lambda^-_{Pre} \Rightarrow \lambda^s \sqsubseteq \lambda^+_{Post}$$

Proof. Note that, similarly to the proof of Lemma 7.2, if the success depends on some called predicate whose abstraction is imprecise, after all predicates have been processed the node is removed, thus not being considered in this Lemma. For every predicate A there are two possibilities.

1. The success condition for this predicate did not change, in this case, nothing is done, and the lemma follows from Lemma 6.2.

2. If the condition changed, for every caller, the new success is computed by joining the semantics of each of the clauses of the predicate in line 32. Then, in `treatChange`, in line 14 if the old success is greater than the new one, then nothing is done and the Lemma holds. Otherwise, the nodes are deleted from the graph and we would not be considering it for this Lemma.

□

As shown in Theorem 4.11, given a well-formed analysis graph, we can ensure correctness of the reanalysis if we guarantee that all calls that need to be reanalyzed are included in Q_α. We want to show that the set Q of queries collected in `treatChange` is enough to guarantee the correctness of the result.

Lemma 7.4 (Queries collected in PREPROC)**.** *Under the conditions of Lemma 7.2, i.e.,* $(\mathscr{A}', R) = \text{PREPROC}(P, \Delta_{As}, \mathscr{A}_0)$*, R guarantees that* $\mathscr{A} = \text{GUIDEDINCANALYZE}(P, R, \emptyset, \mathscr{A}')$ *is correct for P and $\gamma(Q_\alpha)$.*

Proof. We split the proof into two cases: (a) The assertions change only for one predicate: because $\mathscr{A}$ is correct, by Theorem 4.11, since $\mathscr{A}$ is an over-approximation of $[\![P]\!]_Q$, and Theorem 4.11 is true.

(b) The assertions change for more than one predicate: after processing the first predicate $\mathscr{A}$ may not be correct, as `treatChange` removes nodes. However, every node that is removed is added to the set of queries. This means that the nodes that are unreachable when processing the following predicates were already stored before, and therefore, Theorem 4.11 also holds. □

Theorem 7.5 summarizes all correctness and precision guarantees of the algorithm.

Theorem 7.5 (Correctness of GIAwAC)**.** *Let P and P' be programs with correct assertion conditions that differ by Δ_{Cls} and Δ_{As}, and Q_α a set of abstract queries. If $\mathscr{A}_0 = \text{GIAwAC}(P', Q_\alpha, \emptyset, \emptyset, \emptyset)$, then the analysis $\mathscr{A} = \text{GIAwAC}(P, Q_\alpha, \Delta_{Cls}, \Delta_{As}, \mathscr{A}_0)$:*

(a) is correct for P and $\gamma(Q_\alpha)$,

(b) $\forall \langle A, \lambda^c \rangle \in \mathscr{A}.\mathsf{calls}(A, Pre) \in P \Rightarrow \lambda^c \sqsubseteq \lambda^+_{Pre}$, and

(c) $\forall \langle A, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}.\forall\ \mathsf{success}(A, Pre, Post) \in P.\lambda^c \sqsubseteq \lambda^-_{Pre} \Rightarrow \lambda^s \sqsubseteq \lambda^+_{Post}$.

Proof. (a) follows from Lemmas 7.1, 4.1 and 7.4 and Theorem 6.4. (b) and (c) follow from Lemmas 6.1, 6.2, 7.2, and 7.3. □

7.4 Related work

Languages like C++ require specializing all parametric polymorphic code (e.g., templates [172]) to monomorphic variants. While this is more restrictive than *runtime polymorphism* (variants must be statically known at compile time), it solves the analysis precision problem, but not without additional costs. First, it is known to be slow, as templates must be instantiated, reanalyzed, and recompiled for each compilation unit. Second, it produces many duplicates which must be removed later by the linker. Rust [100] takes a similar approach for *unboxed* types.

Runtime polymorphism or dynamic dispatch can be used in C++ (virtual methods), Rust (boxed traits), Go [51] (interfaces), or Haskell's [88] type classes. However, in this case compilers and analyzers do not usually consider the particular instances, except when a single one can be deduced (e.g., in C++ devirtualization [131]).

Mora et al. [125] perform modular symbolic execution to prove that some (versions of) libraries are equivalent with respect to the same client. Chatterjee et al. [29] analyze libraries in the presence of callbacks incrementally for data dependence analysis. That is, they preanalyze the libraries and when a client uses it reuses the analysis and adds incrementally possible calls made by the client. We argue that when using our Horn clause encoding, both high analysis precision and compiler optimizations can be achieved more generally by combining the incremental static global analysis that we have proposed with abstract specialization [143].

8

Experimental Evaluation

In this chapter we present the experimental evaluation of the algorithms proposed in this thesis. We start by describing how they have been implemented within the `CiaoPP` framework. Then we show stress tests for the different configurations of the algorithm. We finish by evaluating the algorithms when (re)analyzing after making different changes to the code of a real-life application, `LPdoc`, the `Ciao` documenter. Additional experiments are presented in Chapter 9 as part of the extension of the system to perform *on-the-fly* verification.

8.1 Implementation within the `CiaoPP` framework

The different new incremental and modular algorithms have been implemented within the `CiaoPP` framework, as detailed in Sec. 2.5.1 (Fig. 17).[1] In the original implementation of the monolithic incremental algorithm, the incremental analysis state was kept in the process so that incremental operation required keeping the analysis process running. The internal databases have been made instead persistent after the analysis is performed, and are reused and updated with program changes. Also, much improved tracking of programs changes has been implemented within the front-end. This front-end is in charge of checking the current clause database to find which clauses differ, and communicate the difference to the Static Analyzer. The Static Analyzer uses the reported changes and the current state of the analyzer (a correct analysis for the previous version of the program) to recompute the fixpoint reusing as much as possible.

8.2 Incremental and Modular Analysis: Stress test

We start our analysis with a stress test of the incremental and modular analysis, for which we use a selection of well-known benchmarks from previous studies of

1 https://github.com/ciao-lang/ciaopp

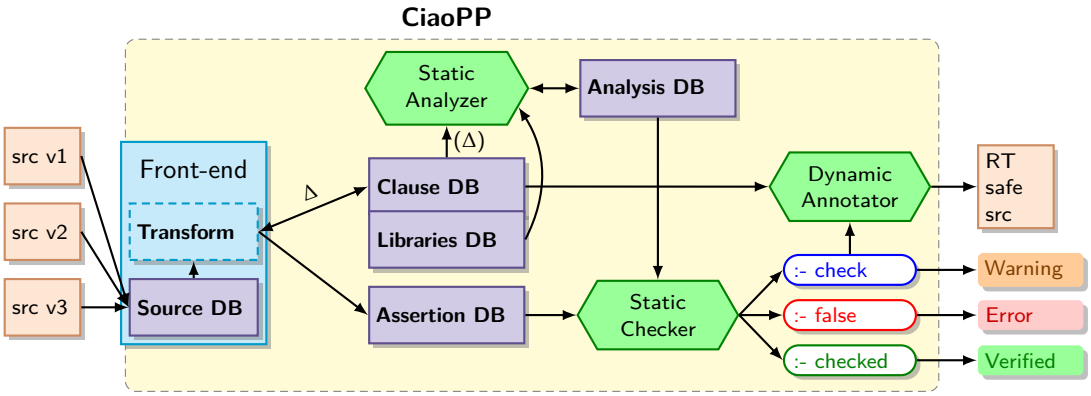


Fig. 17: Architecture of the `CiaoPP` framework supporting incrementality.

incremental analysis, e.g., `ann` (a parallelizer) and `boyer` (a theorem prover kernel), which are programs with a relatively large number of clauses located in a small number of modules. In contrast, `bid` is a more modularized program (see Table 8.1 for more details, and https://github.com/ciao-lang/ciaopp_tests/tree/master/tests/incanal for the source code of the benchmarks). We used the original modular structure as modular partition, and evaluated five strategies:

- `mon`: the baseline non-modular, non-incremental algorithm [129], i.e., IncAnalyze described in Sec. 4.1 with initial results always empty.
- `mon-inc`: the monolithic incremental algorithm IncAnalyze of Sec. 4.1.
- `mon-scc`: the monolithic incremental algorithm IncAnalyze of Sec. 4.1 with the bottom-up deletion strategy of [86].
- `mod`: as a coarse-grain modular algorithm, which consists on ModIncAnalyze without keeping each of the local analysis graphs. Note that this is not the same as the ModAnalyze of Sec. 4.2, as it did not consider modifying the program.
- `mod-inc`: the proposed modular incremental algorithm ModIncAnalyze (Chapter 5).
- `mod-scc`: for the experiments that include deleting clauses, ModIncAnalyze and the alternative strategy for updating the analysis following the strongly connected components of the program (Sec. 5.3.1).

We performed experiments with four different abstract domains: a simple reachability domain (`pdb`), a groundness domain (`gr`), a dependency tracking via propositional

Bench	# Modules	# Predicates	# Clauses	LOC
`hanoi`	2	4	6	46
`aiakl`	4	8	15	71
`qsort`	3	8	17	49
`progeom`	2	10	18	73
`bid`	7	21	48	207
`rdtok`	5	15	57	293
`cleandirs`	3	36	81	528
`read`	3	25	94	352
`warplan`	3	37	114	281
`boyer`	4	29	145	279
`peephole`	3	33	169	377
`witt`	4	69	176	618
`ann`	3	69	229	641
`manag_proj`	8	105	143	805
`check_links`	4	220	504	2042

Table 8.1: Benchmark characteristics sorted by lines of code.

clauses domain [53] (`def`), and the sharing and freeness abstract domain [128] (pointer sharing and uninitialized pointers, `shfr`). We use the exported predicates from the main module (with $\top$ call pattern) as the set of initial queries (i.e., no additional information is provided in the program).

We ran all experiments on a Linux machine (kernel 4.9.0-8-amd64) with Debian 9.0, a Xeon Gold 6154 CPU, and 16 GB of RAM. However, running the test in a standard laptop shows similar performance.

8.2.1 Overhead of incremental analysis: analyzing from scratch

We first study the analysis from scratch of all the benchmarks for all approaches, to observe the overhead introduced by the bookkeeping of the algorithms. The analysis times in milliseconds are shown in Table 8.2. For each benchmark, four rows are shown, corresponding to the four analysis algorithms mentioned earlier: monolithic (`mon`), monolithic incremental (`mon-inc`), modular (`mod`), and, lastly, modular incremental (`mod-inc`), i.e., the proposed approach. In the monolithic setting, the overhead introduced is negligible. Interestingly, the incremental modular analysis performs better overall than simply modular even in analysis from scratch. This is due to the reuse of local information specially in complex benchmarks such as `ann`, `peephole`, `warplan`, or `witt`. In the best cases (e.g., `witt`, `cleandirs`, or `check_links` analyzed with `shfr`) the performance of incremental modular competes

with monolithic thanks to the incremental updates, dropping from 20s to 3s, from 1.2s to 0.8s, and from 2.5s to 1.2s respectively.

Note that a smaller program does not necessarily imply that the analyzer will run faster, it depends on the structure of the code and the kind of data that the program operates with. Also, the cost of performing a modular analysis highly depends on the module scheduling policy, and whether the modular partitions were correctly produced. In this case, if the programmer divided the program in a reasonable manner. For example, analyzing `boyer` (with any domain) modularly comes at no cost, while in the case of `cleandirs` it is 3 times slower than doing it monolithically.

Clause addition/deletion experiment. As a stress test for the proposed algorithm, we measured the cost of re-analyzing the program incrementally adding (or removing) one clause at a time, until the program is completed (or empty). That is, for the addition experiment, the analysis was first run for the first clause only. Then the next clause was added and the resulting program (re)analyzed. This process was repeated until all the clauses in all the modules were added. For the deletion experiment, starting from an already analyzed program, the last clause was deleted and the resulting program (re)analyzed. This process was repeated until no clauses were left. The experiment was performed for all the approaches using the initial (top-down) deletion strategy (`mod-inc`) and the SCC-partition deletion strategy of Sec. 5.3.1 (`mod-scc`).

In our experiments we observed that the analyses performed with the `gr` and `def` domains were the most relevant to evaluate the usefulness of the algorithm. This is due to the domain operations being fairly simple (when compared with the cost of executing the fixpoint algorithm), so that, the complexity of the algorithm is not completely hidden by the complexity of the domain operations (e.g. `shfr`). At the same time, they are complex enough that there is some fixpoint iteration (which does not occur in, e.g., `pdb`, since $\top$ is assumed for every call pattern). Therefore, in this section we focus mainly in the analysis with the `def` domain, but include as well some discussion about `gr`. Nevertheless, for the results for the remaining domains we refer the reader to Appendix A.

Fig. 18 shows the addition and deletion experiments for the `warplan` benchmark analyzed with `def`. Each point represents the time taken to reanalyze the program after incrementally adding/deleting one clause. The horizontal axis denotes the number of clauses added/deleted at that point of the experiment. We observe that the proposed incremental algorithm outperforms overall the non-incremental settings when the time needed to reanalyze is large. We find that for smaller benchmarks our algorithm performs up to 8 times faster than the traditional monolithic, non-incremental algorithm, and, in the worst cases performs as fast as the traditional

Benchmark	pdb	gr	def	shfr	Benchmark	pdb	gr	def	shfr
hanoi (mon)	5.2	2.9	2.2	10.7	warplan	46.0	24.5	20.1	63.3
(mon-inc)	5.3	3.0	2.2	10.2		41.0	26.6	16.6	64.3
(mod)	12.3	7.2	5.8	22.1		71.7	52.7	35.8	180.7
(mod-inc)	10.0	6.2	4.6	18.3		57.0	37.1	24.1	102.9
aiakl	5.9	6.4	4.6	7.9	boyer	38.3	24.1	14.9	50.0
	7.6	7.3	5.8	8.4		37.0	31.5	17.4	51.5
	15.0	18.9	14.0	18.0		48.3	39.3	21.5	68.2
	16.0	15.5	13.0	16.4		44.9	37.3	19.1	65.4
qsort	7.8	8.3	4.0	9.5	peephole	67.0	43.2	19.2	157.6
	8.0	8.5	4.3	10.5		64.1	45.6	21.2	156.4
	21.7	21.6	13.5	24.9		155.8	75.6	43.6	392.8
	19.5	20.3	10.0	20.1		115.1	62.4	40.2	267.0
progeom	5.4	5.4	5.1	6.4	witt	183.4	11.6	33.5	2490.4
	6.1	5.4	5.4	7.1		186.0	16.4	38.8	2491.2
	24.1	23.6	21.6	28.7		1134.6	6.7	120.8	20550.3
	18.4	18.7	14.8	20.3		414.8	9.8	71.1	3222.9
bid	18.8	14.8	9.9	22.9	ann	84.5	58.4	35.0	120.5
	17.7	15.4	10.2	26.8		85.4	64.1	38.5	123.7
	61.0	55.1	39.1	68.3		264.1	174.5	89.7	296.6
	42.4	42.1	32.4	55.7		145.3	127.0	60.6	241.5
rdtok	33.5	44.0	15.7	63.3	manag_proj	111.0	24.1	51.3	18049.2
	51.3	29.2	17.8	66.0		98.3	28.3	48.8	17967.3
	85.1	61.3	40.2	122.0		291.3	54.7	150.3	37184.9
	52.3	53.5	36.6	90.9		221.8	44.4	104.7	34595.0
cleandirs	33.2	27.6	26.2	384.1	check_links	701.7	301.6	167.5	803.3
	31.7	29.1	27.7	389.0		678.9	251.5	178.5	819.2
	145.5	123.8	140.3	1189.2		1292.6	680.8	600.5	2530.3
	93.8	77.2	80.5	778.3		776.1	360.8	267.2	1162.5
read	217.5	116.5	47.8	399.0					
	172.3	105.0	35.0	400.7					
	192.0	118.4	45.1	422.4					
	189.9	126.9	45.5	472.4					

Table 8.2: Analysis times from scratch (ms).

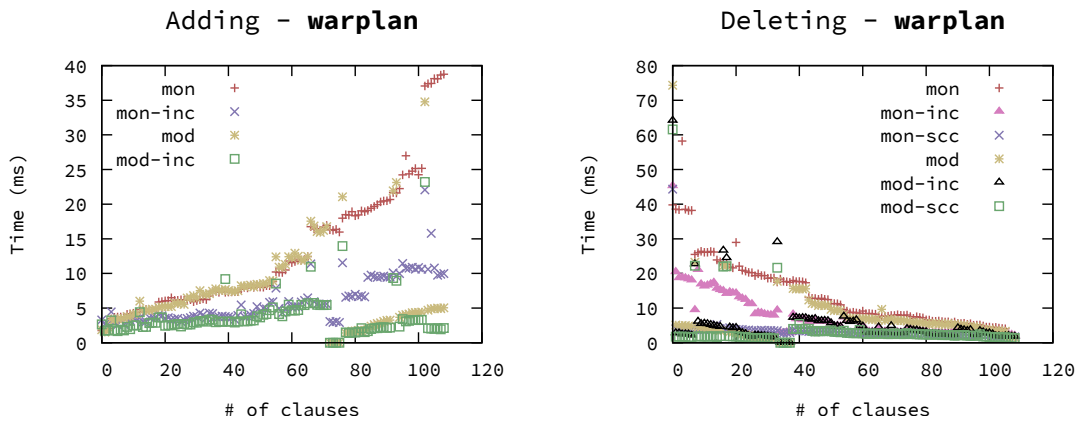


Fig. 18: Analysis time (ms) for `warplan` with `def` for both experiments.

modular algorithm. The detailed analysis times per iteration for the remaining benchmarks are available in Appendix A.

We observe that, even when analyzing takes less time, i.e., when the program has fewer clauses, the analysis time of the algorithm proposed is faster overall. Moreover, as the analysis grows in complexity, the cost our approach grows significantly slower than that of the traditional algorithm. In the case of the deletion experiments, we observe also clear advantages, specially when using the strategy of partitions in SCC presented in Sec. 5.3.1.

8.2.2 *Analysis time per action*

In order to get an overall idea of the cost in terms of the time taken by the analysis we have included Tables 8.3 and 8.4 for the addition and deletion experiment respectively. They show, split by benchmark and analysis configuration, the `mean`, `maximum`, and `minimum` analysis times after each modification made in the experiment, for each program. The objective is to provide intuition for the "response times" of the analyses after each such modification. We center our attention on the costlier *instances* of the benchmarks, i.e., the (re)analysis runs which take the longest after a modification is performed in the program for the traditional, *monolithic* analysis. In absolute terms these are `check_links`, with the largest analysis time (608.6ms), followed by `witt` (177.7ms), `read` (177.4ms), and `manag_proj` (100.8ms). In terms of overall cost (*mean*) of the reanalysis we have `read` (38.5ms), `check_links` (29.3ms), and `boyer` (21.5ms). These high differences between the mean and maximum analysis times are due to the very small values for the first additions, in which the program is very small

	mon			mon-inc			mod			mod-inc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.2	5.6	1.0	1.8	5.9	1.3	1.8	14.9	0.5	1.7	14.1	0.5
ann	7.2	76.8	1.1	3.3	51.8	1.6	2.5	156.8	0.3	2.3	133.9	0.6
bid	3.4	18.1	1.5	2.6	11.9	1.9	2.5	32.7	0.3	2.3	26.6	0.5
boyer	21.5	46.4	1.5	3.2	13.0	1.4	6.0	15.1	0.4	1.9	14.8	0.5
check_links	29.3	608.6	2.6	14.5	571.5	5.5	21.6	889.4	0.4	7.1	664.1	0.8
cleandirs	9.1	28.9	1.5	4.5	18.6	1.9	6.2	96.5	0.9	3.7	63.4	0.9
hanoi	2.4	4.7	0.7	1.8	5.3	1.1	2.9	11.5	0.4	2.1	10.0	0.4
manag_proj	19.7	100.8	4.4	8.3	35.2	4.8	6.9	97.7	0.3	4.6	69.5	0.4
peephole	9.6	64.8	1.5	2.7	30.2	1.6	2.1	95.8	0.4	1.9	76.5	0.6
progeom	2.3	5.4	1.1	1.6	3.4	0.7	2.3	10.2	0.8	2.2	9.7	0.8
read	38.9	177.4	1.1	13.4	151.3	1.3	18.5	214.2	0.8	9.9	165.2	0.6
qsort	2.8	11.0	1.1	1.8	4.4	1.1	2.8	10.5	0.4	2.4	9.5	0.5
rdtok	8.2	31.1	1.2	6.2	57.6	1.3	6.0	27.2	0.3	6.0	59.3	0.4
warplan	10.2	35.0	0.8	4.2	18.5	1.5	5.4	30.7	0.7	2.5	21.3	0.9
witt	15.2	177.7	1.5	5.5	142.0	2.2	11.1	653.2	0.3	4.8	323.4	0.4

Table 8.3: Analysis times (ms) per action of the clause *addition* experiment with `def`.

and there are no iterations. The analysis times for the remaining experiments are available in detailed analysis times for each step are provided in Appendix A. These should be in principle the benchmarks that we should focus on incrementalizing, as more time is saved. This applies not only to the monolithic analysis, but also to the modular analysis. To observe the increase in performance obtained, Table 8.5 shows the speedup of our algorithm with respect to: `mon-inc`, in the case of the addition experiments, and `mod-scc`, in the case of the deletion experiment. The analysis times for the remaining speedups are provided in Appendix A

8.2.3 Accumulated analysis time

To observe in a more detailed manner how the analyzers behave we present in Figs. 19 and 20 the accumulated analysis times, i.e., the analysis time of all the experiments aggregated by benchmark, *divided by how much time was spent in the different parts of the algorithm*. The results are for the `gr` and `def` abstract domains, and each of the bars shows of the full set of addition and deletion experiments.

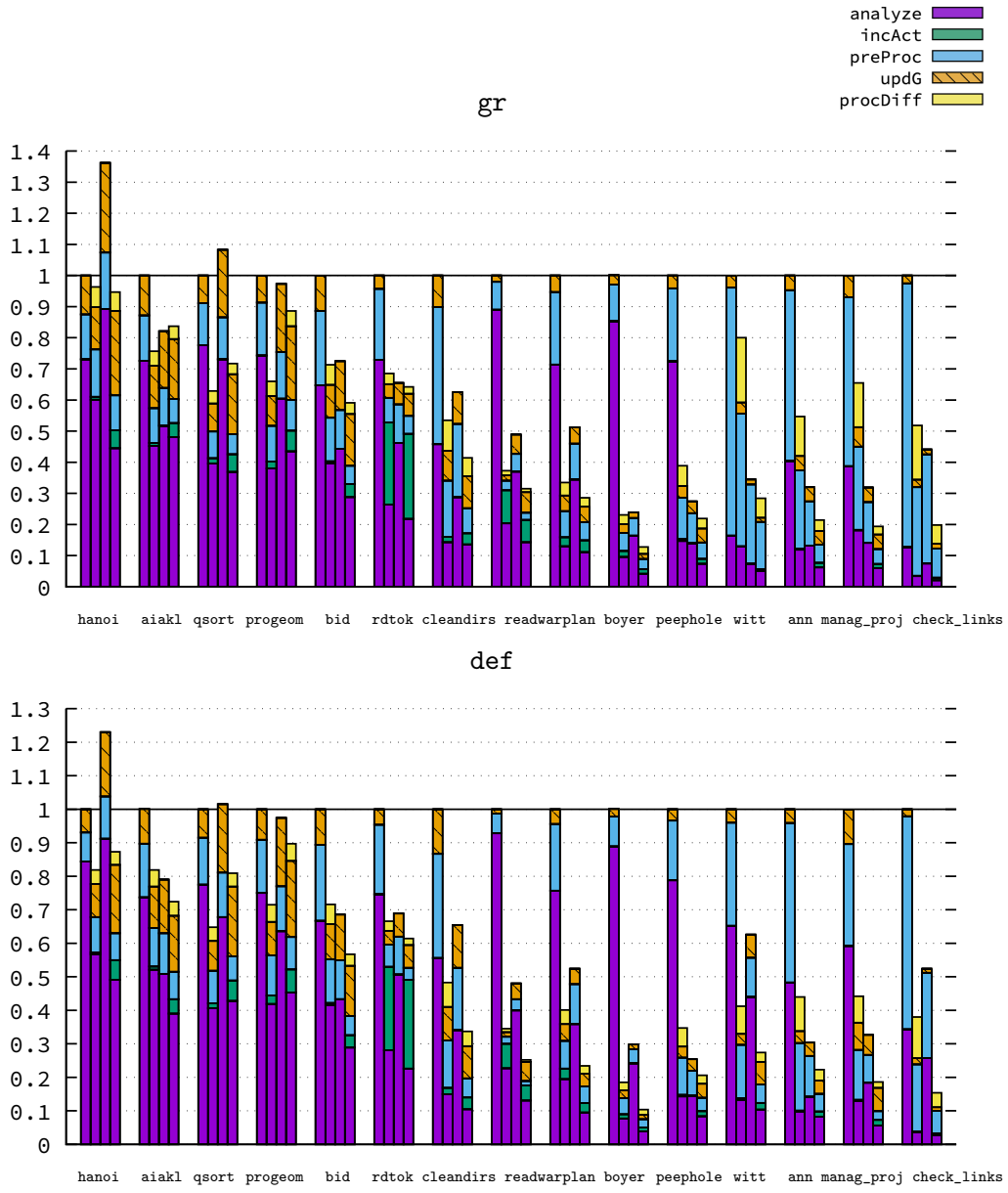


Fig. 19: Accumulated analysis time (normalized w.r.t `mon`) adding clauses. The order inside each set of bars is: |`mon`|`mon-inc`|`mod`|`mod-inc`|.

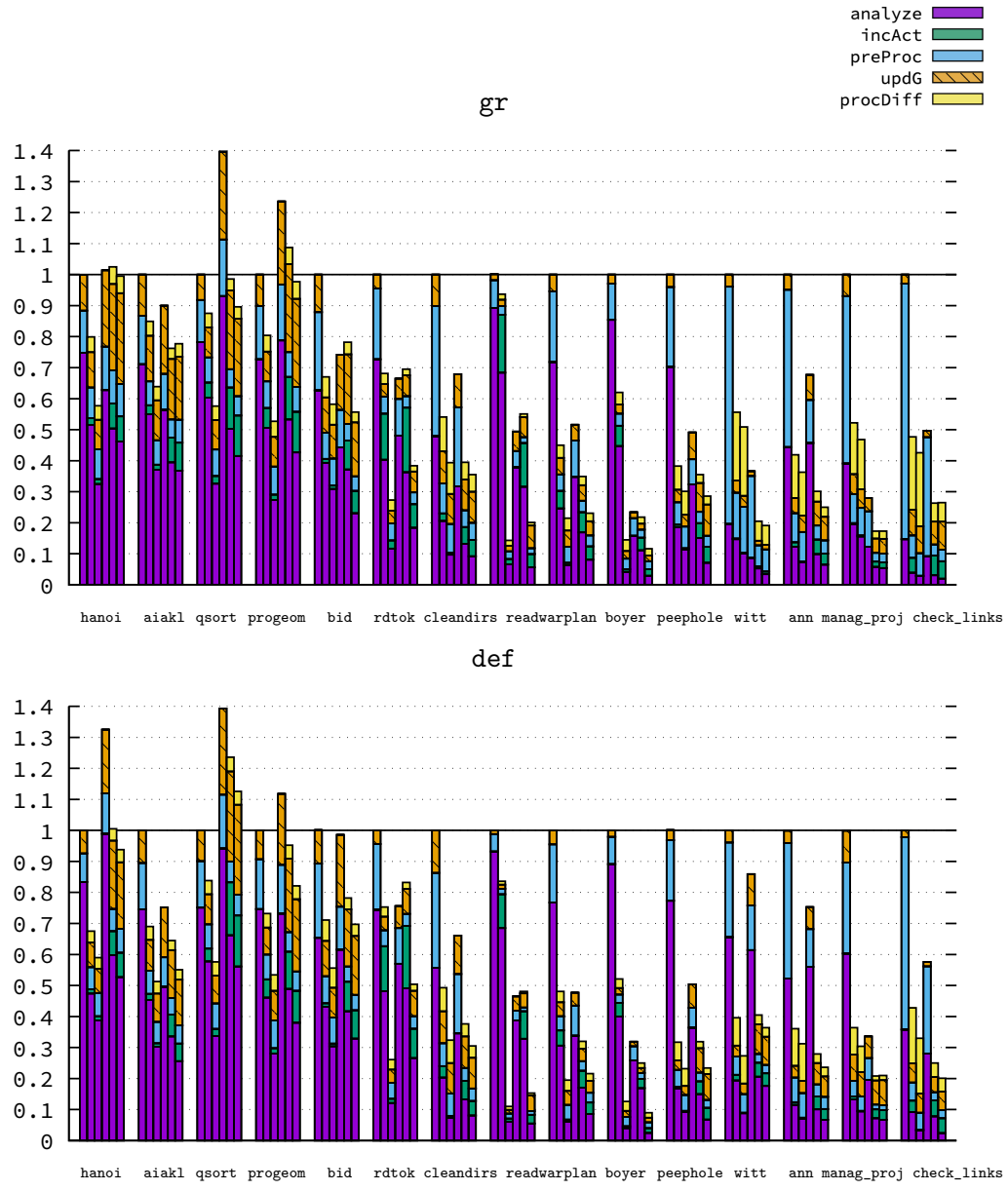


Fig. 20: Accumulated analysis time (normalized w.r.t `mon`) deleting clauses. The order inside each set of bars is: |`mon`|`mon-inc`|`mon-scc`|`mod`|`mod-inc`|`mod-scc`|.

	mon			mon-inc			mon-scc			mod			mod-inc			mod-scc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.6	7.2	1.3	1.9	6.9	1.3	1.4	6.1	0.9	2.0	15.2	0.5	1.8	13.4	0.4	1.5	13.7	0.3
ann	7.9	85.8	1.2	4.0	80.4	1.8	3.3	80.3	1.5	8.8	197.1	0.3	3.4	153.6	0.6	2.8	153.1	0.6
bid	3.0	17.1	1.6	2.5	19.4	1.6	1.9	18.3	1.4	3.4	59.8	0.5	2.9	53.1	0.4	2.6	50.3	0.4
boyer	21.1	51.5	1.3	11.4	35.3	1.3	2.2	36.2	1.1	6.3	44.9	0.4	5.3	45.3	0.6	1.6	45.1	0.3
check_links	29.0	635.6	1.5	28.3	674.8	6.4	20.9	598.2	5.5	22.6	946.0	0.3	17.0	720.1	0.7	13.3	716.0	0.5
cleandirs	9.2	27.5	1.6	5.6	32.1	1.7	3.3	30.1	1.4	6.2	128.9	0.9	4.4	90.8	0.7	3.5	86.5	0.6
hanoi	2.6	5.3	0.8	1.8	5.6	0.9	1.5	5.9	0.7	3.5	13.8	0.5	2.7	12.4	0.7	2.5	12.6	0.5
manag_proj	20.9	103.6	4.5	9.3	98.3	4.5	7.7	90.1	4.1	7.9	259.3	0.3	5.7	212.1	0.3	5.8	217.7	0.3
peephole	10.3	100.6	1.7	3.4	67.4	1.5	2.3	65.4	1.3	5.6	138.6	1.1	3.7	113.4	1.0	2.7	116.2	0.9
progeom	2.3	5.3	1.1	1.8	5.5	1.0	1.2	6.3	0.9	2.7	22.0	0.7	2.5	19.3	0.7	2.1	18.6	0.6
read	39.0	184.7	1.1	33.8	189.1	1.1	3.9	171.9	1.0	17.9	185.7	0.7	19.4	191.3	0.7	5.8	187.2	0.6
qsort	2.5	7.5	1.1	2.2	8.5	1.2	1.4	9.3	0.8	3.6	22.3	0.4	3.4	20.4	0.6	3.1	20.8	0.5
rdtok	8.2	29.7	1.5	7.3	31.8	1.3	2.2	31.8	1.1	6.6	61.8	0.3	8.2	54.1	0.6	4.8	54.8	0.3
warplan	11.1	52.7	1.0	6.2	40.2	1.3	2.0	39.3	1.0	5.5	63.2	0.8	4.1	56.0	1.0	2.7	53.7	0.7
witt	14.6	174.8	1.2	7.1	179.3	2.1	4.5	185.2	1.9	15.0	1,021.7	0.3	7.9	461.3	0.4	7.1	423.9	0.4

Table 8.4: Analysis times (ms) per action of the clause *deletion* experiment with `def`.

Fig. 19 shows the accumulated analysis time for the addition experiments. As mentioned before the bars are split to show the time taken in each operation: `analyze` is the time spent in the module analyzer, `incAct` is the time spent updating the local analysis results, `preProc` is the time spent processing clause relations (e.g., calculating the SCCs), `updG` is the time spent updating $\mathcal{G}$, and `procDiff` the time spent applying the changes to the analysis. This last parameter only appears in the incremental settings. The bars are normalized with respect to the monolithic non-incremental (`mon`) algorithm, which is assumed to take "1" measure of time to execute. For instance, if analyzing `rdtok` with the `gr` domain for the monolithic non-incremental setting is taken as 1, the modular incremental (`mod-inc`) setting takes approx. 0.6, so it is approx. 1.67 times faster.

As before, the benchmarks are sorted by number of LOC. Because of this, it can be observed that the incremental analysis does tend to be more useful as program size grows. Overall, the incremental settings (`mon-inc`, `mod-inc`) are always faster than the corresponding non-incremental settings (`mon`, `mod`). Furthermore, while the traditional modular analysis is sometimes slower than the monolithic one (for the small benchmarks: `hanoi` and `qsort`), our modular incremental algorithm always outperforms both, obtaining 10× overall speedup over monolithic in the best cases (`boyer` analyzed with `def` or `peephole` analyzed with `shfr`). Furthermore, in the

	addition: `mod-inc`			deletion: `mod-scc`				
bench	vs. `mon`	vs. `mon-inc`	vs. `mod`	vs. `mon`	vs. `mon-inc`	vs. `mon-scc`	vs. `mod`	vs. `mod-inc`
`aiakl`	1.3	1.1	1.0	1.7	1.2	0.9	1.3	1.2
`ann`	3.1	1.4	1.1	2.9	1.4	1.2	3.2	1.2
`bid`	1.5	1.1	1.1	1.2	1.0	0.7	1.3	1.1
`boyer`	11.6	1.7	3.2	13.0	7.1	1.4	3.9	3.3
`check_links`	4.1	2.1	3.1	2.2	2.1	1.6	1.7	1.3
`cleandirs`	2.5	1.2	1.7	2.6	1.6	1.0	1.8	1.3
`hanoi`	1.2	0.9	1.4	1.1	0.7	0.6	1.4	1.1
`manag_proj`	4.3	1.8	1.5	3.6	1.6	1.3	1.4	1.0
`peephole`	5.0	1.4	1.1	3.8	1.3	0.9	2.1	1.4
`progeom`	1.1	0.7	1.1	1.1	0.9	0.6	1.3	1.2
`read`	3.9	1.4	1.9	6.8	5.9	0.7	3.1	3.4
`qsort`	1.2	0.7	1.2	0.8	0.7	0.5	1.2	1.1
`rdtok`	1.4	1.0	1.0	1.7	1.5	0.5	1.4	1.7
`warplan`	4.1	1.7	2.2	4.2	2.3	0.8	2.1	1.6
`witt`	3.1	1.1	2.3	2.1	1.0	0.6	2.1	1.1

Table 8.5: Speedups of the clause *addition* (left) and *deletion* (right) experiments with `def`.

larger benchmarks modular incremental outperforms even the monolithic incremental approach.

Fig. 20 shows the results of the deletion experiment. The analysis performance of the incremental approaches is in general better than the non-incremental approaches, except some cases for small programs. Again, our proposed algorithm shows very good performance, in the best case 10× speedup (`read` analyzed with `shfr`), and overall 5× speedup (`ann`, `peephole`, and `witt`), competing with monolithic incremental `scc` and outperforming in general monolithic incremental `td`. The SCC-guided deletion strategy seems to be more efficient than the top-down deletion strategy. This confirms that the top-down deletion strategy tends to be quite pessimistic when deleting information, and modular partitions limit the scope of deletion. For the accumulated analysis time of the remaining domains, please see Appendix A.

8.2.4 *Distribution of analysis times*

Next, we study how the analysis time of the experiments is distributed. Figs. 21 and 22 show histograms that illustrate the number of analyzed instances of the experiments with respect to the analysis time, regardless of the order in which the experiments were performed, and for each configuration. In the vertical axis we plot the number of tests, i.e., *how many different instances of the addition or deletion experiment that were performed could be analyzed in that time or less.* In the

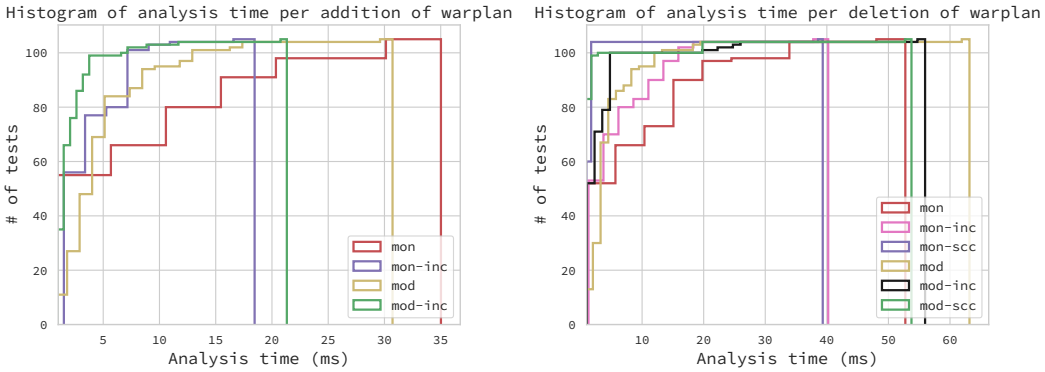


Fig. 21: Distribution over time of instances of the addition (left) and deletion (right) experiments for `warplan` with `def`.

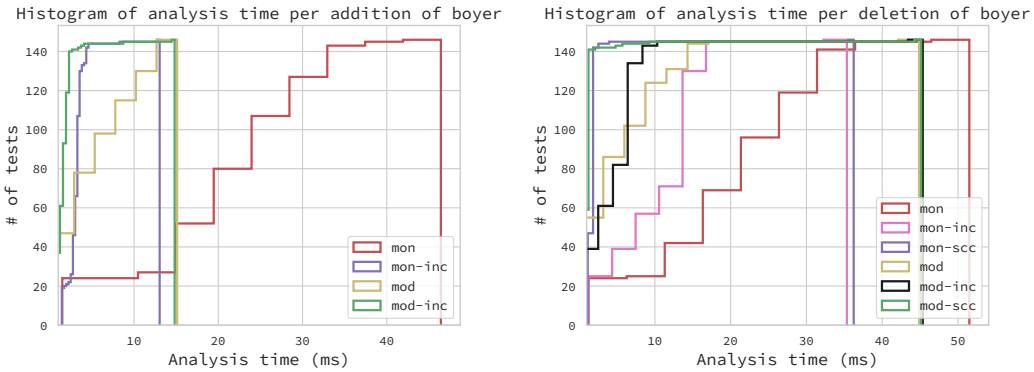


Fig. 22: Distribution over time of instances of the addition (left) and deletion (right) experiments for `boyer` with `def`.

horizontal axis we represent the analysis time. For example, on the left-hand side of Fig. 21, for $5ms$ in the vertical axis, starting from the bottom of the graph we first find the red line corresponding to the monolithic analysis (`mon`). This means that approx. 55 of the analyses performed in `warplan` finished in $5ms$ or less. Then we find the yellow line (`mod` analysis): for this setting, 70 of the instances of the addition experiment were analyzed in $5ms$ or less. The next line that we find is the purple line, corresponding to the `mon-inc` configuration. In this case 78 instances were analyzed in $5ms$ or less. Finally, we have our configuration, `mod-inc`, that was able to analyze 99 instances of the addition experiment in $5ms$. Figs. 21 and 22 show that, overall, the analysis time of the proposed algorithm is faster than that of the previous configurations.

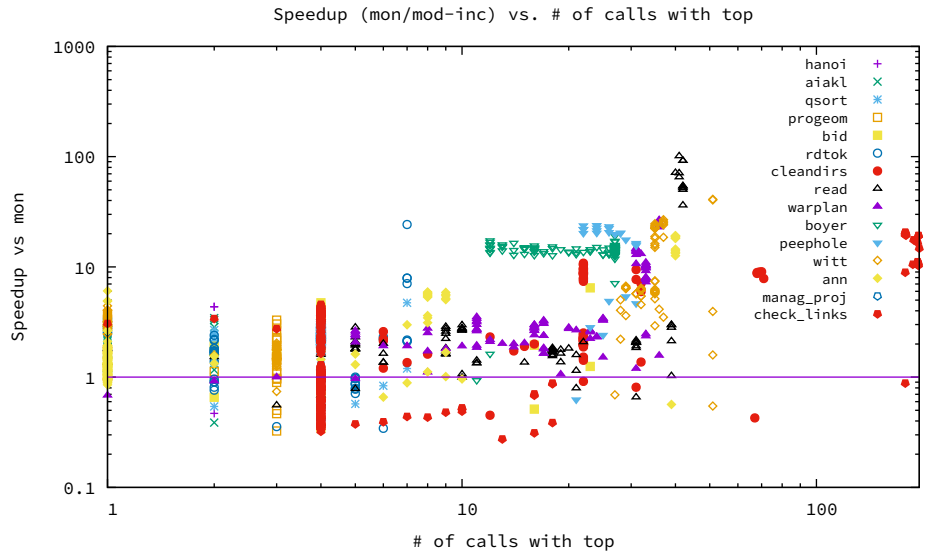


Fig. 23: Speedup vs. monolithic depending on the number of nodes in the analysis graph.

8.2.5 *Correlations to benchmark and analysis graph characteristics*

We also looked for correlations between benchmark characteristics and the speedups observed. While this topic would require a study of its own, we have observed some correlations with benchmark-related analysis characteristics. Figs. 23 and 24 show scatter plots of the speedup obtained with respect to two such characteristics: the number of nodes in the analysis graph an the number of calls to $\top$. The plots show that there is some correlation between the size of the analysis graph and the speedup obtained: we observed that the incremental and modular analysis proposed is beneficial for larger analysis graphs. The sizes of the analysis graphs depend themselves on the complexity of the abstract domain (due to the algorithm being multivariant), the size of the program, and the size of the strongly connected components of the program. Also, we observed that the slowdowns encountered correspond to very small runtimes of the algorithms, e.g., for smaller programs, which are likely to be due to the overhead of the additional bookeeping required by the algorithm. This is, however, not very concerning, as they are small.

8.2.6 *Memory Usage*

We also studied the memory usage for the structures needed for analysis, that is, the analysis graphs, and the other structures needed for memoizing. Table 8.6 contains

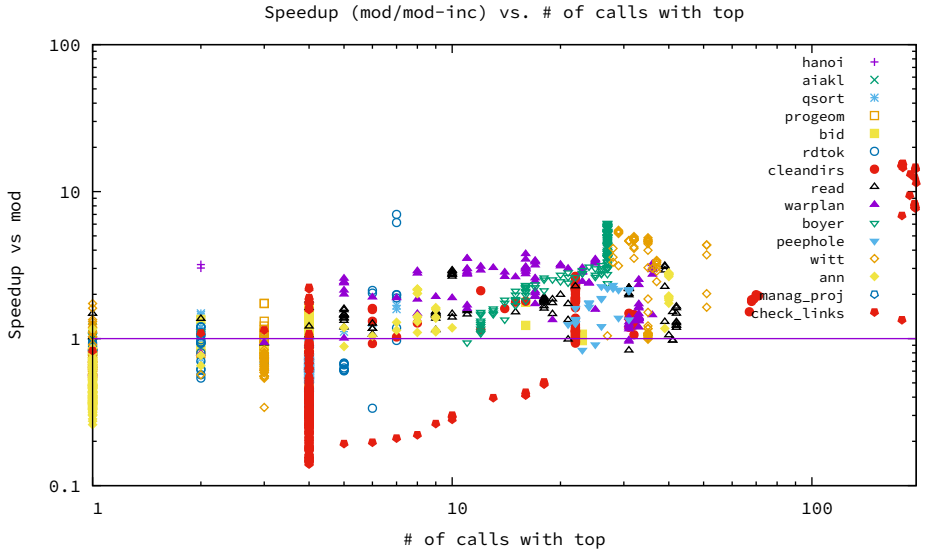


Fig. 24: Speedup vs. modular depending on the number of calls to $\top$.

the maximum memory needed for these structures for any of the modifications analyzed for each benchmark, i.e., the memory high water mark. For the monolithic case, this is the maximum memory necessary to keep the analysis results, and for the modular case, the maximum size of the analysis results of a module and the intermodular information. We do not show any distinction between the different deletion strategies of the incremental algorithm as the necessary bookkeeping of both is the same.

First, note that, since the incremental algorithms (`mon-inc` and `mod-inc`) need to perform additional bookkeeping, they always need more memory than the corresponding non incremental ones (`mon` and `mod`). However, this difference is small and arguably a very reasonable price to pay for the significant reductions in analysis times. Also note that the modular analyses (`mod` and `mod-inc`) always bring a reduction in the memory required to be able to complete every analysis instance. This is of course important because, while it is always possible to wait a bit longer for an analysis result, if the analysis does not fit in the available memory, either the performance will be much worse, due to swapping, or the analysis simply cannot be completed, if virtual memory is depleted.

More importantly, we observe that we obtain a reduction in the memory use of the proposed modular incremental algorithm, `mod-inc`, with respect to the original monolithic incremental algorithm, `mon-inc`. This is shown in the last column of Table 8.6. The memory usage reduction obtained ranges between 59% for the `ann` benchmark and 13% for the `read` benchmark. Ideally, we would like to achieve a

bench	mon	mon-inc	mod	mod-inc	reduction (mon-inc vs mod-inc)
hanoi	16K	16K	12K	12K	0.75 **(25%)**
aiakl	28K	28K	8K	16K	0.57 **(43%)**
qsort	28K	32K	12K	16K	0.50 **(50%)**
progeom	24K	32K	20K	20K	0.63 **(37%)**
bid	80K	80K	36K	40K	0.50 **(50%)**
rdtok	100K	112K	68K	80K	0.71 **(29%)**
cleandirs	200K	204K	144K	152K	0.75 **(25%)**
read	304K	308K	260K	268K	0.87 **(13%)**
warplan	144K	156K	116K	128K	0.82 **(18%)**
boyer	140K	144K	76K	80K	0.55 **(44%)**
peephole	200K	208K	108K	116K	0.56 **(44%)**
witt	504K	524K	352K	364K	0.69 **(30%)**
ann	316K	324K	120K	132K	0.41 **(59%)**
manag_proj	464K	460K	248K	268K	0.58 **(42%)**
check_links	2.3M	2.3M	1.8M	1.8M	0.78 **(22%)**

Table 8.6: Maximum memory usage for the experiments with `def` in bytes.

reduction of memory proportional to the number of modules in which the program is distributed, but on one hand there is overhead due to the fact that each module needs to keep information for the calls to predicates imported from other modules, and in addition a very large reduction in maximum memory usage requires the partitions to be of similar size, quite independent, and with similarly-sized analysis graphs. Our benchmarks instead typically contain a module with the main functionality and some libraries with simpler code, so that the distribution of code among the modules is not even, and so the correlation between memory usage reduction and number of modules in the program is not direct. However, we expect that in actual applications, which tend to have a much larger number of modules and use a good number of libraries, the memory usage reduction will be much larger.

8.3 Studying the effect of using assertions during analysis

In order to study the effect of using assertions during analysis we have implemented GIAwAC (and GuidedIncAnalyze) within the `CiaoPP` system [84] and performed some preliminary experiments to test the use case described in Example 7.1. Our test case is the `LPdoc` documentation generator tool [78, 81], which takes a set of Prolog files with assertions and machine-readable comments and generates a reference manual from them. `LPdoc` consists of around 150 files, of mostly (`Ciao`)

domain	no backend	+ texinfo	+ man	+ html
reachability	1.7	2.1	3.4	3.9
reachability *inc*	1.7	1.2	1.0	1.6
gr	2.1	2.2	2.3	2.6
gr *inc*	2.1	1.4	0.9	1.8
def	6.0	7.1	7.8	9.7
def *inc*	6.0	2.2	1.3	3.5
sharing	27.2	28.1	24.2	28.5
sharing *inc*	27.2	3.9	1.4	5.1

Table 8.7: Analysis time for LPdoc adding one backend at a time (time in seconds).

Prolog code,with assertions (most of which, when written, were meant mainly for documentation generation), as well as some auxiliary scripts in Lisp, JavaScript, bash, etc. The Prolog code analyzed is about 22K lines. This is a tool in everyday use that generates for example all the manuals and web sites for the Ciao system (http://ciao-lang.org, http://ciao-lang.org/documentation.html), as well as for all the different *bundles* developed internal or externally, processing around 20K files and around 1M lines of Prolog and interfaces to other languages. The LPdoc code has also been adapted as the documentation generator for the XSB system [168].

LPdoc is specially relevant in our context because it includes a number of backends in order to generate the documentation in different formats such as texinfo, Unix man format, html, ascii, etc. The front end of the tool generates a documentation tree with all the content and formatting information and this is passed to one out of a number of these backends, which then does the actual, specialized generation in the corresponding typesetting language. We analyzed all the LPdoc code with a reachability domain, a groundness domain (gr), a domain tracking dependencies via propositional clauses [53] (def), and a sharing domain with cliques [132]. The experiment consisted in preanalyzing the tool with no backends and then adding incrementally the backends one by one. In Table 8.7 we show how much time it took to analyze in each setting, i.e., for the different domains and with the incremental algorithm or analyzing from scratch. The experiments were run on a MacBook Pro with an Intel Core i5 2.7 GHz processor, 8GB of RAM, and an SSD disk. These preliminary results support our hypothesis that the proposed incremental analysis brings performance advantages when dealing with these use cases of generic code.

To conclude, in this chapter we presented the experimental evaluation of the modular incremental fixpoint algorithm showing that we obtain almost immediate response when the changes do not affect the result, up to 13× speedup w.r.t. the original non-incremental algorithm. Being aware of modular structures is useful: up to 2× speedup when compared with the original incremental algorithm. Modular analysis from scratch is improved up to 9×, and the maximum size of analysis graphs reduced. We also showed that keeping structures for incrementality produces small overhead. For the algorithms using assertions we saw that assertions help in reducing computation times when reanalyzing after instantiating new implementations of generic code interfaces.

9

Application: On-the-fly Assertion Checking

A driving motivation throughout the thesis has been the fact that assertion checking is an invaluable programmer's tool for finding many classes of errors or verifying their absence in dynamic languages such as Prolog. In this chapter we explore how incremental abstract interpretation can help when verification is done *on-the-fly*, to give continuous correctness feedback during program development, a context where fast response times are essential. In particular, we illustrate how the incremental analyses introduced in the thesis contribute to achieving a high level of reactivity in an integration of the static analysis and verification framework within an integrated development environment (IDE) that reflects analysis and verification results back as colorings and tooltips directly on the program text—the `CiaoPP` tool's VeriFly mode.

This mode was built on an existing Emacs-based development environment for the `Ciao` language, and reuses in part off-the-shelf "on-the-fly" *syntax* checking capabilities offered by the Emacs `flycheck` package. Emacs was chosen because it is a solid platform and preferred by many experienced users. However, this low-maintenance approach should be easily reproducible in other modern extensible editors and mature program development environments.

9.1 Assertion verification

As mentioned before, the `CiaoPP` verification framework uses analyses based on the abstract interpretation technique (the Static Analyzer in Fig. 25), in order to statically compute safe approximations of the program semantics at different relevant program points. Given a program, a set of abstract domains are automatically selected, as determined by their relevance to the properties that appear in the assertions written by the programmer or present in libraries. That is, domains are selected by determining if they can (efficaciously) abstract the properties in the assertions. As mentioned, domains implement the transfer functions of the built-in operations that are relevant to them. If a property in an assertion is understood by a domain, the domain is run. Typically a property may be understood by more than one domain.

In such cases, the tool applies heuristics to avoid redundancies. The analysis with the selected domains is performed and the resulting safe approximations are compared directly with these assertions (the Static Checker in Fig. 25) in order to prove the program correct or incorrect with respect to them. For each assertion originally with status `check`, the result of this process (boxes on the right of Fig. 25) can be: that it is verified (the new status is `checked`), that a violation is detected (the new status is `false`), or that it is not possible to decide either way, in which case the assertion status remains as `check`, as detailed in Sec. 2.5.2. In such cases, optionally, a warning may be displayed and/or a run-time test generated for (the part of) the assertion that could not be discharged at compile-time, test cases generated, etc.

The main components for assertion checking in `CiaoPP` were proposed in [148, 145, 140]. We recall the definitions for checking calls and success conditions for a monovariant analysis.

Definition 9.1 (Checked calls condition)**.** A calls condition $\mathsf{calls}(p(V_1,\ldots,V_n), Pre)$ is abstractly *checked* for a predicate p w.r.t. Q_α in D_α if there is $\langle L, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ *s.t.* $\exists$ a renaming σ, $L = p(V'_1,\ldots,V'_n) = p(V_1,\ldots,V_n)\sigma, \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma,P)}$.

Definition 9.2 (False calls condition)**.** A calls condition $\mathsf{calls}(p(V_1,\ldots,V_n), Pre)$ is abstractly *false* for a predicate $p \in P$ w.r.t. Q_α in D_α if there is $\langle L, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ *s.t.* $\exists$ a renaming σ, $L = p(V'_1,\ldots,V'_n) = p(V_1,\ldots,V_n)\sigma, \lambda^c \sqcap \lambda^+_{TS(Pre\ \sigma,P)} = \bot$.

Note that in these definitions we do not use directly the *Pre* and *Post* conditions, although they already are abstract substitutions. This is because the properties in the conditions stated by the user in assertions might not exist as such in D_α. The fact that the resulting approximations are safe ensures correctness of the procedure when checking both calls and success conditions.

Definition 9.3 (Checked success condition)**.** A success condition $\mathsf{success}(p(V_1,\ldots,V_n), Pre, Post)$ is abstractly *checked* for predicate $p \in P$ w.r.t. Q_α in D_α if there is $\langle L, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ *s.t.* $\exists$ a renaming σ, $L = p(V'_1,\ldots,V'_n) = p(V_1,\ldots,V_n)\sigma$ $\lambda^c \sqcap \lambda^+_{TS(Pre\ \sigma,P)} = \bot$ or $\lambda^s \sqsubseteq \lambda^-_{TS(Post\ \sigma,P)}$.

Definition 9.4 (False success condition)**.** A success condition $\mathsf{success}(p(V_1,\ldots,V_n), Pre, Post)$ is abstractly *false* for $p \in P$ w.r.t. Q_α in D_α if there is $\langle L, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ *s.t.* $\exists$ a renaming σ, $L = p(V'_1,\ldots,V'_n) = p(V_1,\ldots,V_n)\sigma, \lambda^c \sqcap \lambda^-_{TS(Pre\ \sigma,P)} \neq \bot$ and $(\lambda^s \sqcap \lambda^+_{TS(Post\ \sigma,P)} = \bot)$.

9.2 VeriFly: The On-the-fly IDE Integration

Fig. 25 shows the overall architecture of VeriFly, the integration of the `CiaoPP` framework with the new IDE components, represented by the new box to the left,

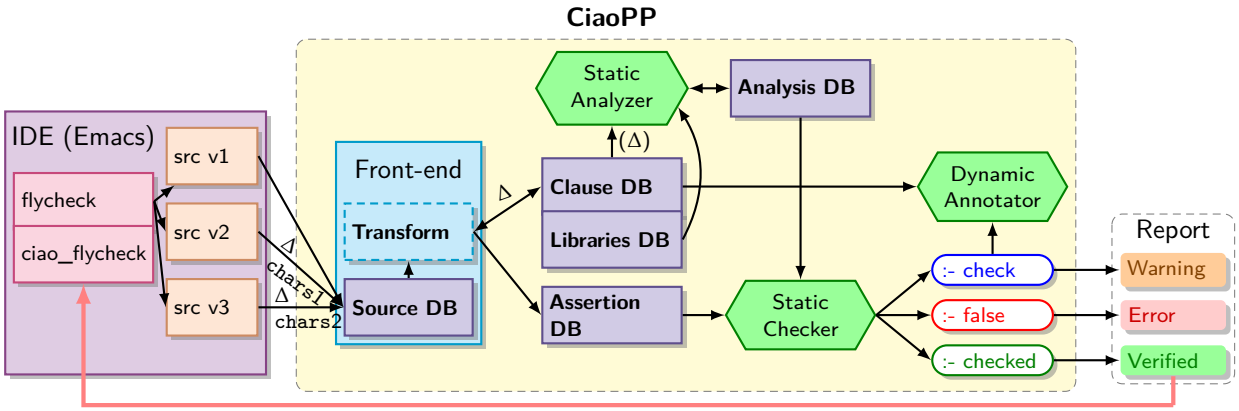


Fig. 25: Integration of the `CiaoPP` framework in the Emacs-based IDE.

and the communication to and from `CiaoPP`. The original output of ciaopp ("Report" box on the right) is parsed and used to highlight code and verified/falsified assertions. As mentioned before, the tool interface is implemented within Emacs and the on-the-fly support is provided by the Emacs "`flycheck`" package.[1] Flycheck is an extension developed for GNU Emacs originally designed for on-the-fly *syntax* checking, but we use it here in a semantic context. However, as also mentioned before, a similar integration is possible with any reasonably extensible IDE.

The overall architecture consists of a `flycheck` adaptor (implementing different `Ciao`-based checkers, from syntactic to full analysis), a `CiaoPP` process that runs in the background in daemon mode, and a lightweight client to `CiaoPP`. When a file is opened, modified, or saved, as well as after some small period of inactivity, an editor checking event is triggered. Edit events notify `CiaoPP` (via the lightweight client) about program changes, which can be both in code and assertions. The `CiaoPP` daemon receives these changes, and, behind the scenes, transforms them into changes at the clause level (also checking for syntactic errors), and then incrementally (re-)analyzes the code and (re-)checks any reachable assertions. The latter can be in libraries, other modules, language built-in specifications, or of course (but not necessarily) in user code. The results (errors, verifications, and warnings), from static (and possibly also dynamic checks) are returned to the IDE and presented as colorings and tooltips directly on the program text. This overall behavior is what we have called in our tool the "VeriFly" mode of operation.

Details on the architecture. Currently, `flycheck` requires saving the contents of the source being edited (Emacs *buffer*) into a temporary file and then invoking an

1 https://github.com/flycheck/flycheck

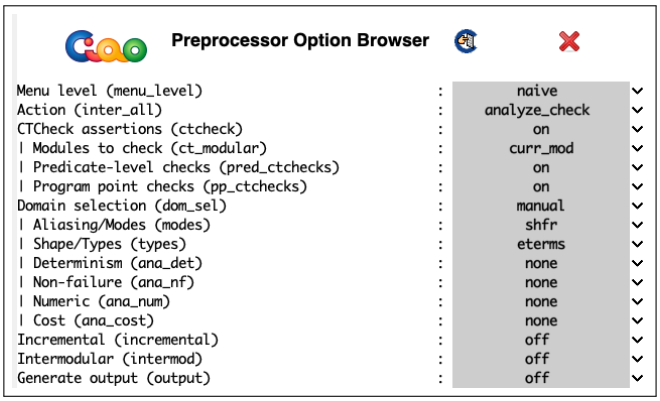


Fig. 26: The `CiaoPP` option browser.

external command. In our case the external command is a lightweight client that communicates with a running `CiaoPP` process, executing as a an *active module*, a daemon process that executes in the background and reacts to simple JSON-encoded queries via a socket.[2] This `CiaoPP` process is started once and kept alive for future analyses, ensuring that no time is unnecessarily wasted in startup and cleanups, as well as allowing caching some common analysis data, etc. for libraries. The approach similar to other like LSP (Language Server Protocol). Finally, `Ciao` implements a "shadow module" mechanism that allows the compiler to read alternative versions for some given modules. We use this mechanism to make `CiaoPP` (and other `Ciao`-based checkers) read the contents of temporary Emacs buffers during edition (saved as temporary files with *shadowed* names). This is specially useful to work in intermodular analysis where the analysis root is not necessarily the active buffer.

Customizing the analysis. In general, `CiaoPP` can be run fully automatically and does not *require* the user to change the configuration or provide invariants or assertions, and, for example, selects automatically abstract domains, as mentioned before. However, the system does have a configuration interface that allows manual selection of abstract domains, and of many parameters such as whether passes are incremental or not, the level of context sensitivity, error and warning levels, the type of output, etc. Fig. 26 shows the option browsing interface of the tool, as well as some options (abstract domain selections) in the menus for the cost analysis, shape and type analysis, pointer (logic variable) aliasing analysis, and numeric analysis.

2 JSON-encoded interaction capabilities have been added recently to support tool interoperability and browser-based interactions, and to simplify future extensions.

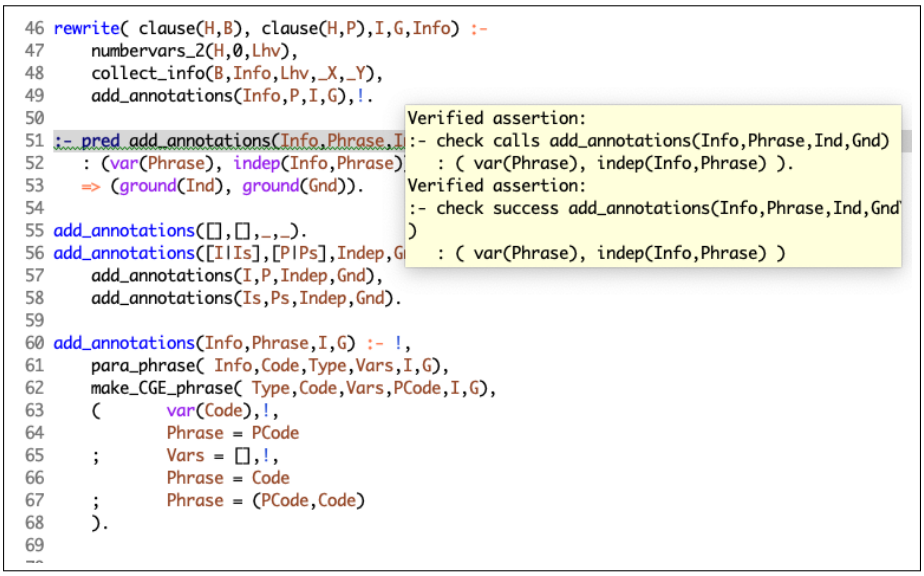


Fig. 27: An assertion within a parallelizer (`ann`).

9.2.1 *VeriFly in action*

We now show some simple examples of the system in action. Fig. 27 shows an assertion being verified within a medium-sized program implementing an automatic program parallelizer. The `add_annotations` loop traverses recursively a list of blocks and transforms sequential sections into parallel expressions. Upon opening the file the assertion is underlined in green, meaning that it has been verified (`checked` status). This ensures that upon entering the procedure there is no variable (pointer) sharing between `Info` (the input) and `Phrase`, i.e., `indep(Info,Phrase)`; that `Phrase` will arrive always as a free variable; and that on output from the procedure, `Ind` and `Gnd` will be ground terms (i.e., will contain no null pointers). Furthermore, this procedure is guaranteed not to fail. The corresponding information is also highlighted in the tool-tip (in yellow).

In Fig. 28 we show an implementation of quick-sort using open-ended ("difference") lists to construct the output lists (i.e., using pointers to append in constant time).

Examples 29, 30a, and 30b show static detection of, respectively, a property incompatibility bug, an illegal call to a library predicate, and a simple non-termination.

Fig. 31 shows naive reverse written using the functional notation library and illustrates the detection of an error regarding unintended behavior w.r.t. cost. The assertion in lines 5 and 6 of Fig. 31 (left) states that predicate `nrev` should be linear in the length of the (input) argument `A`. This is expressed by the property

```
1  :- module(_,[qsort/2],[assertions,nativeprops]).
2
3  :- entry qsort(X,Y) : (ground(X),list(X),var(Y)).
4
5  :- pred qsort(X,Y) => ground(Y).
6  qsort(X,Y) :- qsort_(X,Y,T), T=[].
7
8  :- pred qsort_(X,Y,Z) : (list(X),var(Y),var(Z),indep(Y,Z)) => ground(X).
9  qsort_([], Result, Result).
10 qsort_([First|Rest],ResultB,ResultE) :-
11     partition(Rest,First,Sm,Lg),
12     qsort_(Sm,SmB,SmE),
13     qsort_(Lg,LgB,LgE),
14     ResultB=SmB,
15     SmE=[First|LgB],
16     ResultE=LgE.
17
18 :- pred partition(L,P,Lg,Sm)
19    => (list(Lg), list(Sm), ground(Lg), ground(Sm)).
20 partition([],_,[],[]).
21 partition([X|Y],F,[X|Y1],Y2) :-
22     X @=< F,
23     partition(Y,F,Y1,Y2).
24 partition([X|Y],F,Y1,[X|Y2]) :-
25     X @> F,
26     partition(Y,F,Y1,Y2).
```

```
Verified assertion:
:- check calls qsort_(X,Y,Z)
   : ( list(X), var(Y), var(Z), indep(Y,Z) )
.
Verified assertion:
:- check success qsort_(X,Y,Z)
   : ( list(X), var(Y), var(Z), indep(Y,Z) )
```

Fig. 28: Sorting with incomplete data structures.

`steps_o(length(A))`, meaning that the cost, in terms of resolution steps, of any call to `nrev(A, B)` with `A` bound to a list and `B` a free variable, is in $O(\texttt{length(A)})$. However, this worst case asymptotic complexity stated in the user-provided assertion is incompatible with a safe, quadratic lower bound on the cost of such calls $(\frac{1}{2}\ length(A)^2 + \frac{3}{2}\ length(A) + 1)$ inferred by the static analyzer. In contrast, assertion in lines 5 and 6 of Fig. 32 (left) states that predicate `nrev` should have a quadratic worst case asymptotic complexity in the length of `A`, which is proved to hold by means of the upper-bound cost function inferred by analysis (which coincides with the lower bound above). Fig. 32 also shows the verification of determinacy, non-failure, and termination properties.

9.3 Some Performance Results

We provide some performance results from our tool using the well-known `chat-80` program[3], which contains 5.2k lines of Prolog code across 27 files, and uses a number of system libraries containing different Prolog built-ins and library predicates. The experiments consisted in opening a specific module in the IDE, and activating the checking of assertions with global analysis, i.e., analyzing the whole application as well as the libraries, and then performing a series of small edits, observing the total response time, i.e., the time from edit to graphical update of assertion coloring

3 https://github.com/ciao-lang/chat80

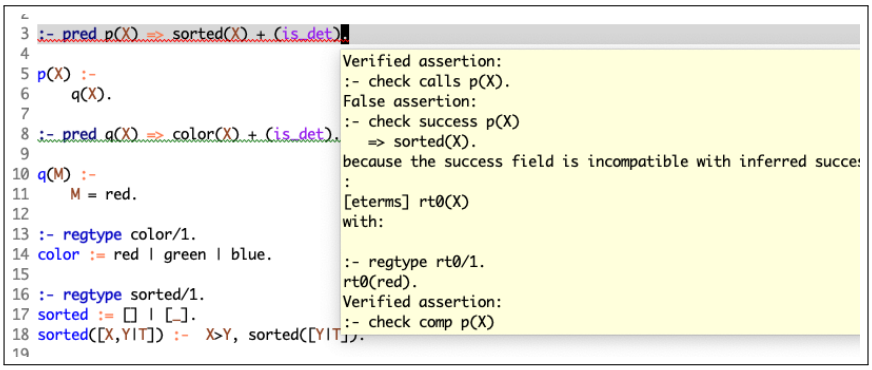
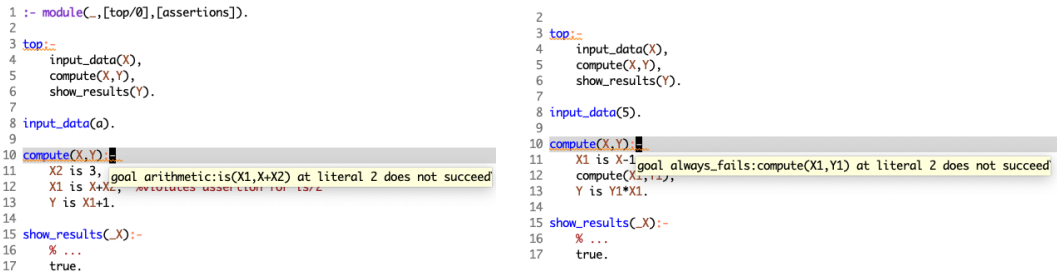


Fig. 29: A property incompatibility bug detected statically.



(a) Detection of illegal call to a library.

(b) Detection of simple non-termination.

Fig. 30: Static detection of bugs without the need for assertions.

in the IDE. Concretely, we performed two kinds of edits, predicate and assertions (**E1**), and only assertions (**E2**). The edits were performed on three selected files: `aggreg`, `readin`, and `talkr`. To study whether incrementality improves response times significantly, we included experiments enabling and disabling it. The experiments were performed in a MacBook Air with the Apple M1 chip and 16 GB of RAM. We evaluated the tool with three well-known abstract domains: a classic pair sharing [121] (`pairSh`), a dependency tracking via propositional clauses domain [53] (`def`), and sharing+groundness with clique-based widening [132] (`ShGrC`). The latter is the most precise, and, hence, the most expensive. We used sharing/groundness domains because they are known to be costly and at the same time, beyond mode inference, necessary to ensure correctness of most other analyses in (C)LP systems that support logical variables, and furthermore in any language that has pointers (aliasing).

Tables 9.2 and 9.1 show the response times of analyzing and checking assertions in experiments **E1** and **E2** respectively. For each of the files that were modified the table shows three columns: *noinc* is the response time in the non-incremental analysis

Fig. 31: Static verification of determinacy, termination, and cost (errors detected).

Fig. 32: Static verification of determinacy, termination, and cost (verified).

setting, *inc* is the response time in the incremental setting, and *speedup* is the speedup of *inc* vs. *noinc*. Each of the rows in the table correspond to each of the abstract domains. The reported time is the average of *total* roundtrip assertion checking time, *measured from the IDE*, that is, what the programmer actually perceives.

In all of the experiments incrementality provides significant speedups. In the first experiment, **E1** (Table 9.2), for the `pairSh` and `def` domains, it allows keeping the response time, crucial for the interactive use case of the analyzer that we propose, under 2 s. For `ShGrC`, a significant speedup is also achieved (more than $\times 3.5$). The response time is borderline at 5 *s*. The reason for this is, as mentioned, that the domain is more precise, an thus more computationally expensive, which in fact allows (dis)proving more assertions. For the **E2** experiment (Table 9.1), the tool detects that no changes are required in the analysis results, and the assertions can be rechecked w.r.t. the available (previous) analysis. The performance analyzing with `ShGrC` is improved significantly (more than $\times 9$) but, more importantly, the response times

	aggreg			readin			talkr		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
`pairSh`	2.8	1.6	×1.8	2.9	1.5	×1.9	2.8	1.6	×1.8
`def`	3.0	1.6	×1.9	2.7	1.5	×1.8	2.9	1.7	×1.7
`ShGrC`	18.1	5.1	×**3.5**	18.3	5.1	×**3.6**	18.1	4.5	×**4.0**

Table 9.1: Average response time (seconds) for the experiments only changing assertions.

	aggreg			readin			talkr		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
`pairSh`	2.8	1.7	×1.6	2.7	1.6	×1.7	2.9	1.7	×1.7
`def`	3.1	1.5	×2.0	2.9	1.4	×2.0	3.0	1.6	×1.9
`ShGrC`	18.2	2.0	×**9.1**	18.1	1.9	×**9.6**	18.2	1.9	×**9.6**

Table 9.2: Average response time (seconds) for the experiments with any program edit.

are around 2 *s*. All in all, the incremental features allow using many domains and, at least in some cases, even the most expensive domains.

Note that the experiments reveal also that an interesting configuration of this tool is to run different analyses in a portfolio, where which analyses to run is decided depending on the kind of change occurred. If only assertions have changed, it is enough to recheck only with `ShGrC`. However if both the code and the assertions changed, analysis for all domains can be run in parallel giving fast, less precise feedback to the programmer as the results are available, and then refine the results when the more precise results are ready.

Aside from the data in the tables, we observed a constant overhead of 0.4 s for loading the code—parsing and prior transformations—in the tool. This is currently still not fully incremental and has not been optimized yet to load only the parts that change. Verification times are negligible w.r.t. analysis times and are approx. 0.1 s; this is also non incremental, since we found that it is not currently a bottleneck, although we plan to make it incremental in the future to push the limits of optimizations forward.

9.4 Related work

The topic of assertion checking in logic programming, and in Prolog in particular, has received considerable attention. A family of approaches involves defining static type systems for logic programs [130, 104, 139] and several strongly-typed logic programming systems have been proposed, notable examples being Mercury [165] and Gödel [87]. Approaches for combining strongly typed and untyped Prolog modules were proposed in [162, 161]. Most of these approaches impose a number of restrictions that make them less appropriate for dynamic languages like Prolog. The `Ciao` model introduced an alternative for writing safe programs without relying on full static typing, but based instead the notions of safe approximations and abstract interpretation, providing a more general and flexible approach than in previous work, since assertions are optional and can contain any abstract property. This approach is specially useful for dynamic languages—see, e.g., [79] for a discussion of this topic. Some aspects of the `Ciao` model have been adopted or applied in other recent Prolog-based approaches, such as, e.g., [161, 174] or the library for run-time checking of assertions in SWI-Prolog. It can be considered an antecedent of the now popular *gradual-* and *hybrid-typing* approaches [57, 164, 151]. There has been work on incrementality within theorem proving-based verification approaches, such as, e.g., [157, 171], and within SMT-based approaches [56, 9]. Additional references can be found in the `CiaoPP` overview papers and the other citations provided.

9.5 Conclusion

We have shown how the integration of the `CiaoPP` static analysis and verification framework within an integrated development environment (IDE) can take advantage of incrementality to achieve a high level of reactivity at different levels of granularity. Our initial experience with this integrated tool shows quite promising results, with low latency times that provide early, continuous, and precise "on-the-fly" semantic feedback to programmers during the development process. This allows detecting many types of errors including swapped variables, property incompatibilities, illegal calls to library predicates, violated numeric constraints, unintended behavior w.r.t. termination, resource usage, determinism, covering and failure, etc. While presented using the Emacs and the `flycheck` package, we argue that our techniques and results should be applicable to any VeriFly-style integration into a modern extensible IDE. We plan to continue our work to achieve further reactivity and scalability improvements, enhanced presentations of verification results, and improved diagnosis, contributing to further improve the programming environments available to the (C)LP programmer.

10

Conclusion

Every time we get to the finish line they move it. Every time.

— Mary Jackson, *Hidden Figures* (2016)

Motivated by the increasing importance of the reliability of large software projects, this thesis has aimed to design and evaluate scalable static analysis algorithms that allow having semantic feedback during program development. The thesis is supported in the theory of abstract interpretation, concretely for logic programs, and aimed to provide a unified view of existing abstract interpretation-based algorithms together with some new proposed algorithms. The latter algorithms were implemented and evaluated and ultimately proven to be useful in the context of a novel interactive semantic program development tool.

The work in this thesis has been developed motivated by the importance of software in our daily lives. Its increased presence, and growing size and complexity, makes code reliability a key factor, from the point of view of guarantees for the client and software maintenance. As mentioned, formal methods-based techniques for automatic static analysis of programs offer reliability guarantees about all program executions, based on rigorous mathematical formalisms. These guarantees are obtained without having to actually execute the program, and without the need of user interaction. The reduced user interaction achievable by these techniques make them really good candidates for their use in the continuous integration/continuous deployment (CI/CD) pipelines as well as for tight integration with IDEs. However, the automatic nature of the techniques is a double edge sword. Being completely automatic logically means having a huge search space. So algorithms and abstractions need to be carefully designed. As already stressed, scalability of such tools is crucial, otherwise, the analysis result can become obsolete, if software changes occur faster than tools report. Our approach to tackle the scalability challenges is to reuse as much as possible previous analysis results, in order to avoid recomputing from scratch.

At this point, we would like to summarize the contributions of the thesis and relate them with these goals, as also stated in the introduction (Sec. 1.1), i.e., G1 to G4.

First, we studied the effects of the inherent precision loss of abstract interpretation on false alarms. To this end we proposed a classification of programs in terms of being complete/incomplete for an abstract interpretation (G4), based on the notion of *incompleteness/completeness cliques*. We proved that the equivalence induced by the abstract semantics on programs is an index set of partial recursive functions if and only if the abstraction is trivial. We considered the strongest possible scenario in order to establish when incompleteness can be injected. We proved that incompleteness can be injected in every program, even when the abstraction is designed to be the most precise one. We proved this using a transformation based on the structure of the abstract domain. This result has important consequences in program analysis and abstract interpretation: first, it shows that any non-trivial abstraction of extensional (functional) properties of programs is susceptible to their intensional structure. This means that any non-trivial abstract interpretation always unveils implicitly properties concerning the way programs are written. Second, program analysis behaves as other well known intensional properties of programs, like computational complexity. Lastly, we studied the class $\overline{\mathbb{C}}(P, \mathsf{A})$, which is the space of action of any code protecting transformations whose aim is to foil program analysis (and therefore foil any tool supporting reverse engineering). We proved that the set of all programs that are incomplete for any non-trivial abstraction is a Turing complete language. This means that it is possible to build a compiler that compiles any program P into an equivalent program that is incomplete for A, therefore justifying code transformations that protect code against program analysis. On the other side, the expressivity of the class $\mathbb{C}(\mathsf{A})$ of all programs that are complete for a non-trivial abstraction A is still obscure. We know that for terminating non-trivial program analyses we cannot find a many-to-one reduction of $\overline{\mathbb{C}}(P, \mathsf{A})$ into $\mathbb{C}(P, \mathsf{A})$. This implies that $\mathbb{C}(\mathsf{A})$ cannot be always Turing complete.

Second, we presented a new common framework for fixpoint computation algorithms for logic programs, in where we formalized the guarantees of existing algorithms. We extended these algorithms to obtain a modular and incremental context-sensitive algorithm (G1). Our algorithm takes care of propagating the fine-grain change information across module boundaries and implements all the actions required to recompute the analysis fixpoint incrementally after additions and deletions in the program. We have shown that the algorithm is correct and computes the most precise analysis for finite abstract domains, while supporting widening for dealing with infinite domains. We have also provided some new results for the baseline algorithms. Moreover, we have implemented and benchmarked the proposed approach within the `CiaoPP` framework, and our preliminary results show promising speedups

for programs of medium and larger size when compared to existing non-modular, fine-grain incremental analysis techniques, as well as improvements in memory consumption. In addition, the finer granularity of the proposed modular incremental fixpoint algorithm also brings improvements with respect to modular analysis alone (which only preserved analysis results at the module boundaries), producing better results even in the limit case of analyzing the whole program from scratch.

Furthermore, we proposed a user-guided multivariant incremental fixpoint algorithm that makes use of assertion information (G2), and we have provided formal results on the influence of such assertions on correctness. We have extended the semantics of the guidance (and all) assertions to deal with both the cases in which the program execution will and will not incorporate run-time tests for unverified assertions, as well as the cases in which the assertions are intended for refining the information or instead for losing precision in order to gain efficiency. We have shown that these annotations are not only useful when dealing with incomplete code but also provide the analyzer with recursion/loop invariants for speeding up global convergence.

Lastly, regarding G3, our preliminary results support the conclusion that the novel incremental analysis proposed brings promising analysis performance advantages for generic code using traits. The notion and encoding of traits used is very close to the underlying mechanisms used in other languages for implementing dynamic dispatch or run-time polymorphism (like Go's interfaces, Rust's traits, or a limited form of Haskell's type clases), so we believe that our techniques and results can be generalized to other languages. This also applies to our proposed algorithm for incremental analysis with assertion changes, which can be applied to different languages through the standard technique of translation to Horn-clause representation.

We closed this thesis with an application of incremental analysis in the context of *on-the-fly* assertion verification. We have shown how the `CiaoPP` static analysis and verification framework is connected within an integrated development environment (IDE) and takes advantage of incrementality to achieve a high level of reactivity at different levels of granularity. Our initial experience with the integrated tool shows quite promising results, with low latency times that provide early, continuous, and precise "on-the-fly" semantic feedback to programmers during the development process. This allows the early detection of many types of errors, including swapped variables, property incompatibilities, illegal calls to library predicates, violated numeric constraints, unintended behaviour w.r.t. termination, resource usage, determinism, covering and failure (exceptions), etc.

10.1 Future work

We want to end this thesis with some lines of future work.

Applications of fine-grain incremental analysis. Among the research lines that clearly follow this thesis, one key goal is to research how this technique is relevant to the numerous other applications of abstract interpretation.

- Improving performance when abstract interpretation is combined with program transformations or partial evaluation. A technique for performing both techniques in a tightly interconnected, interleaved fashion was proposed in [143], and shown to be more powerful than either technique alone or non-interleaved combinations.
- Semantic code search. In [63] we presented a technique for code search based on abstract interpretation. It consisted in, first, statically *preanalyzing* a source code database, e.g., the components of a software project, making a "*best effort*" on the precision of the analysis. Queries to the database can then made using assertions, to find predicates that meet such specifications, and this is done by checking the assertions against the preanalysis results. The preanalysis phase can clearly benefit from incremental algorithms as they improve the recomputation of this information when the source code changes. Second, when a query is made, the preanalysis may not be precise enough to find relevant information. If this is the case, the analysis result can be incrementally refined, reusing as much as possible the existing analysis so that more code can be found precisely to fit the assertion.
- Parallel fixpoint computation. Being able to isolate fixpoint computation of modular partitions of programs allows running the analysis of each of them in parallel. When the semantics of the called modules is available, it can be incrementally incorporated in the caller modules. This approach could be combined with the algorithm proposed recently in [98].

Heuristics for parameter tuning. We have observed that, depending on the program edit, with some settings of the algorithm the analysis is recomputed faster than with others. A shallow and syntactic characterization has been presented here but a more deep, and, probably, semantic characterization of such program changes could help in automatically selecting which reanalysis settings will work better.

Abstract domains amenable to incrementality. Related with the previous point, it may be worth studying which (abstract) data structures can make the proposed

algorithms run at their best performance. As an example of the worst case, if checking if a subgraph needs to be recomputed is as costly as recomputing it from scratch, then this abstract domain is not amenable to incrementality.

Providing incremental analysis and verification to programs in other source languages. Lastly, when analyzing CLP programs obtained by transforming a program in a different source language, incrementality-aware transformations can help obtain the maximum performance of the proposed techniques.

Bibliography

[1] S. Abramsky. Intensionality, definability and computation. In A. Baltag and S. Smets, editors, *Johan van Benthem on Logic and Information Dynamics*, pages 121–142. Springer, 2014.

[2] U. A. Acar. Self-adjusting computation: (an overview). In G. Puebla and G. Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 1–6. ACM, 2009.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.

[4] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Incremental Resource Usage Analysis. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 25–34. ACM Press, January 2012.

[5] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.

[6] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[7] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–576. Elsevier, 1990.

[8] S. Arzt and E. Bodden. 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, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 288–298. ACM, 2014.

[9] S. Asadi, M. Blicha, A. E. J. Hyvärinen, G. Fedyukovich, and N. Sharygina. Incremental Verification by SMT-based Summary Repair. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 77–82. IEEE, 2020.

[10] A. Asperti. The intensional content of rice's theorem. 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, California, USA, January 7-12, 2008*, pages 113–119. ACM, 2008.

[11] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In G. Levi and B. Steffen, editors, *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *LNCS*, pages 135–148. Springer, January 2004.

[12] G. Banda and J. P. Gallagher. Analysis of Linear Hybrid Systems in CLP. In M. Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008*, volume 5438 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2009.

[13] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[14] A. M. Ben-Amram and N. D. Jones. Computational complexity via programming languages: constant factors do matter. *Acta Inf.*, 37(2):83–120, 2000.

[15] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn Clause Solvers for Program Verification. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[16] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In Logozzo and Fähndrich [114], pages 105–125.

[17] A. Bossi, M. Gabbrieli, G. Levi, and M. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1,2):3–47, 1994.

[18] P. Bourdoncle. Interprocedural abstract interpretacion of block structured languages with nested procedures, aliasing and recursivity. In *Proceedings of the International Workshop PLILP'90*, number 456 in Lecture notes in Computer Science, pages 84–97. Springer–Verlag, 1990.

[19] C. Braem, B. L. Charlier, S. Modart, and P. V. Hentenryck. Cardinality analysis of Prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.

[20] R. Bruni, R. Giacobazzi, and R. Gori. Code obfuscation against abstraction refinement attacks. *Formal Asp. Comput.*, 30(6):685–711, 2018.

[21] R. Bruni, R. Giacobazzi, R. Gori, I. Garcia-Contreras, and D. Pavlovic. Abstract Extensionality – On the Properties of Incomplete Abstract Interpretations. In *Proc. ACM Symposium on Principles of Programming Languages 2020*, January 2020.

[22] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[23] F. Bueno, D. Cabeza, M. V. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[24] F. Bueno, M. G. de la Banda, M. V. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[25] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[26] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, 1990.

[27] D. Cabeza and M. V. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[28] M. Carroll and B. Ryder. Incremental data flow analysis via dominator and attribute updates. In *15th ACM Symposium on Principles of Programming Languages (POPL)*, pages 274–284. ACM Press, 1988.

[29] K. Chatterjee, B. Choudhary, and A. Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *PACMPL*, 2(POPL):30:1–30:30, 2018.

[30] M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 451–464, Charleston, South Carolina, 1993. ACM.

[31] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009.

[32] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN Notices **19**(6))*, pages 247–258. ACM Press, 1984.

[33] J. Correas, G. Puebla, M. V. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS, pages 163–178. Springer-Verlag, April 2006.

[34] N. Courant and C. Urban. Precise widening operators for proving termination by abstract interpretation. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.

[35] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In R. Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, Lecture Notes on Computer Science, pages 138–156. Springer Berlin Heidelberg, 2001.

[36] P. Cousot. Is Static Analysis Successful? In A. Farzan, editor, *Online seminar series on Verification beyond 2020.* U. Toronto, Canada, Tuesday, July 7 , 2020.

[37] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.

In *ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[38] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[39] P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Eleventh International Conference on Compiler Construction, CC 2002*, number 2304 in LNCS, pages 159–178. Springer, 2002.

[40] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *14th European Symposium on Programming, ESOP 2005*, pages 21–30, April 2005.

[41] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design (FMSD)*, 35(3):229–264, December 2009.

[42] P. Cousot, R. Giacobazzi, and F. Ranzato. Program analysis is harder than verification: A computability perspective. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2018.

[43] U. Dal Lago. A Short Introduction to Implicit Computational Complexity. In N. Bezhanishvili and V. Goranko, editors, *Lectures on Logic and Computation - ESSLLI 2010 Copenhagen, Denmark, August 2010, ESSLLI 2011, Ljubljana, Slovenia, August 2011, Selected Lecture Notes*, volume 7388 of *Lecture Notes in Computer Science*, pages 89–109. Springer, 2011.

[44] M. Dalla Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[45] E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, and M. Proietti. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *Theory and Practice of Logic Programming*, August 2021.

[46] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis*

of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, volume 8413 of *Lecture Notes in Computer Science*, pages 568–574. Springer, 2014.

[47] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In M. Falaschi and E. Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 91–102. ACM, 2015.

[48] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[49] S. Debray, P. Lopez-Garcia, and M. V. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[50] D. Delmas and J. Souyris. Astrée: From research to industry. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007*, pages 437–451, 2007.

[51] A. A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Professional Computing. Addison-Wesley, October 2015.

[52] S. Drape, C. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, editors, *ISC - Information Security*, volume 4779 of *Lecture Notes in Computer Science*, pages 299 – 314. Springer Verlag, 2007.

[53] V. Dumortier, G. Janssens, W. Simoens, and M. García de la Banda. Combining a Definiteness and a Freeness Abstraction for CLP Languages. In *Workshop on Logic Program Synthesis and Transformation*, 1993.

[54] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of prolog based static analyses. In M. Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2007.

[55] M. Fähndrich and F. Logozzo. Static Contract Checking with Abstract Interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software, FoVeOOS'10*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[56] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Incremental verification of compiler optimizations. In J. M. Badger and K. Y. Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 300–306. Springer, 2014.

[57] C. Flanagan. Hybrid Type Checking. In J. G. Morrisett and S. L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 245–256. ACM, 2006.

[58] Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[59] J. Gallagher, M. V. Hermenegildo, B. Kafle, M. Klemen, P. Lopez-Garcia, and J. Morales. From big-step to small-step semantics and back with interpreter specialization (invited paper). In *Proceedings of the Eighth International Workshop on Verification and Program Transformation (VPT 2020)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 50–65. Open Publishing Association (OPA), 2020. Co-located with ETAPS 2020.

[60] I. Garcia-Contreras, J. Morales, and M. V. Hermenegildo. Experiments in Context-Sensitive Incremental and Modular Static Analysis in CiaoPP. In *10th Workshop on Tools for Automatic Program Analysis (TAPAS'19)*, October 2019. (Extended Abstract).

[61] I. Garcia-Contreras, J. Morales, and M. V. Hermenegildo. Multivariant Assertion-based Guidance in Abstract Interpretation. In *Post-Proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18)*, number 11408 in LNCS, pages 184–201. Springer-Verlag, January 2019.

[62] I. Garcia-Contreras, J. Morales, and M. V. Hermenegildo. Incremental Analysis of Logic Programs with Assertions and Open Predicates. In *Proceedings of the 29th International Symposium on Logic-based Program Synthesis and*

Transformation (LOPSTR'19), volume 12042 of *LNCS*, pages 36–56. Springer, 2020.

[63] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Semantic Code Browsing. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue*, 16(5-6):721–737, September 2016.

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

[65] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of the 6th IEEE Int. Conferences on Software Engineering and Formal Methods (SEFM '08)*, pages 7–20. IEEE Press, 2008.

[66] R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63–72. ACM Press, 2012.

[67] R. Giacobazzi, F. Logozzo, and F. Ranzato. Analyzing program analyses. In Rajamani and Walker [149], pages 261–273.

[68] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In A. Miné and D. Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2012.

[69] R. Giacobazzi and I. Mastroeni. Making abstract models complete. *Mathematical Structures in Computer Science*, 26(4):658–701, 2016.

[70] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

[71] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.

[72] D. Gopan and T. Reps. Guided static analysis. In *International Static Analysis Symposium*, pages 349–365. Springer, 2007.

[73] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[74] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416. ACM, 2012.

[75] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, number 9206 in LNCS, pages 343–361. Springer, July 2015.

[76] R. Hähnle and M. Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In B. Steffen and G. J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 345–373. Springer, 2019.

[77] K. S. Henriksen and J. P. Gallagher. Abstract Interpretation of PIC Programs through Logic Programming. In *SCAM '06, Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.

[78] M. V. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[79] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla. The Ciao Approach to the Dynamic vs. Static Language Dilemma. In *Proceedings for the International Workshop on Scripts to Programs (STOP'11)*, New York, NY, USA, 2011. ACM.

[80] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.

[81] M. V. Hermenegildo and J. Morales. The LPdoc Documentation Generator. Ref. Manual (v3.0). Technical report, UPM, July 2011. Available at `http://ciao-lang.org`.

[82] M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[83] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[84] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[85] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

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

[87] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[88] P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *Haskell Special Issue, ACM Sigplan Notices*, 27(5):1–164, 1992.

[89] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[90] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 758–766. Springer, 2012.

[91] N. D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52(17(1)):307–339, 2004.

[92] B. Kafle, J. P. Gallagher, and J. F. Morales. RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 261–268. Springer, 2016.

[93] G. Kahn. Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, page 22–39, Berlin, Heidelberg, 1987. Springer-Verlag.

[94] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. JayHorn: A framework for verifying Java programs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 352–358. Springer, 2016.

[95] G. A. Kavvos. On the semantics of intensionality. In J. Esparza and A. S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 550–566, 2017.

[96] A. Kelly, K. Marriott, H. Søndergaard, and P. Stuckey. A generic object oriented incremental analyser for constraint logic programs. In *Proceedings of the 20th Australasian Computer Science Conference*, pages 92–101, 1997.

[97] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In L. J. Hendren, editor, *Compiler Construction, 17th International Conference, CC 2008, Budapest, Hungary, March 29 - April 6, 2008*, volume 4959 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2008.

[98] S. K. Kim, A. J. Venet, and A. V. Thakur. Deterministic parallel fixpoint computation. *Proc. ACM Program. Lang.*, 4(POPL):14:1–14:33, 2020.

[99] A. King, L. Lu, and S. Genaim. Detecting Determinacy in Prolog Programs. In S. Etalle and M. Truszczynski, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*,

volume 4079 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006.

[100] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, San Francisco, CA, USA, 2018.

[101] M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo. Static Performance Guarantees for Programs with Run-time Checks. In *20th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'18)*. ACM Press, September 2018.

[102] A. Krall and T. Berger. Incremental global compilation of Prolog with the vienna abstract machine. In *International Conference on Logic Programming*. MIT Press, June 1995.

[103] A. Krall and T. Berger. The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In M. G. de la Banda, G. Janssens, and P. Stuckey, editors, *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[104] T. Lakshman and U. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *International Logic Programming Symposium*. MIT Press, 1991.

[105] V. Laviron and F. Logozzo. Refining abstract interpretation-based static analyses with hints. In *Proc. of APLAS'09*, volume 5904 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, 2009.

[106] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo. Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks. In F. Fioravanti and J. P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*. Springer, 2018.

[107] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In M. V. Eekelen and U. D. Lago, editors, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, volume 9964 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2016.

[108] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In G. Gupta and R. Peña, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.

[109] Y. A. Liu, J. Brandvein, S. D. Stoller, and B. Lin. Demand-driven incremental object queries. In J. Cheney and G. Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 228–241. ACM, 2016.

[110] Y. A. Liu and S. D. Stoller. Dynamic Programming via Static Incrementalization. *High. Order Symb. Comput.*, 16(1-2):37–62, 2003.

[111] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static Caching for Incremental Computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, 1998.

[112] J. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[113] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, number 4349 in LNCS. Springer, Jan 2007.

[114] F. Logozzo and M. Fähndrich, editors. *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *LNCS*. Springer, 2013.

[115] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. *New Generation Computing*, 28(2):117–206, 2010.

[116] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, 18(2):167–223, March 2018. arXiv:1803.04451.

[117] M. Madsen, M. Yee, and O. Lhoták. From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, pages 194–208. ACM, 2016.

[118] A. Majumdar, C. D. Thomborson, and S. Drape. A survey of control-flow obfuscations. In A. Bagchi and V. Atluri, editors, *Information Systems Security, Second International Conference, ICISS 2006, Kolkata, India, December 19-21, 2006, Proceedings*, volume 4332 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2006.

[119] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–196. ACM Press, 1990.

[120] K. Marriott and H. Sondergaard. Bottom-up dataflow analysis of logic programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 733–748, Seattle,Washington, August 1988. MIT Press.

[121] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. Technical report 93/7, Univ. of Melbourne, 1993.

[122] K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.

[123] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in Lecture Notes in Computer Science, pages 154–168. Springer-Verlag, August 2007.

[124] E. Mera, P. Lopez-Garcia, M. Carro, and M. V. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[125] F. Mora, Y. Li, J. Rubin, and M. Chechik. Client-specific equivalence checking. In M. Huchard, C. Kästner, and G. Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 441–451. ACM, 2018.

[126] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989*

North American Conference on Logic Programming, pages 166–189. MIT Press, October 1989.

[127] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[128] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[129] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[130] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[131] M. Namolaru. Devirtualization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 125–133, 2006.

[132] J. Navas, F. Bueno, and M. V. Hermenegildo. Efficient Top-Down Set-Sharing Analysis Using Cliques. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL'06)*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[133] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.

[134] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[135] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.

[136] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In Logozzo and Fähndrich [114], pages 238–258.

[137] J. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.

[138] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J. Morales, and M. V. Hermenegildo. Cost Analysis of Smart Contracts via Parametric Resource Analysis. In D. Pichardie and M. Sighireanu, editors, *Proceedings of the 27th Static Analysis Symposium (SAS 2020)*, volume 12389 of *LNCS*, pages 7–31. Springer, November 2020.

[139] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[140] P. Pietrzak, J. Correas, G. Puebla, and M. V. Hermenegildo. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[141] G. Plotkin. A structural approach to operational semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[142] L. Pollock and M. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, 1989.

[143] G. Puebla, E. Albert, and M. V. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *The 13th International Static Analysis Symposium (SAS'06)*, number 4134 in LNCS, pages 107–126. Springer, August 2006.

[144] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[145] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[146] G. Puebla, J. Correas, M. V. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[147] G. Puebla and M. V. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, September 1996.

[148] G. Puebla and M. V. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[149] S. K. Rajamani and D. Walker, editors. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015.

[150] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, Charleston, South Carolina, 1993. ACM.

[151] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In Rajamani and Walker [149], pages 167–180.

[152] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[153] H. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[154] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[155] H. Rogers. *Theory of recursive functions and effective computability*. The MIT press, 1992.

[156] B. Rosen. Linear cost is sometimes quadratic. In *Eighth ACM Symposium on Principles of Programming Languages (POPL)*, pages 117–124. ACM Press, 1981.

[157] K. Rustan, M. Leino, and V. Wüstholz. Fine-grained caching of verification results. In D. Kroening and C. S. Pasareanu, editors, *Proc. of the 27th International Conference on Computer Aided Verification, CAV 2015*, volume 9206 of *LNCS*, pages 380–397. Springer, July 2015.

[158] B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.

[159] B. Ryder, T. Marlowe, and M. Paull. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, 1988.

[160] M. A. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M. Hermenegildo. VeriFly: On-the-fly Assertion Checking with CiaoPP. In *6th Workshop on Formal Integrated Development Environment (F-IDE 2021)*, May 2021.

[161] T. Schrijvers, V. S. Costa, J. Wielemaker, and B. Demoen. Towards typed prolog. In M. G. de la Banda and E. Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 693–697. Springer, 2008.

[162] T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards Typed Prolog. In E. Pontelli and M. M. G. de la Banda, editors, *International Conference on Logic Programming*, number 5366 in LNCS, pages 693–697. Springer Verlag, December 2008.

[163] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences., 1978.

[164] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[165] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[166] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis. *Science of Computer Programming*, 155:3–26, April 2018. Selected and Extended papers from the 2016 International Symposium on Principles and Practice of Declarative Programming.

[167] T. Swift. Incremental Tabling in Support of Knowledge Representation and Reasoning. *Theory and Practice of Logic Programming*, 14(4-5):553–567, 2014.

[168] T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, Jan 2012.

[169] T. Szabó, S. Erdweg, and M. Voelter. 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. ACM, 2016.

[170] M. Thakur and V. K. Nandivada. Mix your contexts well: Opportunities unleashed by recent advances in scaling context-sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, pages 27–38, New York, NY, USA, 2020. Association for Computing Machinery.

[171] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In G. Barthe, A. Pardo, and G. Schneider, editors, *Proc. of the 9th International Conference on Software Engineering and Formal Methods, SEFM 2011*, volume 7041 of *LNCS*, pages 382–398. Springer, November 2011.

[172] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[173] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[174] I. Wingen and P. Körner. Effectiveness of annotation-based static type inference. In M. Hanus and C. S. Coen, editors, *Functional and Constraint Logic Programming - 28th International Workshop, WFLP 2020, Bologna, Italy, September 7, 2020, Revised Selected Papers*, volume 12560 of *Lecture Notes in Computer Science*, pages 74–93. Springer, 2020.

[175] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[176] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of

Lecture Notes in Computer Science, pages 414–432. Springer-Verlag, Berlin, 1999.

[177] Y. Zhang and Y. A. Liu. Automating Derivation of Incremental Programs. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, page 350. ACM, 1998.

A

Additional plots

A.1 Additional experimental results

A.1.1 Detailed analysis times per step for analysis with `def`

A.1.2 Average analysis times split by domain

	mon			mon-inc			mod			mod-inc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	1.8	4.1	1.1	1.6	4.0	1.1	1.9	16.6	0.5	1.3	10.7	0.5
ann	4.0	28.0	1.4	3.1	15.2	1.9	1.3	32.4	0.3	1.3	27.4	0.5
bid	2.5	10.0	1.4	1.9	5.5	1.5	1.8	22.5	0.5	1.6	19.6	0.5
boyer	7.6	13.2	1.1	2.6	4.1	1.4	2.2	6.4	0.3	1.5	3.9	0.6
check_links	7.2	98.1	2.2	13.2	96.2	5.7	3.5	221.6	0.4	5.7	212.2	0.8
cleandirs	7.2	22.6	1.1	4.4	16.1	1.7	5.4	86.4	0.7	3.0	54.1	0.6
hanoi	1.2	1.8	0.8	1.2	2.1	0.8	1.3	4.7	0.4	1.4	4.5	0.5
manag_proj	11.4	39.0	4.4	7.1	20.6	4.5	4.6	56.4	0.4	3.2	42.6	0.5
peephole	3.7	20.1	1.5	2.3	7.0	1.6	1.1	25.5	0.4	1.0	20.4	0.4
progeom	1.9	3.9	0.9	1.5	2.7	0.9	2.2	10.0	0.7	2.2	10.1	0.9
read	9.6	31.8	1.3	4.7	20.5	1.3	7.1	32.9	0.8	4.9	28.0	0.9
qsort	2.0	4.0	1.2	1.5	2.6	1.2	1.5	7.3	0.5	1.5	7.2	0.6
rdtok	5.1	13.9	1.5	5.1	21.9	1.3	3.8	12.8	0.5	4.8	23.8	0.4
warplan	6.0	13.5	0.9	2.3	4.4	1.2	3.4	10.6	0.6	2.1	6.3	1.0
witt	5.9	27.1	1.4	3.6	16.5	2.3	3.0	59.8	0.3	2.2	47.1	0.4

Table A.1: Analysis time (ms) per benchmark of the *add* experiment (`pdb`).

	mon			mon-inc			mon-scc			mod			mod-inc			mod-scc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.1	4.3	1.3	2.0	5.7	1.3	1.4	5.7	1.0	1.8	13.6	0.4	1.9	13.2	0.5	1.6	13.1	0.4
ann	4.0	29.4	1.2	3.6	32.7	1.8	2.8	33.3	1.5	2.4	48.4	0.3	1.9	57.2	0.6	1.9	57.8	0.5
bid	2.6	8.7	1.8	2.4	12.8	1.8	1.7	10.1	1.3	1.8	35.7	0.5	1.9	33.8	0.5	1.3	28.4	0.3
boyer	7.7	14.7	1.0	6.9	15.1	1.1	1.8	14.5	1.0	2.1	12.5	0.3	2.5	15.9	0.3	1.2	15.3	0.3
check_links	7.4	99.3	1.8	20.0	232.9	5.7	17.9	109.9	5.5	3.7	277.0	0.3	10.0	255.1	0.8	9.2	242.6	0.6
cleandirs	7.5	28.9	1.4	5.3	28.5	1.5	3.1	25.4	1.3	4.8	98.4	0.5	3.6	78.1	0.6	3.1	76.8	0.5
hanoi	1.5	3.1	0.7	1.2	2.5	0.8	1.0	2.5	0.7	1.4	5.0	0.4	1.7	6.3	0.6	1.4	6.0	0.4
manag_proj	12.0	40.4	4.3	7.7	46.3	4.1	7.1	41.7	4.3	4.4	104.8	0.3	3.5	94.4	0.4	3.3	94.3	0.4
peephole	3.3	17.2	1.4	2.6	22.5	1.6	1.9	22.6	1.2	2.3	35.4	0.7	1.9	35.5	0.8	1.6	33.1	0.7
progeom	2.1	4.1	1.2	1.8	5.1	1.1	1.1	4.9	0.8	2.6	20.9	0.9	2.6	18.7	0.9	2.6	23.9	0.6
read	9.1	28.5	1.1	10.5	35.7	1.1	2.0	30.7	1.0	6.2	38.9	0.5	8.4	45.6	0.6	2.9	42.6	0.5
qsort	1.9	3.7	1.2	1.8	4.6	1.1	1.2	4.4	0.9	1.9	11.3	0.5	1.8	10.9	0.5	1.6	11.4	0.5
rdtok	4.9	14.2	1.5	5.3	17.5	1.5	2.0	15.6	1.3	4.0	36.1	0.4	4.9	32.6	0.4	3.0	35.8	0.3
warplan	5.8	16.2	1.0	3.5	14.4	1.3	1.6	15.1	0.8	3.7	29.4	0.8	2.8	23.6	0.9	1.8	22.6	0.7
witt	6.1	26.1	1.4	4.5	32.3	2.2	3.8	32.9	2.0	3.4	90.1	0.3	2.6	70.0	0.4	2.4	69.1	0.4

Table A.2: Analysis time (ms) per benchmark of the *del* experiment (pdb).

	mon			mon-inc			mod			mod-inc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.5	6.9	1.2	1.9	6.5	1.2	2.1	18.3	0.5	2.2	19.0	0.6
ann	5.2	50.0	1.0	3.4	36.5	1.7	2.1	98.5	0.3	1.8	83.9	0.5
bid	3.1	14.8	1.7	2.3	8.7	1.8	2.4	38.8	0.5	2.2	26.9	0.5
boyer	16.0	26.1	1.4	3.1	11.2	1.5	3.3	11.9	0.5	1.7	11.2	0.6
check_links	8.8	158.7	2.0	13.6	160.8	5.3	5.3	312.6	0.3	6.1	278.1	0.7
cleandirs	5.2	23.5	1.3	3.3	17.7	1.3	3.6	72.5	0.3	3.1	64.1	0.6
hanoi	1.5	2.3	0.9	1.4	2.9	0.9	2.0	7.0	0.5	1.4	5.6	0.4
manag_proj	6.9	13.4	5.0	5.9	10.8	4.3	2.8	10.3	0.3	2.2	12.1	0.4
peephole	6.8	42.0	1.7	2.3	20.2	1.4	1.6	62.4	0.4	1.5	43.9	0.5
progeom	2.1	4.8	0.9	1.4	2.6	0.8	2.1	9.9	0.7	2.0	7.7	0.7
read	23.3	102.8	1.1	8.7	65.6	1.1	11.1	100.7	0.6	7.5	81.4	0.8
qsort	2.7	7.2	1.1	1.7	3.5	1.0	2.9	12.2	0.5	2.0	10.0	0.4
rdtok	7.4	26.0	1.2	5.8	37.4	1.4	5.1	23.7	0.4	5.6	39.3	0.5
warplan	8.6	26.4	1.0	2.9	7.9	1.2	4.7	21.3	0.6	2.7	11.8	1.2
witt	1.6	2.7	1.2	3.0	7.0	2.0	0.7	2.4	0.3	1.1	2.5	0.4

Table A.3: Analysis time (ms) per benchmark of the *add* experiment (gr).

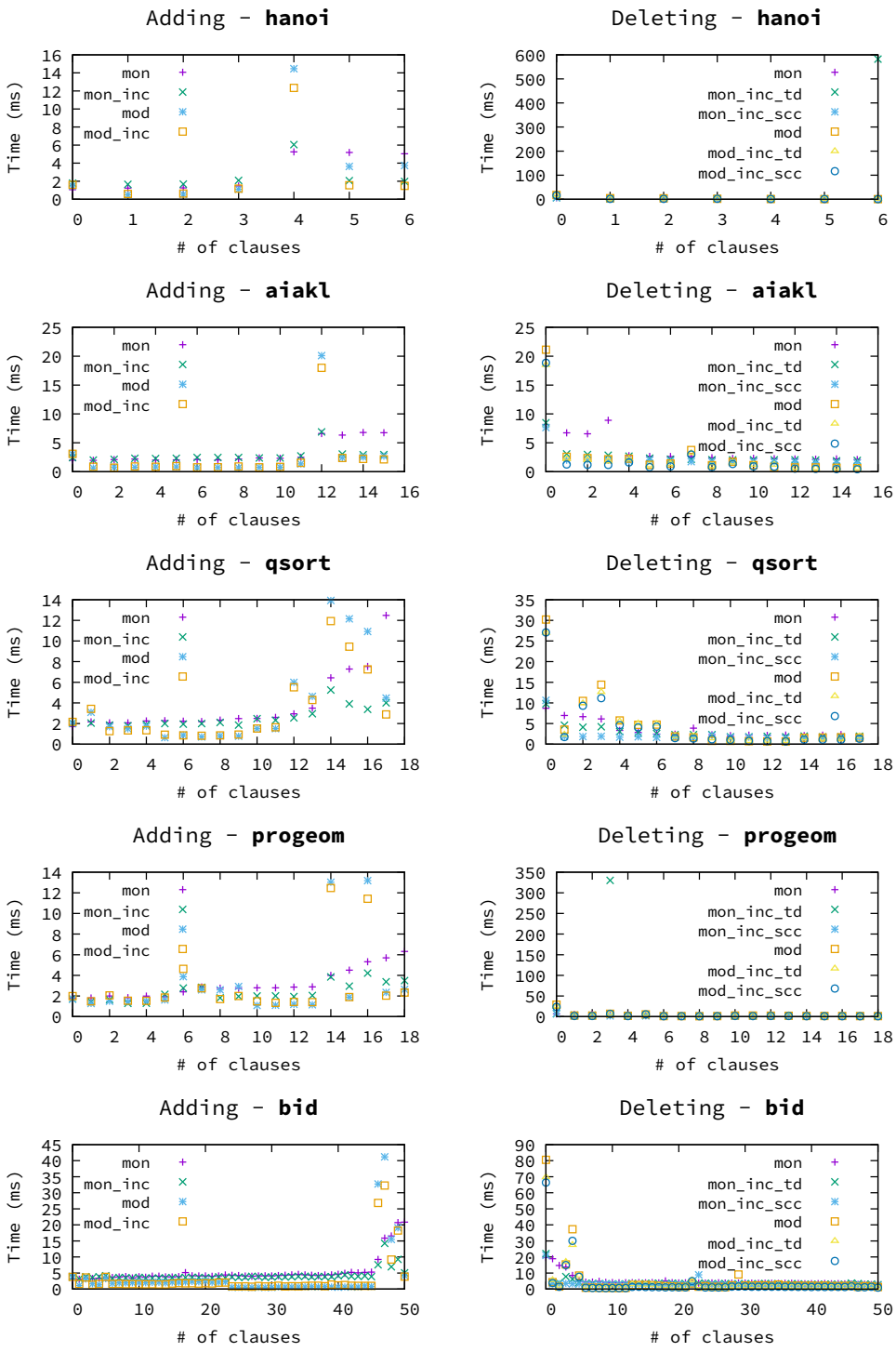


Fig. 33: Analysis times (ms) for both experiments with `def` for *smaller* benchmarks.

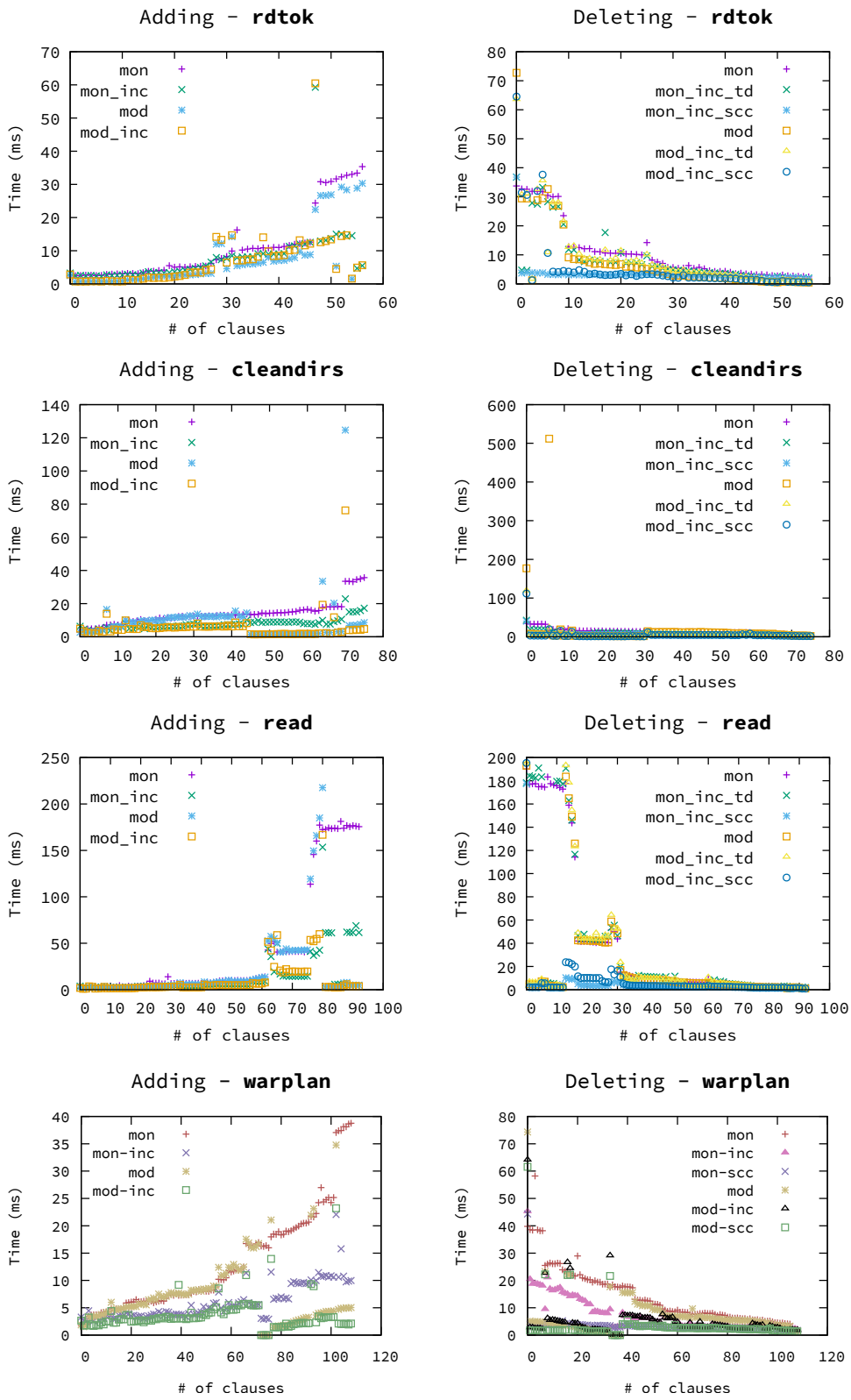


Fig. 34: Analysis times (ms) for both experiments with `def` for *larger* benchmarks (1).

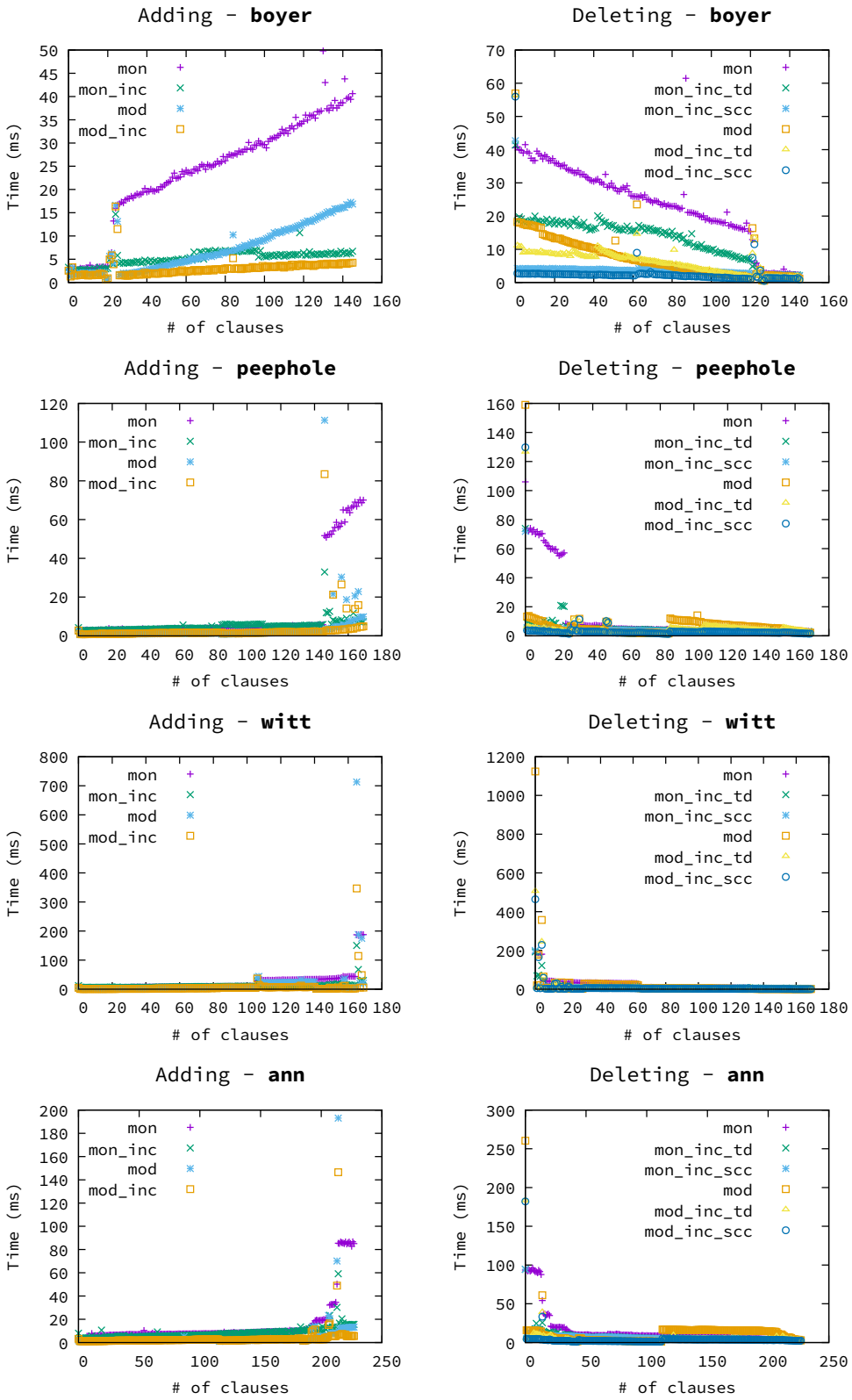


Fig. 35: Analysis times (ms) for both experiments with `def` for *larger* benchmarks (2).

	mon			mon-inc			mon-scc			mod			mod-inc			mod-scc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.3	5.9	1.1	2.1	8.0	1.3	1.5	7.9	0.9	2.2	20.8	0.4	1.9	16.2	0.4	1.9	18.3	0.5
ann	5.9	74.0	1.2	3.9	59.0	1.8	3.2	55.2	1.5	6.5	134.6	0.3	3.1	127.1	0.6	2.5	121.3	0.5
bid	2.8	12.9	1.8	2.2	16.0	1.5	1.9	15.8	1.4	2.3	45.2	0.4	2.8	44.8	0.5	1.9	39.3	0.3
boyer	15.7	30.5	1.3	10.3	26.2	1.1	2.0	23.6	1.0	3.1	33.1	0.3	3.4	36.7	0.4	1.6	35.6	0.5
check_links	10.0	175.5	1.6	22.9	367.2	5.9	20.0	186.0	5.6	6.4	411.7	0.5	12.9	335.0	0.8	12.9	337.4	0.6
cleandirs	5.7	23.0	1.5	4.3	26.5	1.8	2.9	26.8	1.3	4.1	95.6	0.4	3.3	83.3	0.5	2.9	82.1	0.4
hanoi	1.7	2.6	1.0	1.4	2.7	1.0	1.0	2.7	0.6	1.7	6.4	0.3	1.8	6.4	0.4	1.8	7.9	0.5
manag_proj	7.1	14.0	5.1	6.7	20.4	4.7	5.9	20.1	4.0	2.6	29.5	0.3	2.3	35.1	0.3	2.2	36.9	0.3
peephole	7.2	50.1	1.7	3.0	46.5	1.5	2.2	47.2	1.4	4.0	66.0	1.0	3.1	61.6	1.0	2.4	61.9	1.0
progeom	2.1	4.5	0.9	1.8	5.9	0.9	1.1	5.4	0.8	2.7	21.0	0.7	2.5	19.2	0.7	2.3	19.4	0.7
read	22.6	109.2	1.0	22.6	118.7	1.3	2.9	101.2	0.9	11.0	110.8	0.6	13.2	113.8	0.5	4.5	113.6	0.4
qsort	2.5	7.0	0.9	2.3	8.4	1.2	1.4	8.7	0.8	3.5	21.5	0.4	2.6	17.2	0.4	2.4	16.9	0.4
rdtok	7.5	29.3	1.4	6.1	27.8	1.2	2.1	26.7	1.1	5.3	53.5	0.3	6.4	44.9	0.4	3.4	43.7	0.3
warplan	9.0	23.1	1.0	4.8	25.4	1.3	1.9	29.6	0.9	4.9	48.1	0.8	3.8	39.6	1.2	2.4	37.2	0.7
witt	2.0	3.0	1.4	3.5	8.4	2.2	3.1	8.4	1.9	0.9	8.2	0.3	1.2	5.3	0.4	1.0	5.2	0.3

Table A.4: Analysis time (ms) per benchmark of the *del* experiment (gr).

	mon			mon-inc			mod			mod-inc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.6	6.8	1.1	1.8	7.1	1.2	1.7	15.2	0.4	1.6	13.8	0.4
ann	10.3	123.2	1.3	3.9	84.5	2.1	4.2	201.9	0.4	2.8	175.9	0.6
bid	4.2	22.6	2.1	3.1	13.9	2.2	2.9	39.0	0.4	2.1	26.3	0.4
boyer	31.2	47.4	1.2	3.9	24.9	1.3	6.6	28.9	0.5	2.0	26.4	0.5
check_links	34.0	709.6	2.6	17.5	704.8	6.0	28.1	2,012.9	0.3	10.8	1,002.6	0.8
cleandirs	38.0	418.1	1.1	9.0	353.6	2.3	47.3	1,108.3	0.7	12.9	740.4	0.8
hanoi	4.9	9.6	1.3	3.1	10.2	1.6	5.7	22.6	0.5	3.8	18.0	0.6
manag_proj	1,151.8	17,937.2	13.4	186.2	15,469.2	12.5	43.8	1,589.1	0.3	17.1	1,086.8	0.4
peephole	19.7	155.0	1.8	3.6	58.9	1.7	4.1	167.2	0.3	3.1	120.1	0.5
progeom	2.9	6.6	1.2	1.7	3.3	1.0	2.7	12.3	0.7	2.1	9.4	0.7
read	84.3	413.2	1.1	24.6	321.8	1.3	37.5	424.4	1.0	16.7	341.4	0.7
qsort	2.6	9.1	0.9	1.8	4.9	1.0	3.2	12.6	0.4	2.5	10.2	0.6
rdtok	14.1	61.1	1.5	8.4	67.0	1.3	10.5	58.9	0.5	8.0	69.7	0.5
warplan	15.9	61.1	0.9	5.3	39.1	1.2	9.1	111.9	0.9	3.8	61.1	1.1
witt	140.4	2,494.9	1.4	28.0	1,759.9	2.1	128.3	10,786.0	0.3	20.6	2,062.8	0.4

Table A.5: Analysis time (ms) per benchmark of the *add* experiment (shfr).

	mon			mon-inc			mon-scc			mod			mod-inc			mod-scc		
bench	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min	mean	max	min
aiakl	2.6	7.4	1.2	2.0	8.2	1.2	1.5	8.1	1.0	1.6	15.0	0.3	1.8	16.3	0.4	1.5	15.1	0.3
ann	10.1	123.6	1.3	4.3	117.9	2.0	3.1	117.2	1.5	4.5	260.4	0.3	2.9	232.0	0.6	2.3	232.1	0.6
bid	3.9	23.4	2.1	2.6	23.8	1.7	2.2	23.2	1.5	2.8	62.1	0.3	2.7	54.4	0.4	2.4	52.3	0.3
boyer	32.8	50.6	1.6	14.0	62.3	1.3	2.4	50.8	1.1	6.7	60.7	0.4	5.5	61.6	0.5	1.8	60.8	0.3
check_links	33.5	740.9	2.0	36.5	1,799.2	6.0	19.3	728.9	5.5	28.9	2,091.5	0.3	22.4	1,362.3	0.7	9.9	1,126.0	0.6
cleandirs	38.0	388.9	1.1	10.8	381.9	2.2	8.2	386.7	1.5	52.1	1,169.5	0.9	18.4	782.8	0.8	18.4	813.0	0.7
hanoi	5.0	9.6	1.4	3.0	10.3	1.4	2.7	10.5	1.2	5.2	20.8	0.4	3.7	18.3	0.4	4.3	21.9	0.6
manag_proj	1,131.4	18,049.1	12.8	229.0	18,121.6	12.9	145.1	18,081.0	12.5	366.6	37,511.5	0.3	321.2	34,731.8	0.4	322.5	34,924.4	0.4
peephole	19.2	158.2	1.6	4.9	212.4	1.6	3.0	156.4	1.4	5.2	386.1	0.4	3.7	273.3	0.5	3.4	272.6	0.5
progeom	2.7	6.0	1.0	2.3	7.7	1.1	1.4	7.7	0.9	3.5	29.0	0.8	3.0	24.4	0.8	2.5	22.8	0.6
read	85.9	415.5	1.3	71.5	423.7	1.6	6.2	390.0	1.0	37.1	418.6	0.8	38.9	435.4	0.7	8.1	408.5	0.5
qsort	2.9	9.7	1.3	2.5	11.1	1.1	1.4	10.6	0.8	3.4	22.4	0.3	3.9	24.8	0.6	3.5	25.0	0.4
rdtok	13.1	60.4	1.4	12.0	63.4	1.6	2.9	67.4	1.3	11.5	117.7	0.3	12.4	88.4	0.6	7.9	91.6	0.4
warplan	16.3	61.6	1.1	7.7	64.2	1.1	2.5	65.7	1.0	9.6	159.7	0.9	5.6	109.0	1.2	3.9	114.6	0.9
witt	137.9	2,447.8	1.3	38.8	2,424.4	2.2	17.6	2,424.8	1.8	192.5	19,997.6	0.3	49.0	3,144.8	0.4	49.1	3,192.7	0.4

Table A.6: Analysis time (ms) per benchmark of the *del* experiment (`shfr`).

A.1.3 Speedup split by domain

	addition: mod-inc			deletion: mod-scc				
bench	vs. mon	vs. mon-inc	vs. mod	vs. mon	vs. mon-inc	vs. mon-scc	vs. mod	vs. mod-inc
aiakl	1.4	1.2	1.4	1.3	1.3	0.9	1.1	1.2
ann	3.0	2.3	0.9	2.1	1.9	1.5	1.3	1.0
bid	1.6	1.2	1.1	1.9	1.8	1.3	1.4	1.5
boyer	5.1	1.8	1.5	6.5	5.8	1.5	1.8	2.1
check_links	1.3	2.3	0.6	0.8	2.2	1.9	0.4	1.1
cleandirs	2.4	1.5	1.8	2.4	1.7	1.0	1.5	1.2
hanoi	0.9	0.9	1.0	1.1	0.9	0.7	1.0	1.2
manag_proj	3.5	2.2	1.5	3.6	2.3	2.1	1.3	1.0
peephole	3.7	2.3	1.1	2.1	1.7	1.2	1.4	1.2
progeom	0.9	0.7	1.0	0.8	0.7	0.4	1.0	1.0
read	2.0	1.0	1.5	3.1	3.6	0.7	2.1	2.9
qsort	1.3	1.0	1.0	1.2	1.1	0.7	1.2	1.1
rdtok	1.1	1.1	0.8	1.6	1.8	0.7	1.3	1.6
warplan	2.8	1.1	1.6	3.2	1.9	0.9	2.0	1.5
witt	2.6	1.6	1.3	2.6	1.9	1.6	1.4	1.1

Table A.7: Speedups of the clause *addition* (left) and *deletion* (right) experiments (pdb).

	addition: mod-inc			deletion: mod-scc				
bench	vs. mon	vs. mon-inc	vs. mod	vs. mon	vs. mon-inc	vs. mon-scc	vs. mod	vs. mod-inc
aiakl	1.1	0.9	0.9	1.2	1.1	0.8	1.1	1.0
ann	2.9	1.9	1.1	2.4	1.6	1.3	2.6	1.2
bid	1.4	1.1	1.1	1.5	1.1	1.0	1.2	1.4
boyer	9.2	1.8	1.9	9.8	6.4	1.2	1.9	2.1
check_links	1.4	2.2	0.9	0.8	1.8	1.5	0.5	1.0
cleandirs	1.7	1.1	1.2	1.9	1.5	1.0	1.4	1.1
hanoi	1.0	1.0	1.4	1.0	0.8	0.5	1.0	1.0
manag_proj	3.1	2.7	1.3	3.2	3.0	2.6	1.2	1.0
peephole	4.6	1.5	1.1	3.0	1.2	0.9	1.6	1.3
progeom	1.1	0.7	1.0	0.9	0.8	0.5	1.2	1.1
read	3.1	1.2	1.5	5.0	5.0	0.6	2.4	2.9
qsort	1.3	0.8	1.5	1.0	1.0	0.6	1.5	1.1
rdtok	1.3	1.0	0.9	2.2	1.8	0.6	1.6	1.9
warplan	3.2	1.1	1.7	3.7	2.0	0.8	2.0	1.6
witt	1.5	2.8	0.7	2.0	3.4	3.0	0.9	1.2

Table A.8: Speedups of the clause *addition* (left) and *deletion* (right) experiments (gr).

	addition: `mod-inc`			deletion: `mod-scc`				
`bench`	vs. `mon`	vs. `mon-inc`	vs. `mod`	vs. `mon`	vs. `mon-inc`	vs. `mon-scc`	vs. `mod`	vs. `mod-inc`
`aiakl`	1.7	1.2	1.1	1.8	1.3	1.0	1.1	1.2
`ann`	3.7	1.4	1.5	4.4	1.9	1.4	2.0	1.3
`bid`	2.0	1.5	1.4	1.6	1.1	0.9	1.2	1.1
`boyer`	16.0	2.0	3.4	17.8	7.6	1.3	3.6	3.0
`check_links`	3.1	1.6	2.6	3.4	3.7	2.0	2.9	2.3
`cleandirs`	2.9	0.7	3.7	2.1	0.6	0.4	2.8	1.0
`hanoi`	1.3	0.8	1.5	1.2	0.7	0.6	1.2	0.9
`manag_proj`	67.2	10.9	2.6	3.5	0.7	0.4	1.1	1.0
`peephole`	6.4	1.2	1.3	5.7	1.5	0.9	1.5	1.1
`progeom`	1.4	0.8	1.3	1.1	0.9	0.6	1.4	1.2
`read`	5.0	1.5	2.2	10.6	8.8	0.8	4.6	4.8
`qsort`	1.0	0.7	1.3	0.8	0.7	0.4	1.0	1.1
`rdtok`	1.8	1.0	1.3	1.7	1.5	0.4	1.5	1.6
`warplan`	4.2	1.4	2.4	4.2	2.0	0.6	2.5	1.4
`witt`	6.8	1.4	6.2	2.8	0.8	0.4	3.9	1.0

Table A.9: Speedups of the clause *addition* (left) and *deletion* (right) experiments (`shfr`).

A.1.4 Accummulated analysis times

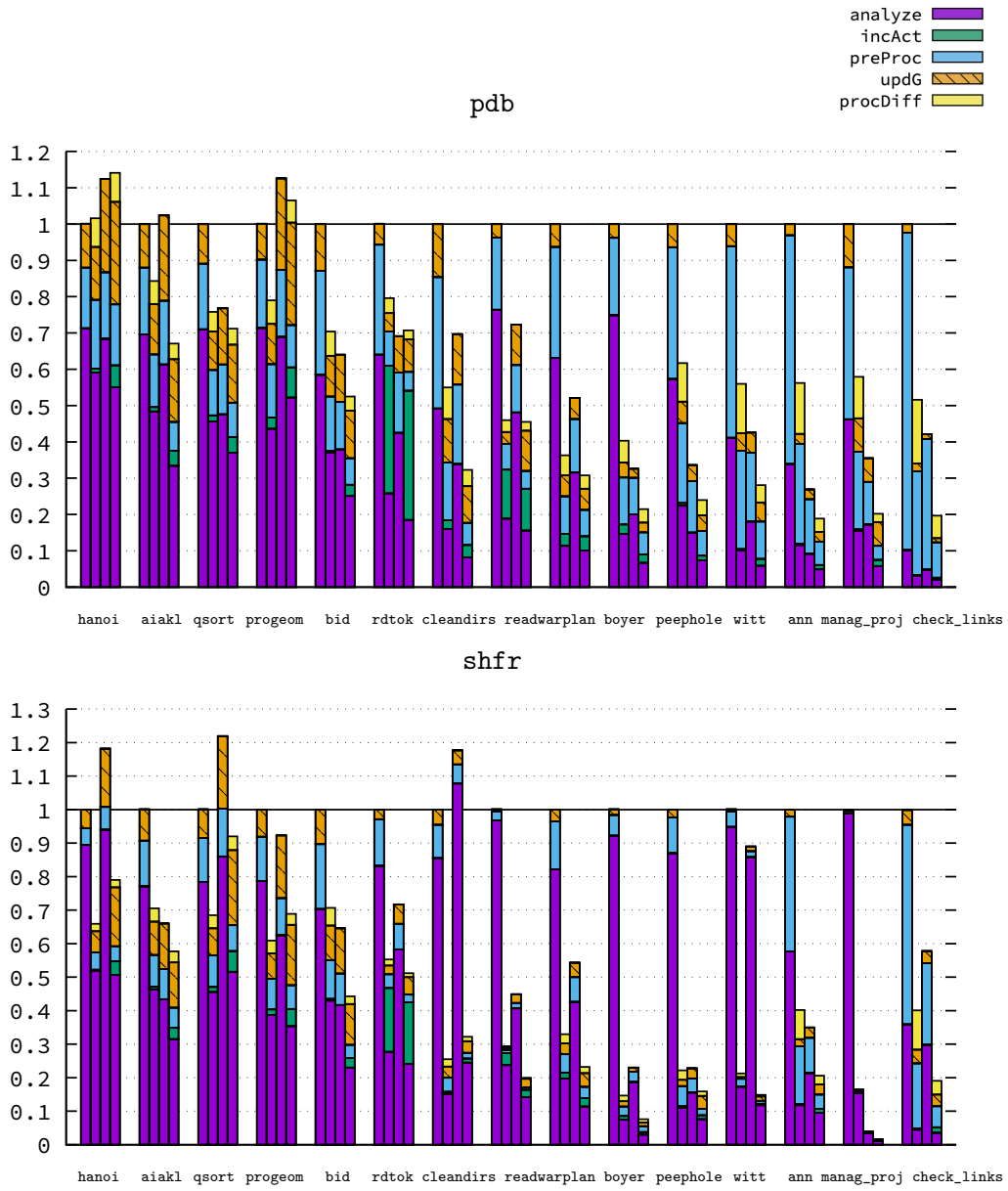


Fig. 36: Accumulated analysis time (normalized w.r.t `mon`) adding clauses. The order inside each set of bars is: |`mon`|`mon_inc`|`mod`|`mod_inc`|.

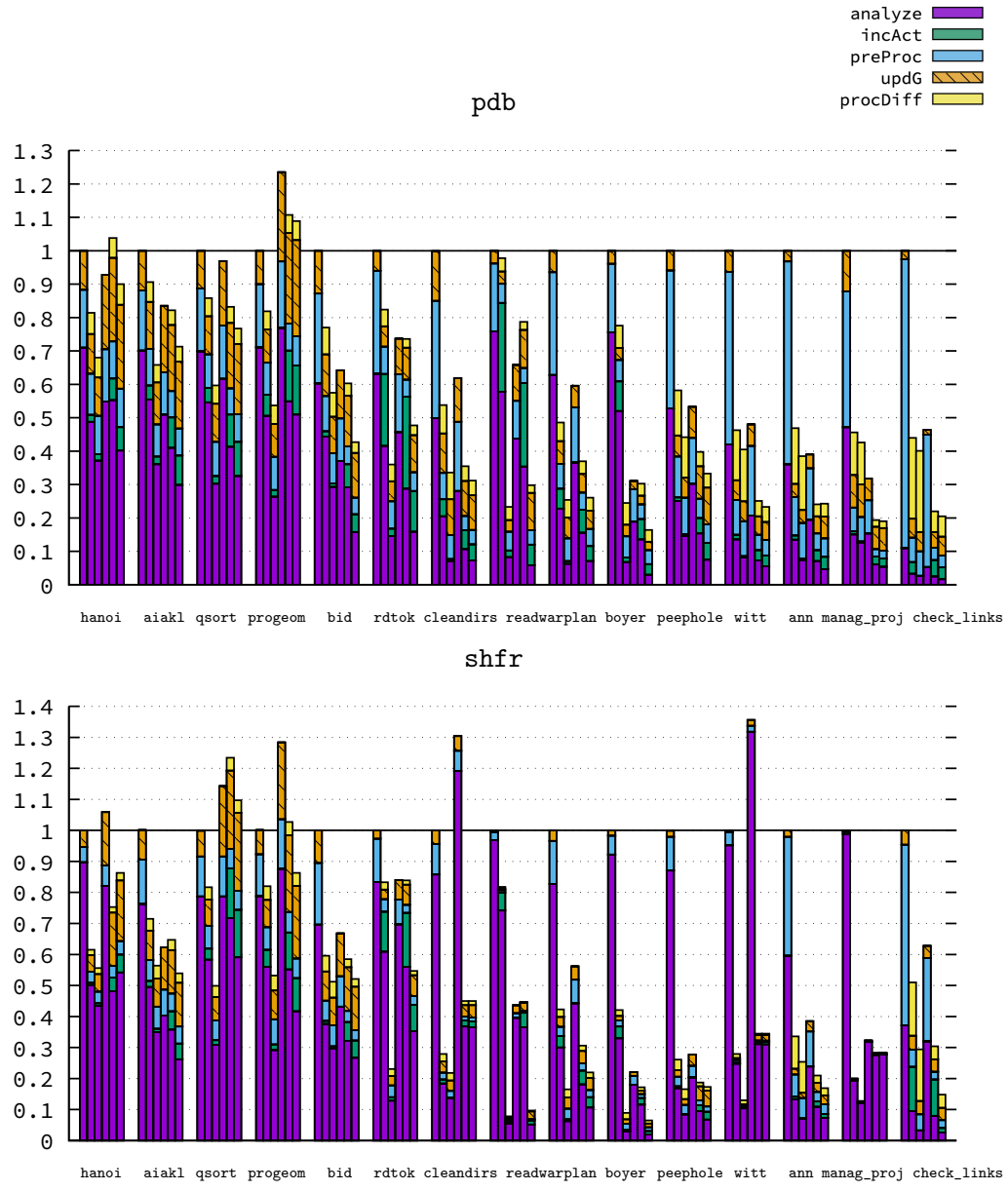


Fig. 37: Accumulated analysis time (normalized w.r.t `mon`) deleting clauses. The order inside each set of bars is: |`mon`|`mon_td`|`mon_scc`|`mod`|`mod_td`|`mod_scc`|.

A.1.5 Speedup vs. size of the analysis

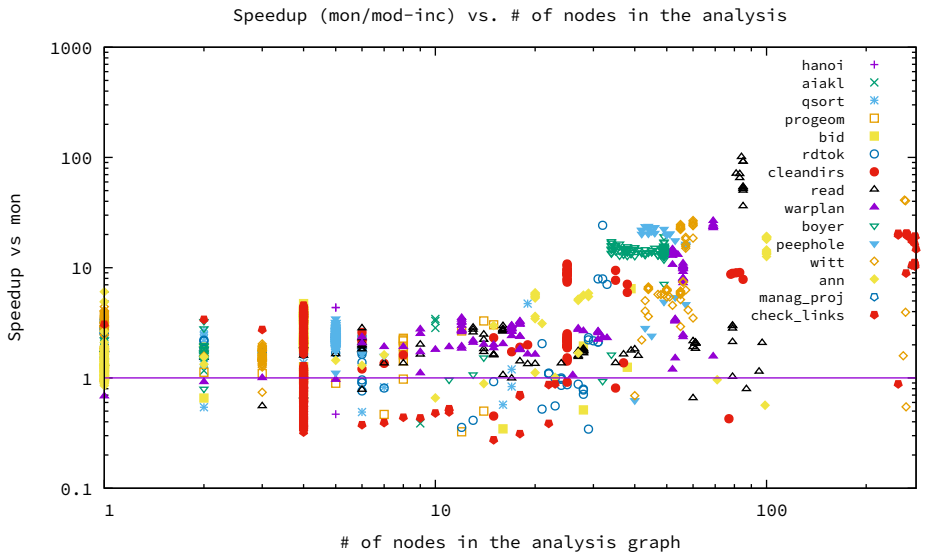


Fig. 38: Speedup vs monolithic depending on the number of nodes in the analysis graph.

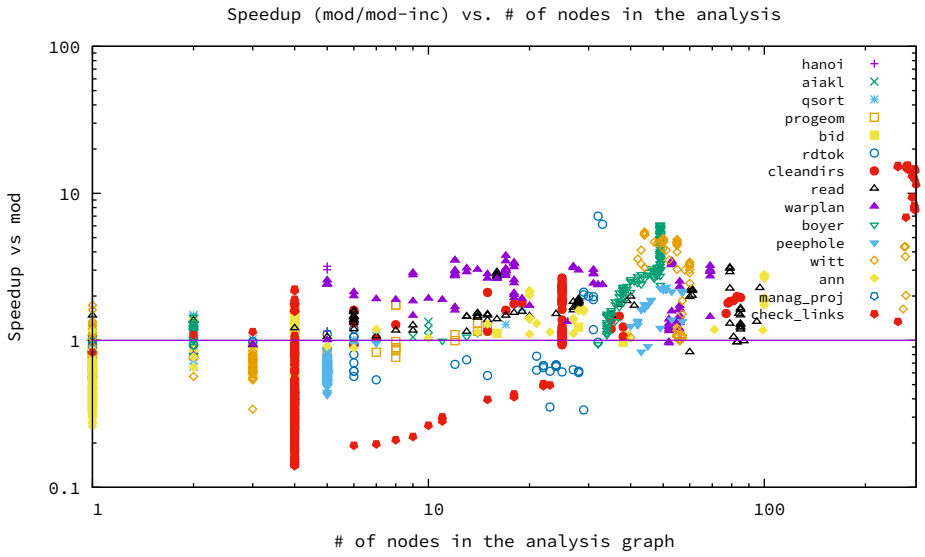


Fig. 39: Speedup vs modular depending on the number of nodes in the analysis graph.