Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Improving Run-time Checking in Dynamic Programming Languages

Ph.D Thesis

Nataliia Stulova
M.Sc. in Artificial Intelligence

2018

Departamento de Lenguajes y Sistemas Informáticos e Ingenieria de Software

Escuela Técnica Superior de Ingenieros Informáticos

Improving Run-time Checking in Dynamic Programming Languages

Submitted in partial fulfillment of the requirements for the degree of:
Doctor of Philosophy in Software, Systems and Computing

Author: **Nataliia Stulova, MSc**
Advisor: **José F. Morales, PhD**
Co-Advisor: **Manuel V. Hermenegildo, PhD**

2018

THESIS COMMITTEE

Chair	
Germán Vidal	*Full Professor*
	Technical University of Valencia (UPV)
	Spain
Secretary	
Manuel Carro	*Associate Research Professor*
	Technical University of Madrid (UPM)
	Spain
Members	
Alessandra Gorla	*Assistant Research Professor*
	IMDEA Software Institute
	Spain
Viktor Kuncak	*Associate Professor*
	Swiss Federal Institute of Technology in Lausanne (EPFL)
	Switzerland
Tom Schrijvers	*Research Professor*
	Catholic University of Leuven (KU Leuven)
	Belgium
Additional Members	
Maria Garcia de la Banda	*Full Professor*
	Monash University
	Australia
Francisco López Fraguas	*Full Professor*
	Complutense University of Madrid (UCM)
	Spain

This dissertation has been revised on 07.06.2018
to include thesis committee feedback.

RESUMEN

Detectar comportamientos incorrectos en los programas es una parte importante en el ciclo de desarrollo de software. Es una tarea compleja y tediosa, especialmente en el contexto de los lenguajes dinámicos. Se han propuesto numerosas técnicas que ayudan en el proceso, entre las cuales nos hemos centrado en el uso de construcciones a nivel de lenguaje para describir el comportamiento esperado del programa, y en las herramientas necesarias para comparar el comportamiento real del programa en contraposición con el esperado, como, por ejemplo, analizadores/verificadores estáticos de código y entornos de verificación en tiempo de ejecución.

En la práctica, sin embargo, el alto coste durante la ejecución hace que el uso de estas herramientas sea poco viable, especialmente para propiedades complejas. Esto reduce el interés en hacer comprobaciones en tiempo de ejecución desde el punto de vista de los programadores y programadoras, quienes esporádicamente permitirán comprobaciones de condiciones muy sencillas pero tenderán a desactivarlas para propiedades complejas. Algunos trabajos optan por limitar la expresividad del lenguaje de aserciones para reducir este coste.

Con esta motivación, el objetivo de esta tésis es doble:

- primero, pretendemos mejorar la expresividad del lenguaje de aserciones para reflejar todas las características relacionadas con el lenguaje de programación, incluyendo, por ejemplo, construcciones de orden superior, haciéndolo de forma que el/la programador/a pueda escribir especificaciones sin necesidad de aprender o programar para ello;

- al mismo tiempo, nuestra meta es comprobar de forma eficiente dichas especificaciones, reduciendo el coste asociado en tiempo de ejecución en la medida de lo posible y sin comprometer las garantías de seguridad que proporcionan dichas comprobaciones.

Esta tésis presenta varias mejoras para la comprobación de especificaciones en tiempo de ejecución entre las que se encuentran:

- un mecanismo discreto de memorización de resultados intermedios de comprobación, de forma que pueden ser reutilizados en el proceso de comprobación en lugar de recalcularlos;

- un técnica que combina comprobación en tiempo de compilación y en tiempo de ejecución, que usa las propiedades de esta

última como información adicional en tiempo de compilación, lo que implica que más propiedades se puedan comprobar estáticamente, aligerando el trabajo en tiempo de ejecución;

- y otra técnica para mejorar la inferencia de estructuras durante el análisis estático de programas, que aprovecha las reglas de visibilidad de términos del entorno modular subyacente, lo que permite simplificar las comprobaciones de propiedades del programa consiguiendo un sobrecoste constante en casos relevantes.

Finalmente, para atacar el problema de la expresividad limitada de los lenguajes de especificaciones, esta tésis se enfoca en el caso concreto de aportar especificaciones detalladas para rutinas de orden superior.

Las técnicas y herramientas estudiadas en esta tésis se presentan, por concreción, en el entorno de comprobación en tiempo de ejecución Ciao. No obstante, los resultados son generales e independientes del sistema, y creemos que pueden trasladarse de forma sencilla a otros lenguajes de programación declarativos. Además, dados los avances en verificación en gran parte de los lenguajes de programación, incluyendo los imperativos, mediante la traducción a cláusulas de Horn y probando propiedades a este nivel, y el hecho de que este enfoque está totalmente soportado en el sistema Ciao, argumentamos que nuestros resultados se pueden adaptar fácilmente a un espectro mucho mas amplio de lenguajes.

ABSTRACT

Detecting incorrect program behaviors is an important part of the software development life cycle. It is also a complex and tedious one, in which dynamic languages bring special challenges. A number of techniques have been proposed to aid in the process, among which we center our attention on the use of language-level constructs to describe expected program behavior, and of associated tools to compare actual program behavior against expectations, such as static code analyzers/verifiers and run-time verification frameworks.

In practice, however, the run-time overhead associated with these tools often remains impractically high, specially for non-trivial properties, or complex data structure tests. This reduces the attractiveness of run-time checking to programmers, who may allow sporadic checking of very simple conditions, but will tend to turn off run-time checking for more complex properties in favor of faster execution. Some approaches even opt for limiting the expressiveness of the assertion language in order to reduce the overhead.

Our research objective in this thesis is twofold:

- first, we aim to enhance the expressiveness of the assertion language to reflect all the features of the related programming language, including, e.g., higher-order constructs, and to do so in a way that allows the programmer to write precise program specifications while not imposing a learning or programming burden on them;
- at the same time, our goal is to efficiently check specifications, mitigating the associated run-time overhead as much as possible without compromising the safety guarantees that the checks provide.

With respect to checking specifications efficiently this dissertation presents several improvements for run-time specification checking, including:

- a mechanism for unobtrusive caching of intermediate run-time checking results so that they can be re-used in the checking process instead of being re-evaluated, contributing to undesirable (and unnecessary) run-time overhead;
- a technique of combining compile- and run-time checking in a way that uses the properties from the program specification as an additional information source during static specification checking, which results in more properties checked statically and fewer of them turned into run-time checks;

- and another technique for improving term shape inference during static program analysis, exploiting term visibility rules of the underlying module system, which allows to simplify property checks in a program in a way that constant run-time overhead is achievable in relevant cases.

Finally, to address the limited expressiveness of the specification languages, this dissertation targets the concrete case of providing detailed specifications for higher-order program routines.

The techniques and tools discussed in this thesis are presented for concreteness in the context of the Ciao run-time checking framework. Nevertheless, these results are general and system-independent, and we believe they can be straightforwardly transferred to the contexts of other declarative languages. In addition, given the recent advances in verification of a wide class of programming languages, including imperative ones, by translation into Horn clauses and proving properties at this level, and the fact that this approach is fully supported in the Ciao system, we argue that our results can easily be adapted to a much broader spectrum of languages.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the two people without whom none of this work would have ever been even started, let alone accomplished, my advisors Manuel Hermenegildo and José F. Morales. Manuel, almost 6 years ago you offered this nervous master's student a problem to work on that was completely alien to her, and look where we ended up. I am most grateful for your continuous support and advice, both on scientific matters and on life in general, especially in my darkest PhD periods. José, I am infinitely thankful for all the hours you have found to help me with the technical bits and pieces, for the advice you have offered, and for many passionate talks on compilers and other things I never imagined to actually be so exciting.

Next, as no scientific work is lone work, I would like to thank all the people who have read and reviewed, and criticized, and in general helped with this dissertation and related publications. Thanks to my thesis committee: Germán Vidal, Manuel Carro, Alessandra Gorla, Tom Schrijvers, Viktor Kunčak, Maria Garcia de la Banda, and Francisco López Fraguas. Thanks to Julio Mariño and Manuel Carro for commenting on the thesis and its presentation during the pre-defense. And thanks to you, Anonymous Reviewers, for without your service the publications this thesis draws from would have never seen the light of the proceedings.

On a more personal note I would like to thank all members of the CLIP group, present and past, who stoically listened to me talking about my work, came to hear my talks, and whom I have collaborated with: Luthfi Darmawan, Umer Liqat, Joaquín Arias, Pedro Lopez, Maximiliano Klemen, and especially to Isabel García Contreras, who is a pleasure to share an office with, and who helped to translate the abstract into human-readable Spanish. Thanks to all the wonderful people I have met at IMDEA Software, who contributed to so many positive memories: Ilya Sergey, Giovanni Bernardi and Andrea Cerone, Wouter Lueks, Álvaro García Pérez, Srdjan Matic, Miriam García, Goran Doychev, Luca Nizzardo, Miguel Ambrona, Platon Kotzias, Bogdan Kulynych, Borja de Regil, and Arianna Blasi. Special thanks to Julián Samborski-Forlese and Damir Valput, for showing that there are other stories. More thanks also to Antonio Faonio for introducing me to Jordi, the cutest and nicest cat I have met, whose contributions to my emotional health cannot be underestimated. And, of course, many thanks to Artem Khyzha, who was there when all this started. Thanks for the constant support, for putting up through my darkest moods, and for the cakes, obviously.

каменяр(к)ам

CONTENTS

1

INTRODUCTION

One of the things that distinguishes human intelligence is our ability for abstract thought and rational reasoning. During the most part of the history of our species these tasks were only performed by humans — philosophers, scholars, engineers. This started to change in the XVII century with the idea that computations can be performed semi-automatically with devices such as logarithmic rulers and mechanical calculators. Fast-forward to XXI century and computations are performed fully automatically by what we call nowadays computers.

Not to be confused with human computers, *a profession that existed in the XVII-XX centuries.*

The Digital Revolution of the last decades has changed the role of computers from military and scientific equation solvers to being an integral part of our lives. Yet, while the changes in the physical organization, the *hardware*, are drastic, it is the *software*, the programs being executed, that have made computers our inseparable companions. As of writing, software is everywhere: in watches, smartphones, cars, planes, self check-out cashiers at the supermarkets, it is regulating traffic flow in the streets, it issues speed limit violation tickets for drivers, and the list goes on and on.

There is no doubt that modern software-based technological solutions offer benefits and comfort both for the engineers and for the end users. This comfort, however, highly depends on the quality of the software. While factors like efficiency or convenience of the user interaction mostly affect the competition between similar products, errors in software might have severe real-life consequences. In 1999 NASA lost a $125 million Mars orbiter because a software component written by an engineering company performed computations in Imperial units while the rest of the software used the more conventional metric system. In 2012 a "software glitch" after an update of the trading software of the Knight Capital company caused a $440 million loss in about 30 minutes. Yet another example is known nowadays as the "Northeast blackout of 2003," when widespread power outage occurred throughout parts of the United States and the Canadian province of Ontario due to a "bug" in the alarm system of the electricity grid controllers, resulting in almost 100 human fatalities. It is no wonder then that companies and service providers that rely on software in their critical operations, such as NASA and Airbus, to name a few, invest in software validation and verification.

1.1 APPROACHES TO ASSERTION-BASED DEBUGGING AND VERIFICATION AND THE CIAO MODEL

For the reasons mentioned above, detecting and avoiding incorrect program behaviors is an important part of the software development life cycle. It is also a complex and tedious one (in which dynamic languages bring special challenges), and thus a number of techniques have been proposed to aid in this process. Among these techniques, we center our attention on the use *of programming language-level constructs* to describe expected program behavior (i.e., to express *specifications*), and of associated tools to compare actual program behavior against such expectations. This language-based approach has the advantage that the specifications can also be used to clarify interfaces and meanings and facilitate "programming in the large" by making large programs more maintainable and better documented.

By dynamic (programming) language *we mean an untyped programming language with run-time checking of various properties (types, modes, etc.).*

A classical approach in this context is the use of *theorem provers and proof assistants*, such as, e.g., HOL, ACL2, Isabelle, Coq, etc., to construct proofs of the validity (or not) of *assertions* (the language-level constructs) about the program. The advantage of this approach is that it can deal with arbitrary properties, but at the price of manual intervention from the programmer, which is also required to have significant expertise in the tools and their underlying theory. While this approach is of clear value when complex properties need to be proved and the required expertise is available, our interest herein is in *automated* approaches, i.e., approaches that can be made an intrinsic part of the development process such as, e.g., by being called at each compilation iteration or even being embedded in the compiler, essentially without programmer involvement.

The classical example of the latter is that of *traditional strongly-typed systems*, such as those used, for example, in some functional and logic languages (e.g., Haskell [45], Gödel [44] or Mercury [97]). Here the language-level constructs are the type declarations and the verification process the type checking/inference. An advantage of this approach is that it meets the automation objective in that the checking/inference process is mechanical and done routinely by the compiler at every cycle. It has also been shown to scale well for industrial applications. As a result, many languages across different programming paradigms adopt strong typing as a mechanism for ensuring data manipulation correctness. The disadvantage is that for the same reasons traditional type systems are required to be decidable, i.e., to always be able to prove in finite (and, in fact, short) time whether an assertion (i.e., a type declaration) holds or not. Traditionally, this has meant limiting in practice the properties that can be captured by the types and/or limiting the programming language, and imposing additional requirements such as that all types (and, when relevant, modes) have to be defined explicitly or that all procedures have to be

"well-typed" and/or "well-moded," absence of subtyping, etc. In this approach programs that are untypable or do not conform to these rules are rejected.

An alternative approach that lies between the two extremes above is to make a best effort to infer and check the required properties through automatic, rigorous analysis tools. The fundamental technique in this context is abstract interpretation [26], which allows inferring *provably safe approximations* of program semantics in an automatic way. The use of approximations implies accepting up front the fact that complete verification or validation may not always be possible. But in return automation can be achieved without imposing too many limitations in the properties or the programming language.

A canonical example of this approach, which it pioneered, is the Ciao programming language, and its associated analyzer and assertion handling model [65, 66, 68, 10, 82, 12, 84, 42, 85, 43, 61, 79]. This model combines a language of (optional) assertions with a methodology for dealing with such assertions, based on abstract interpretation, that is automatic, in contrast to theorem provers and proof assistants, while at the same time allowing a much richer class of properties than traditional type systems. This includes, e.g., modes, moded types, determinacy, non-failure, sharing/aliasing, term linearity, intervals, constraints, cost, etc. The price is not always being able to discharge all the assertions statically, but then, rather than always rejecting the program, run-time checks are introduced (optionally) to detect at run-time any violations of the assertions. The model also allows verifiable program certification [2]. The Ciao model and its implementation will be used as the conceptual reference framework for our work in this dissertation.

1.2 RELATED WORK

The Ciao model draws many synergies from combining various components, such as an assertion language, abstract-interpretation-based static analysis, run-time checking, and testing, to name a few. The combination of compile-time and run-time checking is related to the NU–Prolog debugger [72], which performed compile-time checking of decidable (regular) types and also allowed calling Prolog predicates at run time as a form of dynamic type checks, and to the *soft typing* approach of Cartwright [18], which introduces run-time checks for untyped parts of programs. However, as mentioned before, the Ciao model is not restricted to types, nor requires properties to be decidable. The later proposal of [51] for Prolog IV that combined compile- and run-time checking was inspired by Ciao.

A number of other approaches have been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of *algorithmic*

debugging in [54]. Abstract interpretation for debugging in imperative programs was studied by Bourdoncle [7] and in logic programs by Comini et al., for the particular case of algorithmic debugging and diagnosis for declarative properties [23, 22]. Both of these approaches focused on some specific semantics: context-insensitive and bottom-up (i.e., non query-driven), respectively. The Ciao model is designed to support context sensitivity/multi-variance and standard program control-flow semantics, so that both declarative and procedural properties can be checked, for multiple procedure contexts and with multiple corresponding assertions (pre-/post-condition pairs) for the same procedure or program point. Safe approximations have also been used to reduce the burden posed on programmers by declarative debuggers [21], also addressed by Boye et al. [8]. The general topic has also been summarized by Cousot in [25].

The ideas of allowing properties that cannot always be decided statically, using safe approximation inference by abstract interpretation as proof method, introducing run-time checks for properties not verified statically, etc., are gradually having impact in many contexts.

For example, *gradual* typing has also become a hot research topic in the functional programming community, a notable example of a language incorporating it being Scheme [96, 111], which has also served as a model language for alternative approaches of *occurrence* typing [110] and *contract*-based extensions [32, 30]. Similar work has been carried out for the Racket [77, 76, 78] programming language (an evolution of Scheme), with particular focus on efficiency and practicality of this typing discipline reviewed in [107, 108]. A discussion on introducing gradual typing to Prolog and the implementation challenges is proposed in [93]. Moreover, gradual typing has been successfully introduced in the imperative programming paradigm (TypeScript [89]).

More recently, *refinement* types have gained attention in program verification, adding greater flexibility to the program properties, as well as extending verification approach. Notable examples of languages for which systems implementing this typing approach have been developed include Haskell (Liquid Haskell [113]) and Ruby [47].

In object-oriented programming a similar evolution of tools has been followed [52], with program *contracts* being added to verification frameworks for .NET (Code Contracts [57, 64, 31]) and Java (JML /Spec# [53]).

1.3 RUN-TIME CHECKING OVERHEAD

As mentioned before, the Ciao model, as well as most of the later approaches discussed above, typically involves a certain degree of run-time testing. A practical limitation is that these checks can incur significant run-time performance overhead, even in the simple case

of performing just type checks between typed and untyped parts of programs [89, 108].

In [61] overhead reductions are obtained by limiting the points at which the tests are performed and the instrumentation, as well as by inlining, but some types of tests still incur significant costs. Other approaches opt for limiting the expressiveness of the assertion language in order to reduce the overhead (see [91] for some recent case studies). Some proposals have been made for reducing the run-time overhead of assertion checking based on optimizing the run-time checking mechanisms themselves, at the expense of increased memory consumption [50, 101]: the time overhead of repeated checks on immutable recursive data structures is traded for increased memory use via caching and/or tabling techniques.

As also mentioned before, in the Ciao model static analysis is used to minimize the number and cost of the run-time checks that need to be placed in the program. A number of (abstract interpretation-based) static analyses are combined in order to verify assertions to the largest extent possible at compile time. Recent work in the context of run-time monitoring frameworks for imperative programs uses similar ideas to exploit static analyses in order to reduce the run-time overhead of the monitors as, e.g., proposed in [6] for Java programs.

Despite all these advances, run-time overhead often remains impractically high, for example for properties which require deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may activate sporadic checking of very simple conditions, but tend to turn off run-time checking for more complex properties.

Reducing this run-time checking overhead is one of the main objectives of this dissertation.

1.4 LANGUAGE INDEPENDENCE

We develop the discussion throughout the thesis in the context of *(Horn Clause) Logic Programs*, which allows us to take advantage of the availability of mature program analysis and transformation tools, and a well developed assertion language and assertion processing framework (in particular, that of the Ciao system). However, we argue that the results are applicable to other programming paradigms, either directly (e.g., to other forms of declarative programming), or to imperative programs, via semantic transformation into Horn clauses. The use of Horn clauses in the Ciao system as an intermediate language to support programs in other languages was described in [60]. Some concrete examples of the application of this approach that we have explored within Ciao include cost analysis of Java bytecode programs [74, 75] and energy bound inference in binaries stemming from C-style programs [56, 55]. Recently [34] proposed an approach for us-

ing Horn clauses as an intermediate language which is quite similar to Ciao's [60].

The Horn clause-based transformational approach is currently receiving considerable interest (see, e.g., [36, 37, 58]), and is even the subject of the "Horn clause-based Verification and Synthesis" workshop series [5]. In [28] encouraging results are reported for the direct inference of the verification conditions of safety properties for C programs based on their (C)LP representation. Similar approaches have been used to translate to other formalisms, such as term rewrite systems [35].

1.5 THE CIAO STATIC/DYNAMIC DEBUGGING AND VERIFICATION FRAMEWORK

In this section we provide a tutorial overview of a subset of the Ciao assertion language, relevant for this dissertation, and of the verification framework, following the presentation of [40]. This will provide the context for introducing the different contributions made by the thesis.

CIAO ASSERTIONS Assertions in Ciao are linguistic constructs which allow expressing properties of programs. Syntactically they appear as declarations, and semantically they allow talking about preconditions, (conditional-) postconditions, whole executions, program points, etc. Herein we will focus on the most commonly-used subset of the Ciao assertion language: `pred` assertions. A detailed description of the full language can be found in [84, 11].

Such `pred` assertions are used to describe predicates by stating sets of preconditions and postconditions on the state of the computation before and after calls to predicates, as well as global properties of such computations (such as, e.g., the number of execution steps, determinacy, or the usage of some other resource). Figure 1.1 includes a number of `pred` assertions. The assertion in line 4 expresses that calls to predicate `nrev/2` with the first argument instantiated to a list are admissible, and that if such calls succeed then the second argument should also be instantiated to a list. `list/1` is an example of a *(state) property*: a predicate which expresses constraints on the values of a variable or a set of variables. Note that `A` in `list(A)` above refers to the first argument of `nrev/2`. Properties can also involve several variables and/or be parametric. As an example of the latter, the assertion in line 5 of Figure 1.1 uses `list_of/2` to express that for any call to predicate `nrev/2` with the first argument instantiated to a list of `colors`, if the call succeeds, then the second argument is also instantiated to a list of `colors`. Properties are defined by the user and exported/imported as normal predicates. In Figure 1.1 properties `list/1`, `list_of/2`, and `color/1` are imported from the user module `someprops`

```
1  :- module(_, [nrev/2], [assertions]).
2  :- use_module(someprops,[list/1,list_of/2,color/1]).
3
4  :- pred nrev(A, B) : list(A) => list(B).
5  :- pred nrev(A, B) : list_of(color, A) => list_of(color, B).
6  :- pred nrev(A, _) : list(A) + (not_fails, is_det, terminates).
7
8  nrev([],[]).
9  nrev([H|L],R) :- nrev(L,R0), conc(R0,[H],R).
10
11 conc([],L,L).
12 conc([H|L], K, [H|T]) :- conc(L,K,T).
```

Figure 1.1: Naive reverse with some assertions.

in line 2. In any case properties need to meet some restrictions, e.g., their execution should terminate for any possible call since, as discussed later, properties will not only be checked at compile time, but may also be involved in run-time checks. Types are a particular case (further restriction) of state properties and different type systems are implemented as libraries. Most properties are "runnable" (useful for run-time checking), and can be interacted with, i.e., the answers to a query: `?- use_package(someprops), list(X).` are: `X = []`, `X = [_]`, `X = [_,_]`, `X = [_,_,_]`, etc. Finally, `not_fails/1` is an example of a *computational property*: a predicate which expresses constraints on the *execution* of calls, not failing in this case. However, this thesis concentrates mostly on state properties. As Figure 1.1 shows, there can be several `pred` assertions for the same predicate.

Assertion status: Each assertion has a *verification status*, marked by prefixing the assertion with the keywords `check`, `trust`, `true`, `checked`, and `false`. This specifies respectively whether the assertion is provided by the programmer and is to be checked or to be trusted, or is the output of static analysis and thus correct (safely approximated) information, or the result of processing an input assertion and proving it correct or false. The `check` status is assumed by default when no explicit status keyword is present.

Uses of assertions: assertions find many uses in Ciao, ranging from testing to verification and documentation (for the latter, see `lpdoc` [39]). In addition to describing the properties of the module in which they appear, assertions also allow programmers to describe properties of modules / classes which are not yet written or are written in other languages.[1] This makes it possible to run checkers / verifiers / documenters against partially developed code.

THE CIAO VERIFICATION FRAMEWORK We now describe the Ciao verification framework [12, 42, 84], implemented in the Ciao preprocessor, CiaoPP. Figure 1.2 depicts the overall architecture. Hexagons

1 This is also done in other languages but, in contrast with Ciao, different kinds of assertions for each purpose are often used.

represent tools and arrows indicate the communication paths among them. It is a design objective of the framework that most of this communication be performed also in terms of assertions. This has the advantage that at any point in the process the information is easily readable by the user. The input to the process is the user program, *optionally* including a set of assertions; this set always includes any assertion present for predicates exported by any libraries used (left part of Figure 1.2).

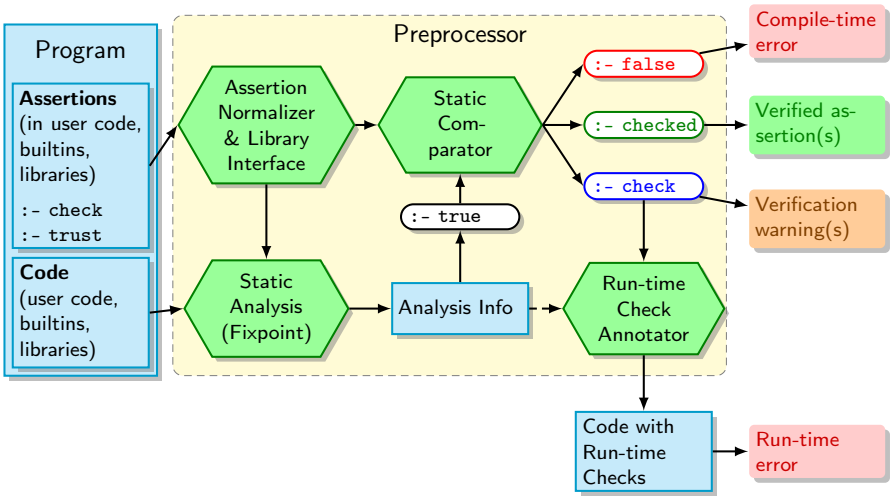


Figure 1.2: The Ciao Verification Framework.

Run-time checking of assertions: after (assertion) normalization in the *Assertion Normalizer* component (which, e.g., takes away syntactic sugar) the *RT-check* module transforms the program by adding run-time checks to it that encode the meaning of the assertions (assume for now that the *Comparator* simply passes the assertions through). Note that the fact that properties are written in the source language and *runnable* is very useful in this process. Failure of these checks raises run-time errors referring to the corresponding assertion. *Correctness* of the transformation requires that the transformed program only produce an error if the assertion is in fact violated.

Compile-time checking of assertions: even though run-time checking can detect violations of specifications, it cannot guarantee that an assertion holds. Also, it introduces run-time overhead. The framework performs compile-time checking of assertions by *comparing* the results of *Static Analysis* (Figure 1.2) with the assertions [12, 42]. This analysis is typically performed by abstract interpretation [26] or any other mechanism that provides *safe* upper or lower approximations of relevant properties, so that comparison with assertions is meaningful despite precision losses in the analysis. The type of analysis may be selected by the user or determined automatically based on the properties appearing in the assertions. Analysis results are given using also the assertion language, to ensure interoperability and make them understandable by the programmer. As a possible result of the

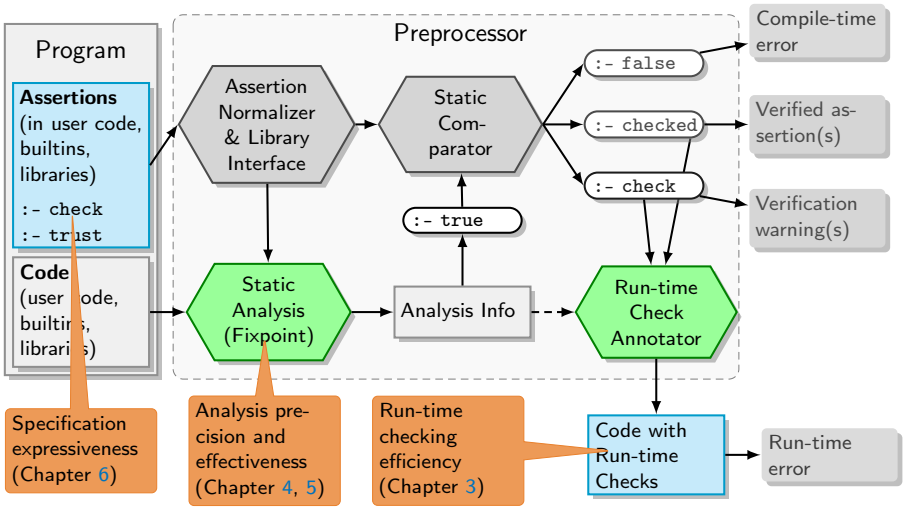


Figure 1.3: Overview of the Contributions (orange boxes) w.r.t. the Ciao Verification Framework Components.

comparison, assertions may be proved to hold, in which case they get `checked` status –Figure 1.2. As another possible result, assertions can be proved not to hold, in which case they get `false` status and a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, if it is not possible to prove nor to disprove (part of) an assertion, then such assertion (or part) is left as a `check` assertion, for which optionally run-time checks can be generated as described above. This can optionally produce a *verification warning*.

1.6 OBJECTIVES AND CONTRIBUTIONS PER CHAPTER

The overall objective of the dissertation is to push the state of the art in several aspects of the assertion-based, combined static/dynamic debugging and verification approach represented by the Ciao model, with the intention that the results will be applicable in the many related systems currently gaining popularity within functional, constraint, and imperative programming. More precisely, the concrete objectives and contributions of the dissertation are depicted in Figure 1.3, which shows the relation between each dissertation chapter and the aspects of the overall approach addressed in that chapter. The actual contents and contributions of the chapters are presented below.

ORGANIZATION OF THE TEXT The chapters are mostly self-contained. However, Chapters 3 to 6 rely on the definitions and notation introduced in Chapter 2. It is also advisable to read Chapter 4 before Chapter 5 as the latter uses a similar experimental evaluation framework.

CHAPTER 3 *(Figure 1.3: Improving run-time checking efficiency)* This chapter presents an approach for reducing run-time checking overhead that is based on the use of memoization to cache intermediate results of check evaluation, avoiding repeated checking of previously verified properties. Compared to approaches that reduce checking frequency, our proposal has the advantage of being exhaustive (i.e., all tests are checked at all points) while still being much more efficient than standard run-time checking. Compared to the limited previous work on memoization, it performs the task without requiring modifications to data structure representation or checking code. We also report on a prototype implementation and provide some experimental results that support that using a relatively small cache leads to significant decreases in run-time checking overhead.

CHAPTER 4 *(Figure 1.3: Improving analysis precision and effectiveness via the run-time checking semantics)* In this chapter we explore the effectiveness of abstract interpretation in detecting parts of program specifications that can be statically simplified to true or false, as well as the impact of such analyses in reducing the cost of the run-time checks required for the remaining parts of these specifications. Starting with a semantics for programs with assertion checking, and for assertion simplification based on static analysis information obtained via abstract interpretation, we propose and study a number of practical assertion checking "modes," each of which represents a tradeoff between code annotation depth, execution time slowdown, and program safety. We then explore these modes in two typical, library-oriented scenarios. We also propose program transformation-based methods for taking advantage of the run-time checking semantics to improve the precision of the analysis. Finally, we study experimentally the performance of these techniques. Our experiments illustrate the benefits and costs of each of the assertion checking modes proposed, as well as the benefits obtained from analysis and the proposed transformations in these scenarios.

CHAPTER 5 *(Figure 1.3: Improving analysis precision and effectiveness via term hiding)* While static analysis can greatly reduce run-time checking overheads, the gains depend strongly on the quality of the information inferred. Reusable libraries, i.e., library modules that are precompiled independently of the client, pose special challenges in this context. We propose a technique which takes advantage of module systems which can hide a selected set of functor symbols to significantly enrich the shape information that can be inferred for reusable libraries, as well as an improved run-time checking approach that leverages the proposed mechanisms to achieve large reductions in overhead, closer to those of static languages, even in the reusable-library context. Our method maintains the full expressiveness of the

assertion language in this context. In contrast to other approaches it does not introduce the need to switch the language to a (static) type system, which is known to change the semantics in languages like Prolog. We also study the approach experimentally and evaluate the overhead reduction achieved in the run-time checks.

CHAPTER 6 *(Figure 1.3: Improving specification expressiveness)* Higher-order constructs extend the expressiveness of first-order (Constraint) Logic Programming ((C)LP) both syntactically and semantically. At the same time assertions have been in use for some time in (C)LP systems helping programmers detect errors and validate programs. However, assertion-based extensions to (C)LP have not been integrated well with higher-order to date. This chapter contributes to filling this gap by extending the assertion-based approach to error detection and program verification to the higher-order context within (C)LP. We propose an extension of properties and assertions as used in (C)LP in order to be able to fully describe arguments that are predicates. The extension makes the full power of the assertion language available when describing higher-order arguments. We provide syntax and semantics for (higher-order) properties and assertions, as well as for programs which contain such assertions, including the notions of error and partial correctness. We also discuss several alternatives for performing run-time checking of such programs.

1.7 LIST OF PUBLICATIONS

The following list indicates publications corresponding to the chapters of the dissertation:

CHAPTER 2 AND CHAPTER 6 are based on the following paper [99]:

> Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. *Assertion-based Debugging of Higher-Order (C)LP Programs*. In 16th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14), pages 225–235. ACM Press, September 2014.

An extended abstract of this paper has been also presented as the following informal publication [100]:

> Nataliia Stulova, José F. Morales, Manuel V. Hermenegildo. *Towards Assertion-based Debugging of Higher-Order (C)LP Programs*. 30th International Conference on Logic Programming (ICLP'14), Theory and Practice of Logic Programming Special Issue, On-line Supplement, Vol. 14, Num. 4-5, pages 209-210, Cambridge University Press, July 2014.

CHAPTER 3 is based on the following paper [101]:

> Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. *Practical Run-time Checking via Unobtrusive Property Caching*. Theory and Practice of Logic Programming, 31st International Conference on Logic Programming (ICLP'15) Special Issue, 15(04-05):726–741, September 2015.

CHAPTER 4 is based on the following paper [105]:

> Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. *Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis*. Science of Computer Programming, Vol. 155, pages 3-26, Elsevier North-Holland, April 2018. Selected and Extended papers from the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP'16).

which is an extended journal version of a conference paper [102]:

> Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. *Reducing the Overhead of Assertion Run-time Checks via Static Analysis*. 18th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16), pages 90-103, ACM Press, September 2016.

CHAPTER 5 is based on the following paper [104]:

> Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. *Exploiting Term Hiding to Reduce Run-time Checking Overhead*. In Francesco Calimeri, Kevin Hamlen, and Nicola Leone, editors, 20th International Symposium on Practical Aspects of Declarative Languages (PADL 2018), LNCS Vol. 10702, pp. 99–115, Springer-Verlag, January 2018.

An extended abstract of this paper has been also presented as the following informal publication [103]:

> Nataliia Stulova, José F. Morales, Manuel V. Hermenegildo. *Towards Run-time Checks Simplification via Term Hiding*. Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017), OpenAccess Series in Informatics (OASIcs), Vol. 58, pages 1-3, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

2

A SEMANTICS OF (CONSTRAINT) LOGIC PROGRAMS WITH ASSERTIONS

This chapter introduces the reader to the formal notation of (Constraint) Logic Programming ((C)LP) and the Ciao system, establishing the technical language for the following chapters. First, the standard concepts and notation from (C)LP theory, that are used throughout this dissertation are recalled. Then the Ciao assertions are introduced in a formal way.

2.1 (C)LP NOTATION AND SEMANTICS

Sets of variable, function, and predicate symbols, are denoted by VS, FS, and PS respectively. Variables start with a capital letter. An anonymous variable, denoted _ , represents a variable that is distinct from any other variable appearing in the same scope. Each $p \in \mathsf{PS}$ and $f \in \mathsf{FS}$ is associated to a natural number called its *arity*, written $\mathsf{ar}(p)$ or $\mathsf{ar}(f)$. The set of terms TS is inductively defined as follows: $\mathsf{VS} \subset \mathsf{TS}$, if $f \in \mathsf{FS}$ and $t_1, \ldots, t_n \in \mathsf{TS}$ then $f(t_1, \ldots, t_n) \in \mathsf{TS}$ where $\mathsf{ar}(f) = n$. An *atom* has the form $p(t_1, ..., t_n)$ where $p \in PS$, $\mathsf{ar}(p) = n$, and $t_1, ..., t_n \in TS$. A *constraint* is essentially a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). A *literal* is either an atom or a constraint. *Constants* are introduced as o-ary symbols. A *goal* is a finite sequence of literals. A *clause* is of the form $H \leftarrow B$ where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of clauses.

The σ symbol represents a variable renaming and $\sigma(X)$ represents the result of applying the renaming σ to some syntactic object X (a term, atom, literal, goal, etc.). The *definition* of an atom A in a program, $\mathsf{cls}(A)$, is the set of program clauses whose head has the same predicate symbol and arity as A, renamed-apart (i.e., all variables are renamed into distinct new variables). In the following it is assumed that all clause heads and clause body literals are normalized, i.e., any literal L in a program clause is of the form $p(X_1, ..., X_n)$ where the $X_1, ..., X_n$ are distinct free variables. This is not restrictive since programs can always be normalized, and it facilitates the presentation. Restricting the constraint θ to the variables of the syntactic object L is denoted as $\overline{\exists}_L \theta$. *Constraint entailment* is denoted by $\models$, so that $\theta_1 \models \theta_2$ denotes that θ_1 entails θ_2. In such case we say θ_2 is *weaker* than θ_1. In the rest of the dissertation it is assumed that there is a single program,

However, for conciseness in the examples non-normalized programs are used sometimes.

so that all sets of clauses, etc. refer to that implicit program and it is not necessary to refer to it explicitly in the notation.

Very often, the properties of a program which we are interested in expressing by means of assertions are related to the run-time behavior of the program. For this, we need to consider the operational semantics of the program. The operational semantics of a program is given in terms of its *derivations*, which are sequences of *reductions* between *states*. A *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store (or *store* for short) θ. We use :: to denote concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. We use $S \rightsquigarrow S'$ to indicate that a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. As a basis of the reductions used in the rest of the dissertation, we define reduction steps as follows:

Definition 2.1 (Reductions). *A state $S = \langle L :: G \mid \theta \rangle$ where L is a literal can be* reduced *to a state S' as follows:*

1. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ *if L is a constraint and $\theta \wedge L$ is satisfiable.*
2. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle B :: G \mid \theta \rangle$ *if L is an atom of the form $p(t_1, \ldots, t_n)$, for some clause $(L \leftarrow B) \in \mathsf{cls}(L)$.*

We denote by $D_{[i]}$ the i-th state of the derivation. As a shorthand, given a non-empty derivation D, $D_{[-1]}$ denotes the last state. We use $S \rightsquigarrow S'$ to indicate that a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow^* S'$ indicates that there is a sequence of reduction steps from state S to state S'.

A *query* is a pair (L, θ), where L is a literal and θ a store, for which the (C)LP system starts a computation from state $\langle L \mid \theta \rangle$. The set of all derivations from the query Q is denoted $\mathsf{derivs}(Q)$. The observational behavior of a program is given by its "answers" to queries. A finite derivation from a query (L, θ) is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle \square \mid \theta' \rangle$, where $\square$ denotes the empty goal sequence. In that case, the constraint $\exists_L \theta'$ (denoting the projection of θ onto the variables of L) is an *answer* to (L, θ). We denote by $\mathsf{answers}(Q)$ the set of answers to a query Q. A finished derivation is *failed* if the last state is not of the form $\langle \square \mid \theta \rangle$. A query Q *finitely fails* if $\mathsf{derivs}(Q)$ is finite and contains no successful derivation.

Note that $\mathsf{derivs}(Q)$ contains not only finished derivations but also all intermediate derivations from a query.

2.2 ASSERTION LANGUAGE

PROPERTIES Conditions on the constraint store are assumed to be expressed with *properties*. Properties and the other predicates composing the program are written in the same language. This approach is motivated by the direct correspondence between the declarative

and the operational semantics of constraint logic programs. In what follows property literals are referred to as *prop* literals.

Example 2.1. *The following property describes a sorted list:*

```
sorted([]).
sorted([_]).
sorted([X,Y|L]) :- X =< Y, sorted([Y|L]).
```

Definition 2.2. *The meaning of a prop literal L denoted $|L|$, is the set of constraints given by* answers$((L, true))$.

Intuitively, the meaning of a prop literal is the set of "weakest" constraints for which the literal holds:

Example 2.2. *Prop literal* `list/1` *can be defined by:*

```
list([]).
list([_|L]) :- list(L).
```

The meaning of `list/1` *and of* `sorted/1` *from Example 2.1 is given by:*

$$\begin{aligned} |list(A)| = &\ \{A = [], A = [B|C] \wedge list(C)\} \\ |sorted(A)| = &\ \{A = [], A = [B], \\ &\ A = [B, C|D] \wedge B \leq C \wedge E = [C|D] \wedge sorted(E)\}. \end{aligned}$$

An important observation is that in constraint logic programming it seems natural to define the meaning of prop literals as (C)LP programs rather than as (recursive) sets. Thus, the admissible prop literals are restricted to those literals L_p for which a definition of the corresponding predicate p exists such that answers$((L_p, true)) = |L_p|$. This is not too strong a restriction given the high expressive power of (C)LP languages.[1] Note that the approach also implies that the program must contain the definitions of all the predicates p for literals L_p used in conditions of assertions. We believe that this choice of a language for writing conditions is in fact of practical interest because it facilitates the job of programmers, which do not need to learn a specification language in addition to the (C)LP language they are already familiar with.

The following definition from [85] defines when the condition represented by a prop literal holds[2] for a given store:

Definition 2.3 (Succeeds Trivially)**.** *A prop literal L* succeeds trivially *for θ, denoted $\theta \Mapsto L$, iff $\exists \theta' \in$ answers$((L, \theta))$ such that $\theta \models \theta'$. A DNF formula of prop literals succeeds trivially for θ if all of the prop literals of at least one conjunct of the formula succeed trivially.*

1 Note that the scheme of [84, 83] allows approximate definitions of such predicates and sufficient conditions for proving and disproving them.

2 Lemma 1 in [85] establishes that the notions of "Holding Trivially" and "Succeeding Trivially" are equivalent, which allows us to simplify and base our semantics on the latter notion.

Intuitively, a prop literal L succeeds trivially if L succeeds for θ without adding new "relevant" constraints to θ.

Example 2.3. *Consider prop literals* `list(A)` *and* `sorted(B)` *and the predicate definitions of Example 2.2*

- *Assume that* $\theta = (A = f)$. *Since* $\forall \theta' \in |list(A)| : \theta \not\models \theta'$, *as we would expect,* $\theta \not\Rightarrow list(A)$.
- *Assume now that* $\theta = (A = [\_|Xs])$. *Though* A *is compatible with a list, it is not actually a (nil terminated) list. Again in this case* $\forall \theta' \in |list(A)| : \theta \not\models \theta'$ *and thus again* $\theta \not\Rightarrow list(A)$ *The intuition behind this is that we cannot guarantee that* A *is actually a list given* θ, *since a possible instance of* A *in* θ *is* $A = [\_|f]$, *which is clearly not a list.*
- *Finally, assume that* $\theta = (A = [B] \wedge B = 1)$. *In such case* $\exists \theta' = (A = [B|C] \wedge C = [])$ *such that* $\theta \models \theta'$ *and* $\exists c = (B = 1)$ *such that* $(c \wedge \theta' \not\models false) \wedge (\theta' \wedge c \models \theta)$. *Thus, in this last case* $\theta \Mapsto list(A)$.

Another class of the property checks of interest to a programmer are compatibility *properties, which are not considered in this dissertation. A discussion of the two can be found in [42] and Section 2.9.2 of [84].*

This means that prop literals are considered as *instantiation* checks: they are true iff the variables they check for are at least as constrained as their predicate definition requires.

Definition 2.4 (Test Literal)**.** *A prop literal* L *is a* test *iff* $\forall \theta$ *either* $\theta \Mapsto L$ *or* (L, θ) *finitely fails.*

ASSERTIONS Assertions are linguistic constructions for expressing properties of programs and are one of the ways of providing program specifications. They are used for detecting deviations of the program behavior (symptoms) with respect to such assertions, or to ensure that no such deviations exist (correctness). This thesis concentrates on use of the `pred` assertions of the Ciao assertion language [42, 85, 40], following the formalization of [99, 105], given that such assertions are the most frequently used in practice, and they subsume the other assertion schemas in that language. In the following the term assertion is used to refer to a `pred` assertion.

Assertions allow specifying certain conditions on the constraint store that must hold at certain points of program derivations. The main intent behind the construction of a specification for a predicate using `pred` assertions is to define the set of all admissible preconditions for this predicate, and for each such precondition in turn specify the respective postcondition. I.e., `pred` assertions allow stating sets of related *preconditions* and *conditional postconditions* for a given predicate. These pre- and postconditions are formulas containing prop literals introduced earlier. This provides a direct link between the properties used in assertions and the corresponding run-time tests, which constitute (instrumented) calls to the predicates defining the properties. This also allows defining specifications that are more general than, e.g., classical types.

More formally, the set of assertions for a given predicate represented by a *Head* atom is composed of the (possibly empty) set of all statements of the form:

$$\texttt{:- pred } Head \texttt{ : } Pre_1 \texttt{ => } Post_1\texttt{.}$$
$$\ldots$$
$$\texttt{:- pred } Head \texttt{ : } Pre_n \texttt{ => } Post_n\texttt{.}$$

where *Head* is the same normalized atom, that denotes the predicate that the assertions apply to, and the Pre_i and $Post_i$ are assumed to be DNF formulas of prop literals that refer to the variables of *Head*. It is assumed that variables in assertions are renamed such that the *Head* atom is identical for all assertions for a given predicate. A set of assertions as above states that in any execution state $\langle Head :: G \mid \theta \rangle$ at least one of the Pre_i conditions should hold, and that, given the $(Pre_i, Post_i)$ pair(s) where Pre_i holds, then, if *Head* succeeds, the corresponding $Post_i$ should hold upon success.

Example 2.4. *The procedure* `qsort(A,B)` *is the usual one that relates lists* `A` *and their sorted versions* `B`*. The following assertions:*

```
:- pred qsort(A,B) : list(A) => (sorted(B), list(B)).
:- pred qsort(A,B) : list(B) => (permutation(B,A),list(A)).
```

state that (restrict the meaning of `qsort/2` *to):*

- `qsort(A,B)` *should be called either with* `A` *constrained to a list or with* `B` *constrained to a list;*
- *if* `qsort(A,B)` *succeeds when called with* `A` *constrained to a list then on success* `B` *should be a sorted list; and*
- *if* `qsort(A,B)` *succeeds when called with* `B` *constrained to a list then on success* `A` *should be a list which is a permutation of* `B`*.*

From this point on the set of assertions for a predicate represented by *Head* is denoted by $\mathcal{A}\langle Head \rangle$, and the set of all assertions in a program by $\mathcal{A}$.

ASSERTION CONDITIONS The different checks on the constraint store imposed by a set of assertions are normalized into a set of corresponding *assertion conditions* as follows:

Definition 2.5 (Assertion Conditions for a Predicate)**.** *Given a predicate represented by a normalized atom Head, if the corresponding set of assertions is* $\mathcal{A}\langle Head \rangle = \{A_1 \ldots A_n\}$*, with* $A_i =$ "`:- pred` $Head$ `:` Pre_i `=>` $Post_i$`.`" *the set of* assertion conditions *for Head is* $\mathcal{A}_C\langle Head \rangle = \{C_0, C_1, \ldots, C_n\}$*, where:*

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

The following assumptions are implicitly made:

- If there are no assertions associated with *Head* then the corresponding set of assertion conditions $\mathcal{A}_C\langle Head\rangle$ is empty.
- The set of assertion conditions for a program, denoted $\mathcal{A}_C$, is the union of the assertion conditions for each of the predicates in the program.
- Also, given a single assertion A_i its corresponding set of assertion conditions is defined as $\{C_0, C_i\}$.

The $\mathsf{calls}(Head, \bigvee_{i=1}^{n} Pre_i)$ conditions encode the checks that ensure that the calls to the predicate represented by *Head* are within those admissible by the set of assertions, and we thus call them the *calls assertion conditions*. The conditions $\mathsf{success}(Head, Pre_i, Post_i)$ encode the checks for compliance of the successes for particular sets of calls, and we thus call them the *success assertion conditions*. Informally, such a set of assertions states that in any execution state $\langle Head :: G \mid \theta\rangle$ at least one of the Pre_i conditions should hold, and that, given the $(Pre_i, Post_i)$ pair(s) where Pre_i holds, then, if the predicate succeeds, the corresponding $Post_i$ should hold upon success.

Example 2.5. *The assertion conditions corresponding to the predicate assertions for* `qsort/2` *are as follows:*

$$\begin{aligned}
&\mathit{calls}(qsort(A,B),(list(A)\vee list(B)))\\
&\mathit{success}(qsort(A,B),list(A),(sorted(B)\wedge list(B)))\\
&\mathit{success}(qsort(A,B),list(B),(permutation(B,A)\wedge list(A)))
\end{aligned}$$

In order to define the semantics of assertion conditions, the auxiliary partial functions prestep and step are introduced as follows:

$$\begin{aligned}
\mathsf{prestep}(L_a, D) = (\theta,\sigma) &\triangleq D_{[-1]} = \langle L :: G \mid \theta\rangle \wedge \exists\sigma\ L = \sigma(L_a)\\
\mathsf{step}(L_a, D) = (\theta,\sigma,\theta') &\triangleq D_{[-1]} = \langle G \mid \theta'\rangle\ \wedge \exists\sigma\ L = \sigma(L_a)\\
&\qquad\wedge\ \exists i\ D_{[i]} = \langle L :: G \mid \theta\rangle
\end{aligned}$$

Given a derivation whose current state is a call to L (normalized atom), the prestep function returns the substitution σ for L, and the constraint store θ at the predicate *call* (i.e., just before the literal is reduced). Given a derivation whose current state corresponds exactly to the return from a call to L, the step function returns the substitution σ for L, the constraint store θ at the call to L, and the constraint store θ' at L's *success* (i.e., just after all literals introduced from the body of L have been fully reduced). Using these functions, the semantics of *calls* and *success* assertion conditions are given by the following definition:

Definition 2.6 (Valuation of an Assertion Condition on a Derivation)**.** *Given a* calls *or* success *assertion condition C, the* valuation of C on a derivation D*, denoted* $\mathsf{solve}(C, D)$ *is defined as follows:*

$$\begin{aligned}\mathsf{solve}(\mathsf{calls}(L, Pre), D) &\triangleq (\mathsf{prestep}(L, D) = (\theta, \sigma)) \\ &\Rightarrow (\theta \Rightarrow \sigma(Pre)) \\ \mathsf{solve}(\mathsf{success}(L, Pre, Post), D) &\triangleq (\mathsf{step}(L, D) = (\theta, \sigma, \theta')) \\ &\Rightarrow ((\theta \Rightarrow \sigma(Pre)) \Rightarrow (\theta' \Rightarrow \sigma(Post)))\end{aligned}$$

where L is a normalized atom.

ASSERTION STATUS As the intended use of assertions is to perform error detection and verification with respect to partial correctness, i.e., to ensure that the program does not produce unexpected results for *valid* ("expected") queries.[3] the notion of program is extended to include assertions and valid queries.

Definition 2.7 (Annotated Program)**.** *An annotated program is a tuple $(P, \mathcal{Q}, \mathcal{A})$ where P is a constraint logic program $\mathcal{Q}$ is a set of valid queries, and $\mathcal{A}$ is a set of assertions. As before, $\mathcal{A}_C$ denotes the set of* calls *and* success *assertion conditions derived from $\mathcal{A}$.*

In the context of annotated programs we extend *derivations* to operate on the set of valid queries as follows: $\mathsf{derivs}(\mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} \mathsf{derivs}(Q)$.

Definition 2.8 (Assertion Condition Status)**.** *Given the set of queries $\mathcal{Q}$, the assertion condition C can be either* checked *or* false*, as follows:*

$$\begin{aligned}\mathsf{checked}(C) &\triangleq \forall D \in \mathsf{derivs}(\mathcal{Q}) \,.\, \mathsf{solve}(C, D) \\ \mathsf{false}(C) &\triangleq \exists D \in \mathsf{derivs}(\mathcal{Q}) \,.\, \neg\mathsf{solve}(C, D)\end{aligned}$$

Definition 2.9 (Assertion Status)**.** *In an annotated program $(P, \mathcal{Q}, \mathcal{A})$ an assertion $A \in \mathcal{A}$ is* checked *(*false*) if all (any) of the corresponding assertion conditions are* checked *(*false*).*

Definition 2.10 (Partial Correctness)**.** *An annotated program $(P, \mathcal{Q}, \mathcal{A})$ is* partially correct *w.r.t. the set of assertions $\mathcal{A}$ and the set of queries $\mathcal{Q}$ iff $\forall A \in \mathcal{A}$, A is checked for $\mathcal{Q}$.*

Note that it follows immediately that a program is partially correct if all its assertion conditions are checked. The goal of assertion checking is thus to determine whether each assertion A is false or checked for $\mathcal{Q}$. Again, for this it is sufficient to prove the corresponding assertions conditions false or checked. There are two kinds of approaches to doing this (which can also be combined). While it is in general not

3 In practice, this set of expected queries is determined from module interfaces that define the set of exported predicates.

possible to try all derivations stemming from $\mathcal{Q}$, an alternative is to explore a hopefully representative set of them [61]. Though this does not allow fully validating the program in general, it makes it possible to detect many incorrectness problems. An alternative approach is to use global analysis techniques and is based on computing safe approximations of the program behavior statically [12, 43].

Finally, in addition to checked and false assertions, *true* assertions will be considered. True assertions differ from checked assertions in that true assertions hold in the program for any set of queries $\mathcal{Q}$.

Definition 2.11 (True Assertion)**.** *An assertion A is* true *iff for its corresponding assertion conditions C_i it holds that $\forall \mathcal{Q},\ \forall D \in \mathit{derivs}(\mathcal{Q}) : \mathit{solve}(C_i, D)$.*

Clearly, any assertion which is true in the program is also checked for any $\mathcal{Q}$, but not vice-versa. Since true assertions hold for any possible query they can be regarded as query-independent properties of the program. Thus, true assertions can be used to express analysis information, as already done, for example, in [10]. This information can then be reused when analyzing the program for different queries.

2.3 SEMANTICS WITH ASSERTIONS

An operational semantics which checks whether assertion conditions hold or not while computing the derivations from a query is provided below. Every assertion condition C is related to a unique identifier c via a mapping $\mathsf{id}(C) = c$, and the identifiers are used to keep track of any violated assertion conditions. The $\mathsf{err}(c)$ literal denotes a special goal that marks a derivation finished because of the violation of the assertion condition with identifier c. A finished derivation from a query (L, θ) is now *successful* if the last state is of the form $\langle \square \mid \theta' \rangle$, *erroneous* if the last state is of the form $\langle \mathsf{err}(c) \mid \theta' \rangle$, or *failed* otherwise. The set of literals is extended with *check literals*, syntactic objects of the form $\mathsf{check}(c)$ where c is an identifier for an assertion condition.Thus, a *literal* is now a constraint, an atom, or a check literal.[4]

Note that this operational semantics assumes that program execution terminates as soon as any one assertion condition is violated. An alternative one that collects violated assertion condition identifiers is discussed in Chapter 6.

Definition 2.12 (Operational Semantics for Programs with Assertions)**.** *A state $S = \langle L :: G \mid \theta \rangle$ can be* reduced *to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, as follows:*

1. *If L is a constraint then $S' = \langle G \mid \theta \wedge L \rangle$ if $\theta \wedge L$ is satisfiable.*

4 While check literals are simply instrumental here, note that they are also directly useful for supporting program point assertions (which are basically check literals that appear in the body of clauses) [84].

2. *If L is an atom and $\exists(L \leftarrow B) \in \mathsf{cls}(L)$, then the new state is obtained as*

$$S' = \begin{cases} \langle \mathit{err}(c) \mid \theta \rangle & \text{if } \exists\, C = \mathit{calls}(L, \mathit{Pre}) \in \mathcal{A}_C\langle L \rangle \\ & \quad \wedge \mathit{id}(C) = c \wedge \theta \not\Rightarrow \mathit{Pre} \\ \langle B :: G' \mid \theta \rangle & \text{otherwise} \end{cases}$$

and $G' = \mathit{check}(c_1) :: \ldots :: \mathit{check}(c_n) :: G$ such that $C_i = \mathit{success}(L, \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A}_C\langle L \rangle \wedge \mathit{id}(C_i) = c_i \wedge \theta \Rightarrow \mathit{Pre}_i$.

3. *If L is a check literal $\mathit{check}(c)$, then S' is obtained as*

$$S' = \begin{cases} \langle \mathit{err}(c) \mid \theta \rangle & \text{if } \exists\, C = \mathit{success}(L', \_, \mathit{Post}) \in \mathcal{A}_C\langle L' \rangle \\ & \quad \wedge\ \mathit{id}(C) = c\ \wedge\ \theta \not\Rightarrow \mathit{Post} \\ \langle G \mid \theta \rangle & \text{otherwise} \end{cases}$$

The set of derivations for a program from its set of queries $\mathcal{Q}$ using the semantics with assertions is denoted $\mathsf{derivs}_{\mathcal{A}}(\mathcal{Q})$.

RUN-TIME CHECKING OF ASSERTIONS The main idea behind run-time checking of assertions is, given a set of queries $\mathcal{Q}$, and a set of assertions $\mathcal{A}$, to directly apply Definition 2.8 in order to determine whether the respective assertion conditions in $\mathcal{A}_C$ are checked or false, i.e., obtaining (a subset of) the derivations by running the program and determining whether they belong to the error set of the assertions. It is not to be expected that Definition 2.12 can be used to determine that an assertion is checked, as this would require checking the derivations from all valid queries, which is in general an infinite set and thus checking would not terminate.

In this situation, and as mentioned before, an alternative is to perform run-time checking for a hopefully representative set of queries. Though this does not allow fully validating the program in general, it allows detecting many incorrectness problems. Theorem 2.1 below guarantees that the behavior of a partially correct program is the same under the operational semantics of the Definition 2.1 and under the semantics with assertions of the Definition 2.12.

Definition 2.13 (Error-erased Derivation)**.** *The set of* error-erased *derivations from $\rightsquigarrow_{\mathcal{A}}$ is obtained by a syntactic rewriting $(-)^{\circ}$ that removes states that begin by a check literal, and check literals from goals. It is recursively defined as follows:*

$$\{D_1, \ldots, D_n\}^\circ = \{D_1^\circ, \ldots, D_n^\circ\}$$

$$(S_1, \ldots, S_m, S_{m+1})^\circ = \begin{cases} (S_1, \ldots, S_m)^\circ \\ \quad \textit{if } S_{m+1} = \langle \mathit{check}(\_) :: \_ \mid \_ \rangle \\ (S_1, \ldots, S_m)^\circ \parallel ((S_{m+1})^\circ) \\ \quad \textit{otherwise} \end{cases}$$

$$\langle G \mid \theta \rangle^\circ = \langle G^\circ \mid \theta \rangle$$

$$(L :: G)^\circ = \begin{cases} G^\circ & \textit{if } L = \mathit{check}(\_) \\ L :: (G^\circ) & \textit{otherwise} \end{cases}$$

$$\square^\circ = \square$$

where $\parallel$ stands for sequence concatenation.

Theorem 2.1 (Correctness and Completeness Under Assertion Checking)**.** *For any annotated program $(P, \mathcal{Q}, \mathcal{A})$, given $\mathcal{D} = \mathit{derivs}(\mathcal{Q})$ and $\mathcal{D}' = \mathit{derivs}_{\mathcal{A}}(\mathcal{Q})$, $\mathcal{D} = (\mathcal{D}')^\circ$.*

In other words, for any annotated program the *error-erased* derivations obtained from $\rightsquigarrow_{\mathcal{A}}$ and the derivations obtained from $\rightsquigarrow$ are equivalent after filtering out check literals.

Proof. We will prove $\mathcal{D} = (\mathcal{D}')^\circ$ by showing that $\mathcal{D} \subseteq (\mathcal{D}')^\circ$ and $\mathcal{D} \supseteq (\mathcal{D}')^\circ$:

- ($\subseteq$) For all $D \in \mathcal{D}$ exists $D' \in \mathcal{D}'$ so that $D = (D')^\circ$.
- ($\supseteq$) For all $D' \in \mathcal{D}'$, $D = (D')^\circ \in \mathcal{D}$.

We will prove each case:

- ($\subseteq$) Let $D = (S_1, \ldots, S_n)$, $S_i = \langle L_i \mid \theta_i \rangle$, for some $Q = (L_1, \theta_1) \in \mathcal{Q}$ and $S_i \rightsquigarrow S_{i+1}$. Proof by induction on the length n of D:
  - Base case ($n = 1$). Let $S'_1 = \langle L_1 \mid \theta_1 \rangle$. It holds that $(S'_1)^\circ = \langle L_1 \mid \theta_1 \rangle)^\circ = \langle L_1^\circ \mid \theta_1 \rangle = \langle L_1 \mid \theta_1 \rangle = S_1$ (since L_1 does not contain any check literal). Thus, $(D')^\circ = ((S'_1))^\circ = ((S'_1)^\circ) = (S_1) = D$.
  - Inductive case (show $n + 1$ assuming n holds). For each $D_2 = (S_1, \ldots, S_n, S_{n+1})$ there exists $D'_2 = (S'_1, \ldots, S'_m, S'_{m+1})$ such that $(D'_2)^\circ = D_2$. Given the induction hypothesis it is enough to show that for each $S_n \rightsquigarrow S_{n+1}$ there exists $S'_m \rightsquigarrow_{\mathcal{A}} S'_{m+1}$, such that $(S'_{m+1})^\circ = S_{n+1}$. According to $\rightsquigarrow_{\mathcal{A}}$ (see Def. 2.12), L'_{m+1} and θ'_{m+1} are obtained in the same way than in $\rightsquigarrow$ (see Def. 2.1), except for the introduction of check literals. Since all check literals are removed in error-erased states, it follows that $(S'_{m+1})^\circ = S_{n+1}$. $\square$

- ($\supseteq$) Let $D' = (S'_1, \ldots, S'_m)$, $S'_i = \langle L'_i \mid \theta'_i \rangle$, for some $Q = (L'_1, \theta'_1) \in \mathcal{Q}$ and $S'_i \rightsquigarrow_{\mathcal{A}} S'_{i+1}$. Proof by induction on the length m of D':
  - Base case ($m = 1$). It holds that $(S'_1)^\circ = S_1$ (showed in base case for $\subseteq$). Then $(D')^\circ = D \in \mathcal{D}$.
  - Inductive case (show $m+1$ assuming m holds). We want to show that given $D'_2 = (S'_1, \ldots, S'_m, S'_{m+1})$, $(D'_2)^\circ = D_2 \in \mathcal{D}$. Given the induction hypothesis it is enough to show that for each $S'_m \rightsquigarrow_{\mathcal{A}} S'_{m+1}$ there exists $S_n \rightsquigarrow S_{n+1}$ such that $S_{n+1} = (S'_{m+1})^\circ$ (so that $(S_1, \ldots, S_n, S_{n+1}) \in \mathcal{D}$) or $S_n = (S'_{m+1})^\circ$ ($D_2 = D \in \mathcal{D}$). According to cases of Def. 2.12:
    - If L'_m begins with a check literal then $(L'_{m+1})^\circ = (L'_m)^\circ$. Thus $(S'_{m+1})^\circ = (S'_m)^\circ = S_n$.
    - Otherwise, it holds that $(S'_{m+1})^\circ = S_{n+1}$ using the same reasoning than in the inductive case for $\subseteq$.

□

This result implies that the semantics with assertions can also be used to obtain all answers to the original query. Furthermore, the following theorem guarantees that the proposed operational semantics for annotated programs can be used in order to detect (all) violations of assertions:

Definition 2.14 (Run-time Valuations of an Assertion Condition on a Derivation)**.** *The run-time valuation of an assertion condition C on a derivation D is given by:*

$$\begin{aligned} \mathit{rtsolve}(C, D) \triangleq \forall C', \sigma, L\ (C' \in \mathcal{A}_C\langle L \rangle \wedge \sigma(C) = C' \wedge \mathit{id}(C') = c) \\ \Rightarrow D_{[-1]} \neq \langle \mathit{err}(c) \mid \_ \rangle \end{aligned}$$

I.e., condition $\mathsf{rtsolve}(C, D)$ is valid if none of the identifiers c of assertion conditions C in the program appear inside the error goal in the final state of the derivation D.

Theorem 2.2 (Run-time Error Detection)**.** *For any annotated program $(P, \mathcal{Q}, \mathcal{A})$, $C \in \mathcal{A}_C$ is false iff $\exists\ D \in \mathit{derivs}_{\mathcal{A}}(\mathcal{Q})$ s.t. $\neg \mathit{rtsolve}(C, D)$.*

Proof. Let us assume assertion condition $A \in \mathcal{A}_C$ is false
$\Leftrightarrow$ from Def. 2.9 and Def. 2.5 $\exists \{C_c, C_s\}$ assertion conditions s.t. $\mathsf{false}(C_c) \vee \mathsf{false}(C_s)$, where $C_c = \mathsf{calls}(L, Pre)$ and $C_s = \mathsf{success}(L, Pre, Post)$ correspond to A. Let us first prove the $\neg\mathsf{rtsolve}(C_c, D)$ case and then the $\neg\mathsf{rtsolve}(C_s, D)$ one:

- $\mathsf{false}(C_c)$
  $\Leftrightarrow$ from Def. 2.8 $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\neg\mathsf{solve}(C_c, D)$
  $\Leftrightarrow$ from Def. 2.6 $(\mathsf{prestep}(L, D) = (\theta, \sigma) \wedge \theta \not\Rightarrow \sigma(Pre))$
  $\Leftrightarrow$ from Def. 2.12 $\exists\ S \rightsquigarrow_{\mathcal{A}} S'$ where:

$$S = \langle L :: G \mid \theta \rangle \text{ s.t. } \exists\ C = \mathsf{calls}(L, Pre) \in \mathcal{A}_C\langle L \rangle \wedge \mathsf{id}(C) = c$$
$$S' = \langle \mathsf{err}(c) \mid \theta \rangle$$

$\Leftrightarrow$ from Def. 2.14 $\neg\mathsf{rtsolve}(C_c, D)$ □

- $\mathsf{false}(C_s)$
  $\Leftrightarrow$ from Def. 2.8 $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\neg\mathsf{solve}(C_s, D)$
  $\Leftrightarrow$ from Def. 2.6 $(\mathsf{step}(L, D) = (\theta, \sigma, \theta') \wedge \theta \mapsto \sigma(Pre) \wedge \theta' \not\Rightarrow \sigma(Post))$
  $\Leftrightarrow$ from Def. 2.12 $\exists\ S \rightsquigarrow^*_{\mathcal{A}} S' \rightsquigarrow_{\mathcal{A}} S''$ where

$$
\begin{aligned}
S =& \langle L :: G \mid \theta \rangle \\
& \exists\ C = \mathsf{success}(L, Pre, Post) \in \mathcal{A}_C\langle L \rangle \\
& \wedge\, \mathsf{id}(C) = c \wedge \theta \mapsto Pre \\
S' =& \langle \mathsf{check}(c) :: G \mid \theta' \rangle \wedge \theta' \not\Rightarrow Post \\
S'' =& \langle \mathsf{err}(c) \mid \_ \rangle
\end{aligned}
$$

  $\Leftrightarrow$ from Def. 2.14 $\neg\mathsf{rtsolve}(C_s, D)$

□

Theorem 2.2 states that assertion condition C is false iff there is a derivation D in which the run-time valuation of the assertion condition of C in D is false (i.e., if at least one instance of the assertion condition A is in the error set for such derivation D). Given a set of $false$ assertion conditions we can easily derive the set of $false$ assertions using Def. 2.5. In order to prove that any assertion is checked this has to be done for all possible derivations for all possible queries, which is often not possible in practice. This is why analysis based on abstractions is often used in practice for this purpose.

3

RUN-TIME CHECKING WITH PROPERTY CACHING

Run-time checking frameworks, either as components of an IDE or as separate tools, are usually responsible for instrumenting programs with checks from the program annotations. However, run-time testing in these frameworks can generally incur high penalty in execution time and/or space over the standard program execution without tests.

The standard operational semantics with run-time checking revisited in Chapter 2 (see Def. 2.12) has the same potential problems as other approaches which perform exhaustive tests: it can be prohibitively expensive, both in terms of time and memory overhead.

Example 3.1 (Complexity Jump)**.** *Consider the usual* `length/2` *predicate that returns the length of its list input argument:*

```
:- pred length(L,N) : list(L) => num(N).

length([],0).
length([H|T],N) :- length(T,M), N is M + 1.
```

Checking that the first argument of the `length/2` *predicate is a list at each recursive step turns the standard $O(n)$ algorithm into $O(n^2)$.*

Our objective in this chapter is to develop an approach to run-time testing that is efficient while being minimally obtrusive and remaining exhaustive. We present an approach based on the use of memoization to cache intermediate results of check evaluation in order to avoid repeated checking of previously verified properties over the same data structure. Memoization has of course a long tradition in (C)LP in uses such as tabling resolution [109, 29, 114], including also sharing and memoizing tabled sub-goals [119], for improving termination. Memoization has also been used in program analysis [117, 68], where tabling resolution is performed using abstract values. However, in tabling and analysis what is tabled are call-success patterns and in our case the aim is to cache the results of test execution.

Using the Ciao assertion model [42, 85, 43] as a basis allows us to provide an operational semantics of programs with checks and caching, as well as a concrete implementation from which we derive experimental results. We also present a program transformation for implementing the run-time checks that is more efficient than previous proposals [85, 61, 62]. Our experimental results provide evidence that using a relatively small cache leads to significant decreases in run-time checking overhead.

3.1 OPERATIONAL SEMANTICS WITH CACHING

We base our approach on an operational semantics which modifies the run- time checking to maintain and use a *cache store*:

Definition 3.1 (Cache Store)**.** *The* cache store $\mathbb{M}$ *is a special constraint store which temporarily holds results from the evaluation of* prop *literals w.r.t. the standard constraint store* θ*.*

We introduce an extended program state of the form $\langle G \mid \theta \mid \mathbb{M}\rangle$ and a *cached* version of "*succeeds trivially*"from the Definition 2.3:

Definition 3.2 (Succeeds Trivially with Cache)**.** *Given a prop literal* L*, it succeeds trivially for* θ *and* $\mathbb{M}$ *in program* P*, denoted* $\theta \overset{\mathbb{M}}{\Mapsto} L$*, iff* $L' = \sigma(L)$ *and either* $L' \in \mathbb{M}$ *or* $\theta \mapsto L'$*.*

Also, the cache store is updated based on the results of the prop checks, formalized in the following definitions:

Definition 3.3 (Updates on the Cache Store)**.** *Let us consider a DNF formula* $Props = \bigvee_{i=1}^{n}(\bigwedge_{j=0}^{m(i)} L_{ij})$*, where each* L_{ij} *is a prop literal. By* $\mathit{lits}(Props) = \{L_{ij} | i \in [1:n], j \in [0:m(i)]\}$ *we denote the set of all literals which appear in* $Props$*. The* cache update *operation is defined as a function* $\mathit{upd}(\theta, \mathbb{M}, Props)$ *such that:*

$$\mathit{upd}(\theta, \mathbb{M}, Props) \subseteq \mathbb{M} \cup \{L | (\theta \mapsto L) \wedge (L \notin \mathbb{M}) \wedge (L \in \mathit{lits}(Props)\}$$

Note that a precise definition of cache update is left open in this semantics. Contrary to θ, updates to the cache store $\mathbb{M}$ are not monotonic since we allow the cache to "forget" information as it fills up, i.e., we assume from the start that $\mathbb{M}$ is of limited capacity. However, that information can always be recovered via recomputation of property checks. In practice the exact cache behavior depends on parts of the low-level abstract machine state that are not available at this abstraction level.

Definition 3.4 (Reductions with Assertions and Cache Store)**.** *A state* $S = \langle L :: G \mid \theta \mid \mathbb{M}\rangle$*, where* L *is a literal, can be* reduced *to a state* S'*, as follows:*

1. *If* L *is a constraint then* $S' = \langle G \mid \theta \wedge L \mid \mathbb{M}\rangle$ *if* $\theta \wedge L$ *is satisfiable.*
2. *If* L *is an atom and* $\exists (L \leftarrow B) \in \mathsf{cls}(L)$*, then*

$$S' = \begin{cases} \{\langle \mathit{err}(c) \mid \_ \mid \_\rangle\} & \text{if } \exists\, C = \mathit{calls}(L, Pre) \in \mathcal{A}_C\langle L\rangle \\ & \wedge \mathit{id}(C) = c \wedge \theta \overset{\mathbb{M}}{\not\Mapsto} Pre \\ \langle B :: G' \mid \theta \mid \mathbb{M}'\rangle & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathbb{M}' &= \mathit{upd}(\theta, \mathbb{M}, Pre) \\ G' &= \mathit{check}(c_1) :: \ldots :: \mathit{check}(c_n) :: G \end{aligned}$$

such that $C_i = \mathit{success}(L, \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A}_C\langle L\rangle \wedge\ \mathit{id}(C_i) = c_i \wedge \theta \overset{\mathbb{M}}{\mapsto} \mathit{Pre}_i$.

3. *If* L *is a check literal* $\mathsf{check}(c)$*, then* S' *is obtained as*

$$S' = \begin{cases} \langle \mathit{err}(c) \mid \theta \mid \_\rangle & \mathit{if}\ \exists\ C = \mathit{success}(L', \_, \mathit{Post}) \in \mathcal{A}_C\langle L'\rangle \\ & \wedge \mathit{id}(C) = c \wedge \theta \overset{\mathbb{M}}{\not\mapsto} \mathit{Post} \\ \langle G \mid \theta \mid \mathbb{M}'\rangle & \mathit{otherwise} \end{cases}$$

and $\mathbb{M}' = \mathit{upd}(\theta, \mathbb{M}, \mathit{Post})$.

Again, the order in which the $\mathsf{check}(c)$ *literals are selected is irrelevant.*

3.2 IMPLEMENTATION OF RUN-TIME CHECKING WITH CACHING

We use the traditional definitional transformation [85] as a basis of our implementation of the operational semantics with cached checks. This consists of a program transformation that introduces *wrapper* predicates that check calls and success assertion conditions while running on a standard (C)LP system. However, we propose a novel transformation that, in contrast to previous approaches, groups all assertion conditions for the same predicate together to produce optimized non-repetitive checks.

For every predicate p the transformation replaces all program clauses $p(\bar{x}) \leftarrow \mathit{body}$ by $p'(\bar{x}) \leftarrow \mathit{body}$, where p' is a new predicate symbol, and inserts the wrapper clauses given by $\mathsf{wrap}(p(\bar{x}), p')$. The wrapper generator is defined as follows:

$$\mathsf{wrap}(p(\bar{x}), p') = \left\{ \begin{array}{l} p(\bar{x}) \leftarrow p_C(\bar{x}, \bar{r}), p'(\bar{x}), p_S(\bar{x}, \bar{r}). \\ p_C(\bar{x}, \bar{r}) \leftarrow \mathit{ChecksC}. \\ p_S(\bar{x}, \bar{r}) \leftarrow \mathit{ChecksS}. \end{array} \right\}$$

where *ChecksC* and *ChecksS* are the optimized compilation of pre- and postconditions $\bigvee_{i=1}^{n} \mathit{Pre}_i$ and $\bigwedge_{i=1}^{n} (\mathit{Pre}_i \rightarrow \mathit{Post}_i)$ respectively, for $C_0 = \mathsf{calls}(p(\bar{x}), \bigvee_{i=1}^{n} \mathit{Pre}_i)$, $C_i = \mathsf{success}(p(\bar{x}), \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A}_C\langle p(\bar{x})\rangle$, and the additional *status* variables $\bar{r}$ are used to communicate the results of each Pre_i evaluation to the corresponding $(\mathit{Pre}_i \rightarrow \mathit{Post}_i)$ check. This way, without any modifications to the literals calling p in the bodies of clauses in the program (and in any other modules that contain calls to p), after the transformation run-time checks will be performed for all these calls to p since p (now p') will be accessed via the wrapper predicate.

The compilation of checks for assertion conditions emits a series of calls to a `reify_check(P,R)` predicate,

```
:- pred reify_check(P,Res) : prop(P) => bool(Res).
```

which accepts as the first argument a property and unifies its second argument with `1` or `0`, depending on whether the property check succeeded or not. The results of those reified checks are then combined and evaluated as boolean algebra expressions using bitwise operations and the Prolog `is/2` predicate. That is, the logical operators $(A \vee B)$, $(A \wedge B)$, and $(A \rightarrow B)$ used in encoding assertion conditions are replaced by their bitwise logic counterparts `R is A \/ B`, `R is A /\ B`, `R is (A # 1) \/ B`, respectively.

The purpose of reification and this compilation scheme is to make it possible to optimize the logic formulae containing properties that result from the combination of several `pred` assertions (i.e., the assertion conditions). The optimization consists in reusing the reified status R when possible, which happens in two ways. First, the *prop* literals which appear in *Pre* or *Post* formulas are only checked once (via `reify_check/2`) and then their reified status R is reused when needed. Second, the reified status of each *Pre* conjunction is reused both in *ChecksC* and *ChecksS*, where checks of *prop* literals are substituted by `reify_check/2` evaluation results.

In practice the wrap$(p(\bar{x}),p')$ clause generator shares the minimum number of status variables and omits *trivial* assertion conditions, i.e., those with `true` conditions in one of their parts. For instance, excluding $p_S(\bar{x},\bar{r})$ preserves low-level optimizations such as last call optimization.[1]

The translation procedure for assertion conditions has three principal phases. During the first phase two routines occur:

1. All assertion conditions in the program are collected and grouped by their respective predicate.
2. From each such set two smaller sets are derived:
   a) a set of unique pairs (variable,property);
   b) a set of unique property combinations.

This is illustrated in more detail in the example 3.2 below.

Example 3.2 (Program transformation)**.** *Consider the following annotated program:*

```
:- pred p(X,Y) : (int(X) , var(Y)) => (int(X), int(Y)). % A1
:- pred p(X,Y) : (int(X) , var(Y)) => (int(X), atm(Y)). % A2
:- pred p(X,Y) : (atm(X) , var(Y)) => (atm(X), atm(Y)). % A3

p(1,42).
p(2,gamma).
p(a,alpha).
```

1 Even though in this work the $p_C(\bar{x},\bar{r})$ and $p_S(\bar{x},\bar{r})$ predicates follow the usual bytecode-based compilation path, note that they have a concrete structure that is amenable to further optimizations (like specialized WAM-level instructions or a dedicated interpreter).

From the set of assertions $\{A1, A2, A3\}$ *the following assertion conditions are constructed:*

$$
\begin{aligned}
C_0 &= \mathit{calls}(p(X,Y), (int(X) \wedge var(Y)) \vee ((atm(X) \wedge var(Y)))) \\
C_1 &= \mathit{success}(p(X,Y), (int(X) \wedge var(Y)), (int(X) \wedge int(Y))) \\
C_2 &= \mathit{success}(p(X,Y), (int(X) \wedge var(Y)), (int(X) \wedge atm(Y))) \\
C_3 &= \mathit{success}(p(X,Y), (atm(X) \wedge var(Y)), (atm(X) \wedge atm(Y)))
\end{aligned}
$$

The resulting optimized program transformation is:

```
p(X,Y) :-
    p_c(X,Y,R3,R4),
    p'(X,Y),
    p_s(X,Y,R3,R4).

p_c(X,Y,R3,R4) :-
    reify_check(atm(X),R0),
    reify_check(int(X),R1),
    reify_check(var(Y),R2),
    R3 is R1/\R2,
    R4 is R0/\R2,
    Rc is R3\/R4,
    error_if_false(Rc).
```

```
p_s(X,Y,R3,R4) :-
    reify_check(atm(X),R5),
    reify_check(int(X),R6),
    reify_check(atm(Y),R7),
    reify_check(int(Y),R8),
    Rs is (R3#1\/(R6/\R8))
       /\ (R3#1\/(R6/\R7))
       /\ (R4#1\/(R5/\R7)),
    error_if_false(Rs).

p'(1,42).
p'(2,gamma).
p'(a,alpha).
```

Note that `A1` *and* `A2` *have identical preconditions, and this is reflected in having only one property combination,* `R3`*, for both of them. The same works for individual properties: in* C_0 *literal* $int(X)$ *appears twice, literal* $var(Y)$ *three times, but all such occurrences correspond to only one check in the code respectively.*

The error-reporting predicates `error_if_false/1` in the instrumented code implement the final state transition in the operational semantics of the Definition 3.4. These predicates abstract away the details of whether errors produce exceptions, are reported to the user, or are simply recorded.

The cache itself is accessed fundamentally within the `reify_check/2` predicate. Although the concrete details for a particular use case (and a corresponding set of experiments) will be described later, we discuss the main issues and trade-offs involved in cache implementation in this context. First, although the cache will in general be software-defined and dynamically allocated, in any case the aim is to keep it small with a bounded limit (typically a fraction of the stacks), so that it does not have a significant impact on the memory consumption of the program.

Also, in order to ensure efficient lookups and insertions of the cache elements, it may be advantageous not to store the property calls literally but rather their memory representation. This means however that, e.g., for *structure-copying* term representation, a property may appear more than once in the cache for the same term if its representation appears several times in memory.

Furthermore, insertion and removal (*eviction*) of entries can be optimized using heuristics based on the cost of checks (e.g., not caching simple checks like `integer/1`), the entry index number (such as direct-mapped), the history of entry accesses (such as Least Recently Used (LRU)), or caching *contexts* (such as caching depth limits during term traversal in regular type checks).

Finally, failure and some of the stack maintenance operations such as reallocations for stack overflows, garbage collection, or backtracking need updates on the cache entries (due to invalidation or pointer reallocation). Whether it is more optimal to evict some or all entries, or update them is a nontrivial decision that defines another dimension in heuristics.

3.3 APPLICATION TO REGULAR TYPE CHECKING

As concrete properties to be used in our experiments we select a simple yet useful subset of the properties than can be used in assertions: the regular types [27] often used in (C)LP systems. Regular types are properties whose definitions are *regular programs*, defined by a set of clauses, each of the form:

$$p(x, v_1, \ldots, v_n) \leftarrow B_1, \ldots, B_k$$

1. x is a linear term (whose variables, which are called *term variables*, are unique)
2. In all clauses defining $p/(n+1)$ the terms x do not unify except maybe for one single clause in which x is a variable.
3. v_1, ..., v_n are unique variables, which are called *parametric variables*.
4. Each B_i is of the form:
   a) $t(z)$ where z is one of the *term variables* and t is a *regular type expression*;
   b) $q(y, t_1, \ldots, t_m)$ where $q/(m+1)$ is a *regular type*, $t_1, \ldots, t_m$ are *regular type expressions*, and y is a *term variable*.
5. Each term variable occurs at most once in the clause body (and should be as the first argument of a literal).

A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables. A parametric type functor is a regular type, defined by a regular program.

INSTANTIATION CHECKS A standard technique to check membership on regular types is based on *tree automata*. In particular, the regular types defined above are recognizable by top-down deterministic automata. This also includes parametric regtypes, provided their parameters are instantiated with concrete types during checking, since then they can be reduced to non-parametric regtypes. [2]

Let us recall some basics on deterministic tree automata, as they will be the basis of our regtype checking algorithm. A tree automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, Q_f \rangle$ where Σ, Q, Δ, Q_f are finite sets such that: Σ is a signature, Q is a finite set of states, Δ is the set of transitions of the form $f(q_1, \ldots, q_n) \to q$ where $f \in \Sigma$, $q, q_1, \ldots, q_n \in Q$ with n being the arity of f, and $Q_f \subseteq Q$ is the set of final states. The automaton is *top-down deterministic* if $|Q_f| = 1$ and for all $f \in \Sigma$ and all $q \in Q$ there exists at most one sequence $q_1, \ldots, q_n$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$.

Translation of regular types (or instances of parametric regular types for particular types) from Prolog clauses into deterministic top-down tree automata rules is straightforward. This representation is suitable for low-level encoding (e.g., using integers for q_i states and a map between each q_i state and its definition).

Example 3.3. *The following* `bintree/2` *regular type describes a binary tree of elements of type* `T`*. The corresponding translation into tree automata rules for the* `bintree(int)` *instance with* $Q_f = \{q_b\}$ *is shown to its right.*

```
:- regtype bintree/2.
bintree(empty,T).
bintree(tree(LC,X,RC),T) :-
        bintree(LC,T),
        T(X),
        bintree(RC,T).
```

$$\Delta = \{ \quad empty \to q_b \qquad tree(q_b, q_{int}, q_b) \to q_b \ \}$$

ALGORITHM FOR CHECKING REGULAR TYPES WITH CACHES We describe the RegCheck algorithm for regtype checking using caches in Algorithm 3.1. The `reify_check/2` predicate acts as the interface between RegCheck and the runtime checking framework. The algorithm is derived from the standard definition of *run* on tree automata. A run of a tree automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, Q_f \rangle$ on a tree $x \in T_\Sigma$ (terms over Σ) is a mapping ρ assigning a state to each occurrence (subterm) of $f(x_1, \ldots, x_n)$ of x such that:

$$f(\rho(x_1), \ldots, \rho(x_n)) \to \rho(f(x_1, \ldots, x_n)) \in \Delta$$

A term x is recognized by $\mathcal{A}$ if $\rho(x) \in Q_f$. For deterministic top-down recognition, the algorithm starts with the single state in Q_f

2 Note that checks are performed via entailment checks w.r.t. primitive (Herbrand) constraints. That means that $term(X)$ (which is always true) and $ground(X)$ (denoting all possible ground terms), despite having the same minimal Herbrand models as predicates, do not have the same s-model and are not interchangeable as regtype instantiation checks.

Algorithm 3.1 Check that the regular type of the term stored at x is t, at cache depth d.

```
function RegCheck(x, t, d)
    Find C ∈ Constructors(t) so that Functor(C) = Functor(x),
    otherwise return False
    if Arity(x) = 0 then                          ▷ Atomic value, not cached
        return True
    else if CacheLookup(x, t) then                ▷ Already in cache
        return True
    else if ∀i ∈ [1, Arity(x)].RegCheck(Arg(i, x), Arg(i, C), d + 1)
then
        if d < depthLimit then                    ▷ Insert in cache
            CacheInsert(x, t)
        return True                               ▷ In regtype
    else
        return False                              ▷ Not in regtype
```

(which for simplicity, we will use to identify each regtype and its corresponding automata) and follows the rules *backwards*. The tree automata transition rules for a regtype are consulted with the functions $Constructors(t) = \{C | C \to t \in \Delta\}$, $Arg(i, u)$ (the i-th argument of a constructor or term u), and $Functor(u)$ (the functor symbol, including arity, of a constructor or term u). Once there is a functor match, the regtypes of the arguments are checked recursively. To speed up checks, the cache is consulted (CacheLookup(x, t) searches for (x, t)) before performing costly recursion, and *valid* checks inserted (CacheInsert(x, t) inserts (x, t)) if needed (e.g., using heuristics, explained below). The cache for storing results of regular type checking is implemented as a *set* data structure that can efficiently insert and look up (x, t) pairs, where x is a term address[3] and t a regular type identifier. The specific implementation depends on the cache heuristics, as described below.

COMPLEXITY It is easy to show that complexity has $O(1)$ best case (if x was cached) and $O(n)$ worst case, with n being the number of tree nodes (or term size). In practice, the caching heuristics can drastically affect performance. For example, assume a full binary tree of n nodes. Caching all nodes at levels multiple of c will need $n/(2^{c+1} - 1)$ entries, with a constant cost for the worst case check (at most $2^{c+1} - 1$ will be checked, independently of the size of the term).

3 Since regtype checks are monotonic, this is safe as long as cache entries are properly invalidated on backtracking, stack movements, and garbage collection. Using addresses is a pragmatic decision to minimize the overheads of caching.

CACHE IMPLEMENTATION AND HEURISTICS In order to decide what entries are added and what entries are evicted to make room for new entries on cache misses, we have implemented several caching heuristics and their corresponding data structures. Entry eviction is controlled by *replacement policies*:

- Least-recently used (LRU) replacement and fully associative. Implemented as a hash table whose entries are nodes of a doubly linked list. The most recently accessed element is moved to the head and new elements are also added to the head. If cache size exceeds the maximal size allowed, the cache is pruned.
- Direct-mapped cache with collision replacement, with a simple hash function based on modular arithmetic on the term address. This is simpler but less predictable.

The insertion of new entries is controlled by the caching *contexts*, which include the regular type being checked and the location of the check:

- We do not cache simple properties (like primitive type tests, e.g., `integer/1`, etc), where caching is more expensive than recomputing.
- We use the check depth level in the cache interface for recursive regular types. Checks beyond this threshold depth limit are not cached. This gives priority to roots of data structures over internal subterms which may pollute the cache.

LOW-LEVEL C IMPLEMENTATION. In our prototype, this algorithm is implemented in C with some specialized cases (as required for our WAM-based representation of terms, e.g., to deal with atomic terms, list constructors, etc.).[4] The regtype definition is encoded as a map between functors (name and arity) and an array of q states for each argument. For a small number of functors, the map is implemented as an array. Efficient lookup for many functors is achieved using hash maps. Additionally, a number of implicit transition rules exist for primitive types (any term to q_{any}, integers to q_{int}, etc.) that are handled as special cases.

3.4 EXPERIMENTAL EVALUATION

To study the impact of caching on run-time overhead, we have evaluated the run-time checking framework on a set of 7 benchmarks,

4 Even though the algorithm can be easily implemented as a deterministic Prolog program, we chose in this work a specialized, lower-level implementation that can interact more directly with the optimized cache data structures.

for regular types. We consider benchmarks where we perform a series of element insertions in a data structure. Benchmarks `amqueue`, `set`, `B-tree`, and (binary) `tree` were adapted from the Ciao libraries; benchmarks `AVl-tree`, `RB-tree` and `heap` were adapted from the YAP libraries. These benchmarks can be divided into 4 groups:

A. simple list-based data structures: `amqueue`, `set`;

B. balanced tree-based structures that do not change the structural properties of their nodes on balancing: `AVl-tree`, `heap`;

C. balanced tree-based structures that change node properties: `B-tree` (changes the number of node children), `RB-tree` (changes node color);

D. unbalanced tree structures (`tree`).

A following example of regular types that we use in assertions is taken from the red-black trees and is provided below:

```
:- regtype node/1.
node('').
node(red(L,K,V,R))   :- node(L), term(K), term(V), node(R).
node(black(L,K,V,R)) :- node(L), term(K), term(V), node(R).

:- regtype rb_tree/1.
rb_tree(t(L,R)) :- node(L), node(R).
```

For each run of the benchmark suite the following parameters were varied: cache replacement policy (LRU, direct mapping), cache size (1 to 256 cells), and check depth threshold (1 to 5, and "infinite" threshold for unlimited check depth). Table 3.1 summarizes the results of the experiments. For each combination of the parameters it reports the optimal caching policy, LRU (L) or direct mapping (D). Also, for each of the benchmarks it reports an interval within which the worst case check depth varies.

The experiments show that the overhead of checks with depth threshold 2 (storing the regtype of the check argument and the regtypes of its arguments) is smaller than or equal to the one obtained with unlimited depth limit (Fig 3.1). A depth limit of 1 does not allow checks to store enough useful information about terms of most of the data structures (compare the overhead increase for `amqueue` with this and bigger limits), while unlimited checks tend to overwrite this information multiple times, so that it cannot be reused.

At the same time, for data structures represented by large nested terms (e.g., nodes of `B-trees`), deeper limits (3 or 4) for small inputs seem more beneficial for capturing such term structure. It can also be observed that the lower cost of element insert/lookup operations with the DM cache replacement policy results in having lower total overhead than with the LRU policy.

Table 3.1: Benchmarks

benchmark	assertions	regtypes	depth lim	cache size 256	128	64	32	max depth DM	LRU
amqueue	4	1	2	D	D	D	D	2	2
			∞	D	D	D	D		
set	4	1	2	D	D	D	D	1	1
			∞	D	D	D	D		
AVL-tree	8	1	2	L	L	L	D	[7:11]	[3:11]
			∞	L	L	L	D		
heap	7	2	2	L	L	L	D	[5:11]	[1:11]
			∞	L	L	L	D		
B-tree	9	5	2	L	D	D	D	[13:21]	[4:21]
			∞	L	D	D	D		
RB-tree	15	2	2	L	L	L	D	[13:21]	[4:21]
			∞	L	D	D	D		
tree	2	1	2	D	D	D	D	[9:20]	[6:20]
			∞	L	D	D	D		

While even with caching the cost of the run-time checks still remains significant,[5] caching does reduce overhead by 1-2 orders of magnitude with respect to the cost of run-time checking without caching (Fig. 3.2). Also, the slowdown ratio of programs with run-time checks using caching is almost constant, in contrast with the linear (or worse) growth in the case where caching is not used.

An important issue that has to be taken into account here is that most of the benchmarks are rather simple, and that performing insert operations is much less costly that performing run-time checks on the arguments of this operation. This explains the observation that checking overhead is the highest for the `set` benchmark (Fig 3.1), while it is one of the simplest used in the experiments.

Another factor that affects the overhead ratio is cache size. For smaller caches cell rewritings occur more often, and thus the optimal cache replacement policy in such cases is the one with the cheapest operations.

For instance, for cache size 32 the optimal policy for all benchmark groups is DM, while for other cache sizes LRU is in some cases better as it allows optimizing cell rewritings. This observation is also confirmed by the maximal check depth in the worst case, which is

5 Note that in general run-time checking is a technique for which non-trivial overhead can be expected for all but the most trivial properties. It can be conceptually associated with running the program in the debugger, which typically also introduces significant cost.

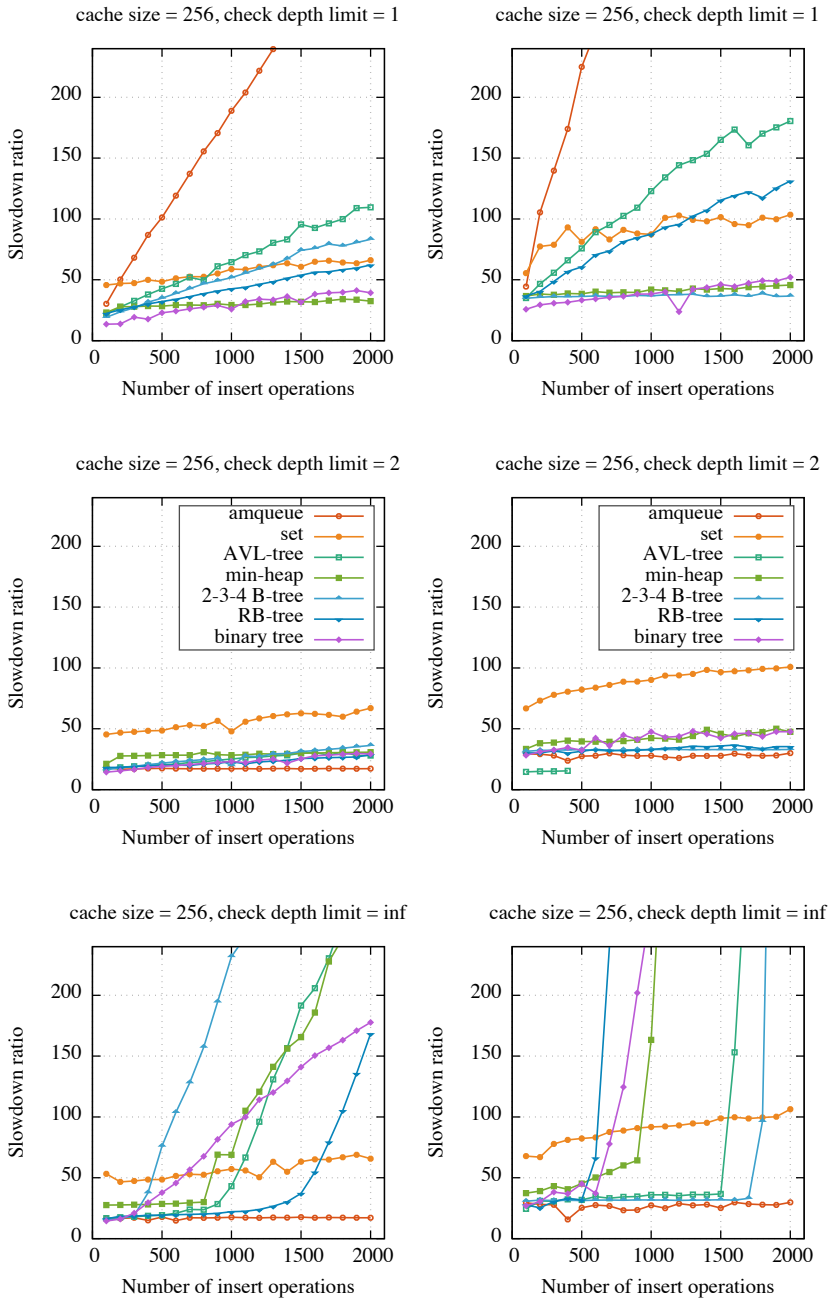


Figure 3.1: Run-time check overhead ratios for all benchmarks with check depth thresholds of 1, 2, ∞, and DM (left column) and LRU (right column) replacement policies in cache of 256 elements.

almost half on average for the benchmarks for which LRU is the optimal policy (Fig 3.3).

In the simple data structures of group (a) the experiments show that it is beneficial to have cheaper cache operations (like those of caches with DM caching policy), since such structures do not suffer from cache cell rewritings as much as more complex structures. The

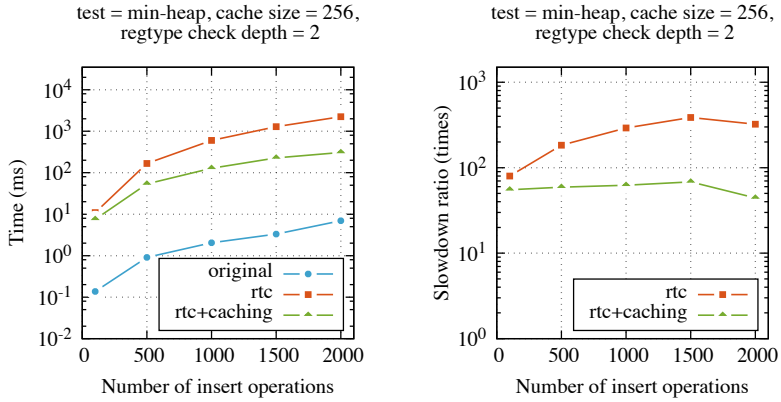
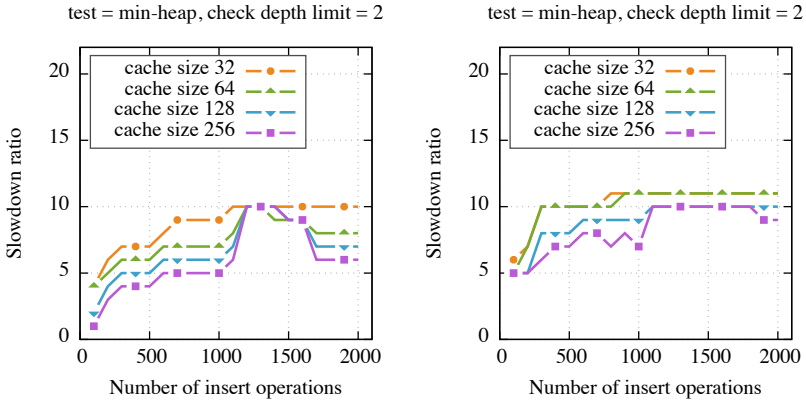


Figure 3.2: Absolute and relative running times of the `heap` benchmark with different rtchecks configurations, LRU caching policy.



(a) LRU (b) DM

Figure 3.3: Worst case regtype check depth for benchmarks from groups (b) and (c), with LRU and DM cache replacement policies respectively.

same observation is still true for group (d), where for some inputs the binary tree might grow high and regtype checks of leaves will pollute the cache with results of checks for those inner nodes on the path, that are not in the cache, overwriting cache entries with regtypes of previously checked nodes.

The DM policy also happens to show better results for group (c) for a similar reason. Since data structures in this group change essential node properties during the tree insertion operation, this in practice means that sub- terms that represent inner tree nodes are (re-)created more often. As a result, with the LRU caching policy the cache would become populated by check results for these recently created nodes, while the DM caching policy would allow preserving (and reusing) some of the previously obtained results. The only group that benefits from LRU is (b), where this policy helps preserving check results

for the tree nodes that are closer to the root (and are more frequently accessed) and most of the overwrites happen to cells that store leaves.

More plots are available in the Appendix B.1.

3.5 CONCLUSIONS

We have presented an approach to reducing the overhead implied by run-time checking of properties based on the use of memoization to cache intermediate results of check evaluation, avoiding repeated checking of previously verified properties. We have provided an operational semantics with assertion checks and caching and an implementation approach, including a more efficient program transformation than in previous proposals. We have also reported on a prototype implementation and provided experimental results that support that using a relatively small cache leads to very significant decreases in run-time checking overhead.

The idea of using memoization techniques to speed up checks has attracted some attention recently [50]. Their work (developed independently from ours) is based on adding fields to data structures to store the properties that have been checked already for such structures. In contrast, our approach has the advantage of not requiring any modifications to data structure representation, or to the checking code, program, or core run-time system.

Compared to the approaches that reduce checking frequency our proposal has the advantage of being exhaustive (i.e., all tests are checked at all points) while still being much more efficient than standard run-time checking. Our approach greatly reduces the overhead when tests are being performed, while allowing the parts for which testing is turned off to execute at full speed without requiring recompilation. While presented for concreteness in the context of the Ciao run-time checking framework, we argue that the approach is general, and the results should carry over to other programming paradigms.

4

COMBINING STATIC AND DYNAMIC CHECKING

Despite various advances in run-time checking overhead reduction, it often remains impractically high, for example for properties which require deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may activate sporadic checking of very simple conditions, but tend to turn off run-time checking for more complex properties.

Motivated by this problem, assertion-based frameworks have been proposed where static analysis is used to minimize the number and cost of the run-time checks that need to be placed in the program to detect incorrect program behaviors. Intuitively, this model can offer a more appealing trade-off of performance vs. safety guarantees. However, while there has been evidence supporting this hypothesis from the regular use of the Ciao system, there has been little systematic experimental work presented to date verifying this, i.e., measuring the actual impact of analysis on reducing run-time checking overhead. Some results were reported in [62] in the context of computational properties, such as resource consumption. Supporting evidence also comes from studies of the effectiveness of abstract interpretation (combined with abstract specialization [87]) in the reduction of run-time checking of sufficient conditions for independence in automatic parallelization [13].

In this chapter we explore the effectiveness of abstract interpretation-based compile-time analysis in detecting parts of program specifications that can be simplified before they are turned into run-time checks. Again, the objective of such simplification is to achieve a system that can detect the same (or a larger) set of incorrect behaviors in a program, but with a significant reduction in the impact on the running time of the program.

Starting with a semantics for programs with assertion checking and for assertion simplification based on analysis information obtained via abstract interpretation, we propose and study a number of practical *assertion checking modes*, each of which represents a *trade-off* between code annotation depth, execution time slowdown, and program behavior safety guarantees. The proposed modes are specially tailored to the scenario of annotating and pre-processing libraries to ensure their correctness prior to their use by client programs (i.e., scenario 1 of [88]). We also define a transformation-based approach in order to implement each one of these modes.

We then concentrate on the reduction of the number of run-time tests via (abstract interpretation-based) program analysis. To this end

we propose a technique that enhances analysis precision by taking into account that any assertions that cannot be proved statically will be the subject of run-time testing. We then report on an implementation of the proposed techniques (within the CiaoPP system) and study their impact in practice, by measuring the reduction in run-time checking overhead achieved.

4.1 ASSERTION CHECKING MODES

As mentioned before, a typical motivation for introducing run-time checks into the source code of a program is to detect and report erroneous program behaviors. Such checks may range from simple variable instantiation checks to describing complex program invariants and behavioral contracts.

When a program is being instrumented with run-time checks, the choice of instrumentation strategy is determined by several factors and considerations. Most of these factors can typically be generalized to a compromise between thoroughness of the code annotation (complexity of the properties, annotation depth) and the resulting performance penalties (increases in execution time, code size, and memory use).

We propose a view on this compromise that differentiates among various levels of behavioral safety guarantees embodied in different *assertion checking modes*. We consider for concreteness the context of developing a standalone library that provides an *open* interface to its clients. By this we mean that at the time of analyzing and instrumenting the library the clients are not known and can be expected to call the library in both correct and incorrect ways, i.e., we do not require the clients to verify that the calls to the library adhere to the interface. Also, we do not expect the library to be recompiled (or reanalyzed) depending on the needs of each client.[1] Thus, the library has to be analyzed and checked independently of the clients. We define three scenarios in this context, depending on the level of guarantees that the library provides to the clients that use it.

UNSAFE CHECKING MODE This checking mode corresponds to a scenario where no execution time slowdown is tolerated at run time, even at the cost of providing no safety guarantees to the clients. I.e., no run-time checks are generated from the assertions of the library. Formally, this corresponds to using the standard semantics of the Definition 2.1 of Chapter 2, and thus ignoring all the assertions in the code. This of course eliminates any overhead but at the cost of not being able to ensure correctness. However, we still consider it, first

1 This is all in contrast with the scenario in which the whole set of modules involved is available and can be processed as a whole, monolithically or modularly [86, 24]. Similarly, we also do not address directly in this work link-time optimizations.

```
:- module(_, [p]). % export p

:- check pred p : Pre => Post.


p :- body. % no calls to p
           % for simplicity


q :- p.
```

(a) Initial program fragment.

```
:- module(_, [p]).

% C0 = calls(p, Pre) ∧ status(c0, check)
% C1 = success(p, Pre, Post) ∧ status(c1, check)

p :- p_inner. % the link clause

p_inner :- body.

q :- p_inner.
```

(b) The same program fragment after the transformation.

Figure 4.1: Client-safe program transformation.

because it represents a baseline to compare to, and also because of the frequent –even if not recommendable– practice of turning off run-time checks for production code, in order to avoid overhead, which is typically done if it is perceived that sufficient testing was carried on the code out prior to delivery.

CLIENT-SAFE CHECKING MODE In this checking mode the library provides the client with behavior guarantees on its interface, but does not check any of the assertions for the internal procedures. Run-time checks are thus generated only for the assertion conditions for the exported predicates of the library. More formally, assuming that the set of (atoms of) exported predicates is given by Exp, the run-time semantics under such mode is:

1. If L is a constraint then $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ if $\theta \wedge L$ is satisfiable.
2. If L is an atom such that $L \notin Exp$, and $\exists (L \leftarrow B) \in \mathsf{cls}(L)$, then $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle B :: G' \mid \theta \rangle$.
3. If L is an atom such that $L \in Exp$, and $\exists (L \leftarrow B) \in \mathsf{cls}(L)$, then from the initial state $S = \langle L :: G \mid \theta \rangle$ the new state S' is obtained as:

$$S' = \begin{cases} \langle \mathsf{err}(c) \mid \theta \rangle & \text{if } \exists\, C = \mathsf{calls}(L, Pre) \in \mathcal{A}_C\langle L \rangle \wedge \mathsf{id}(C) = c \\ & \quad \wedge\, \theta \not\Rightarrow Pre \\ \langle B :: G' \mid \theta \rangle & \text{otherwise} \end{cases}$$

   and $G' = \mathsf{check}(c_1) :: \ldots :: \mathsf{check}(c_n) :: G$ such that $C_i = \mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C\langle L \rangle \wedge \mathsf{id}(C_i) = c_i \wedge \theta \mapsto Pre_i$.

4. If L is a check literal $\mathsf{check}(c)$, then from the initial state $S = \langle L :: G \mid \theta \rangle$ the new state S' is obtained as:

$$S' = \begin{cases} \langle \mathsf{err}(c) \mid \theta \rangle & \text{if } C = \mathsf{success}(L', \_, Post) \in \mathcal{A}_C\langle L' \rangle \wedge \mathsf{id}(C) = c \\ & \quad \wedge\, \theta \not\Rightarrow Post \\ \langle G \mid \theta \rangle & \text{otherwise} \end{cases}$$

The modified semantics above ensures that checks are performed only for the predicates in the library interface. However, all calls within the library to the exported predicates, including recursive calls, would also be checked, which is not required by the definition of the scenario, which only establishes the checking of the calls that cross the interface. In order to avoid this, and to ensure that the checks are performed only on the external calls, we assume that the program transformation given in Fig. 4.1 is applied to all exported predicates. This transformation introduces intermediate *link* predicates for the exported predicates so that the module interface is preserved but all the internal calls are replaced by calls to the wrapper predicates, for which no checks are performed. This way the checks for the exported predicates are not repeated in the internal library calls, allowing for execution time reduction for the checks. This combination of program transformation and run-time checking policy allows obtaining safety guarantees at the library boundaries with minimal run-time checking execution time overhead.

SAFE-RT EXECUTION MODE In this mode the library provides behavior guarantees both on its interface and its internals. Run-time checks are thus generated for all assertions of the library. This corresponds to using the semantics with assertions of the Definition 2.12 of the Chapter 2. The performance penalty here is the largest.

SOURCE CODE TRANSFORMATIONS The checking modes described above require different source transformations to be performed on a program during compile time (see Fig. 4.2). Before any such transformations take place, the assertions are normalized and expanded into assertion conditions. This allows ensuring that no syntactic errors are present in the assertion conditions and that no undefined properties (i.e., properties that are not defined in the program or imported from libraries) appear in such conditions.

In the *Unsafe* mode nothing is done and the assertion conditions are simply ignored during compilation. In the *Safe-RT* mode the source transformation is quite straightforward: all the assertion conditions for all assertions in the program are turned into run-time checks directly. In the *Client-safe* mode, as mentioned before, the program transformation of Figure 4.1 is first performed for all the exported predicates, and then run-time checks are generated only for the assertion conditions of those exported predicates.

To this end, we introduce a variation on one the previous run-time checking modes, namely the ***Safe-CT-RT Checking Mode***, where static verification is performed in order to eliminate as many of the properties in the program assertions to be checked at run time as possible. Run-time checks are still generated for all program assertions but in contrast to the *Safe-RT* case the assertions are simplified before the

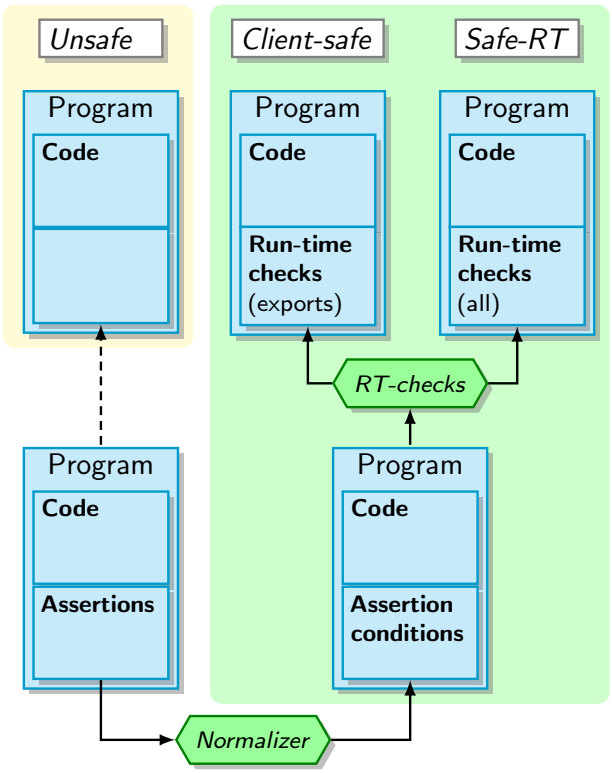


Figure 4.2: Source transformation differences per checking mode.

checks are generated from them. In this mode the run-time checks for the *calls* assertion conditions of the exported predicates are left untouched in any case, in order to ensure the safety of calls in our open-library context.

4.2 OPTIMIZING RUN-TIME CHECKS VIA STATIC ANALYSIS

ABSTRACT INTERPRETATION-BASED ANALYSIS For analysis we use the technique of abstract interpretation [26], which safely approximates the execution of a program on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain*[2] (D). Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$.[3] The operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) over abstract values λ mimic those of 2^D in a precise sense:

$$\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')$$

2 In what follows we assume the concrete domains to have a powerset structure, but the framework is not limited to such domains and can be applied to domains of arbitrary structure.

3 Strictly, only the concretization function is required.

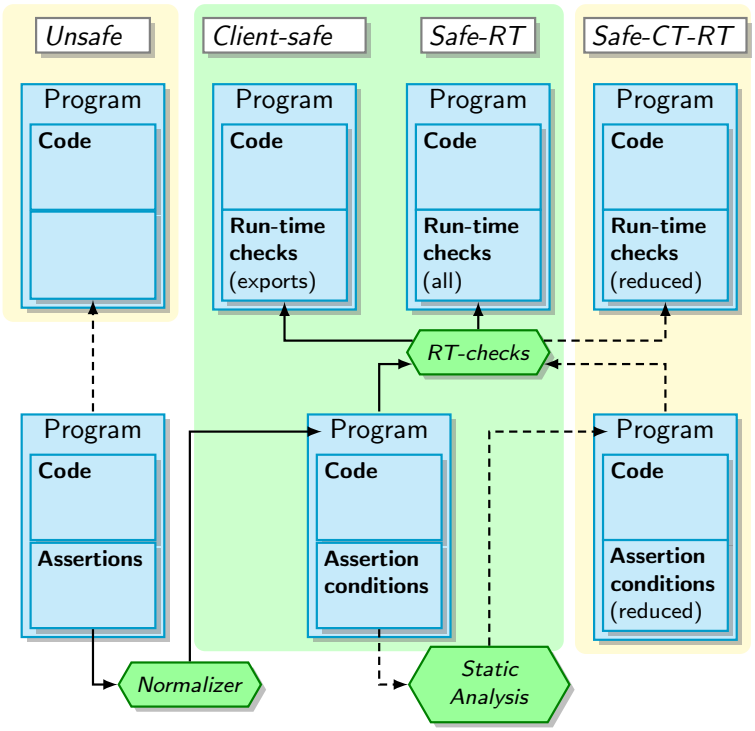


Figure 4.3: Source transformation differences per checking mode, including compile-time analysis.

As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \varnothing$ (and represents unreachable code), whereas $\top$ denotes the most general abstract constraint, i.e., $\gamma(\top) = D$.

The concrete framework that we will use in the static analysis component is the Ciao PLAI abstract interpretation system [65, 66, 68]. Below we adapt some definitions and notation from [85] to illustrate the analysis process implemented by PLAI.

The goal-dependent abstract interpretation performed by PLAI takes as input a program P, an abstract domain D_α,[4] and a description $\mathcal{Q}_\alpha$ of the possible initial queries to the program, given as a set of *abstract queries*. Each such abstract query is a pair (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ an abstraction of a set of concrete initial program states (e.g., substitutions or constraints). Thus, a set of abstract queries $\mathcal{Q}_\alpha$ represents a set of concrete queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. The PLAI abstract interpretation process computes a set of (connected) triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s \rangle \mid p \text{ is a predicate of } P\}$, where λ^c and λ^s are abstract constraints that describe sets of calls (entry) and success (exit) states for p such that λ^c safely approximates a set of call states at p and λ^s safely

4 In fact, the analysis supports analysis using a number of different abstract domains, but, for simplicity, and without loss of generality –a set of abstract domains can always be encoded as a single domain– we use only one domain in the presentation.

approximates the set of success states at p for all calls contained in λ^c. In what follows we will refer to such triplets also as *memo table entries*.[5]

The analysis (as the assertion language, to be introduced later) is designed to discern among the various usages of a predicate. Thus, multiple usages (contexts) of a procedure can result in multiple descriptions in the analysis output, i.e., for a given predicate p multiple $\langle L_p, \lambda^c, \lambda^s \rangle$ triples may be inferred. More precisely, the analysis is said to be *multivariant on calls* if more than one triple $\langle L_p, \lambda^c_1, \lambda^s_1 \rangle$, $\ldots$, $\langle L_p, \lambda^c_n, \lambda^s_n \rangle$ $n \geq 0$ with $\lambda^c_i \neq \lambda^c_j$ for some i, j may be computed for the same predicate. Independently of the number of triples computed, the set of all λ^c_i together (i.e., the union of the concretizations of all the λ^c_i) safely approximates the set of possible concrete calls made to p during any program execution. In any case, for simplicity of presentation, we assume that the analysis computes exactly one tuple $\langle L_p, \lambda^c, \lambda^s \rangle$ for each (reachable) predicate p.

ASSERTION PROCESSING BY STATIC ANALYSIS We now return to the issue of optimizing run-time checks via (abstract interpretation-based) static program analysis, in order to reduce the number of run-time tests and thus the overhead from run-time testing, following the Ciao model. To this end, we recall the basic abstract interpretation-based analysis approach used and the memo table representation of the analysis results and describe how run-time tests are optimized using the information in the analysis memo table. Based on this in the following section we will present our approach for taking advantage of the run-time checking semantics to improve the precision of the analysis.

The steps of the verification process are represented by associating a notion of "status" to each assertion:

$$
\begin{array}{l}
\texttt{:-}\ [\mathit{Status}]\ \texttt{pred}\ \mathit{Head}\ \texttt{:}\ \mathit{Pre}_1\ \texttt{=>}\ \mathit{Post}_1\,. \\
\ldots \\
\texttt{:-}\ [\mathit{Status}]\ \texttt{pred}\ \mathit{Head}\ \texttt{:}\ \mathit{Pre}_n\ \texttt{=>}\ \mathit{Post}_n\,.
\end{array}
$$

This optional *Status* flag indicates whether the assertion refers to intended or actual properties, and possibly some additional information, as shown in the top part of Table 4.1 (see also Figure 1.2).

The reasoning about the statuses of assertion conditions is performed in the following terms. Given a literal L and a program P, the *trivial success set* of L in P is $TS(L, P) = \{\bar{\exists}_L \theta \mid \theta \Rightarrow L\}$.

An abstract constraint $\lambda^-_{TS(L,P)}$ is an *abstract trivial success subset* of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L, P)$. An abstract constraint $\lambda^+_{TS(L,P)}$ is an *abstract trivial success superset* of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L, P)$. Given

5 The analysis also provides information at body literals (also referred to as "program points").

6 We will use only `true` assertions in the rest of the dissertation for simplicity.

Table 4.1: Assertion status.

Status	Source	Description
`check`	user	The assertion expresses part of the intended semantics. It may or may not hold in the current version of the program. It is the default status that is assumed for assertions written without an explicit status.
`checked`	static checking	The assertion was a `check` assertion which has been proved to actually hold in the current version of the program for any valid initial call (for the given Q_α).
`false`	static checking	Similarly, a `check` assertion is rewritten with the status `false` when it is proved not to hold for some valid initial query (for the given Q_α).
`true`	static analyses	Such an assertion expresses (a part of) the actual semantics of the program, normally automatically inferred by analysis. In particular, each triple (memo table entry) $\langle L_p, \lambda^c, \lambda^s \rangle$ computed by the analysis is presented to the user by including a corresponding assertion of the form "`:- true pred` P `:` λ^c `=>` λ^s`.`" in the program.
`trust`	user	Provided by the user (or other tools) in order to guide analysis (increase precision).[6]

the program P, the concrete and abstract sets of queries $\mathcal{Q}$ and $\mathcal{Q}_\alpha$[7] respectively, where $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$, and $\langle L, \lambda^c, \lambda^s \rangle \in \mathit{Analysis}(P, \mathcal{Q}_\alpha, D_\alpha)$, the status of an assertion condition C, associated with it by the mapping $\mathsf{status}(c, \mathit{Status})$ where c is the corresponding identifier, is determined as follows:

- If $C = \mathsf{calls}(L, \mathit{Pre})$ s.t. $\mathsf{id}(C) = c$ then:
  - $\mathsf{status}(c, \texttt{checked})$ if $\lambda^c \sqsubseteq \lambda^-_{TS(Pre,P)}$.
  - $\mathsf{status}(c, \texttt{false})$ if $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\mathsf{prestep}(L, D) = (\theta, \sigma)$ $\wedge \overline{\exists}_L \theta \neq \varnothing$ and $\lambda^c \sqcap \lambda^+_{TS(Pre,P)} = \bot$.
- If $C = \mathsf{success}(L, \mathit{Pre}, \mathit{Post})$ s.t. $\mathsf{id}(C) = c$ then:

7 In the implementation of PLAI, $\mathcal{Q}_\alpha$ is obtained from the calls conditions of the assertions of exported predicates (or, if no such assertions are present, a "topmost" abstract state is assumed), or from specific "`entry`" assertions.

- $\mathsf{status}(c, \texttt{checked})$ if (1) $\lambda^c \sqcap \lambda^+_{TS(Pre,P)} = \bot$ or (2) $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$;
- $\mathsf{status}(c, \texttt{false})$ if $\lambda^c \sqcap \lambda^-_{TS(Pre,P)} \neq \bot$ and $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ and $\exists\ \theta \in \gamma(\lambda^c \sqcap \lambda^-_{TS(Pre,P)}) : \exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\mathsf{step}(L, D) = (\theta, \sigma, \theta') \wedge \overline{\exists}_L \theta' \neq \varnothing$.

Putting it more informally, the compile-time checking process can be seen as a revision of the assertion statuses where for each predicate literal L its *annotation* composed from the respective assertion conditions $\mathcal{A}_C\langle L\rangle^{usr} = \{C \mid C \in \mathcal{A}_C\langle L\rangle \wedge \mathsf{id}(C) = c \wedge \mathsf{status}(c, S) \wedge S \in \{\texttt{check}, \texttt{true}\}\}$ given the analysis output of the form $\mathcal{A}_C\langle L\rangle^{ana} = \mathsf{id}(C) = c \wedge \mathsf{status}(c, \texttt{true})\}$ is rewritten into $\{C \mid C \in \mathcal{A}_C\langle L\rangle^{usr} \wedge \mathsf{id}(C) = c \wedge \mathsf{status}(c, S) \wedge S \in \{\texttt{check}, \texttt{checked}, \texttt{false}\}\}$.

4.3 TAKING ADVANTAGE OF THE RUN-TIME CHECKING SEMANTICS DURING ANALYSIS

The standard analysis introduced in Section 1.5 of the Chapter 2 safely approximates the traditional semantics (i.e., the semantics without assertions or run-time checks of the Definition 2.1).[8] However, if we know that run-time checks will be performed for sure for a certain set of (`check`) assertions (as, e.g., for all assertions in the *Safe-RT* execution mode, or the ones corresponding to interface predicates in the *Client-safe* mode), it is possible to use this information during analysis to improve precision:

- It is possible to assume that the calls assertion conditions hold after the predicate has entered the predicate definition (since, according to the semantics of Section 2.3 either the checks for these calls assertion conditions have already succeeded or the program has exited with error).
- It is also possible to assume the relevant success assertion conditions after the predicate has exited (since, again, at this point either these success assertion conditions have already succeeded or the program has exited with error).

As an example, consider the Ciao Prolog program of Figure 4.4.[9] There, `p/2` is an exported predicate, `q/2` and `r/2` are local predicates,

8 Assertions with `true` and `trust` status (Table 4.1) are in fact read and applied by the traditional analysis during its fixpoint calculation. However, in this discussion we refer to incorporating into the analysis the information present in `check` assertions, i.e., from the assertions being checked at compile time or run time. These assertions are not normally taken into account by the analysis since they may or may not hold and, in general, run-time tests may or may not be included in the compiled program.

9 In the examples we use just simple regular types (and in some cases constraints) as properties for simplicity of presentation, but even in this case please note that their use is *moded*, i.e., the assertions here express *states of instantiation*.

```
1  :- module(_,[p/2]).        % p/2 is exported
2  :- use_module(lib,[e/2]). % e/2 is imported
3
4  p(X,Y) :- q(X,Y).
5
6
7  :- pred q(X,Y) : (int(X), X>3) => (int(Y), Y>0).
8
9  q(X,Y) :- r(X,Y).
10
11
12 :- pred r(X,Y) : (int(X), X>0) => (int(Y), Y>16).
13
14 r(X,Y) :- e(X,Y).
```

Figure 4.4: Example for analysis improvement.

and `e/2` is imported. We allow both `p/2` and `e/2` to be called without any restriction, and we do not specify any constraints either regarding their successes. However, we want to enforce (through the two assertions) that `q/2` and `r/2` will always be called with their first argument `X` bound to an integer greater than 3, and that their second argument `Y` be bound to a positive integer upon success. Since any type of call is allowed to `p/2`, without information on the presence of run-time checks the analysis cannot infer anything about the calls conditions for `q/2` and `r/2`, or for the success conditions of these two predicates, and will report warnings for unchecked conditions for all of them (and the two assertions will remain in `check` status).

However, note that, if we know that we will be generating run-time checks for those assertion conditions, the call to `r/2` in the body of `q/2` can only be reached if the calls condition for `q/2` holds, i.e., if `X` is bound to an integer, and greater than 3 (since otherwise execution would have been aborted by the failing run-time check). Thus, this information can be incorporated into the analysis and propagated to the call to `r/2`, and it can be determined that the calls condition for `r/2` (i.e., that its first argument will be bound to a positive integer) always holds. Consequently, this calls condition for `r/2` gets status `checked` and no run-time test needs to be generated for it.

Similarly, the run-time test for the success condition for `r/2` ensures that if the call to `r/2` in the body of `q/2` returns, then its second argument is guaranteed to be bound to an integer and greater than 16. Therefore, the success condition for `q/2` will also get status `checked` and no run-time test needs to be generated for it either.

TRANSFORMATION A straightforward method to incorporate the information from successful checks into the analysis, so that it takes the semantics with run-time checking into account, would be to analyze the transformed program (i.e., the program including the code that performs the run-time tests) instead of the original one. This is the approach implied by the original transformational definitions of

the assertion language. On the other hand, programs transformed for run-time testing contain numerous optimizations and instrumentation that make their analysis less efficient and can potentially affect precision. An alternative would be to use a very simple (even if inefficient) run-time checking transformation just for analysis. Inspired by this idea, we propose herein a different, even more direct approach, based on introducing additional assertions and link predicates in the program that together capture the run-time checking semantics and provide the additional information source for the analysis, in order to increase precision. This is performed as a program transformation T that precedes the analysis and is applied to every annotated predicate in a program:

$$T(L) = \langle \{L \leftarrow L_{inner}\} \cup \mathsf{cls}(L_{inner}), \mathcal{A}_C^{link} \cup \mathcal{A}_C^{inner} \rangle$$

where $L = p(\vec{X})$, and the literal $L_{inner} = p_{inner}(\vec{X})$ is obtained with a new predicate symbol p_{inner}, and:

$$\begin{aligned}
\mathsf{cls}(L_{inner}) &= \{L_{inner} \leftarrow B \mid L \leftarrow B \in \mathsf{cls}(L)\} \\
\mathcal{C} &= \{C \in \mathcal{A}_C\langle L\rangle \mid \mathsf{id}(C) = c \wedge \mathsf{status}(c, \texttt{check})\} \\
\mathcal{A}_C^{link} &= \{C^l \mid C^l = C \wedge C \in \mathcal{C} \wedge \mathsf{id}(C^l) = c^l\} \\
&\quad \text{and } \forall C^l \in \mathcal{A}_C^{link} \text{ we extend} \\
&\quad \text{the status relation s.t. } \mathsf{status}(c^l, S^l), \text{ s.t.:} \\
S^l &= \begin{cases} \texttt{check} & \text{if } C = \mathsf{calls}(_,_) \\ \texttt{true} & \text{if } C = \mathsf{success}(_,_,_) \end{cases} \\
\mathcal{A}_C^{inner} &= \{C^i \mid C^i = C \wedge C \in \mathcal{C} \wedge \mathsf{id}(C^i) = c^i\} \\
&\quad \text{and } \forall C^i \in \mathcal{A}_C^{inner} \text{ we extend} \\
&\quad \text{the status relation s.t. } \mathsf{status}(c^i, S^i), \text{ s.t.:} \\
S^i &= \begin{cases} \texttt{true} & \text{if } C = \mathsf{calls}(_,_) \\ \texttt{check} & \text{if } C = \mathsf{success}(_,_,_) \end{cases}
\end{aligned}$$

The objective of the transformation is to improve the precision and reduce the cost of the analysis, while preserving program behavior when the `check` assertion conditions are expanded into run-time checks. The transformation modifies all predicates with `check` assertions for which it is known that run-time checks will be generated. For each such predicate p, the original predicate symbol is renamed into p_{inner} and a single-clause wrapper predicate for p (which we will refer to as a *link* clause), is introduced which calls the p_{inner} predicate.

The set of assertion conditions for the initial predicate p is duplicated for the p_{inner} counterpart, including their original statuses. However, the statuses of the `success` assertion conditions for p in the link clause and the `calls` assertion conditions of p_{inner} are set to `true`.

```
:- check calls   q(X,Y) : (int(X), X>3).
:- true  success q(X,Y) : (int(X), X>3) => (int(Y), Y>0).

q(X,Y) :- q_inner(X,Y).


:- true  calls   q_inner(X,Y) : (int(X), X>3).
:- check success q_inner(X,Y) : (int(X), X>3) => (int(Y), Y>0).

q_inner(X,Y) :- r(X,Y).
```

Figure 4.5: CTRT program transformation example (output).

As a result, the calls assertion conditions for p (i.e., $C^l = \mathsf{calls}(L, \_)$ with $\mathsf{id}(C^l) = c^l$ and $\mathsf{status}(c^l, \texttt{check})$) will still be checked in the version with run-time checks, but they will be assumed in p_{inner} (i.e., $C^i = \mathsf{calls}(L_{inner}, \_)$ with $\mathsf{id}(C^i) = c^i$ and $\mathsf{status}(c^i, \texttt{true})$).

For the success part the assertion conditions will still be checked for the inner predicate (i.e., $C^i = \mathsf{success}(L_{inner}, \_, \_)$ with $\mathsf{id}(C^i) = c^i$ and $\mathsf{status}(c^i, \texttt{check})$) and the information will be assumed upon exiting p (i.e., $C^i = \mathsf{success}(L, \_, \_)$ with $\mathsf{id}(C^i) = c^i$ and $\mathsf{status}(c^i, \texttt{true})$). The transformation guarantees that the same run-time tests will be performed, that no duplication of checks will occur (since there are no intermediate states between the calls to p and p_{inner} and exits from p_{inner} to p), and that the analysis will gather the right information.

An example of the CTRT transformation for the `q/2` predicate from the program in Fig. 4.4 is shown in Fig. 4.5. The `true` assertions here correspond to the additional information that can be safely used in the analysis. Since all predicates with assertions undergo this transformation, a number of inner calls coming from the link clauses are added to the program. Yet such calls are relatively inexpensive and the resulting runtime overhead is negligible. Even more, should the analysis verify the calls assertion condition of the link clause or the success assertion condition of the inner clause, the link clause then becomes unnecessary and can be completely removed.

Lemma 4.1 (Correctness of the CTRT Transformation)**.** *Let P be a program and $Q = (L, \theta)$ a query to P. Then $\forall\, D \in \mathit{derivs}(Q)$ that are* finished *the final state $D_{[-1]}$ is the same in the versions of P with and without the CTRT transformation (modulo variable renaming).*

Proof. First, let us prove the correctness of the transformation for the *calls* assertion conditions.

Let $\mathcal{A}_C\langle L\rangle = \{C\}$ where $C = \mathsf{calls}(L, Pre)$ s.t. $\mathsf{id}(C) = c \wedge \mathsf{status}(c, \texttt{check})$ and $\exists(L \leftarrow B) \in \mathsf{cls}(L)$. The possible reduction sequences from the $S_0 = \langle L :: G \mid \theta\rangle$ state:

$$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: G \mid \theta\rangle = S_{succ} \quad \text{if } \theta \models Pre$$
$$S_0 \rightsquigarrow_{\mathcal{A}} \langle \mathsf{err}(c) \mid \theta\rangle = S_{err} \quad \text{if } \theta \not\models Pre$$

Now let us add the *link* clause for L and rename its other clauses s.t. $\mathsf{cls}(L) = \{L \leftarrow L_{inner}\}$ and $\exists L_{inner} \leftarrow B \in \mathsf{cls}(L_{inner})$, and let's add an assertion condition for L_{inner}: $C^{inner} = \mathsf{calls}(L_{inner}, Pre)$ with $\mathsf{id}(C^{inner}) = c^i$ and $\mathsf{status}(c^i, \texttt{check})$. The possible reduction sequences from the S_0 state now are:

$S_0 \rightsquigarrow_{\mathcal{A}} \langle L_{inner} :: G \mid \theta \rangle \rightsquigarrow_{\mathcal{A}} S_{succ}$ if $\theta \Mapsto Pre$

$S_0 \rightsquigarrow_{\mathcal{A}} S_{err}$ if $\theta \not\Mapsto Pre$

The $S_0 \rightsquigarrow_{\mathcal{A}} \langle L_{inner} :: G \mid \theta \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{err}(c^i) \mid \theta \rangle$ reduction sequence is impossible since it would require $\theta \Mapsto Pre$ to hold in the first reduction step and $\theta \not\Mapsto Pre$ to hold in the second reduction step.

This way in both assertion checking modes $D_{[-1]} \in \{S_{succ}, S_{err}\}$ and run-time checks for the *calls* assertion condition C^{inner} (namely, checks for $\theta \Mapsto Pre$ after the checks for $\theta \Mapsto Pre$) could be safely removed by setting $\mathsf{status}(c^i, \texttt{true})$. □

Next, let's consider the case of *success* assertion conditions.

Let $\mathcal{A}_C\langle L \rangle = \{C\}$ where $C = \mathsf{success}(L, Pre, Post)$ s.t. $\mathsf{id}(C) = c \wedge \mathsf{status}(c, \texttt{check})$ and $\exists (L \leftarrow B) \in \mathsf{cls}(L)$. The possible reduction sequences from the $S_0 = \langle L :: G \mid \theta \rangle$ state are:

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{check}(c) :: G \mid \theta \rangle \rightsquigarrow^*_{\mathcal{A}} \langle G \mid \theta \rangle = S_{succ}$ if $\theta \Mapsto Post$

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{check}(c) :: G \mid \theta \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{err}(c) \mid \theta \rangle = S_{err}$ if $\theta \not\Mapsto Post$

Now let us add the *link* clause for $L$and rename its other clauses s.t. $\mathsf{cls}(L) = \{L \leftarrow L_{inner}\}$ and $\exists L_{inner} \leftarrow B \in \mathsf{cls}(L_{inner})$, and let's add an assertion condition for L_{inner}: $C^{inner} = \mathsf{success}(L_{inner}, Pre, Post)$ with $\mathsf{id}(C^{inner}) = c^i$ and $\mathsf{status}(c^i, \texttt{check})$. We also now consider C as C^{link} with its identifier c^l. The possible reduction sequences from the S_0 state now are:

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{check}(c^i) :: \mathsf{check}(c^l) :: G \mid \theta \rangle \rightsquigarrow^*_{\mathcal{A}} \langle G \mid \theta \rangle = S_{succ}$
if $\theta \Mapsto Post$

$S_0 \rightsquigarrow_{\mathcal{A}} \langle B :: \mathsf{check}(c^i) :: \mathsf{check}(c^l) :: G \mid \theta \rangle \rightsquigarrow^*_{\mathcal{A}}$
$\langle \mathsf{err}(c^l) \mid \theta \rangle = S_{err}$
if $\theta \not\Mapsto Post$

Although the assertion condition identifiers for the two S_{err} are different, the checks performed in these states are equal ($\theta \not\Mapsto Post$).

This way the run-time checks for the c^l assertion condition are duplicating the checks for c^i and could be safely removed by setting $\mathsf{status}(c^l, \texttt{true})$. □

4.4 OPTIMIZING CHECKS AT THE CLIENT-LIBRARY BOUNDARIES

We now consider another aspect of our library scenario: optimizing the checks at the client-library boundaries. We will remain within the case in which the library provides an open interface to its clients, i.e., the clients are not known when analyzing and compiling the library, these clients can be expected to call the library in arbitrary ways, and we do not want the library to be reanalyzed or recompiled for each

particular client. As seen in Section 4.1, in this scenario the reusability of the library forces us at least in principle to keep the run-time checks for the assertions at the library interface to ensure correctness. However, on the client side it may be possible to detect places where there is a call in the client module to a library predicate, such that the checks or analysis performed in the client module guarantee that the calls conditions of the library predicate will be satisfied. Detecting this could allow us to optimize away the checks at the client- library boundaries, and thus reduce run-time checking overhead.

Again, while inter-modular analysis could be used to this end, the advantage of fixing the library boundaries is that the library modules, once analyzed and compiled, can be reused without repeating the analysis or re-analyzing for new abstract call states. This reanalysis may not be really practical in the case of pre-compiled libraries, and also implies in any case additional cost, which may be prohibitive for some applications. Also, in inter-modular analysis and optimization the module boundaries change dynamically during analysis and this can happen after a change in any module. Another advantage of fixing the library boundaries is thus that it avoids having to recompile the client if there are changes in the library source code (and vice-versa), provided that the interface of the library itself is not changed. I.e., there are advantages to being able to fix the interface at certain boundaries.

The alternative that we propose is to provide a fixed interface, but one that provides two entry points for each predicate exported by the library: the standard one, that performs the run-time checks for the assertions in the library interface, and another one that provides direct access to the exported predicates bypassing the boundary assertion checks (in particular, the `_inner` versions produced by the CTRT transformation). We also propose a matching transformation for the client module that allows selecting, for each literal in the client that calls a library predicate, which of the two versions of that predicate exported by the library interface can safely be used.[10]

On the client side, we assume that the source code of the library predicates that are being imported by the client module is in general not accessible from the client during the analysis in the client. However, we assume that the interface of the library includes also the assertions of its exported predicates (as is the case in Ciao/CiaoPP). Thus, analysis on the client side has to rely solely on the information available in the interface of the library. This is not an issue however, if the library is compiled with the CTRT transformation, as in this case the transformation includes the assertions for the exported predicates (more specifically, the *link* clause assertions) in the library interface.

10 This can obviously be generalized to providing several entry points under several conditions [88], but we will keep the discussion limited to two entry points per predicate for simplicity.

```
:- module(mod,[p/2]).
:- use_module(lib,[e/2]).

:- pred p(X,Y) : int(X)
              => int(Y).

p(X,Y) :- e(X,Y).

:- pred q(X,Y) => int(Y).

q(X,Y) :- e(X,Y).
```

```
:- module(lib,[e/2]).

:- pred e(X,Y) : int(X)
              => int(Y).

e(X,Y) :- ...
```

Figure 4.6: A client-library program.

As an example, consider the client-library program in Figure 4.6 (using just moded types for brevity). There, in the client module `mod`, `p/2` is an exported predicate and `q/2` is a local predicate, and `e/2` is imported from the library `lib`. We want to enforce through the assertions that `p/2` always be called with its first argument `X` bound to an integer, and that its second argument `Y` be bound to an integer upon success (i.e., returning a free variable is not allowed). At the same time, we do not enforce any call-specific way to invoke `q/2`, and we enforce that its second argument `Y` should be bound to an integer upon success.

Both `p/2` and `q/2` call predicate `e/2`, imported from the library. Since `e/2` is an exported predicate in the `lib` module, the check for its calls condition (that its first argument `X` is bound to an integer) will always be performed. But notice that at the point where `e/2` is called from `p/2` the check for its first argument being an integer at run time has already taken place, as the same check was required by the calls conditions for the `p/2` predicate. This check duplication can be avoided if we replace at compile-time the call to `e/2` in the body of `p/2` with a call to `e_inner/2`, which is visible from `mod` during the pre-compilation analysis time. In principle this inner predicate would have to be exported but in practice it is done through the internal visibility mechanism in the compiler, which the user cannot bypass. At the same time we would like to keep the check for the calls condition of `e/2` when it is called from the body of `q/2`, as in that case nothing ensures that its first argument will be bound to an integer.

The optimization that we seek requires us to be able to reason about individual call sites in the bodies of the clauses in the program predicates, also referred to as "program points." For this, we need the analysis information (abstract states) to be available not just at the whole predicate level (call and success) but also at the level of the clause literals. This information is indeed provided by the PLAI analysis that we are using as reference (Chapter 1.5). We also need the interface of the transformed library to be extended by making accessible the *link* predicates generated for all its annotated exported predicates, together with their respective assertions. As mentioned before,

such interface extension will provide us with (at least) two different versions of the library exported predicates, that can be called at different program points in the client. For this kind of reasoning we also require the static analysis performed to be in effect *multivariant* on calls.[11]

```
:- module(mod,[p/2, q/2]).
:- use_module(lib,[e/2,e_inner/2]).

:- check calls   p(X,Y) : int(X).
:- true  success p(X,Y) : int(X) => int(Y).

p(X,Y) :- p_inner(X,Y).

:- true  calls   p_inner(X,Y) : int(X).
:- check success p_inner(X,Y) : int(X) => int(Y).

p_inner(X,Y) :- e_inner(X,Y).

:- check calls   q(X,Y) : term(X).
:- true  success q(X,Y) : term(X) => num(Y).

q(X,Y) :- q_inner(X,Y).

:- true  calls   q_inner(X,Y) : term(X).
:- check success q_inner(X,Y) : term(X) => num(Y).

q_inner(X,Y) :- e(X,Y).
```

```
:- module(lib,[e/2, e_inner/2]).

:- check calls   e(X,Y) : int(X).
:- true  success e(X,Y) : int(X) => int(Y).

e(X,Y) :- e_inner(X,Y).

:- true  calls   e_inner(X,Y) : int(X).
:- check success e_inner(X,Y) : int(X) => int(Y).

e_inner(X,Y) :- ...
```

Figure 4.7: A two-module program after the transformations.

Let ppt denote a program point identifier, which refers to a particular literal position in the body of a particular clause in the program. Let L^{ppt} denote the literal L that is located at program point ppt. We assume thus that the analysis provides the following information:

- The $\langle L_p, \lambda_i^c, \lambda_i^s \rangle$ triples for the predicates in the program, as before.
- In addition, triples $\langle L_p^{\mathsf{ppt}}, \lambda^c, \lambda^s \rangle$ that provide, for each literal L_p, the abstract state before and after the calls to such literal at each program point ppt in which L_p occurs in the body of a clause.

11 In the experiments we used explicit materialization of versions. Note however that this can also be obtained via modular partial evaluation.

We further adapt our notation to *program point*-level reasoning as follows:

- Let $\mathsf{status}^{\mathsf{ppt}}(c, S)$ denote the status of some assertion condition $C = \mathsf{calls}(L, \_)$ or $C = \mathsf{success}(, L, \_)\_$ s.t. $\mathsf{id}(C) = c$ for the literal L at program point ppt.

Now with the information from the multivariant analysis and the statuses of assertions after the checking phase it is straightforward to apply a program-point literal substitution. Since we are considering programs that undergo the CTRT transformation by the time the static analysis and assertion checking are performed, L^{ppt} should be either L or L_{inner}, depending on the abstract state at the program point and the result of the program point assertion checks:

$$L^{\mathsf{ppt}} = \begin{cases} L_{inner} & \text{if } \forall C \in \mathcal{A}_C\langle L\rangle \text{ s.t. } C = \mathsf{calls}(L, \_) \wedge \mathsf{id}(C) = c \\ & \quad \mathsf{status}^{\mathsf{ppt}}(c, \texttt{checked}) \text{ holds} \\ L & \text{otherwise} \end{cases}$$

A result of such program transformation can be seen in Figure 4.7 for the program in Figure 4.6.

4.5 EXPERIMENTAL EVALUATION

As stated throughout the chapter, our objective is to explore the effectiveness of abstract interpretation in detecting parts of program specifications that can be statically simplified to true or false, and to quantify the impact of this application of analysis towards reducing the cost of the run-time checks. In particular, we have studied these issues for the different assertion checking modes that we have defined and for the two scenarios.

EXPERIMENTAL SETUP We have built an experimental harness by extending the Ciao preprocessor, CiaoPP, which implements our baseline assertion verification framework (see [40], Section 4). The architecture of this framework is shown in Figure 4.8. We provide below high-level descriptions of the verification process and internal functionality of the principal components.

The input to the verification process, as mentioned before in the Section 1.5, is the user program, optionally including a set of assertions; this set always includes any assertions present for predicates exported by any libraries used. Any `check`, `trust`, or `true` assertions are normalized and the program is expanded to kernel form (simple Horn clauses), and the result is given as input to the *static analysis*.

We have introduced new front-end passes implementing the new transformations (marked in Figure 4.8) which thus support the de-

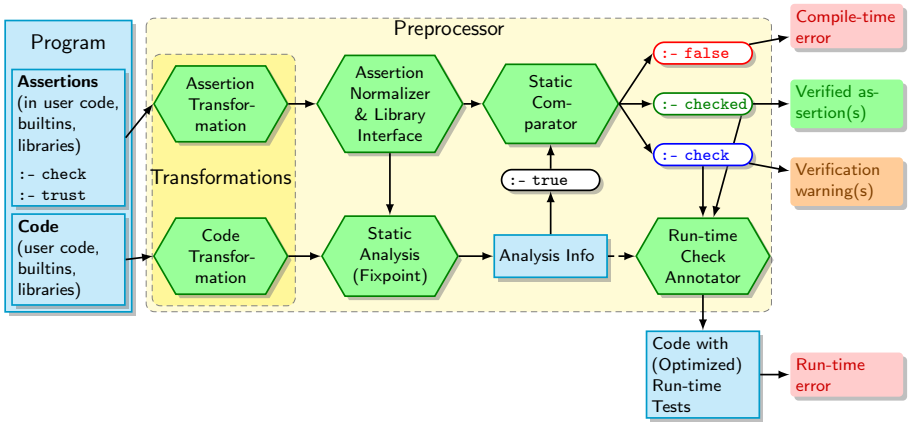


Figure 4.8: Adding the transformations to the Ciao Preprocessor.

fined scenarios, as well as some other minor adaptations and extensions to the interface to select these different scenarios.

In particular, the program transformations used in our experiments for introducing the run-time checks are those of [101], with no caching.

PROPERTIES AND ANALYSIS DOMAINS In our experiments we consider several classes of properties, that are typically of interest to describe the intended semantics of (logic) programs:

- The first one is the *state of variable instantiation*, i.e., which variables are bound to ground terms, or unbound, and, if they are unbound, the sharing ("pointer aliasing") patterns in order to be able to determine independence and transfer accurately grounding information ("strong update"). These properties are approximated safely and quite accurately using the CiaoPP *sharing and freeness* abstract domain [67].
- The second class of properties we will be using refers to the shapes of the data structures constructed by the program in memory. To this end we use the CiaoPP *eterms* [112] abstract domain which infers safe approximations of these shapes as regular trees.
- The third class of properties that we consider refers to the numerical relations among program variables (constraints), in particular linear inequalities over real (floating point) numbers, which are useful to describe properties of numerical parts of programs. To this end we apply CiaoPP's *polyhedra* abstract domain, using the Parma Polyhedra Library (PPL) [3] as back-end solver.

Note that both the Ciao language of assertions and the analyzers in the system support a wide class of additional properties, including sized types, determinacy, non-failure, cardinality, constraints, size relations between variables, consumption of a variety of resources,

etc. [43, 95]. However, we consider the three classes above a suitable study set for our experiments.

Table 4.2: Benchmarks

`boyer`	a theorem prover implementation based on Lisp by R. Boyer (nqthm system), performs symbolic evaluation of a given formula;
`boyerx`	a variant of `boyer` (using generic term manipulation predicates for formula rewrites);
`crypt`	cryptomultiplication puzzles solver;
`deriv`	a program that performs symbolic differentiation of a given formula;
`exp`	exponential calculation;
`factorial`	recursive factorial calculation;
`fft`	fast Fourier transformation calculation;
`fib`	a program that finds N-th Fibonacci number;
`guardians`	prison guards game;
`hamming`	a program that generates the sequence of Hamming numbers;
`hanoi`	hanoi towers puzzle solver for N disks that are moved over three rods;
`jugs`	the water jugs problem;
`knights`	N knights chess problem;
`mmatrix`	matrix multiplication for two matrices with dimensions $n \times n$;
`nreverse`	naive list reversal;
`poly`	a program that raises a polynomial $(1 + x + y + z)$ to the 10th power symbolically;
`primes`	a program that computes N first prime numbers;
`progeom`	a program that constructs a perfect difference set of order N;
`queens`	the N queens program, the number of the queens being the input;
`qsort`	the quicksort program;
`serialize`	a palindrome program;
`tak`	a program that computes the *tak* function;
`witt`	the WITT clustering system implementation;

Table 4.3: Benchmarks set 2, for the *polyhedra* abstract domain.

`ackerman`	Ackermann function computation;
`array`	a generic array API implementation;
`factA`	factorial with multiplication as addition;
`factM`	factorial with direct multiplication;
`incr`	variable increment;
`mc`	McCarthy91 program;
`symstairs`	synchronous increment/decrement for two variables;

BENCHMARKS To study the differences in the run-time overhead levels observed in different assertion checking modes we have selected a set of benchmarks, listed in Tables 4.2 and 4.3.[12]

Given a concrete program, the CiaoPP assertion checking system checks the properties appearing in the assertions in the program and automatically chooses the appropriate abstract domains that have to be used during analysis on order to prove those properties [43]. In our experiments, however, in order to be able to study separately the impact on our proposals for different kinds of properties/domains, we have done the domain selection manually for each benchmark, as follows.

The benchmarks in the first set are symbolic and the properties of their predicates are more naturally expressed using the *eterms* and *sharing and freeness* abstract domains.

The benchmarks in the second set are classical numerical benchmarks, and their properties are more naturally expressed using the *polyhedra* abstract domain (as well as *sharing and freeness* for describing inputs/outputs and absence of sharing/pointer aliasing).

These benchmarks are relatively simple yet diverse programs that represent frequently-occurring programming patterns such as performing symbolic or arithmetic computations, problem solving in fixed domains, processing stream data, etc. In general, they include recursion, search, irregular/dynamic data structures, etc. The relative internal complexity despite their generally small size make them good candidates to answer our main questions, allowing us to concentrate on the properties of interest in each case.

All the benchmarks have been carefully annotated with reasonable program assertions that describe the expected behavior. In the numeric benchmarks the properties contained in the assertions are linear inequalities that should hold for the calls and successes of the predicates involved (see Figure 4.9 for an example). The assertions in the symbolic benchmarks contain shape and sharing/freeness prop-

12 Source available at https://cliplab.org/papers/optchk-scp2017/

```
:- pred mc(A,B) : constraint([A >= 0])
                => constraint([B >= 90, A =< B + 10]).

mc(N,R) :- N > 100,  R is N - 10.
mc(N,R) :-
        T1 is N + 11,
        mc(T1,T2),
        mc(T2,R).
```

Figure 4.9: Assertion with numeric properties (example from `mc`).

erties (Figure 4.10 shows a fragment of the `fft` code as an example).

```
:- regtype complex/1.              % A complex number

complex((A,B)) :- num(A), num(B).

:- pred complex_mul(A, B, C) : (complex(A), complex(B), term(C))
                            => (complex(A), complex(B), complex(C)).

complex_mul((Ra,Ia), (Rb,Ib), (Rc,Ic)) :-
    Rc is Ra*Rb-Ia*Ib,
    Ic is Ra*Ib+Rb*Ia.
```

Figure 4.10: Complex number operations (fragment).

Tables 4.4 and 4.5 present some quantitative characteristics of the benchmarks, such as lines of code (LOC), excluding empty and commented lines, size metrics of the benchmark object file after the compilation, and also the total number of program `pred` assertions. Regarding the sizes after the transformations, note that these transformations only add binary wrapper predicates that incur very little run-time overhead (since arguments do not change order the wrapper predicates translate to a single call instruction, with no argument overhead), so they do not significantly alter the benchmark metrics.

In order to measure the run-time overhead reduction in the client-library interaction scenario (i.e., measuring the gains from eliminating the redundant run-time checks on calls to the library predicates for the *Safe-CTRT* assertion–Section 4.4) we have adapted several of our benchmarks, splitting them into client and library parts. We have selected primarily those benchmarks where such separation is meaningful, i.e., where it is straightforward to identify a part of the benchmark module with a library-like structure that can be placed naturally in a separate module. As an example, we separated the `fft` benchmark into a library for arithmetic operations on complex numbers and the FFT calculations themselves as the client.

We have also concentrated on benchmarks in which there are different call sites to the (now) library predicates, and where some of them required keeping the checks on the calls in the imported predi-

Table 4.4: Benchmarks set 1 metrics (*eterms* and *sharing and freeness* abstract domains).

Code			Assertions
Name	LOC	Size (KB)	total
boyer	853	70	13
boyerx	853	50	12
crypt	76	10	8
deriv	29	9	2
exp	28	6	3
factorial	13	4	2
fft	104	13	10
fib	11	5	3
guardians	78	9	7
hamming	71	9	10
hanoi	44	6	3
jugs	132	10	5
knights	49	9	7
mmatrix	48	6	4
nreverse	14	5	3
poly	81	12	7
primes	33	6	5
progeom	71	8	9
qsort	46	6	6
queens	47	6	7
serialize	81	10	6
tak	18	5	2
witt	651	50	43

cates and others did not, presenting thus good opportunity for study. Table 4.6 lists these benchmarks and the boundary at which the client-library split of each individual benchmark was performed. Note that the *lists* library is listed as used in every benchmark of this client-library interaction study subset. This is because this library provides some of the regular types that are used in the assertions of the client parts of the benchmarks.

EXPERIMENTAL RESULTS (BASE SCENARIO) Tables 4.7 and 4.8 show the compilation time for the benchmarks under the different as-

Table 4.5: Benchmark set 2 metrics (*polyhedra* abstract domain).

Code			Assertions
Name	LOC	Size (KB)	total
ackerman	16	4	1
array	24	6	1
factA	15	4	1
factM	9	4	1
incr	9	4	2
mc	8	4	1
symstairs	15	4	1

Table 4.6: Benchmarks used for the client-library interaction use case.

fft	fast Fourier transformation	lists, complex numbers
hamming	Hamming numbers	lists, queues
hanoi	hanoi towers puzzle encoding	lists
nreverse	naive list reversal	lists
qsort	the quicksort algorithm	lists
witt	the WITT clustering algorithm	lists, sets

sertion checking modes that we have defined.[13] Note that the compilation time for the benchmarks under the *Safe-CT-RT* mode includes the total static analysis and assertion checking times. In all cases the compilation times include the cost of the proposed transformations, except in the unsafe mode, in which no transformations are performed and thus serves also as baseline. The experiments were run on a MacBook Pro with 2.6 GHz Intel Core i5 processor, 8GB RAM, and under the Mac OS X 10.12.5 operating system.

Tables 4.9 and 4.10 show more detail on the analysis and assertion checking times for the *Safe-CT-RT* mode for the different benchmarks. The *load* and *prep* columns indicate the time needed to load the source files and prepare the analyses, and the *shfr*, *eterms*, and *polyhedra* columns the time to perform sharing+freeness, shape (regular types), and numerical analyses, respectively.

13 Times for compilation and analysis assume that the compiler and analyzer are already loaded in memory and ready to execute. Thus, we removed the compiler and CiaoPP start-up time. In the current implementation, the engine needs around 1.4 seconds to load all the necessary bytecode but can then process different programs (e.g., interactively, from within the development environment) without having to be restarted. There exist in any case many solutions to significantly reduce this startup time (keeping code in memory, optimizing the bytecode reader, reduced versions of CiaoPP that contain only the necessary domains, lazy load, etc.).

Table 4.7: Benchmarks: full compilation time (including *eterms* and *sharing and freeness* analysis, assertion checking, and transformations).

Benchmark	Compilation time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
boyer	242	1,271	1,444	469,807
boyerx	221	1,070	1,244	31,426
crypt	174	638	629	2,286
deriv	193	797	765	1,031
exp	167	741	734	1,075
factorial	148	686	1,006	865
fft	181	901	808	3,429
fib	157	608	722	933
guardians	169	673	736	1,580
hamming	187	852	1,085	1,987
hanoi	164	638	635	1,142
jugs	179	827	855	1,590
knights	162	852	974	1,751
mmatrix	174	825	722	1,085
nreverse	163	799	690	989
poly	181	941	909	2,156
primes	173	676	651	1,536
progeom	173	934	845	1,974
qsort	167	770	909	1,341
queens	169	821	1,037	1,405
serialize	167	849	822	1,636
tak	161	959	686	1,035
witt	281	1,866	1,938	180,353

The analyses are actually relatively inexpensive compared to the rest of the compilation passes for most of the benchmarks. The regular type analysis is expensive in `boyer` and `boyerx`. The analysis of the formula rewrite predicates generates many large types whose manipulation is expensive. The `witt` benchmark, despite having more regular data structures (tables of sets and matrices), is also expensive to analyze due to a large number of operations. Note that the *eterms* abstract domain can be controlled in several ways within CiaoPP but we left the analyzer use the automatic, default settings for these experiments. Also note that more efficient –but less precise– domains are available to control analysis cost, many within CiaoPP, such as,

Table 4.8: Benchmarks: full compilation time (including *polyhedra* analysis, assertion checking, and transformations).

Benchmark	Compilation time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
`ackerman`	183	651	460	523
`array`	159	580	568	730
`factA`	146	518	448	649
`factM`	162	505	525	601
`incr`	152	485	480	617
`mc`	149	505	643	632
`symstairs`	153	583	481	661

for example, several widenings for sharing [73, 59], pair sharing domains [98, 94], or other type inference domains [33, 9].

Tables 4.11, 4.12, 4.13, and 4.14 report on the actual execution times for each benchmark using the different assertion checking modes, together with data on the results of assertion checking. For some of the benchmarks, measurements were taken for calls with several input values and this is expressed using the notation $Name(Input)$. The 'Checked Assertion Conditions' column reports the ratios of statically checked *calls* and *success* assertion conditions in the *Safe-CT-RT* checking mode to the total number of respective assertion conditions in the *Safe-RT* checking mode for each benchmark (i.e., N/M means that N out of the M assertion conditions are checked).

In the worst case the overhead in the *Safe-RT* checking mode is three orders of magnitude higher than in *Client-safe*, but *Safe-CT-RT* removes one order of magnitude (`boyerx`, `fft`, `knights`, `witt`). This is expected since run-time checks of complex properties like data shapes cannot be performed in constant time. The run-time checking process changes the complexity of the programs and the overhead increases as the size of the input grows. Note that the *Client-safe* mode also represents the theoretically lowest overhead that we could obtain (assuming a fixed implementation of the instrumentation), by removing all the internal checks, but keeping the library interface checks.

We can observe performance variations due to secondary effects (code layout, cache alignment), due to which sometimes the time in *Safe-CT-RT* mode can be slightly smaller than in *Client-safe* mode (`crypt`). To reduce the measurement noise (also influenced by the computations performed by other processes) we execute each benchmark several times and report the minimal time.[14]

14 The current measurements depend on the C `getrusage()` function, that on Mac OS has microsecond resolution.

Table 4.9: Static analysis time for benchmarks using the *Safe-CT-RT* checking mode with *eterms* and *sharing and freeness* analyses (part of total compilation time).

Benchmark	load	Analysis time, ms				Assertion checking
		prep	*shfr*	prep	*eterms*	
boyer	757.43	9.21	62.59	9.36	737.73	614.33
boyerx	686.38	6.40	53.63	6.68	556.77	408.75
crypt	528.04	1.43	8.46	1.41	39.37	138.05
deriv	478.51	1.04	4.32	1.46	17.36	32.74
exp	460.00	0.49	2.12	0.45	15.17	54.11
factorial	493.96	0.29	1.61	0.25	11.21	16.75
fft	515.95	1.84	9.54	1.91	43.05	162.43
fib	477.50	0.41	2.71	0.90	13.06	17.47
guardians	481.39	1.08	9.24	1.22	28.86	63.74
hamming	536.77	1.22	9.10	1.22	27.85	81.34
hanoi	477.13	0.59	2.91	0.47	15.65	21.79
jugs	494.40	1.02	5.08	1.19	26.75	126.10
knights	527.57	0.90	4.19	1.37	32.31	58.59
mmatrix	482.28	1.28	3.85	0.80	15.12	27.86
nreverse	524.33	0.50	3.23	0.30	3.50	9.01
poly	494.44	1.67	50.94	1.49	52.26	103.17
primes	527.36	0.80	2.64	0.55	17.34	33.09
progeom	481.82	1.30	7.18	1.00	27.66	59.31
qsort	496.13	1.00	5.77	0.64	8.23	22.18
queens	512.68	0.71	4.77	1.32	22.13	48.62
serialize	496.34	1.33	15.28	1.41	24.30	52.89
tak	519.57	0.44	1.75	0.43	13.76	26.67
witt	580.95	15.52	124,284.60	15.88	847.20	1,250.04

In practice, in many programs *Safe-CT-RT* is able to remove most of the checks, except of course those corresponding to the external predicates. We included in the benchmarks two versions of `boyer`. The original translation (which we call here `boyerx`) uses `functor/3` and `arg/3` to implement rewrites of arbitrary terms representing formulas. This makes the domains lose precision. The `boyer` version uses instead a larger predicate that explicitly enumerates possible formula terms.

The benefits of applying the CTRT transformation are not so prominent in the case of numerical analysis, mainly due to the fact that the numerical checks are usually much less costly than the data shape checks. However, in programs that include arithmetic operations that

Table 4.10: Static analysis time for benchmarks using the *Safe-CT-RT* checking mode with *polyhedra* numerical analysis (part of total compilation time).

Benchmark	`load`	Analysis time, ms		Assertion checking
		prep	*polyhedra*	
`ackerman`	476.19	1.02	11.99	13.21
`array`	460.07	0.86	38.55	21.37
`factA`	451.58	1.01	33.67	13.43
`factM`	459.06	0.72	4.01	9.07
`incr`	448.13	0.79	5.19	6.96
`mc`	493.90	0.70	5.41	6.75
`symstairs`	460.86	0.79	7.34	10.12

are not captured well by the *polyhedra* abstract domain the overhead reduction is still noticeable (e.g., compare the running times of the `factA` and `factM` benchmarks, which differ only in the way they perform multiplication). Another challenge for the domain are complex benchmarks like `ackerman` (double recursion) and `mc`.

EXPERIMENTAL RESULTS (CLIENT-LIBRARY SCENARIO) Table 4.15 shows the compilation time for the client-library scenario benchmarks from Table 4.6. As mentioned before, each of the benchmarks was split into client and library modules, and then two versions were generated of the client module: one without any optimization of the calls to the library and the other applying the program-point calls optimization ('Unoptimized' and 'Optimized' columns, respectively). All files were compiled in the *Safe-CT-RT* checking mode. One can notice that sum of the compilation times of client and libraries is proportional to the compilation time of the 'monolithic' version.

Table 4.16 provides the details for the analysis times of the client-library scenario benchmarks. The 'Part=C-u' rows report the analysis times for the client modules without optimizations of the calls to the library modules and the 'Part=C-o' ones report the times for the client modules with the optimized calls. The 'Part=L' rows provide the analysis times for the library modules. The sum of the analysis times of this client-library separated benchmark versions is comparable to the analysis time of the 'monolithic' benchmark versions reported above. The slight increase in the analysis time is expected, since processing a module and the modules at its interface takes some time.

The fact that the analysis times in the two-module scenario do not differ much from the analysis times of the 'monolithic' version of our benchmarks provides evidence supporting the scalability of the transformations that we have proposed, in the sense that, since changes in

Table 4.11: Benchmark execution times under the different modes (all benchmarks).

Benchmark	Execution time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
boyer	11.665	11.350	3,215.894	14.010
boyerx	17.541	17.755	2,621.203	1,254.041
crypt	0.106	0.118	6.601	0.114
deriv	0.013	0.062	4.629	0.071
exp	4.359	4.363	73.321	4.427
factorial	0.008	0.014	0.803	0.015
fft	28.419	32.702	32,112.845	254.773
fib	0.080	0.086	16.052	0.094
guardians	3.637	3.255	6,521.171	3.866
hamming	17.793	18.288	9,860.070	20.197
hanoi (8)	0.057	0.070	122.730	0.086
jugs	0.017	0.026	1.529	0.026
knights	232.922	232.940	18,842.485	250.993
mmatrix (4)	0.005	0.016	0.742	0.017
nreverse	2.438	2.699	10,596.668	3.640
poly	1.172	1.371	428.480	1.404
primes	0.033	0.044	11.066	0.040
progeom (8)	5.702	5.694	2,222.974	6.378
qsort (32)	0.022	0.030	7.382	0.035
queens (8)	2.522	2.527	545.413	2.846
serialize (25)	0.012	0.025	4.998	0.029
tak	2.980	2.991	980.910	3.457
witt	24.027	17.488	1,853.552	389.750

the client code do not affect the library part any more, only that part of the program will have to be recompiled should some changes be made. Even if the largest part of the cost is in the client (e.g., `witt`), note that the observation before is also true with respect to changes in the library, i.e., the client will not have to be reanalyzed for changes in the library.

The actual execution times for the benchmarks in the client-server scenario are given in Table 4.17. Here we are of course interested in the effect of the optimization of the checks at the module boundaries, i.e., in comparing the 'Unoptimized' and 'Optimized' results.

Table 4.12: Checked vs. total assertions (all benchmarks).

Benchmark	Checked Assertion Conditions	
	calls	success
boyer	13/13	12/12
boyerx	11/12	10/11
crypt	7/8	8/8
deriv	1/2	1/1
exp	2/3	2/2
factorial	1/2	1/1
fft	9/10	8/9
fib	2/3	2/2
guardians	6/7	6/6
hamming	9/10	9/9
hanoi (8)	1/2	2/2
jugs	4/5	4/4
knights	6/7	6/6
mmatrix (4)	2/3	3/3
nreverse	2/3	2/2
poly	6/7	7/7
primes	3/4	4/4
progeom (8)	7/8	8/8
qsort (32)	4/5	3/3
queens (8)	5/6	4/4
serialize (25)	4/5	4/4
tak	1/2	1/1
witt	31/43	38/40

The results show that in the optimized case the execution times are reduced and comparable to those in the previous 'monolithic' setup (i.e., to the times in Tables 4.11, 4.13). The minor deviations from that case are due to the noise in the measurements and the use of additional predicate wrappers in the interface of the library (that was not present in the 'monolithic' versions). These wrappers are necessary to distinguish internal from external calls within the library. This effect can be observed in the execution times of the `hamming` benchmark: the current compilation mechanism introduces these wrapper predicates that add some overhead, and since in `hamming` the operations are

Table 4.13: Benchmark execution times under the different modes (benchmarks subset, varied output).

Benchmark	Execution time, ms			
	Unsafe	Safe		
		Client	RT	CT+RT
`hanoi` (2)	0.000	0.012	0.161	0.013
`hanoi` (4)	0.002	0.014	1.517	0.015
`hanoi` (8)	0.057	0.070	122.730	0.086
`mmatrix` (2)	0.001	0.010	0.148	0.010
`mmatrix` (3)	0.002	0.011	0.358	0.013
`mmatrix` (4)	0.005	0.016	0.742	0.017
`progeom` (2)	0.002	0.005	0.615	0.005
`progeom` (4)	0.096	0.098	28.118	0.111
`progeom` (8)	5.702	5.694	2,222.974	6.378
`qsort` (8)	0.002	0.008	0.839	0.008
`qsort` (16)	0.008	0.014	2.664	0.016
`qsort` (32)	0.022	0.030	7.382	0.035
`queens` (4)	0.007	0.009	1.248	0.011
`queens` (6)	0.133	0.136	29.527	0.153
`queens` (8)	2.522	2.527	545.413	2.846
`serialize` (9)	0.004	0.008	0.881	0.011
`serialize` (16)	0.006	0.013	2.343	0.014
`serialize` (25)	0.012	0.025	4.998	0.029

Table 4.14: Benchmark (`polyhedra`) execution times under the different modes and checked vs. total assertions.

Benchmark	Execution time, ms				Checked Assertion Conditions	
	Unsafe	Safe				
		Client	RT	CT+RT	calls	success
`ackerman`	0.042	0.043	4.049	0.045	1/1	1/1
`array`	0.004	0.003	0.043	0.003	1/1	1/1
`factA`	0.002	0.008	0.032	0.018	0/1	1/1
`factM`	0.001	0.008	0.031	0.043	0/1	0/1
`incr`	0.001	0.007	0.128	0.032	1/2	2/2
`mc`	0.007	0.015	0.737	0.312	0/1	1/1
`symstairs`	0.006	0.012	0.601	0.309	0/1	1/1

Table 4.15: Benchmarks: full compilation time (client-library scenario).

Benchmark	Compilation time, ms		
	Client		Library
	Unoptimized	Optimized	
fft	3,253	3,385	1,674
hamming	1,565	1,542	1,463
hanoi	1,219	1,297	1,035
nreverse	1,040	1,203	1,089
qsort	1,377	1,223	1,105
witt	169,944	164,121	2,617

Table 4.16: Static analysis time for benchmarks (B) (client-library scenario, part of total compilation time).

B	Part	load	Analysis time, ms				Assertion checking
			prep	shfr	prep	eterms	
fft	C-u	486.76	1.39	8.38	1.49	40.45	125.55
	C-o	518.09	1.33	8.24	1.31	38.07	121.75
	L	484.69	0.75	5.45	0.64	25.72	82.77
hamming	C-u	483.03	0.92	4.50	0.87	19.09	87.67
	C-o	512.81	1.08	5.11	0.89	19.86	53.74
	L	451.23	0.61	7.28	0.64	8.68	22.41
hanoi	C-u	470.81	0.45	2.36	0.34	11.66	23.17
	C-o	509.23	0.42	2.39	0.32	12.10	15.29
	L	454.93	0.37	2.00	0.31	3.30	11.86
nreverse	C-u	456.46	0.33	2.04	0.26	2.23	10.32
	C-o	492.82	0.50	2.88	0.32	3.25	7.27
	L	447.63	0.26	2.03	0.21	1.87	5.33
qsort	C-u	480.46	0.73	3.87	0.68	7.33	21.92
	C-o	498.62	0.52	3.61	0.51	7.11	16.74
	L	485.56	0.61	3.40	0.39	3.48	8.06
witt	C-u	505.32	12.32	118,807.15	14.41	722.83	1,491.66
	C-o	605.32	14.77	117,395.04	12.43	663.12	1,128.95
	L	488.70	2.08	64.39	2.21	41.74	88.74

very simple this overhead becomes noticeable. However, this overhead does not have a big impact in other benchmarks.

In the case where we have not optimized the checks at the boundaries of the module (the 'Unoptimized' column) execution times are

Table 4.17: Benchmark execution times in the client-library scenario.

Benchmark	Execution time, ms	
	Unoptimized	Optimized
`fft`	2,199.29	271.78
`hamming`	146.85	60.47
`hanoi` (2)	0.03	0.01
`hanoi` (4)	0.16	0.02
`hanoi` (8)	3.21	0.10
`nreverse`	22.13	3.39
`qsort` (8)	0.08	0.01
`qsort` (16)	0.16	0.02
`qsort` (32)	0.32	0.04
`witt`	466.34	426.21

higher than in the 'monolithic' setup and are only superseded by the times with all run-time checks enabled (the *Safe-CT-RT* mode). These experiments clearly demonstrate the positive effect of eliminating run-time checks at module boundaries. It is quite interesting that we are able to achieve these performance gains without generating more versions or specializing the program (which is important in some contexts).

4.6 CONCLUSIONS

Our overall objective is to construct automatic verification and debugging systems for non-trivial properties, that can be used routinely as part of the development process for both prototyping and production code. Our concrete approach is the use of frameworks that combine static and dynamic verification, i.e., systems that combine compile-time and run-time checking of user-provided assertions. In this chapter we have addressed the study of how run-time overhead can be reduced in different scenarios and, specially, via static analysis.

We have defined four practical assertion checking modes, and studied the corresponding trade-offs between the level of guarantees provided by each one and the corresponding execution time slowdown. For these checking modes we have explored the effectiveness of abstract interpretation in detecting the parts of the program's (partial) specifications that can be statically simplified to true or false, concentrating on the practical impact of such analysis in reducing the cost of the run-time checks required for the remaining parts of the specifications. We have also addressed the application of our approach when optimizing run-time checks for the calls across client-library

boundaries. We have described a typical client-library use case and discussed the possibilities for optimizing the run-time checks in this context using an illustrative example. Also, we have proposes a new program point source transformation for avoiding the duplication of run-time checks.

We have also proposed program transformations that allow incorporating the run-time checking semantics into the analysis phase and demonstrated that this approach can increase analysis precision and allow for better and more fine-grained (program-point) check elimination.

Our experiments have shown that there is indeed a significant advantage in using analysis to reduce the overhead implied by the run-time tests. We argue that the results are encouraging, supporting the hypothesis that the combination of run-time checking with analysis can reduce checking overhead sufficiently to allow providing full safety in production code, for non-trivial properties.

While evaluating the effectiveness of our assertion-based approach in finding errors in programs was not directly the objective of this chapter (we concentrated here on measuring the reduction in run-time overhead due to analysis and the enhancements proposed), during our experiments a good number of program errors were flagged by the system. In particular, it is worth mentioning that the analysis of one of the more complex programs, `boyer`, allowed us to spot bugs in the original translation from LISP that had been around for 30 years!

5

SHALLOW RUN-TIME CHECKING

Modular programming has become widely adopted due to the benefits it provides in code reuse and structuring data flow between program components. A tightly related concept is the principle of *information hiding* that allows concealing the concrete implementation details behind a well-defined interface and thus allows for cleaner abstractions. Different programming languages implement these concepts in different ways, some examples being the encapsulation mechanism of classes in object-oriented programming and opaque data types. In the (C)LP context, most mature language implementations incorporate module systems, some of which allow programmers to restrict the visibility of some functor symbols to the module where they are defined, thus both hiding the concrete implementation details of terms from other modules and providing guarantees that only the predicates of that particular module can use those functor symbols as term constructors or matchers.

One of the most attractive features of untyped languages for programmers is the flexibility they offer in term creation and manipulation. However, with such power comes the responsibility of ensuring correctness in the manipulation of data, and this is specially relevant when data can come from unknown clients. A popular solution for ensuring safety is to enhance the language with optional assertions that allow specifying correctness conditions both at the module boundaries and internally to modules. These assertions can be checked dynamically by adding run-time checks to the program, but this can introduce overheads that are in many cases impractical. Such overheads can be greatly reduced with static analysis, but the gains then depend strongly on the quality of the analysis information inferred. Unfortunately, there are some common scenarios where shape/type analyses are necessarily imprecise. A motivational example is the case of reusable libraries, i.e., the case of analyzing, verifying, and compiling a library for general use, without access to the client code or analysis information on it. This includes for example the important case of servers accessed via remote procedure calls. Static analysis faces challenges in this context, since the unknown clients can fake data that is really intended to be internal to the library. Ensuring safety then requires sanitizing input data with potentially expensive run-time checks.

In order to alleviate this problem, in this chapter we present techniques that, by exploiting term hiding and the strict visibility rules of the module system, can greatly improve the quality of the shape

information inferred by static analysis and reduce the run-time overhead for the calls across module boundaries by several orders of magnitude. These techniques can result in improvements in the number and size of checks that allow bringing guarantees and overheads to levels close to those of statically-typed approaches, but without imposing on programs the restriction of being well-typed. In particular, we present a semantics for modular logic programs where the mapping of module symbols is abstract and implementation-agnostic, i.e., independent of the visibility rules of particular module systems.

5.1 AN ABSTRACT APPROACH TO MODULAR LOGIC PROGRAMS

There have been several proposals to date for supporting modularity in logic programs, all of which are based on performing a partition of the set of program symbols into modules. The two most widely adopted approaches are referred to as *predicate-based* and *atom-based* module systems. In predicate-based module systems all symbols involved in terms are global, i.e., they belong to a single global `user` module –a special module from which all modules import the symbols and to which all modules can add symbols. In atom-based module systems [106] only constants and explicitly exported symbols are global, while the rest of the symbols are local to their modules. Ciao [15] adopts a hybrid approach which is as in predicate-based systems but with the possibility of marking a selected set of symbols as local (we will use this model in the examples in Sec. 5.3). Despite the differences among these module systems, by performing module resolution applying the appropriate visibility rules, programs are reducible in all systems to a form that can be interpreted using the same Prolog-style semantics. We will use this property in order to abstract our results away from particular module systems and their symbol visibility rules. To this end we present a formalization of the "flattened" version of a modular program, where visibility is explicit and is thus independent of the visibility conventions of specific module systems. Let MS denote the set of all *module symbols*. The *flattened* form of a modular definite program is defined as follows:

Definition 5.1 (Modular Program)**.** *A* modular program *is a pair of* $(P, \mathsf{mod}(\cdot))$*, where* P *is a definite program and* $\mathsf{mod}(\cdot)$ *is a mapping that assigns for each symbol* $f \in \mathsf{FS}$ *a unique module symbol* $m \in \mathsf{MS}$*. Let* C *be a clause* $H \leftarrow B$ *in* P*,* $\mathsf{mod}(C) \triangleq \mathsf{mod}(H)$*. Let* A *be an atom*[1] *or a term of the form* $f(\ldots)$*. Then* $\mathsf{mod}(A) \triangleq \mathsf{mod}(f)$*.*

In this chapter the FS *set also includes predicate symbols.*

The $\mathsf{mod}(\cdot)$ mapping creates a partition of the clauses in the definite program P. We refer to each resulting equivalence class as a module, and represent it with the module symbol shared by all clauses in

1 In practice constraints are also located in modules. It is trivial to extend the formalization to include this, we do not write it explicitly for simplicity.

that class. The set of all symbols defined by a module m is $\mathsf{def}(m) = \{f | f \in \mathsf{FS}, \mathsf{mod}(f) = m, m \in \mathsf{MS}\}$.

Definition 5.2 (Interface of a Module)**.** *The* interface *of a module m is given by the disjoint sets $\mathsf{exp}(m)$ and $\mathsf{imp}(m)$, s.t. $\mathsf{exp}(m) \subseteq \mathsf{def}(m)$ is the subset of the symbols defined in m that can appear in other modules, referred to as the* export list *of m, and $\mathsf{imp}(m) = \{f | f \in \mathit{FS}, f$ is in symbols of $\mathsf{cls}(p), p \in \mathsf{def}(m)\} \setminus \mathsf{def}(m)$ is a superset of symbols in the bodies of the predicates of m, that are not defined in m, referred to as the* import list *of m.*

To track calls across module boundaries we introduce the notion of *clause end literal*, a marker of the form $\mathsf{ret}(H)$, where H stands for the head of the parent clause, as given in the following definition:

Definition 5.3 (Operational Semantics of Modular Programs)**.** *We redefine the basic derivation semantics of the Definition 2.1, such that goal sequences are of the form $(L, m) :: G$ where L is a literal, and m is the module from which L was introduced, as shown below.*
Then, a state $S = \langle (L, m) :: G \mid \theta \rangle$ can be reduced to a state S' as follows:

1. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ *if L is a constraint and $\theta \wedge L$ is satisfiable.*
2. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle (B_1, n) :: \ldots :: (B_k, n) :: (\mathit{ret}(L), n) :: G \mid \theta \rangle$ *if L is an atom and $\exists (L \leftarrow B_1, \ldots, B_k) \in \mathsf{cls}(L)$ where $\mathsf{mod}(L) = n$ and it holds that $(L \in \mathsf{def}(n) \wedge n = m) \bigvee (L \in \mathsf{exp}(n) \wedge L \in \mathsf{imp}(m) \wedge n \neq m)$.*
3. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \rangle$ *if L is a clause return literal $\mathit{ret}(\_)$.*

Basically, for reduction step 2 to succeed, the L literal should either be defined in module m (and then $n = m$) or it should belong to the export list of module n and be in the import list of module m.

5.2 RUN-TIME CHECKING OF MODULAR PROGRAMS

SEMANTICS WITH RUN-TIME CHECKING OF ASSERTIONS AND MODULES We now present the operational semantics with assertions for modular programs, which checks whether assertion conditions hold or not while computing the derivations from a query in a modular program. The set of derivations for a modular program from its set of queries $\mathcal{Q}$ using the semantics with run-time checking of assertions is denoted by $\mathsf{rtc\text{-}derivs}(\mathcal{Q})$. We also extend the clause return literal to the form $\mathsf{ret}(H, \mathcal{C})$, where $\mathcal{C}$ is the set of identifiers c_i of the assertion conditions that should be checked at that derivation point.

Definition 5.4 (Operational Semantics for Modular Programs with Run-time Checking)**.** *A state $S = \langle (L, m) :: G \mid \theta \rangle$ can be* reduced *to a state S', denoted $S \rightsquigarrow_{\mathsf{rtc}} S'$, as follows:*

1. *If L is a constraint then $S' = \langle G \mid \theta \wedge L \rangle$ if $\theta \wedge L$ is satisfiable.*

2. *If L is an atom and $\exists(L \leftarrow B_1, \ldots, B_k) \in \mathsf{cls}(L)$, then the new state S' is obtained as*

$$S' = \begin{cases} \langle \mathit{err}(c) \mid \theta \rangle & \mathit{if}\ \exists\ C = \mathit{calls}(L, \mathit{Pre}) \in \mathcal{A}_C\langle L \rangle \\ & \quad \wedge\ \mathit{id}(C) = c \wedge \theta \not\Rightarrow \mathit{Pre} \\ \langle (B_1, n) :: \ldots :: (B_k, n) :: (\mathit{ret}(L, \mathcal{C}), n) :: G \mid \theta \rangle & \mathit{otherwise} \end{cases}$$

s.t. $\mathcal{C} = \{c_i \mid C_i = \mathit{success}(L, \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A}_C\langle L \rangle \wedge \mathit{id}(C_i) = c_i \wedge \theta \Rightarrow \mathit{Pre}_i\}$ where $\mathsf{mod}(L) = n$ and it holds that $(L \in \mathsf{def}(n) \wedge n = m) \bigvee (L \in \mathsf{exp}(n) \wedge L \in \mathsf{imp}(m) \wedge n \neq m)$

3. *If L is a clause return literal $\mathit{ret}(\_, \mathcal{C})$, then*

$$S' = \begin{cases} \langle \mathit{err}(c) \mid \theta \rangle & \mathit{if}\ \exists\ c \in \mathcal{C}\ \mathit{s.t.}\ C = \mathit{success}(L', \_, \mathit{Post}) \in \mathcal{A}_C\langle L' \rangle \\ & \quad \wedge \mathit{id}(C) = c \wedge \theta \not\Rightarrow \mathit{Post} \\ \langle G \mid \theta \rangle & \mathit{otherwise} \end{cases}$$

Theorem 5.1 below on the correctness of the operational semantics with run-time checking can be straightforwardly adapted from [99]. The completeness of this operational semantics as presented in Theorem 5.2 below can only be proved for *partial* program derivations, as the new semantics introduces the $\mathsf{err}(\_)$ literal that directly replaces the goal sequence of a state in which a violation of an assertion condition occurs. Below we adapt the Definition 2.13 from the Chapter 2, as it will be instrumental for the Theorem 5.1 proof.

Definition 5.5 (Error-erased Program Derivations)**.** *The set of* error-erased partial *derivations from $\rightsquigarrow_{\mathsf{rtc}}$ is obtained by a syntactic rewriting $(-)^\circ$ that removes the error states and sets of assertion condition identifiers from the clause end literals. It is recursively defined as follows:*

$$(S_1, \ldots, S_m, S_{m+1})^\circ = \begin{cases} (S_1, \ldots, S_m)^\circ & \mathit{if}\ S_{m+1} = \langle \mathit{err}(c) \mid \_ \rangle \\ (S_1, \ldots, S_m)^\circ \parallel ((S_{m+1})^\circ) & \mathit{otherwise} \end{cases}$$

$$\langle G \mid \theta \rangle^\circ = \langle G^\circ \mid \theta \rangle$$

$$(L :: G)^\circ = \begin{cases} (\mathit{ret}(L'), m) :: (G^\circ) & \mathit{if}\ L = \mathit{ret}(L', \_) \\ L :: (G^\circ) & \mathit{otherwise} \end{cases}$$

$$\square^\circ = \square$$

where $\parallel$ stands for sequence concatenation.

Theorem 5.1 (Correctness Under Assertion Checking)**.** *For any tuple $(P, \mathcal{Q}, \mathcal{A})$ it holds that $\forall D' \in \mathit{rtc\text{-}derivs}(\mathcal{Q})\ \exists D \in \mathit{derivs}(\mathcal{Q})$ s.t. D' is equivalent to D (including partial derivations).*

We define that D' is equivalent to D iff $(D')^\circ = D$.

Proof. Let $D' = (S'_1, \ldots, S'_k)$, $S_i = \langle (L'_i, m_i) \mid \theta'_i \rangle$, for $Q = ((L'_1, m_1), \theta'_1) \in \mathcal{Q}$ and $S'_i \rightsquigarrow_{\mathsf{rtc}} S'_{i+1}$. Proof by induction on the length k of D':

- Base case ($k = 1$). $(S'_1)^\circ = \langle (L'_1, m_1)^\circ \mid \theta'_1 \rangle = \langle (L'_1, m_1) \mid \theta'_1 \rangle = S_1$ since L'_1 can be neither the $\mathsf{ret}(\_,\_)$ nor $\mathsf{err}(\_)$ literal, as they require at least one $\rightsquigarrow_{\mathsf{rtc}}$ state reduction to be reached.
- Inductive case (show $k+1$ assuming k holds). In the inductive step it is enough to consider the cases that are different in the $\rightsquigarrow$ and $\rightsquigarrow_{\mathsf{rtc}}$ reductions:
  - If $D' = (S'_1, \ldots, S'_k, S'_{k+1})$ and $S'_{k+1} = \langle \mathsf{err}(\_) \mid \theta'_{k+1} \rangle$ then from Def. 5.5 it immediately follows $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $(D')^\circ = (S'_1, \ldots, S'_k, S'_{k+1})^\circ = (S'_1, \ldots, S'_k) = D$
  - If $D' = (S'_1, \ldots, S'_k, S'_{k+1})$ and $S'_{k+1} = \langle \mathsf{ret}(L, \_) \mid \theta'_{k+1} \rangle$ then from Def. 5.5 it immediately follows $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $(D')^\circ = (S'_1, \ldots, S'_k)^\circ \| (S'_{k+1})^\circ = (S'_1, \ldots, S'_k) \| \langle \mathsf{ret}(L) \mid \theta'_{k+1} \rangle = D$

□

Theorem 5.2 (Partial Completeness Under Assertion Checking)**.** *For any tuple $(P, \mathcal{Q}, \mathcal{A})$ it holds that $\forall D = (S_1, \ldots, S_k, S_{k+1}, \ldots, S_n) \in \mathit{derivs}(\mathcal{Q})$ $\exists D' \in \mathit{rtc\text{-}derivs}(\mathcal{Q})$ s.t. D' is equivalent to D or $(S_1, \ldots, S_k, \langle \mathit{err}(c) \mid \_ \rangle)$.*

The proof of Theorem 5.2 is trivial, based on the same reasoning as in the proof of Theorem 5.1, and is not included.

5.3 SHALLOW RUN-TIME CHECKING

As mentioned before, the main advantage of modular programming is that it allows safe local reasoning on modules, since two different modules are not allowed to contribute clauses to the same predicate.[2] Our purpose herein is to study how in systems where the visibility of function symbols can be controlled, similar reasoning can be performed at the level of terms, and in particular how such reasoning can be applied to reducing the overhead of run-time checks. We will refer to these reduced checks as *shallow* run-time checks, which we will formally define later in this section. We start by recalling how in cases where the visibility of terms function symbols can be controlled, this reasoning is impossible without global (inter-modular) program analysis, using the following example module of the Figure 5.1:

Example 5.1. *Consider a module* `m1` *that exports a single predicate* `p/1` *that constructs* `point/1` *terms, as shown in Figure 5.1 (a). Here, we want to reason about the terms that can appear during program execution at several specific program points: (a) before we call* `p/1` *(point at which execution enters module* `m1`*); (b) when the call to* `p/1` *succeeds (point at which execution leaves the module); and (c) before we call* `q/1` *(point at which execution*

2 In practice, an exception is `multifile` predicates. However, since they need to be declared explicitly, local reasoning is still valid assuming conservative semantics (e.g., *topmost* abstract values) for them.

enters another module). When we exit the module at points (b) and (c) we know that in any `point(X)` *constructed in* $m1$ *either* $X = 1$ *or* $X = 2$. *However, when we enter module* $m1$ *at point (a)* `A` *could have been bound by the calling module to any term including, e.g.,* `point([4,2])`, `point(2)`, `point(a)`, `point(1)`, *etc., since the use of the* `point/1` *functor is not restricted.*

Now we will consider the case where the visibility of terms can be controlled. We start by defining the following notion:

Definition 5.6 (Hidden Functors of a Module)**.** *The set of hidden functors of a module is the set of functors that appear in the module that are local and non-exported.*

```
:- module(m1, [p/1, r/0]). % m1 declared,
                           %   p/1 and r/0 exported
p(A) :- A = point(B),
        B = 1.             % A = user:point(1)

:- use_module(m2,[q/1]).   % import q/1 from a module m2

r :- X = point(2), q(X).   % X = user:point(2)
```

(a) a module with all its functors visible.

```
:- module(m1, [p/1, r/0]).
:- hide point/1.           % point/1 is restricted to m1

p(A) :- A = point(B),
        B = 1.             % m1:point(1),
                           %   not user:point(1)
:- use_module(m2,[q/1]).
                           % m1:point(2) escapes
r :- X = point(2), q(X).   %   through call to q/1
```

(b) the same module but with hidden functors.

Figure 5.1: Example module.

Example 5.2. *In this example we mark instead the* `point/1` *symbol as hidden, as shown in Figure 5.1 (b). We use Ciao module system notation [15], where all function symbols belong to* `user`, *unless marked with a* `:- hide f/N` *declaration. Such symbols are hidden, i.e., local and not exported.*[3]
Let us consider the same program points as in Example 5.1. When we exit the module, we can infer the same results, but with `m1:point/1` *instead of* `user:point/1`. *Now, if we see the* `m1:point(X)` *term at point (a) we know that it has been constructed in* $m1$, *and the* `X` *has to be bound to either* 1 *or* 2, *because the code that can create bindings for* X *is only located in* $m1$ *(and the* `point/1` *terms are passed outside the module at points (b) and (c)).*

As mentioned before, these considerations will allow us to use an optimized form of checking that we refer to as *shallow checking*. In

3 Note that this can be achieved in other systems: e.g., in XSB [106] it can be done with a `:- local/1` declaration, combined with not exporting the symbol.

order to formalize this notion, we start by defining all possible terms that may exist outside a module m as its *escaping terms*. We will also introduce the notion of *shallow properties* as the specialization of the definition of these properties w.r.t. these escaping terms, and we will present algorithms to compute such shallow versions of properties.

Definition 5.7 (Visible Terms at a State)**.** *A property whose model represents all terms that are visible in a state $S = \langle (L, \_) :: G \mid \theta \rangle$ of some derivation $D \in$ rtc-derivs$(\mathcal{Q})$ for a tuple $(P, \mathcal{Q}, \mathcal{A})$ is given by the predicate $\mathsf{vis}_S(X) \leftarrow \bigvee_{V \in \mathsf{Vars}_L}(X = V \wedge \theta)$, where Vars_L denotes the set of variables of literal L.*

Definition 5.8 (Escaping Terms)**.** *Consider all states S in all derivations $D \in$ rtc-derivs$(\mathcal{Q})$ of any tuple $(P, \mathcal{Q}, \mathcal{A})$, where P imports a given module m. A property whose model represents escaping terms w.r.t. m is given by the predicate $\mathsf{esc}_m(X) \leftarrow \bigvee \mathsf{vis}_S(X)$ for each $S = \langle (\_, n) :: \_ \mid \_ \rangle$ with $n \neq m$.*

The set of all public symbols to which a variable X can be bound is denoted as $\mathsf{usr}(X) = \{X | \mathsf{mod}(X) = \texttt{user}\}$. The following lemma states that it is enough to consider the states at the module boundaries to compute $\mathsf{esc}_m(X)$:

Lemma 5.3 (Escaping at the Boundaries)**.** *Consider all derivation steps $S_1 \rightsquigarrow_{\mathsf{rtc}} S_2$ where $S_1 = \langle (L_1, m) :: \_ \mid \_ \rangle$ and $S_2 = \langle (L_2, n) :: \_ \mid \theta \rangle$ with $n \neq m$. That is, the derivation steps when calling a predicate at n from m (if L_1 is a literal) or when returning from m to module n (if L_1 is ret$(_)$). Let $\mathsf{esc}_{m'}(X)$ be the* smallest property *(i.e., the property with the smallest model) such that $\theta \Rightarrow \mathsf{esc}_{m'}(X)$ for each variable X in the literal L_2, and $\mathsf{usr}(X) \Rightarrow \mathsf{esc}_{m'}(X)$. Then $\mathsf{esc}_{m'}(X) \vee \mathsf{usr}(X)$ is equivalent to $\mathsf{esc}_m(X)$.*

Proof. Let $\mathsf{esc}_m(X) \equiv \bigvee_i \bigvee_{V \in Vars_i}(X = V \wedge \theta_i)$ and $\mathsf{esc}_{m'}(X) \equiv \bigvee_i \bigvee_{V \in Vars'_i}(X = V \wedge \theta'_i)$. From the definitions, it can be seen that the set of all θ'_i (at the boundaries, before and after m) is a subset of all θ_i (outside m). The rest of the θ_i correspond to states not preceded by a literal from m. For such states $\bigvee_{V \in Vars_i}(X = V \wedge \theta_i)$ must be: 1) covered by $\mathsf{usr}(X)$ (and thus $\mathsf{esc}_{m'}(X)$); or 2) contain some $X = f(\ldots)$ with f hidden in m. Since f cannot appear in literals from $n \neq m$ then it must have come from some $\theta_b \wedge \theta_o$, where θ_b is some ancestor at the boundaries (already covered), θ_o is a conjunction of constraints introduced outside m (with cannot contain f), and thus it is more specific and also covered by $\mathsf{esc}_{m'}(X)$). □

Algorithm 5.1 computes an over-approximation of the $\mathsf{esc}_m(X)$ property. The algorithm has two parts. First, it loops over the exported predicates in module m. For each exported predicate we use *Post* from the success assertion conditions as a safe over-approximation of the constraints that can be introduced during the execution of the predicate. We compute the union ($\sqcup$, which is equivalent to $\vee$ but it

Algorithm 5.1 ESCAPING_TERMS

```
1:  function ESCAPING_TERMS(M)
2:      Def := usr(X)
3:      for all L exported from M do
4:          for all success(L, _, Post) ∈ A_C⟨L⟩ do
5:              for all P ∈ LITNAMES(Post, vars(L)) do
6:                  Def := Def ⊔ P(X)
7:      for all L imported from M do
8:          for all calls(L, Pre) ∈ A_C⟨L⟩ do
9:              for all P ∈ LITNAMES(Pre, vars(L)) do
10:                 Def := Def ⊔ P(X)
11:     return (esc_m(X) ← Def)
12: function LITNAMES(G, Args)
13:     return set of P such that A ∈ Args and G = (... ∧ P(A) ∧ ...)
```

can sometimes simplify the representation) of all properties that restrict any variable argument in *Post*. The second part of the algorithm performs the same operation on all the properties specified in the *Pre* of the calls assertions conditions. This is a safe approximation of the constraints that can be *leaked* to other modules called from m.

Note that the algorithm can use analysis information to detect more precise calls to the imported predicates, as well as more precise successes of the exported predicates, than those specified in the assertion conditions present in the program.

Lemma 5.4 (Correctness of ESCAPING_TERMS)**.** *The* ESCAPING_TERMS *algorithm computes a safe (over)approximation to* $\mathsf{esc}_m(X)$ *(when using the operational semantics with assertions).*

Proof. Let $Q(X) =$ ESCAPING_TERMS(m), we will show that $Q(X)$ over-approximates $\mathsf{esc}_m(X)$. Since $\mathsf{esc}_m(X)$ is equivalent to $\mathsf{esc}_{m'}(X)$ (Lemma 5.3), it is enough to consider the derivation steps at the boundaries. That is, $S_1 \rightsquigarrow_{\mathsf{rtc}} S_2$ where $S_1 = \langle (L_1, m) :: \_ \mid \_ \rangle$ and $S_2 = \langle (L_2, n) :: \_ \mid \theta \rangle$ with $n \neq m$. If L_1 is a literal (not $\mathsf{ret}(\_)$) then it corresponds to the case of calling an imported predicate. The operational semantics ensures that $\theta \Rrightarrow Pre$ and thus $Q(X)$ over-approximates this case. If L_2 is $\mathsf{ret}(\_)$ then it corresponds to the case of returning from m. The operational semantics ensures that $\theta \Rrightarrow Post$ and thus $Q(X)$ also over-approximates this case. □

SHALLOW PROPERTIES Shallow run-time checking consists in using *shallow* versions of properties in the run-time checks for the calls across module boundaries. While this notion could be added directly to the operational semantics, we will present it as a program transformation based on the generation of shallow versions of the properties, since this also provides a direct implementation path.

Algorithm 5.2 SHALLOW_INTERFACE

```
1: function SHALLOW_INTERFACE(M)
2:     Let M' be M with wrappers for exported predicates
3:       (to differentiate internal from external calls)
4:     Let Q(X) := ESCAPING_TERMS(M')
5:     for all L exported from M do
6:         for all calls(L, Pre) ∈ A_C⟨L⟩ do
7:             Update A_C⟨L⟩ with calls(L, Pre#)
8:         for all success(L, Pre, Post) ∈ A_C⟨L⟩ do
9:             Update A_C⟨L⟩ with success(L, Pre#, Post)
10:    return M'
```

Example 5.3. *Assume that the set of escaping terms of m contains* `point(1)` *and it does not contain the more general* `point(_)`*. Consider the property:*

```
intpoint(point(X)) :- int(X).
```

Checking `intpoint(A)` *at any program point outside m must check first that* `A` *is instantiated to* `point(X)` *and that* `X` *is instantiated to an integer (*`int(X)`*). However, the escaping terms show that it is not possible for a variable to be bound to* `point(X)` *without* `X=1`*. Thus, the latter check is redundant. We can compute the optimized – or* shallow *– version of* `intpoint/1` *in the context of all execution points external to m as* `intpoint(point(_))`*.*

Let SPEC(L, Pre) generate a specialized version L' of predicate L w.r.t. the calls given by Pre (see [81]). It holds that for all θ, $\theta \Rightarrow L$ iff $\theta \wedge Pre \Rightarrow L'$.

Definition 5.9 (Shallow Property)**.** *The shallow version of a property $L(X)$ w.r.t. module m is denoted as $L(X)^{\#}$, and computed as* SPEC$(L(X), Q(X))$*, where $Q(X)$ is a (safe) approximation of the escaping terms of m (*ESCAPING_TERMS(m)*).*

Algorithm 5.2 computes the optimized version of a module interface using shallow checks. It first introduces wrappers for the exported predicates, i.e., predicates `p(X) :- p'(X)`, renaming all internal occurrences of `p` by `p'`. Then it computes an approximation $Q(X)$ of the escaping terms of M. Finally, it updates all Pre in calls and success assertion conditions, for all exported predicates, with their shallow version $Pre^{\#}$. We compute the shallow version of a conjunction of literals $Pre = \bigwedge_i L_i$ as $Pre^{\#} = \bigwedge_i L_i^{\#}$.

Theorem 5.5 (Correctness of SHALLOW_INTERFACE)**.** *Replacing a module m in a larger program by its shallow version does not alter the (run-time checking) operational semantics.*

Proof. By definition, the transformation only affects the checks for $Pre = (\bigwedge_i L_i(X_i))$ conjunctions in assertion conditions of exported predicates in m. These checks correspond to the derivation steps

$S_1 \rightsquigarrow_{\mathsf{rtc}} S_2$ where $S_1 = \langle (\_, n) :: G \mid \theta \rangle$ and $S_2 = \langle (\_, m) :: G \mid \_ \rangle$ with $n \neq m$. Let $Q(X)$ be obtained from ESCAPING_TERMS(m). The shallow version $Pre^{\#} = (\bigwedge_i L_i(X_i))^{\#} = (\bigwedge_i \textsc{Spec}(L_i(X_i), Q(X_i)))$ (Definition 5.9). By Definition 5.8 it holds that $\theta \Rightarrow (\bigwedge_i \mathsf{esc}_m(X_i))$. By Lemma 5.4 it holds that $\theta \Rightarrow (\bigwedge_i Q(X_i))$. By correctness of SPEC, since θ entails each $Q(X_i)$, then the full and specialized versions of L_i can be interchanged. □

DISCUSSION ABOUT PRECISION The presence of any *top* properties in the calls or success assertion conditions will propagate to the end in the ESCAPING_TERMS algorithm (see Algorithm 5.1). For a significant class of programs, this is not a problem as long as we can provide or infer precise assertions which do not use this top element. Note that $\mathsf{usr}(X)$, since it has a void intersection with any hidden term, does not represent a problem. For example, many generic Prolog term manipulation predicates (e.g., `functor/3`) typically accept a *top* element in their calls conditions. We restrict these predicates to work only on user (i.e., not hidden) symbols.[4] More sophisticated solutions, that are outside the scope of this dissertation, include: producing monolithic libraries (creating versions of the imported modules and using abstract interpretation to obtain more precise assertion conditions); or disabling shallow checking (e.g., with a dynamic flag) until the execution exits the context of m (which is correct except for the case when terms are dynamically asserted).

MULTI-LIBRARY SCENARIOS Recall that properties can be exported and used in assertions from other modules. The shallow version of properties in m are safe to be used not only at the module boundaries but also in any other assertion check outside m. Computing the shallow optimization can be performed per-library, without strictly requiring intermodular analysis. However, in some cases intermodular analysis may improve the precision of escaping terms and allow more aggressive optimizations.

5.4 EXPERIMENTAL EVALUATION

We explore the effectiveness of the combination of term hiding and shallow checking in the reusable library context, i.e., in libraries that use (some) hidden terms in their data structures and offer an interface for clients to access/manipulate such terms. We study the four assertion checking modes of [102]: *Unsafe* (no library assertions are checked), *Client-Safe* (checks are generated only for the assertions of the predicates exported by the library, assertions for the internal library predicates are not checked), *Safe-RT* (checks are generated from

4 This can be implemented very efficiently with a simple bit check on the atom properties and does not impact the execution.

Table 5.1: Benchmark metrics.

Name	LOC	Size (KB)	`pred` Assertions	# Hidden Symbols
`AVL-tree`	147	16.7	20	2
`B-tree`	240	22.1	18	3
`Binary tree`	58	8.3	6	2
`Heap`	139	15.1	12	3
`RB-tree`	678	121.8	20	4

assertions both for internal and exported library predicates), and *Safe-CT+RT* (like *RT*, but analysis information is used to clear as many checks as possible at compile-time). We use the lightweight instrumentation scheme from [101] for generating the run-time checks from the program assertions. For eliminating the run-time checks via static analysis we reuse the Ciao verification framework, including the extensions from [102]. We concentrate in these experiments on shape analysis (regular types).

In our experiments each benchmark is composed of a library and a client/driver. We have selected a set of Prolog libraries that implement tree-based data structures. Libraries `B-tree` and `binary tree` were taken from the Ciao sources; libraries `AVL-tree`, `RB-tree`, and `heap` were adapted from YAP, adding similar assertions to those of the Ciao libraries. Table 5.1 shows some statistics for these libraries: number of lines of code (LOC), size of the object file (Size KB), the number of assertions in the library specification considered (Pred Assertions), and the number of hidden functors per library (# Hidden Symbols).

In order to focus on the assertions of the library operations used in the benchmarks (where by an operation we mean the set of predicates implementing it) we do not count in the tables the assertions for library predicates not directly involved in those operations. Library assertions contain instantiation (moded) regular types. For each library we have created two drivers (clients) resulting in two experiments per library. In the first one the library operation has constant ($O(1)$) time complexity and the respective run-time check has $O(N)$ time complexity (e.g., looking up the value stored at the root of a binary tree and checking on each lookup that the input term is a binary tree). Here a major speedup is expected when using *shallow* run-time checks, since the checking time dominates operation execution time and the reduction due to shallow checking should be more noticeable. In the second one the library operation has non-constant ($O(log(N))$) complexity and the respective run-time check $O(N)$ complexity (e.g., inserting an element in a binary tree and checking on each insertion that the input term is a tree). Here obviously a smaller speedup is

Table 5.2: Static analysis time for benchmarks for the *Safe-CT+RT* mode.

Benchmark	Analysis time, ms				
	prep	`shfr`	prep	`eterms`	total
`AVL-tree`	2	10	2	31	45 (2%)
`B-tree`	3	9	3	38	53 (2%)
`Binary tree`	1	9	1	14	25 (2%)
`Heap`	2	7	2	24	35 (2%)
`RB-tree`	13	11	14	35	73 (3%)

Table 5.3: Static checking time for benchmarks for the *Safe-CT+RT* mode.

Benchmark	Assertions	
	checking, ms	unchecked
`AVL-tree`	59 (2%)	2/20
`B-tree`	90 (3%)	3/18
`Binary tree`	33 (2%)	2/6
`Heap`	71 (4%)	2/12
`RB-tree`	298 (10%)	3/20

to be expected with *shallow* checking. All experiments were run on a MacBook Pro, 2.6 GHz Intel Core i5 processor, 8GB RAM, and under the Mac OS X 10.12.3 operating system.

STATIC ANALYSIS Tables 5.2 and 5.3 present the detailed compile-time analysis and checking times for the *Safe-CT+RT* mode. Numbers in parentheses indicate the percentage of the total compilation time spent on analysis, which stays reasonably low even in the most complicated case (13% for the `RB-tree` library). Nevertheless, the analysis was able to discharge most of the assertions in our benchmarks, leaving always only 2-3 assertions unchecked (i.e., that will need run-time checks), for the predicates of the library operations being benchmarked.

RUN-TIME CHECKING After the static preprocessing phase we have divided our libraries into two groups: (a) libraries where the only unchecked assertions left are the ones for the boundary calls (`AVL-tree`, `heap`, and `binary tree`),[5] and (b) libraries with also some unchecked assertions for internal calls (`B-tree` and `RB-tree`). We present run time

5 Due to our reusable library scenario the analysis of the libraries is performed without any knowledge of the client and thus the library interface checks must always remain.

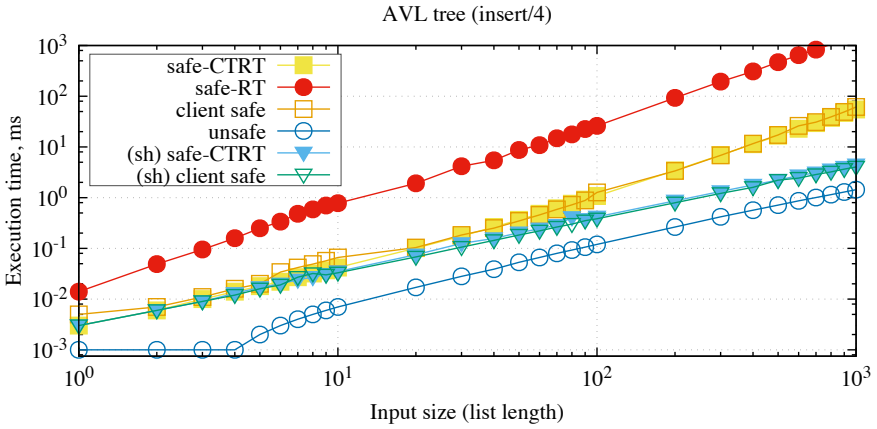


Figure 5.2: Run times in different checking modes, `AVL-tree` library, $O(log(N))$ operation.

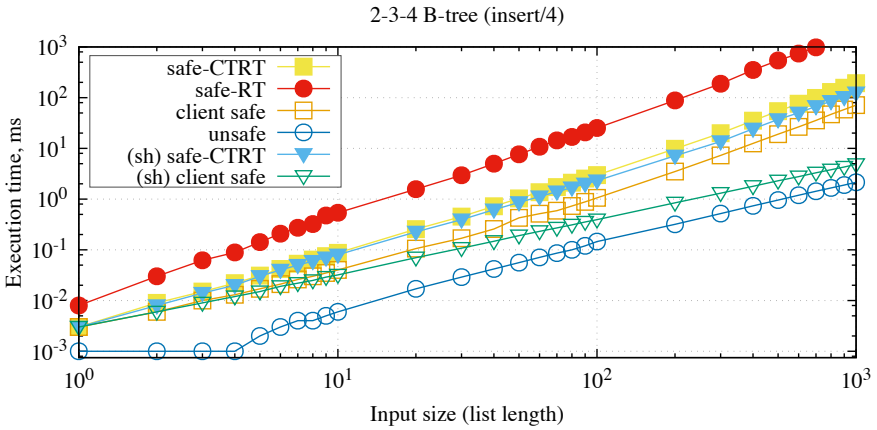


Figure 5.3: Run times in different checking modes, `B-tree` library, $O(log(N))$ operation.

plots[6] for one library of each group. Since the unchecked assertions in the second group correspond to internal calls of the $O(log(N))$ operation experiment, we only show here a set of plots of the $O(1)$ operation experiment for one library, as these plots are very similar across all benchmarks.

Fig. 5.2 illustrates the overhead reductions from using the shallow run-time checks in the `AVL-tree` benchmark for the $O(N)$ *insert* operation experiment. This is also the best case that can be achieved for this kind of operations, since in the *Safe-CT+RT* mode all inner assertions are discharged statically. Fig. 5.3 shows the overhead reductions from using the shallow checks in the `B-tree` benchmark for the $O(log(N))$ *insert* operation experiment. In contrast with the previous case, here the overhead reductions achieved by employing shallow checks are dominated by the total check cost, and while the overhead

6 The current measurements depend on the C `getrusage()` function, that on Mac OS has microsecond resolution.

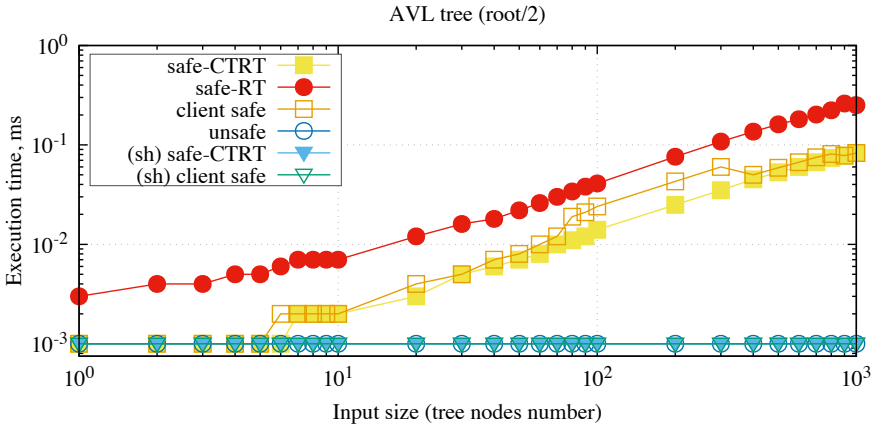


Figure 5.4: Run times in different checking modes, `AVL-tree` library, $O(1)$ operation.

reduction is obvious in the *Client-Safe* mode, it is not significant in the *Safe-CT+RT* mode where some internal assertion was being checked.

Fig. 5.4 presents the overhead reductions in run-time checking resulting from the use of the shallow checks in the `AVL-tree` benchmark for the $O(1)$ *peek* operation experiment on the root. As we can see, using shallow checks allows us to obtain constant overhead on the boundary checks for such cheap operations in all execution modes but *Safe-RT*. In summary, the shallow checking technique seems quite effective in reducing the shape-related run-time checking overheads for the reusable-library scenario.

More plots are available in the Appendix B.2.

5.5 CONCLUSIONS

The topic of modules and logic programming has received considerable attention, dating back to [116, 19, 63] and resulting in standardization attempts for ISO-Prolog [46]. Currently, most mature Prolog implementations adopt some flavor of a module system, *predicate-based* in SWI [118], SICStus [17], YAP [90], and ECLiPSe [92], and *atom-based* in XSB [106]. As mentioned before, Ciao [40, 15] uses a hybrid approach, which behaves by default as in predicate-based systems but with the possibility of marking a selected set of symbols as hidden, making it essentially compatible with that of XSB. Some previous research in the comparative advantages of atom-based module systems can be found in [38].

While traditionally Prolog is untyped, there have been some proposals for integrating it with type systems, starting with [69]. Several strongly-typed Prolog-based systems have been proposed, notable examples being Mercury [97], Gödel [44], and Visual Prolog [80]. An approach for combining typed and untyped Prolog modules has been proposed in [93].

In this work we have described a lightweight modification of a predicate-based module system to support term hiding and explored the optimizations that can be achieved with this technique in the context of combined compile-time/run-time verification. We have studied the challenging case of reusable libraries, i.e., library modules that are pre-compiled independently of the client. We have shown that with our approach the shape information that can be inferred can be enriched significantly and large reductions in overhead can be achieved. The overheads achieved are closer to those of static languages, even in the reusable-library context, without requiring switching to strong typing, which is less natural in Prolog-style languages, where there is a difference between error and failure/backtracking. We have shown herein that even in the challenging context of calls across open module boundaries it is sometimes possible to achieve constant run-time overhead.

6 HIGHER-ORDER EXTENSIONS

Many high-level programming languages support higher-order programming style which adds flexibility to the software development process. Within the (C)LP paradigm, Prolog has included higher-order constructs since the early days, and there have been many proposals for combining the first-order kernel of (C)LP with different higher-order constructs (see, e.g., [115, 71, 20, 70, 16, 14]). Many of these proposals are currently in use in different (C)LP systems and have been found very useful in programming practice, inheriting the well-known benefits of code reuse (templates), elegance, clarity, and modularization.

When higher-order constructs are introduced in the language it becomes necessary to describe properties of arguments of predicates that are themselves also predicates. While the combination of contracts and higher-order has received some attention in functional programming [32, 30], within (C)LP the combination of higher-order with the previously mentioned assertion-based approaches has received comparatively little attention to date. Current Prolog systems simply use basic atomic types (i.e., stating simply that the argument is a `pred`, `callable`, etc.) to describe predicate-bearing variables. The approach of [4] is oriented to meta programming. It allows describing meta-types but there is no notion of directionality (modes), and only a single pattern is allowed per predicate. Although this approach looks promising in the line of reasoning about higher–order calls, there are many limitations that make it undesirable for our purposes.

This chapter contributes to filling the existing gap between higher-order programs and assertions in (C)LP for describing them. Our starting point is the Ciao assertion model, since, as mentioned before, it has been adopted at least in part in a number of the most popular (C)LP systems. We start by extending the traditional notion of programs and derivations in order to deal with higher-order calls (Section 6.1). This part allows us to revisit the traditional model in this new, higher-order context, while introducing a different formalization than the original one of [85]. This formalization, which will be used throughout the chapter, is more compact and gathers all assertion violations as opposed to just the first one, among other differences. We then define an extension of the properties used in assertions and of the assertions themselves to higher-order, and provide corresponding semantics and results (Section 6.2).

6.1 FIRST-ORDER ASSERTIONS ON HIGHER-ORDER DERIVATIONS

We start by extending the traditional notion of programs and derivations to include higher-order calls. We also recall the notions of first-order conditional literals, assertions, program correctness, and runtime checking and adapt them to the case of higher-order derivations.

While this adaptation is not complex, this part allows us to revisit the traditional model in this new, higher-order context, while proposing a richer formalization than that of the Chapter 2. This new formalization, which will be used throughout the chapter, gathers all assertion violations as opposed to just the first one, among other differences.

HIGHER-ORDER PROGRAMS AND DERIVATIONS We start by extending the definition of program, state reduction, and derivations in order to deal with the syntax and semantics of higher-order calls.[1]

Definition 6.1 (Higher-order Programs)**.** Higher-order programs *are a generalization of* constraint logic programs *where:*

- *The set of literals* LS *is extended to include* higher-order literals $X(t_1, \ldots, t_n)$*, where* $X \in \mathsf{VS}$ *and the* $t_i \in \mathsf{TS}$*.*
- *The set of terms* TS *is extended so that* $\mathsf{PS} \subset \mathsf{TS}$ *(i.e., predicate symbols* p *can be used as constants).*

In the following we assume a simple semantics where when a call to a higher-order literal $X(t_1, \ldots, t_n)$ occurs, X has to be constrained to a predicate symbol in the store:[2]

Definition 6.2 (Reductions in Higher-order Programs)**.** *A state* $S = \langle L :: G \mid \theta \rangle$ *where* L *is a literal can be* reduced *to a state* S'*, denoted* $S \leadsto S'$*, as follows:*

1. *If* L *is a constraint and* $\theta \wedge L$ *is satisfiable, then* $S' = \langle G \mid \theta \wedge L \rangle$*.*
2. *If* L *is an atom of the form* $p(t_1, \ldots, t_n)$*, for some clause* $(L \leftarrow B) \in \mathsf{cls}(L)$*, then* $S' = \langle B :: G \mid \theta \rangle$*.*
3. *If* L *is of the form* $X(t_1, \ldots, t_n)$*, then* $S' = \langle G' \mid \theta \rangle$ *where*

$$G' = \begin{cases} p(t_1, \ldots, t_n) :: G & \text{if } \exists p \in \mathsf{PS} \wedge \theta \models (X = p) \wedge \mathsf{ar}(p) = n \\ \epsilon_{uninst\_call} & \text{otherwise} \end{cases}$$

1 While the higher-order programs considered can also be reduced to first order via a defunctionalization transformation (see, e.g., [115]) we prefer herein to treat higher order natively. This is in line with current Prolog implementations which provide syntax and direct implementation support (e.g., call/n etc.) for higher order. Also, the transformation-based approach requires a static pre-processing which would not work in general for modular programs since the number of predicates would be unknown a priori.

2 This is also the most frequent semantics in current systems. Other alternatives, such as residuation [1] (delays), predicate enumeration, etc. can also be used, requiring relatively straightforward adaptations of the model proposed.

The concepts of answers and of finished and successful derivations carry over without change to this notion of higher-order derivations. The notion of (finitely) failed derivation is extended as follows:

Definition 6.3 ((Finitely) Failed Derivation)**.** *A finished derivation from a query* (L, θ) *is* failed *iff its last state is not of the form* $\langle \square \mid \theta' \rangle$ *or* $\langle \epsilon_{uninst\_call} \mid \theta \rangle$.

Finally, we introduce the concept of *floundered* derivations:

Definition 6.4 (Floundered Derivation)**.** *A finished derivation from a query* (L, θ) *is* floundered *iff its last state is of the form* $\langle \epsilon_{uninst\_call} \mid \theta \rangle$.

FIRST-ORDER ASSERTIONS ON HIGHER-ORDER DERIVATIONS In order to keep track of any violated assertion conditions, we introduce an extended program state of the form $\langle G \mid \theta \mid \mathcal{E} \rangle$, where $\mathcal{E}$ denotes the set of identifiers for falsified assertion condition instances. For the sake of readability, we write labels in *negated* form when they appear in the error set.

The definitions below adapt the base first-order definitions from the Chapter 2 to the notation with extended program states.

Definition 6.5 (Reductions in Higher-order Programs with First-order Assertions)**.** *A state* $S = \langle L :: G \mid \theta \mid \mathcal{E} \rangle$*, where* L *is a literal can be* reduced *to a state* S'*, denoted* $S \rightarrowtriangle_{\mathcal{A}} S'$*, as follows:*

1. *If* L *is a constraint and* $\theta \wedge L$ *is satisfiable, then* $S' = \langle G \mid \theta \wedge L \mid \mathcal{E} \rangle$.
2. *If* L *is of the form* $X(t_1, \ldots, t_n)$*, then* $S' = \langle G' \mid \theta \mid \mathcal{E} \rangle$ *where*

$$G' = \begin{cases} p(t_1, \ldots, t_n) :: G & \text{if } \exists p \in \mathit{PS} \wedge \theta \models (X = p) \wedge \mathit{ar}(p) = n \\ \epsilon_{uninst\_call} & \text{otherwise} \end{cases}$$

3. *If* L *is an atom and* $\exists (L \leftarrow B) \in \mathsf{cls}(L)$*, then the new state* $S' = \langle B :: \mathit{PostC} :: G \mid \theta \mid \mathcal{E}' \rangle$ *where:*

$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{\bar{c}\} & \text{if } \exists\, C = \mathit{calls}(L, \mathit{Pre}) \in \mathcal{A}_C\langle L \rangle \text{ s.t.} \\ & \quad \mathit{id}(C) = c \wedge \theta \not\Rrightarrow \mathit{Pre} \\ \mathcal{E} & \text{otherwise} \end{cases}$$

   and PostC is the sequence $\mathit{check}(c_1) :: \ldots :: \mathit{check}(c_n)$ *including all the checks* $\mathit{check}(c_i)$ *such that* $C_i = \mathit{success}(L, \mathit{Pre}_i, \mathit{Post}_i) \in \mathcal{A}_C\langle L \rangle \wedge \mathit{id}(C_i) = c_i \wedge \theta \Rrightarrow \mathit{Pre}_i$.

4. *If* L *is a check literal* $\mathit{check}(c)$*, then* $S' = \langle G \mid \theta \mid \mathcal{E}' \rangle$ *where:*

$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{\bar{c}\} & \text{if } C = \mathit{success}(L, \_, \mathit{Post}) \in \mathcal{A}_C\langle L \rangle \text{ s.t.} \\ & \quad \mathit{id}(C) = c \wedge \theta \not\Rrightarrow \mathit{Post} \\ \mathcal{E} & \text{otherwise} \end{cases}$$

Note that the order in which the PostC check *literals are selected is irrelevant.*

As before, $\looparrowright^*_{\mathcal{A}}$ denotes a series of consequent program state reductions.

To address the changes introduced to the operational semantics by extending the program state, we adapt several definitions and proofs from the Chapter 2, as they will be instrumental later on in this chapter.

Definition 6.6 (Assertion Conditions Renaming)**.** *Given the atoms L and L_a s.t. $L_a = \sigma(L)$, and the set of assertion conditions $\mathcal{A}_C\langle L\rangle$, the set of assertion conditions $\mathcal{A}_C\langle L_a\rangle$ can be obtained from $\mathcal{A}_C\langle L\rangle$, such that: $\exists C \in \mathcal{A}_C$, $C = \mathit{calls}(L, Pre)$ (or $C = \mathit{success}(L, Pre, Post)$), $C_a = \mathit{calls}(L_a, \sigma(Pre))$ (or $C_a = \mathit{success}(L_a, \sigma(Pre), \sigma(Post))$).*

Definition 6.7 (Error-erased Higher-order Derivation)**.** *The set of* error-erased *derivations from $\looparrowright_{\mathcal{A}}$ is obtained by a syntactic rewriting $(-)^\diamond$ that removes states that begin by a check literal, check literals from goals, and the error set. It is recursively defined as follows:*

$$\{D_1, \ldots, D_n\}^\diamond = \{D_1^\diamond, \ldots, D_n^\diamond\}$$

$$(S_1, \ldots, S_m, S_{m+1})^\diamond = \begin{cases} (S_1, \ldots, S_m)^\diamond \\ \quad \textit{if } S_{m+1} = \langle \mathit{check}(\_) :: \_ \mid \_ \mid \_ \rangle \\ (S_1, \ldots, S_m)^\diamond \parallel ((S_{m+1})^\diamond) \\ \quad \textit{otherwise} \end{cases}$$

$$\langle G \mid \theta \mid \mathcal{E} \rangle^\diamond = \langle G^\diamond \mid \theta \rangle$$

$$(L :: G)^\diamond = \begin{cases} G^\diamond & \textit{if } L = \mathit{check}(\_) \\ L :: (G^\diamond) & \textit{otherwise} \end{cases}$$

$$\square^\diamond = \square$$

where $\parallel$ stands for sequence concatenation.

We provide below the proof of the Theorem 2.1 but adapted to the semantics with extended program states of the Definition 6.5:

Proof of Theorem 2.1 (for higher-order derivations). We will prove $\mathcal{D} = (\mathcal{D}')^\diamond$ by showing that $\mathcal{D} \subseteq (\mathcal{D}')^\diamond$ and $\mathcal{D} \supseteq (\mathcal{D}')^\diamond$.

- ($\subseteq$) For all $D \in \mathcal{D}$ exists $D' \in \mathcal{D}'$ so that $D = (D')^\diamond$.
- ($\supseteq$) For all $D' \in \mathcal{D}'$, $D = (D')^\diamond \in \mathcal{D}$.

We will prove each case:

- ($\subseteq$) Let $D = (S_1, \ldots, S_n)$, $S_i = \langle L_i \mid \theta_i \rangle$, for some $Q = (L_1, \theta_1) \in \mathcal{Q}$ and $S_i \looparrowright S_{i+1}$. Proof by induction on the length n of D:

- Base case ($n = 1$). Let $S'_1 = \langle L_1 \mid \theta_1 \mid \emptyset \rangle$. It holds that $(S'_1)^\diamond = \langle L_1 \mid \theta_1 \mid \emptyset \rangle)^\diamond = \langle L_1^\diamond \mid \theta_1 \rangle = \langle L_1 \mid \theta_1 \rangle = S_1$ (since L_1 does not contain any check literal). Thus, $(D')^\diamond = ((S'_1))^\diamond = ((S'_1)^\diamond) = (S_1) = D$.
- Inductive case (show $n+1$ assuming n holds). For each $D_2 = (S_1, \ldots, S_n, S_{n+1})$ there exists $D'_2 = (S'_1, \ldots, S'_m, S'_{m+1})$ such that $(D'_2)^\diamond = D_2$. Given the induction hypothesis it is enough to show that for each $S_n \curvearrowright S_{n+1}$ there exists $S'_m \curvearrowright_{\mathcal{A}} S'_{m+1}$, such that $(S'_{m+1})^\diamond = S_{n+1}$. According to $\curvearrowright_{\mathcal{A}}$ (see Def. 6.5), L'_{m+1} and θ'_{m+1} are obtained in the same way than in $\curvearrowright$ (see Def. 6.2), except for the introduction of check literals. Since all check literals are removed in error-erased states, it follows that $(S'_{m+1})^\diamond = S_{n+1}$. □

- ($\supseteq$) Let $D' = (S'_1, \ldots, S'_m)$, $S'_i = \langle L'_i \mid \theta'_i \mid \mathcal{E}_i \rangle$, for some $Q = (L'_1, \theta'_1) \in \mathcal{Q}$ and $S'_i \curvearrowright_{\mathcal{A}} S'_{i+1}$. Proof by induction on the length m of D':
  - Base case ($m = 1$). It holds that $(S'_1)^\diamond = S_1$ (showed in base case for $\subseteq$). Then $(D')^\diamond = D \in \mathcal{D}$.
  - Inductive case (show $m+1$ assuming m holds). We want to show that given $D'_2 = (S'_1, \ldots, S'_m, S'_{m+1})$, $(D'_2)^\diamond = D_2 \in \mathcal{D}$. Given the induction hypothesis it is enough to show that for each $S'_m \curvearrowright_{\mathcal{A}} S'_{m+1}$ there exists $S_n \curvearrowright S_{n+1}$ such that $S_{n+1} = (S'_{m+1})^\diamond$ (so that $(S_1, \ldots, S_n, S_{n+1}) \in \mathcal{D}$) or $S_n = (S'_{m+1})^\diamond$ ($D_2 = D \in \mathcal{D}$). According to cases of Def. 6.5:
    - If L'_m begins with a check literal then $(L'_{m+1})^\diamond = (L'_m)^\diamond$. Thus $(S'_{m+1})^\diamond = (S'_m)^\diamond = S_n$.
    - Otherwise, it holds that $(S'_{m+1})^\diamond = S_{n+1}$ using the same reasoning than in the inductive case for $\subseteq$.

□

Definition 6.8 (Run-time Valuations of an Assertion Condition on a Derivation (for higher-order derivations))**.** *Let $\mathcal{E}(D)$ denote the error set of the last state of derivation D, $D_{[-1]} = \langle \_ \mid \_ \mid \mathcal{E} \rangle$. The run-time valuation of an assertion condition C on a derivation D is given by:*

$$\mathit{rtsolve}(C, D) \triangleq \forall c, C', \sigma, L\ (C' \in \mathcal{A}_C\langle L \rangle \wedge \mathit{id}(C') = c \wedge \sigma(C) = C') \Rightarrow \mathcal{E}(D) \nvdash \bar{c}$$

Proof of Theorem 2.2 (for higher-order derivations): Let us assume assertion condition $A \in \mathcal{A}_C$ is false $\Leftrightarrow$ from Def. 2.9 and Def. 2.5 $\exists\{C_c, C_s\}$ assertion conditions s.t. $\mathsf{false}(C_c) \vee \mathsf{false}(C_s)$, where $C_c = \mathsf{calls}(L, Pre)$ and $C_s = \mathsf{success}(L, Pre, Post)$ correspond to A. Let us first prove the $\neg\mathsf{rtsolve}(C_c, D)$ case and then the $\neg\mathsf{rtsolve}(C_s, D)$ one:

- $\mathsf{false}(C_c)$
  $\Leftrightarrow$ from Def. 2.8 $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\neg\mathsf{solve}(C_c, D)$
  $\Leftrightarrow$ from Def. 2.6 $(\mathsf{prestep}(L, D) = (\theta, \sigma) \wedge \theta \not\Rightarrow \sigma(Pre))$
  $\Leftrightarrow$ from Def. 6.5 $\exists\, S \not\hookrightarrow_{\mathcal{A}} S'$ where:

$$
\begin{aligned}
S =& \langle L :: G \mid \theta \mid \mathcal{E} \rangle \\
& \text{s.t. } \exists\, C = \mathsf{calls}(L, Pre) \in \mathcal{A}_C\langle L \rangle \wedge \mathsf{id}(C) = c \\
S' =& \langle \_ \mid \theta \mid \mathcal{E}' \rangle \\
& \text{s.t. } \mathcal{E}' = \mathcal{E} \cup \{\bar{c}\}
\end{aligned}
$$

  $\Leftrightarrow$ from Def. 6.8 $\neg\mathsf{rtsolve}(C_c, D)$ □

- $\mathsf{false}(C_s)$
  $\Leftrightarrow$ from Def. 2.8 $\exists D \in \mathsf{derivs}(\mathcal{Q})$ s.t. $\neg\mathsf{solve}(C_s, D)$
  $\Leftrightarrow$ from Def. 2.6 $(\mathsf{step}(L, D) = (\theta, \sigma, \theta') \wedge \theta \Rightarrow \sigma(Pre) \wedge$
  $\theta' \not\Rightarrow \sigma(Post))$
  $\Leftrightarrow$ from Def. 6.5 $\exists\, S \not\hookrightarrow^*_{\mathcal{A}} S' \not\hookrightarrow_{\mathcal{A}} S''$ where

$$
\begin{aligned}
S =& \langle L :: G \mid \theta \mid \_ \rangle \wedge \\
& \exists\, C = \mathsf{success}(L, Pre, Post) \in \mathcal{A}_C\langle L \rangle \\
& \wedge\, \mathsf{id}(C) = c \wedge \theta \Rightarrow Pre \\
S' =& \langle \mathsf{check}(c) :: G \mid \theta' \mid \mathcal{E}' \rangle \wedge \theta' \not\Rightarrow Post \\
S'' =& \langle \_ \mid \_ \mid \mathcal{E}'' \rangle \wedge \mathcal{E}'' = \mathcal{E}' \cup \{\bar{c}\}
\end{aligned}
$$

  $\Leftrightarrow$ from Def. 6.8 $\neg\mathsf{rtsolve}(C_s, D)$

□

6.2 HIGHER-ORDER ASSERTIONS ON HIGHER-ORDER DERIVATIONS

Once we have established basic results for the case of first-order assertions in the context of higher-order derivations, we extend the notion of assertion itself to the higher-order case. The motivation is that in the higher-order context terms can be bound to predicates and our aim is to also be able to state and check properties of such predicates.

ANONYMOUS ASSERTIONS In the higher-order case terms can be bound to predicate names. In this context it is convenient to be able to describe the properties that such predicates must meet. To this end properties of terms that may be bound to predicates but where the predicate name may not be known statically in the code.

We start by generalizing the notion of assertion to include *anonymous assertions*: assertions where the predicate symbol is a variable from VS, which can be instantiated to any suitable predicate symbol from PS to produce non-anonymous assertions. An anonymous assertion is an expression of the from "`:- pred` L `:` Pre `=>` $Post$", where L is of the form $X(V_1, \ldots, V_n)$ and Pre and $Post$ are DNF formulas of *prop* literals.

Example 6.1. *The anonymous assertion:*

```
:- pred X(A,B) : list(A) => list(B).
```

states that for any predicate $p \in P$ that X is constrained to be of arity 2, it should be called with its first argument instantiated to a list, and if it succeeds, then its second argument should be also a list on success.

We now introduce *predprops*, which gather a number of anonymous assertions in order to fully describe variables containing higher-order terms (predicate symbols), similarly to how *prop* literals describe conditions for variables containing first-order terms.

Definition 6.9 (Predprop)**.** *Given Pre_i and $Post_i$ conjunctions of* prop *literals, a* predprop *$pp(X)$ is an expression of the form:*

$$\begin{aligned} &\texttt{pp(X)\{}\ \ \texttt{:- pred}\ X(V_1,\ldots,V_m)\ \texttt{:}\ Pre_1\ \texttt{=>}\ Post_1.\\ &\qquad\ldots\\ &\qquad\texttt{:- pred}\ X(V_1,\ldots,V_m)\ \texttt{:}\ Pre_n\ \texttt{=>}\ Post_n.\ \texttt{\}} \end{aligned}$$

Definition 6.10 (Anonymous Assertion Conditions for a *predprop*)**.** *The corresponding set of* anonymous assertion conditions *for the predprop $pp(X)$ is defined as $\mathcal{A}_H\langle pp\rangle\langle X\rangle = \{H_i\langle X\rangle \mid i = 0..n\}$ where:*

$$H_i\langle X\rangle = \begin{cases} \mathit{calls}(X(V_1,\ldots,V_m), Pre) & i = 0 \\ \mathit{success}(X(V_1,\ldots,V_m), Pre_i, Post_i) & i = 1..n \end{cases}$$

The variable X can be instantiated to a particular predicate symbol $q \in PS$ to produce a set of non-anonymous assertion conditions $\mathcal{A}_C\langle pp\rangle\langle q\rangle$ for q (see Definition 6.12 for one possibility).

Example 6.2. *Consider defining a* `comparator(Cmp)` predprop *that describes predicates of arity 3 which can be used to compare numerical values:*

```
:- prop comparator(Cmp) {
:- pred Cmp(X,Y,R) : int(X),int(Y) => between(-1,1,R).
:- pred Cmp(X,Y,R) : flt(X),flt(Y) => between(-1,1,R).
}.
```

The `comparator(Cmp)` predprop *includes two anonymous assertions describing a set of possible preconditions and postconditions for predicates of this kind. In this example:*

$$\begin{aligned} &\mathcal{A}_H\langle comparator\rangle\langle Cmp\rangle = \{\\ &\ \mathit{calls}(Cmp(X,Y,Res), (int(X) \wedge int(Y)) \vee (flt(X) \wedge flt(Y))),\\ &\ \mathit{success}(Cmp(X,Y,Res), int(X) \wedge int(Y), between(-1,1,Res))\\ &\ \mathit{success}(Cmp(X,Y,Res), flt(X) \wedge flt(Y), between(-1,1,Res))\\ &\} \end{aligned}$$

```
1  :- prop nneg(P)  {:- pred P(X) : true => nnegint(X).}
2  :- prop  neg(P)  {:- pred P(X) : true =>  negint(X).}
3
4  :- pred test_c(P,N) : nneg(P) => true.
5  :- pred test_c(P,N) :  neg(P) => true.
6
7  test_c(P,N) :- P(N).
8
9  :- pred test_s(N,P) : nnegint(N) => nneg(P).
10 :- pred test_s(N,P) :  negint(N) =>  neg(P).
11
12 test_s( 1,P) :- P = z. % bug here, should be P = p
13 test_s(-1,P) :- P = n.
14
15 z(1). z(-2).  p(1). p(2).  n(-1). n(-2).  c(a). c(b).
```

Figure 6.1: Sample program with predprops.

Example 6.3. *Figure 6.1 provides a larger example. It is more stylized for brevity, but it covers a good subset of the relevant cases, used later to illustrate the semantics. Lines 1-2 provide the definitions of two predprops,* `nneg/1` *and* `neg/1` *respectively. The former describes a unary predicate which should have its argument constrained to a non-negative integer on success (expressed by the* `nnegint/1` *property), independently of how the predicate is called (note the* `true` *keyword in the precondition part). Similarly, the latter describes a unary predicate which succeeds with its argument bound to a negative integer (*`negint/1` *property). Predicates* `z/1`, `p/1`, `n/1` *and* `c/1` *are used as arguments in queries to* `test_c/2` *and* `test_s/2` *to trigger the checking of the predprops. While* `p/1` *and* `n/1` *completely satisfy* `nneg/1` *and* `neg/1` *respectively,* `z/1` *and* `c/1` *satisfy neither one of these predprops.*

Note that it would still be possible to define `nneg/1` or `neg/1` without the higher-order assertions. For example, we could define them by considering the meaning of each predicate symbol in our program. However, this approach has some serious limitations. First, we would need global reasoning over the whole program.[3]

Definition 6.11 (Meaning of a *predprop* Literal)**.** *The* meaning *of a* predprop $pp(X)$*, denoted* $|pp(X)|$ *is the set of constraints* $\{X = q \mid q \in PS, \forall C \in \mathcal{A}_C\langle pp\rangle\langle q\rangle : checked(C)\}$.

A predicate given by its predicate symbol $p \in \mathsf{PS}$ is *compatible* with a *predprop* $pp(X)$ if all the assertions resulting from $pp(p)$ are *checked* for all possible queries in an annotated program.

3 For the sake of simplicity we are not using modules in this chapter, but note that the reasoning would also have to include all modules in the program. Second, we would need to reconsider `nneg/1` or `neg/1` every time a new predicate is introduced in the program, which again is error prone, and against reusability and modular design. Our approach of dealing directly with higher order does not suffer from any of those limitations.

OPERATIONAL SEMANTICS FOR HIGHER-ORDER PROGRAMS WITH HIGHER-ORDER ASSERTIONS We now discuss several alternative operational semantics for higher-order programs with higher-order assertions. In all cases the aim of the semantics is to check whether assertions with *predprops* hold or not during the computation of the derivations from a query.

CHECKING WITH STATIC PREDPROPS According to Definition 6.11, a *predprop* literal $pp(X)$ denotes the subset of predicates for which all the associated assertions are checked. When that set of assertions can be statically computed, then $\theta \Rightarrow Cond$ can be used for both prop and predprop $Cond$ literals, and the operational semantics is identical to the one for the higher-order programs and regular assertions.

We will denote as $S \leadsto_{\mathcal{HA}_s} S'$ a reduction from a state S to a state S' under the semantics for higher-order derivations in programs with assertions that may contain higher-order properties, which are statically precomputed.

Thus, state reductions are performed as follows:

$$\frac{\langle G \mid \theta \mid \mathcal{E} \rangle \leadsto_{\mathcal{A}} \langle G' \mid \theta' \mid \mathcal{E}' \rangle}{\langle G \mid \theta \mid \mathcal{E} \rangle \leadsto_{\mathcal{HA}_s} \langle G' \mid \theta' \mid \mathcal{E}' \rangle}$$

The meaning of each predprop, $|pp(X)|$, can be inferred or checked (if given by the user) by static analysis.

In this semantics, given the program shown in Figure 6.1 and the goal `test_c(z,-2)`, assertions are detected to be false since $\{P = \mathtt{z}\} \not\subset |\mathtt{neg(P)}|$ and $\{P = \mathtt{z}\} \not\subset |\mathtt{nneg(P)}|$.

CHECKING WITH DYNAMIC PREDPROPS Given the difficulty in determining the meaning of $|pp(X)|$ statically, we also propose a semantics with dynamic checking. In this semantics we treat the case when a *predprop* $pp(X)$ is interpreted as a set of corresponding anonymous assertion conditions $\mathcal{A}_H\langle pp\rangle\langle X\rangle$ (see Definition 6.9), since in this case $|pp(X)|$ is not known statically. We start with an over-approximation of each predprop $|pp(X)| = \{X = p \mid p \in \mathsf{PS}\}$ and incrementally remove predicate symbols, as violations of assertion conditions are detected:

- we can detect when some assertion condition is violated (Def. 6.5);
- we need a way to obtain a set of assertion conditions from predprops (anonymous assertion conditions);

We do that by defining instantiations of anonymous assertion conditions for particular predicate symbols and the dependencies among those instances.

The following definition extends the notion of assertion conditions from the Definition 2.5 to the case of anonymous assertion conditions and higher-order literals:

Definition 6.12 (Hypothetical Assertion Condition)**.** *Given a predprop* $pp(X)$ *and a predicate symbol* $p \in PS$*,* $\mathcal{A}_C\langle pp\rangle\langle p\rangle$ *denotes the set of* hypothetical *assertion conditions* C_p*, such that for* $H\langle X\rangle \in \mathcal{A}_H\langle pp\rangle\langle X\rangle$ *(Def. 6.10),* $L = X(V_1, \ldots, V_n)$*, and* $L_p = p(V_1, \ldots, V_n)$*,* C_p *is defined as:*

$$C_p = \begin{cases} \mathit{calls}(L_p, Pre) & \text{if } H\langle X\rangle = \mathit{calls}(L, Pre) \\ \mathit{success}(L_p, Pre, Post) & \text{if } H\langle X\rangle = \mathit{success}(L, Pre, Post) \end{cases}$$

and the h *is a unique identifier provided by the* $\mathit{id}(C_p)$*, that can be easily distinguished from the identifiers* c *of the first-order assertion conditions. Please note that anonymous assertion conditions* $H\langle\_\rangle$ *do not have identifiers assigned to them, but rather serve as templates.*

Example 6.4. *Consider the* `comparator(P)` *predprop from Ex. 6.2, where* `P` *is constrained to a predicate symbol* `less/3`*. Then, the set of corresponding hypothetical assertion conditions is constructed as:*

$$\begin{aligned} &\mathcal{A}_C\langle comparator\rangle\langle less\rangle = \{ \\ &\quad C_1 = \mathit{calls}(less(X,Y,R), (int(X) \wedge int(Y)) \vee (flt(X) \wedge flt(Y))), \\ &\quad C_2 = \mathit{success}(less(X,Y,R), int(X) \wedge int(Y), between(-1,1,R)) \\ &\quad C_3 = \mathit{success}(less(X,Y,R), flt(X) \wedge flt(Y), between(-1,1,R)) \\ &\} \end{aligned}$$

and $\mathit{id}(C_i) = h_i$*,* $1 \leq i \leq 3$*.*

This way we obtain "first-order" assertion conditions for `less/3` *similar to the ones that would be obtained from user-provided assertions.*

Violation of such hypothetical assertion conditions has to be treated in a special way, as it does not signal the violation of the conditions themselves, but instead of the corresponding predprop. The error set $\mathcal{E}$ in Def. 6.5 contained negated assertion condition instance identifiers. Now we extend this set with *assertion dependency rules*[4] of the form $\bigwedge(\bigvee \bar{c}) \rightarrow \bar{c}$. For simplicity, we also introduce a special label h_0 to denote the assertion conditions that appeared originally in the program. The following definitions provide the description of how such dependencies are generated.

Definition 6.13 (Literal Simplification)**.** *The* simplification *of a literal* L *w.r.t.* θ *is defined as:*

$$\mathit{simp}(L, \theta) = \begin{cases} L & \text{if } L \text{ is a predprop} \\ true & \text{if } \theta \models\!\!\Rightarrow L \\ false & \text{if } \theta \not\models\!\!\Rightarrow L \end{cases}$$

We extend this definition for a conjunction of literals.

4 Note that those rules are propositional Horn clauses (about negated propositions), a P-complete problem solvable in linear time – a subset solvable for Prolog engines.

Definition 6.14 (Extension of $\mathcal{A}_C$ and $\mathcal{E}$ for dynamic predprop checking)**.** *Given the label c of an assertion condition and a formula of the form $Props = \bigvee_{i=1}^{n}(\bigwedge_{j=0}^{m(i)} Prop_{ij})$, where $Prop_{ij}$ is either a prop or predprop literal s.t. $\mathsf{simp}(Props, \theta) \neq true$, the* extension *of $\mathcal{A}_C$ and $\mathcal{E}$ for dynamic predprop checking, denoted as $\mathsf{ext}(\mathcal{A}_C, c, Props) = (\Delta\mathcal{A}_C, \Delta\mathcal{E})$, is obtained as follows:*

1. *if $\mathsf{simp}(Props, \theta) = false$, then $\Delta\mathcal{A}_C = \emptyset$ and $\Delta\mathcal{E} = \{\bar{c}\}$;*
2. *otherwise: $\Delta\mathcal{A}_C = \bigcup_{i=1}^{n} \mathcal{A}_C^i$, and $\Delta\mathcal{E} = \{\bigwedge_{i=1}^{n}(\bigvee_{h\in H_i} \bar{h}) \rightarrow \bar{c}\}$ where:*

$$\begin{aligned} \mathcal{A}_C^i = \ & \{C \mid C \in \mathcal{A}_C\langle Prop_{ij}\rangle\langle X_{ij}\rangle, 0 \leq j \leq m(i), \\ & \quad \text{and } X_{ij} \text{ is bound to some } q \in \mathsf{PS}\}. \\ H_i = \ & \{h \mid C \in \mathcal{A}_C^i \text{ and } \mathsf{id}(C) = h\} \end{aligned}$$

We will denote as $S \hookrightarrow_{\mathcal{HA}_\mathsf{d}} S'$ a reduction from a state S to a state S' under the current semantics.

Definition 6.15 (Reductions in Higher-order Programs with Higher-order Assertions)**.** *A state $S = \langle L :: G \mid \theta \mid \mathcal{E}\rangle$ where L is a literal can be* reduced *to a state S', denoted $S \hookrightarrow_{\mathcal{HA}_d} S'$, as follows:*

1. *If L is a constraint and $\theta \wedge L$ is satisfiable, then $S' = \langle G \mid \theta \wedge L \mid \mathcal{E}\rangle$.*
2. *If L is of the form $X(t_1, \ldots, t_n)$, then $S' = \langle G' \mid \theta \mid \mathcal{E}\rangle$ where*

$$G' = \begin{cases} p(t_1, \ldots, t_n) :: G & \text{if } \exists p \in \mathsf{PS} \wedge \theta \models (X = p) \wedge \mathsf{ar}(p) = n \\ \epsilon_{uninst\_call} & \text{otherwise} \end{cases}$$

3. *If L is an atom and $\exists(L \leftarrow B) \in \mathsf{cls}(L)$, then for each $C_i \in \mathcal{A}_C\langle L\rangle$ s.t. $\mathsf{id}(C_i) = c_i$:*

$$h_i = \begin{cases} h & \text{if } \exists C_j \in \mathcal{A}_C\langle L\rangle \wedge C_i = \sigma(C_j) \wedge \mathsf{id}(C_j) = h \\ h_0 & \text{otherwise} \end{cases}$$

$$(\Delta_i\mathcal{A}_C, \Delta_i\mathcal{E}) = \begin{cases} \mathsf{ext}(\mathcal{A}_C, c_i, Pre) & \text{if } C_i = \mathsf{calls}(L, Pre) \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases}$$

$$PostC_i = \begin{cases} \mathsf{check}(c_i) & \text{if } C_i = \mathsf{success}(L, Pre_i, Post_i) \\ & \quad \text{and } \mathsf{simp}(Pre_i, \theta) = true \\ true & \text{otherwise} \end{cases}$$

and $S' = \langle B :: PostC :: G \mid \theta \mid \mathcal{E}'\rangle$, where $\mathcal{A}'_C = \mathcal{A}_C \cup \bigcup_i \Delta_i\mathcal{A}_C$, $\mathcal{E}' = \mathcal{E} \cup \bigcup_i\{\bar{c}_i \rightarrow \bar{h}_i\} \cup \bigcup_i \Delta_i\mathcal{E}$, and $PostC$ is the sequence $PostC_1 :: \ldots :: PostC_n$ (simplifying `true` *literals).*

4. *If L is a check literal $\mathsf{check}(c)$ and $C = \mathsf{success}(L', \_, Post) \in \mathcal{A}_C\langle L'\rangle$ s.t. $\mathsf{id}(C) = c$, then $S' = \langle G \mid \theta \mid \mathcal{E}'\rangle$ where:*

a) $(\Delta\mathcal{E}, \Delta\mathcal{A}_C) = ext(\mathcal{A}_C, c, Post)$,
b) $\mathcal{E}' = \mathcal{E} \cup \Delta\mathcal{E}$ *and* $\mathcal{A}'_C = \mathcal{A}_C \cup \Delta\mathcal{A}_C$.

Note that in this semantics we support more than one *calls* assertion condition per predicate (as several predprops may be applied to the same predicate symbol). Also note that in general we cannot prove with dynamic checking that a predprop is *true*. So, as a safe approximation we treat preconditions in such *success* assertion conditions as $false$.

Definition 6.16 (Trivial Assertion Condition)**.** *An assertion condition C is* trivial *if it is of the form $calls(\_, true)$ or $success(\_, \_, true)$. It is also assumed that for any predprop $pp(X)$ $\mathcal{A}_H\langle pp\rangle\langle X\rangle$ does not contain trivial assertion conditions.*

Theorem 6.1 (Higher-order Run-time Checking)**.** *For any annotated program $(P, \mathcal{Q}, \mathcal{A})$, if $\exists D \in derivs_{\mathcal{HA}_d}(\mathcal{Q})$ s.t. $\neg rtsolve(C, D) \Rightarrow C \in \mathcal{A}_C$ is $false$.*

Proof. In this proof we reflect on the case when an assertion condition is falsified because some of its predprops are violated. To do so it is enough to show that at least one predprop was violated. Let us first prove the theorem for the case when the unsatisfied assertion condition is $C_c = \mathsf{calls}(L, pp(X))$ and then for the case $C_s = \mathsf{success}(L, Pre, pp(X))$, where $pp(X)$ is a predprop. Without loss of generality we assume that $\mathcal{A}_H\langle pp\rangle\langle X\rangle$ has cardinality 1 (which is the case when $pp(X)$ consists of one anonymous assertion and one of the corresponding anonymous assertion conditions is trivial).

• Let's assume $\neg\mathsf{rtsolve}(C_c, D)$
$\Leftrightarrow$ From Def. 6.8: $\exists c', C'_c, \sigma, L\ (C'_c \in \mathcal{A}_C\langle L\rangle) \wedge \mathsf{id}(C'_c) = c'$
$(\sigma(C_c) = C'_c) \wedge (\mathcal{E}(D) \vdash \bar{c}')$
$\Rightarrow$ From Def. 6.15 and $\mathcal{E}(D) \vdash \bar{c}'$ it must hold that
$D = (\ldots, S_1, \ldots, S_2, S_3 \ldots, S_4, \ldots)$ where:

$$
\begin{array}{lll}
S_1 = \langle L' :: \_ \mid \theta_1 \mid \_\rangle & \text{s.t. } \exists\ L' \leftarrow B' \in \mathsf{cls}(L), C'' = \mathsf{calls}(L', \sigma(pp(X))), & \\
 & \mathsf{id}(C'') = c', C'' \in \mathcal{A}_C\langle L\rangle, & \\
 & \theta_1 \models (X = q), q \in \mathsf{PS} & \\
S_2 = \langle L_2 :: \_ \mid \_ \mid \mathcal{E}_2\rangle & \text{s.t. } \{\bar{h} \rightarrow \bar{c}', \bar{c}' \rightarrow \bar{h}_0, \} \in \mathcal{E}_2, \mathsf{id}(C_q) = h, & \\
 & C_q \in \mathcal{A}_C\langle pp\rangle\langle q\rangle, L_2 = q(\ldots) & \\
S_3 = \langle \_ \mid \theta_3 \mid \mathcal{E}_3\rangle & \text{s.t. } \{\bar{c}'' \rightarrow \bar{h}\} \in \mathcal{E}_3,\ c''\#C''_c \in \mathcal{A}_C\langle L_2\rangle & \\
S_4 = \langle \_ \mid \_ \mid \mathcal{E}_4\rangle & \text{s.t. } \mathcal{E}_4 \vdash \bar{c}'' &
\end{array}
$$

$\Rightarrow$ From $\mathcal{E}_3 \vdash \bar{c}''$ and Th. 2.2 we know that $\neg\mathsf{checked}(C''_c)$
and thus $(X = q) \notin |pp(X)|$ according to Def. 6.11.
$\Rightarrow$ From Def. 2.3 it follows that $\theta_3 \not\Rightarrow pp(q)$
$\Rightarrow$ Given the state S_1 before the call to L' and the state S_3:
$(\mathsf{prestep}(L, D) = (\theta_3, \sigma)) \wedge (\theta' \not\Rightarrow \sigma(pp(X)))$

$\Rightarrow$ From Def. 2.6 $\neg\mathsf{solve}(C_c, D)$
$\Rightarrow$ From Def. 2.8 $\mathsf{false}(C_c)$ $\square$
•Let's assume $\neg\mathsf{rtsolve}(C_s, D)$
$\Leftrightarrow$ From Def. 6.8: $\exists c', C'_s, \sigma, L\ (C'_s \in \mathcal{A}_C\langle L\rangle) \wedge \mathsf{id}(C'_s) = c' \wedge$
$(\sigma(C_s) = C'_s) \wedge (\mathcal{E}(D) \vdash \bar{c}')$
$\Rightarrow$ From Def. 6.15 and $\mathcal{E}(D) \vdash \bar{c}'$ it must hold that
$D = (\ldots, S_1, S_2, \ldots, S_3, S_4, \ldots, S_5, S_6, \ldots, S_7, \ldots)$ where:

$$
\begin{array}{ll}
S_1 = \langle L' :: \_ \mid \theta_1 \mid \_\rangle & \text{s.t. } \exists\ L' \leftarrow B' \in \mathsf{cls}(L), \mathsf{id}(C'') = c' \\
\quad C'' = \mathsf{success}(L', \sigma(Pre), \sigma(pp(X))) \in \mathcal{A}_C\langle L\rangle, & \\
\quad\quad \theta_1 \Rrightarrow \sigma(Pre) & \\
S_2 = \langle B' :: \mathsf{check}(c') :: \_ \mid \_ \mid \mathcal{E}_2\rangle & \text{s.t. } \{\bar{c}' \rightarrow \bar{h}_0\} \in \mathcal{E}_2 \\
S_3 = \langle \mathsf{check}(c') :: \_ \mid \_ \mid \_\rangle & \\
S_4 = \langle \_ \mid \theta_4 \mid \mathcal{E}_4\rangle & \text{s.t. } \theta_4 \models (X = q), q \in \mathsf{PS}, \\
\quad\quad\quad\quad & \quad \{\bar{h} \rightarrow \bar{c}'\} \in \mathcal{E}_4, \mathsf{id}(C_q) = h, \\
\quad\quad\quad\quad & \quad C_q \in \mathcal{A}_C\langle pp\rangle\langle q\rangle. \\
S_5 = \langle L_5 :: \_ \mid \_ \mid \_\rangle & \text{s.t. } L_5 = q(\ldots) \\
S_6 = \langle \_ \mid \_ \mid \mathcal{E}_6\rangle & \text{s.t. } \{\bar{c}'' \rightarrow \bar{h}\} \in \mathcal{E}_6 \text{where} \\
\quad\quad\quad\quad & \quad C''_s \in \mathcal{A}_C\langle L_5\rangle, \mathsf{id}(C''_s) = c'' \\
S_7 = \langle \_ \mid \theta_7 \mid \mathcal{E}_7\rangle & \text{s.t. } \mathcal{E}_7 \vdash \bar{c}''
\end{array}
$$

$\Rightarrow$ From $\mathcal{E}_7 \vdash \bar{c}''$ and Th. 2.2 we know that $\neg\mathsf{checked}(C''_s)$
and thus $(X = q) \notin |pp(X)|$ according to Def. 6.11.
$\Rightarrow$ From Def. 2.3 it follows that $\theta_7 \not\Rrightarrow pp(q)$
$\Rightarrow$ Given the state S_1 before the call to L' and the state S_7:
$(\mathsf{step}(L, D) = (\theta_1, \sigma, \theta_7)) \wedge (\theta_1 \Rrightarrow \sigma(Pre)) \wedge (\theta_7 \not\Rrightarrow \sigma(pp(X)))$
for $C'_s \in \mathcal{A}_C\langle L\rangle$ s.t. $\mathsf{id}(C'_s) = c'$
$\Rightarrow$ From Def. 2.6 $\neg\mathsf{solve}(C_s, D)$
$\Rightarrow$ From Def. 2.8 $\mathsf{false}(C_s)$ $\square$

Let us trace finished derivations D^1, D^2 and D^3 from the queries $Q_1 = (\texttt{test\_c(n,X)}, true)$, $Q_2 = (\texttt{test\_c(c,X)}, true)$ and $Q_3 = (\texttt{(test\_s(1,P),P(-2))}, true)$, respectively, to the program in Figure 6.1.

In ${D^1}_{[1]}$ (see Table 6.1) we encounter two assertions for `test_c/2` with a predprop in each precondition and trivial postconditions. According to state reduction rules, $\Delta\mathcal{A}_C$ consists of calls assertion condition instance c_1 and two hypothetical assertion conditions h_1 and h_2, derived from predprops `nneg/1` and `neg/1`, and $\Delta\mathcal{E} = \{\bar{c}_1 \rightarrow \bar{h}_0, \bar{h}_1 \wedge \bar{h}_2 \rightarrow \bar{c}_1\}$. In ${D^1}_{[2]}$ and current goal `P(-1)` (which is implicitly reduced as `n(-1)`), success assertion condition instances c_2 and c_3 are derived from the hypotheses h_1 and h_2, and $\Delta\mathcal{E} = \{\bar{c}_2 \rightarrow \bar{h}_1, \bar{c}_3 \rightarrow \bar{h}_2\}$. Consequently, two check literals, $\mathsf{check}(2)$ and $\mathsf{check}(3)$ are added to the goal sequence. In states ${D^1}_{[3]}$ and ${D^1}_{[4]}$ those literals are reduced, which results in adding $\bar{c}_2$ to $\mathcal{E}$ because `nnegint(-1)` property from the

G	$\Delta\theta$	$\Delta\mathcal{E}$	$\Delta\mathcal{A}_C$
`test_c(n,X)`	$P = n$ $N = -1$ $X = N$	$\bar{c}_1 \rightarrow \bar{h}_0$ $\bar{h}_1 \wedge \bar{h}_2 \rightarrow \bar{c}_1$	c_0 h_1 h_2
`P(-1)`	$Z = -1$	$\bar{c}_2 \rightarrow \bar{h}_1$ $\bar{c}_3 \rightarrow \bar{h}_2$	c_2 c_3
$\mathsf{check}(c_2)$, $\mathsf{check}(c_3)$	-	$\bar{c}_2$	-
$\mathsf{check}(c_3)$	-	-	-
$\square$	-	-	-

c_0	$\mathsf{calls}(test\_c(n, X), nneg(n) \vee neg(n))$
c_2	$\mathsf{success}(n(-1), true, nnegint(-1))$
c_3	$\mathsf{success}(n(-1), true, negint(-1))$
h_1	$\mathsf{success}(n(Z), true, nnegint(Z))$
h_2	$\mathsf{success}(n(Z), true, negint(Z))$

Table 6.1: A derivation of the query (`test_c(n,X)`, *true*) to the program in Figure 6.1.

postcondition of c_2 is violated. This example shows that the mechanism of dependencies between assertion conditions allows avoiding "false negative" results in assertion checking.

The derivation D^2 is similar to D^1 (see Table 6.2). The difference is in the ${D^2}_{[4]}$ state, when it becomes possible to infer $\mathcal{E} \vdash \bar{c}_1$ and thus to conclude that `c/1` $\notin |nneg(X)| \wedge$ `c/1` $\notin |neg(X)|$ and that both assertions for `test_c/2` are *false* for this query.

In ${D^3}_{[1]}$ (see Table 6.3) we encounter two assertions with a pred-prop in each postcondition. According to the state reduction rules, $\Delta\mathcal{A}_C$ for this state consists of calls and success assertion condition instances, c_0 and c_1, $\Delta\mathcal{E} = \{\bar{c}_0 \rightarrow \bar{h}_0, \bar{c}_1 \rightarrow \bar{h}_0\}$ for them. Also, a check literal $\mathsf{check}(c_1)$ is added to the goal sequence. After its reduction a hypothetical assertion condition h_2, derived from the `nneg(X)` pred-prop which appears in c_1, is added to $\mathcal{A}_C$ in ${D^3}_{[3]}$, and $\mathcal{E}$ is extended with a dependency rule $\{\bar{h}_2 \rightarrow \bar{c}_1\}$. In state ${D^3}_{[4]}$ an assertion condition instance c_2 is obtained from h_2 and $\Delta\mathcal{E} = \{\bar{c}_2 \rightarrow \bar{h}_2\}$. Finally, in ${D^3}_{[5]}$ the error set contains the following chain of dependency rules: $\mathcal{E} \supset \{\bar{c}_2, \bar{c}_2 \rightarrow \bar{h}_2, \bar{h}_2 \rightarrow \bar{c}_1, \bar{c}_1 \rightarrow \bar{h}_0\}$ and rule $\bar{c}_1 \rightarrow \bar{h}_0$ allows us to detect and report the violation of the assertion condition c_1 for predicate `test_s/2`.

G	$\Delta\theta$	$\Delta\mathcal{E}$	$\Delta\mathcal{A}_C$
`test_c(c,X)`	$P = c$ $N = a$ $X = N$	$\bar{c}_1 \rightarrow \bar{h}_0$ $\bar{h}_2 \wedge \bar{h}_3 \rightarrow \bar{c}_1$	c_1 h_2 h_3
`P(a)`	$Z = a$	$\bar{c}_2 \rightarrow \bar{h}_2$ $\bar{c}_3 \rightarrow \bar{h}_3$	c_2 c_3
$\mathsf{check}(c_2)$, $\mathsf{check}(c_3)$	-	$\bar{c}_2$	-
$\mathsf{check}(c_3)$	-	$\bar{c}_3$	-
$\square$	-	-	-

c_1	$\mathsf{calls}(test\_c(c, X), nneg(c) \vee neg(c))$
c_2	$\mathsf{success}(c(a), true, nnegint(a))$
c_3	$\mathsf{success}(c(a), true, negint(a))$
h_2	$\mathsf{success}(c(Z), true, nnegint(Z))$
h_3	$\mathsf{success}(c(Z), true, negint(Z))$

Table 6.2: A derivation of the query (`test_c(c,X)`, *true*) to the program in Figure 6.1.

6.3 MINIMALISTIC SAMPLE IMPLEMENTATION

The following code (portable to most Prolog systems with minor changes) shows a minimalistic sample implementation (as an interpreter `intr/1`) of the operational semantics for dynamic predprop checking (Def. 6.15). Conciseness and simplicity has been favoured over efficiency. We assume that clauses, assertion conditions, and predprops have been parsed and stored in `cl/2`, `ac/1`, `pp/2` facts, respectively. The interpreter will throw an exception the first time that a failed program assertion is detected (see `ext/2` predicate). E.g., `intr((test_s(1,P),P(1)))` is a valid query while `intr((test_s(1,P),P(-2)))` throws a failed assertion exception. Predicate `reset/0` must be called between `intr/1` queries to reset error status and temporary data. In the handler errors can be gathered (as in the semantics) or execution aborted.

G	$\Delta\theta$	$\Delta\mathcal{E}$	$\Delta\mathcal{A}_C$
`test_s(1,P)`, `P(-2)`	$N = 1$	$\bar{c}_0 \to \bar{h}_0$ $\bar{c}_1 \to \bar{h}_0$	c_0 c_1
`P = z, check(`c_1`),` `P(-2)`	$P = z$	-	-
`check(`c_1`), P(-2)`	-	$\bar{h}_2 \to \bar{c}_1$	h_2
`P(-2)`	$Z = -2$	$\bar{c}_2 \to \bar{h}_2$	c_2
check(c_2)	-	$\bar{c}_2$	-
$\square$	-	-	-

c_0	calls$(test_s(1,P), nnegint(1) \vee negint(1))$
c_1	success$(test_s(1,P), nnegint(1), nneg(P))$
c_2	success$(z(-2), true, nnegint(-2))$
h_2	success$(z(Z), true, nnegint(Z))$

Table 6.3: A finished derivation of the query `((test_s(1,P),P(-2))`, *true*) to the program in Figure 6.1.

```
:- module(_,[reset/0,intr/1],[hiord,dcg,dynamic_clauses]).
:- use_module(library(aggregates)).

%------------------------------------------------------%
% Sample program data and properties

% negint/1 and nnegint/1 properties
eval_prop(negint(X))  :- integer(X), X < 0.
eval_prop(nnegint(X)) :- integer(X), X >= 0.

% predprops nneg/1 and neg/1
pp(nneg(P),ac(P(X), nneg_c1(P)#success(true,nnegint(X)))).
pp(neg(P), ac(P(X), neg_c1(P)#success(true, negint(X)))).

% assertion conditions and clauses for test_s/2
ac(test_s(N,_P), c1#calls((nnegint(N);negint(N)))).
ac(test_s(N,P), c2#success(nnegint(N), nneg(P))).
ac(test_s(N,P), c3#success(negint(N), neg(P))).
cl(test_s( 1,P), P = z).
cl(test_s(-1,P), P = n).

% clauses for z/1, n/1
cl(z(1), true).  cl(z(-2), true).
cl(n(-1), true).  cl(n(-2), true).

%------------------------------------------------------%
% Intepreter

:- dynamic hyp_ac/2. % hypothetical assertion condition
:- dynamic negac/1. % (negated) assertion dependency rule

% Reset errors and hypothetical assertion conditions
reset :- retractall(hyp_ac(_, _)),
         ( retract((negac(_) :- _)), fail ; true ).

% Interpreter with higher-order assertion checking
intr(X) :- ctog(X, X1), !, intr(X1).
```

```
intr(X) :- is_blt(X), !, X.
intr((A,B)) :- !, intr(A), intr(B).
intr((A ; B)) :- !, ( intr(A) ; intr(B) ).
intr(A) :-
    get_acs(A, Acs),
    pre(Acs, Ids, []), cl(A, Body),
    intr(Body), post(Ids, Acs).

% Built-ins
is_blt(true).   is_blt(fail).   is_blt(_ = _).

% From call(N,...) to N(...),where N is a predicate symbol
ctog(X, _) :- var(X), !, throw(inst_error).
ctog(X, X1) :-
    X =.. [call,N|Args],
    ( atom(N) -> true ; throw(inst_error) ),
    X1 =.. [N|Args].

% Get assertion conditions for the given literal A
get_acs(A, Acs) :- ( bagof(Ac, get_ac(A, Ac), Acs)
                   -> true ; Acs = [] ).
get_ac(A, Ac) :- ( ac(A, Ac) ; hyp_ac(A, Ac) ).

pre([]) --> [].
pre([Ac|Acs]) --> pre_(Ac), pre(Acs).
pre_(Id#calls(Pre)) --> { ext(Pre, Id) }.
pre_(Id#success(Pre, _)) --> ( { simp0(Pre, true) }
                             -> [Id] ; [] ).

post([], _Acs).
post([Id|Ids], Acs) :- post_(Id, Acs), post(Ids, Acs).
post_(Id, Acs) :- member(Id0#success(_Pre,Post), Acs),
    Id == Id0, !, ext(Post, Id).
post_(_, _).

% Check/extend assertion conditions
ext(Props, Id) :-
    simp(Props, Props2), ext_(Props2, Id),
    ( negac(A), atom(A)
    -> throw(failed_assertion(A)) ; true ).
ext_(true, _Id) :- !.
ext_(false, Id) :- !, assertz((negac(Id) :- true)).
ext_(Props, Id) :- acsubs(Props, Props2),
    assertz((negac(Id) :- Props2)).

% Add assertion dependency rules
acsubs((A,B), (A2;B2)) :- !,
    acsubs(A, A2), acsubs(B, B2).
acsubs((A ; B), (A2 , B2)) :- !,
    acsubs(A, A2), acsubs(B, B2).
acsubs(ac(L, Id#Ac), negac(Id)) :-
    ctog(L, L2), assertz(hyp_ac(L2, Id#Ac)).

% Condition simplification
simp(true, R) :- !, R = true.
simp((X;Y), R) :- !,
    simp(X, Rx), simp(Y, Ry), or(Rx, Ry, R).
simp((X,Y), R) :- !,
    simp(X, Rx), simp(Y, Ry), and(Rx, Ry, R).
simp(X, R) :- pp(X, Ac), !, R = Ac.
simp(X, R) :- eval_prop(X), !, R = true.
simp(_, R) :- R = false.

% Condition simplification for success preconditions
simp0(true, R) :- !, R = true.
simp0((X,Y), R) :- !,
```

```
104     simp0(X, Rx), simp0(Y, Ry), and(Rx, Ry, R).
105 simp0(X, R) :- eval_prop(X), !, R = true.
106 simp0(_, R) :- R = false.
107
108 or(true, _, R) :- !, R = true.
109 or(false, X, R) :- !, R = X.
110 or(_, true, R) :- !, R = true.
111 or(X, false, R) :- !, R = X.
112 or(X, Y, (X;Y)).
113
114 and(false, _, R) :- !, R = false.
115 and(true, X, R) :- !, R = X.
116 and(_, false, R) :- !, R = false.
117 and(X, true, R) :- !, R = X.
118 and(X, Y, (X,Y)).
```

6.4 CONCLUSIONS

This chapter contributes towards filling the gap between higher-order (C)LP programs and assertion-based extensions for error detection and program verification. To this end we have defined a new class of properties, "predicate properties" (*predprops* in short), and proposed a syntax and semantics for them. These new properties can be used in assertions for higher-order predicates to describe the properties of the higher-order arguments. We have also discussed several operational semantics for performing run-time checking of programs including predprops and provided correctness results.

Our predprop properties specify conditions for predicates that are independent of the usage context. This corresponds in functional programming to the notion of *tight* contract satisfaction [30], and it contrasts with alternative approaches such as *loose* contract satisfaction [32]. In the latter, contracts are attached to higher-order arguments by implicit function wrappers. The scope of checking is local to the function evaluation. Although this is a reasonable and pragmatic solution, we believe that our approach is more general and more amenable for combination with static verification techniques. For example, avoiding wrappers allows us to remove checks (e.g., by static analysis) without altering the program semantics.[5] Moreover, our approach can easily support *loose* contract satisfaction, since it is straightforward in our framework to optionally include wrappers as special predprops.

We have included the proposed predprop extensions in an experimental branch of the Ciao assertion language implementation. This has the immediate advantage, in addition to the enhanced checking, that it allows us to document higher-order programs in much more

5 E.g. `f(g)=g` is not an identity function if wrappers are added to `g` on call. This complicates reasoning about the program, and may lead to unexpected and hard to detect differences in program semantics. Similar examples can be constructed where the presence of predprops in assertions would invalidate many reasonable program transformations.

accurate way. We have also implemented several prototypes for operational semantics with dynamic predprop checking (Section 6.3)

7

CONCLUSIONS AND FUTURE WORK

This chapter closes the dissertation by summarizing the main general results and conclusions from the previous chapters, highlighting what has been achieved with respect to the objectives and plan presented in Chapter 1. It also outlines possible future developments in the methods and tools proposed in the dissertation.

7.1 CONCLUSIONS

A motivation of this dissertation has been the development of improved techniques and tools for efficient assertion-based compile- and run-time software verification for dynamic programming languages. These techniques and tools have been presented in the context of Ciao — a dynamic declarative multi-paradigm programming language with a combined static/dynamic assertion checking framework. However, it is also an argument of the thesis that the results are applicable to other programming paradigms, either directly (e.g., to other forms of declarative programming), or to imperative programs, via the technique of semantic transformation into Horn clauses.

Among the most relevant results and conclusions from this work we can mention:

- We have proposed an unobtrusive run-time check caching mechanism that allowed performing fewer repeated checks and where (Chapter 3):

  SIGNIFICANT RUN-TIME OVERHEAD REDUCTIONS (orders of magnitude) were observed for checks of deep immutable recursive data structures defined by *regular types*;

  SEVERAL CACHING POLICIES were experimentally evaluated for different data structures which demonstrated the effect of caching policy in property caching efficiency.

- We have proposed improvements in the context of combined (multivariant) compile-time and run-time checking that allowed simultaneously improving the efficiency of both verification techniques, such that (Chapter 4):

  IMPROVED STATIC CHECKING PRECISION was achieved by improving the precision of static program analysis through the inclusion of additional multivariant information about reachable and unreachable program states coming from the run-time checks;

SIGNIFICANT REDUCTIONS IN RUN-TIME OVERHEAD were observed due to the fact that more precise static analysis was able to verify more checks at compile time;

SEVERAL SCENARIOS OF RUN-TIME CHECK INSTRUMENTATION were formalized as *assertion checking modes* which allowed discussing about the trade-off between checking thoroughness and program behavior safety guarantees;

- We have presented an approach for run-time overhead reduction in checks for calls across module boundaries in multi-modular programs where (Chapter 5):

  A SEMANTICS FOR MODULAR LOGIC PROGRAMS was proposed that is implementation-agnostic;

  CONSTANT RUN-TIME CHECKING OVERHEAD was achieved in several *assertion checking modes* due to taking module system visibility rules into account when reasoning about term visibility in different execution contexts. These techniques were shown to provide results for shape-style properties similar to strong typing even in open library scenarios.

- We have presented a lightweight extension for the Ciao assertion language for providing specifications of run-time behavior of predicate-bearing arguments of higher-order predicates where (Chapter 6):

  MORE DETAILED SPECIFICATIONS, which are a syntactic and semantic extension of the Ciao assertion language with higher-order properties (*predprops*), allow specifying in detail the run-time behavior of the predicate-bearing arguments of higher-order predicates;

  A BETTER ERROR REPORTING mechanism for controlling blame assignment in the case of *predprop* violation (assertion dependency rules) makes it possible to pinpoint erroneous higher-order calls with greater precision.

7.2 FUTURE WORK

Among the work which this dissertation leaves open for the future we would like to outline the following directions:

CACHING AND GARBAGE COLLECTION The proposed property caching techniques could be improved by a more sophisticated interaction/integration with the garbage collection (GC) mechanism. Currently, to remain consistent, the cache contents are invalidated on GC. This behavior could be replaced by a less intrusive one, where only the parts of the cache that contain terms subject to GC are invalidated, keeping the rest intact.

JUST IN TIME COMPILATION Another interesting direction would be to explore the potential of "just-in-time" compilation techniques in the context run-time check optimization.

IMPROVED BLAME ASSIGNMENT While the Ciao system already has facilities for reporting and locating errors stemming from assertion checking, the current blame assignment mechanisms could be improved, specially in the context of higher-order calls. There have been interesting recent advances in this area for functional programming languages.

COMBINATIONS WITH STATIC/DYNAMIC PROFILING The proposal here is to develop tools for statically/dynamically tracking how run-time checking affects overall program costs and how the run-time checking is combined with standard program execution flow. Our argument here is that if run-time checks are present within program "hot spots" – frequent and costly operations – investing in the optimization of not only the hot spots themselves, but also specifically the associated run-time checks might significantly contribute to run-time overhead reductions. In this context, we have already made some early progress in a method for static profiling of programs with run-time checks in order to detect combined costs, detect these hot spots, and provide static performance guarantees [49, 48].

SPECIFICATION INFERENCE FROM TEXTUAL DOCUMENTATION There already exist proposals and techniques for program specification inference from natural language comments, or constructing specifications using natural language processing tools over variable and function names of programs. In the context of the Ciao assertions model it seems an attractive research direction since the textual documentation appears within assertion contexts and thus strong connections can be established between the textual documentation and the formal properties. Also helping towards this end is the availability of extensive documentation of this kind since all the system manuals are extracted from this combination of specifications adorned with text (see, e.g., the LPDoc system [41, 39]).

ENHANCING SPECIFICATIONS WITH MINED DATA Numerous techniques have been proposed for inferring program invariants or partial specifications by mining execution traces (usually in the context of using run-time *monitors*). We feel this would constitute a promising complementary approach for obtaining or correcting existing specifications for higher-order predicates.

BIBLIOGRAPHY

[1] Hassan Aït-Kaci. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 52–68. MIT Press, 1993.

[2] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2), 2008.

[4] C. Beierle, R. Kloos, and G. Meyer. A Pragmatic Type Concept for Prolog Supporting Polymorphism, Subtyping, and Meta-Programming. In *Proc. of the ICLP'99 Workshop on Verification of Logic Programs, Las Cruces*, Electronic Notes in Theoretical Computer Science, volume 30, issue 1. Elsevier, 2000.

[5] N. Bjørner, Fabio Fioravanti, A. Rybalchenko, and V. Senni, editors. *Workshop on Horn Clauses for Verification and Synthesis*, July 2014. Electronic Proceedings in Theoretical Computer Science.

[6] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially Evaluating Finite-state Runtime Monitors Ahead of Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, June 2012.

[7] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[8] J. Boye, W. Drabent, and J. Małuszyński. Declarative Diagnosis of Constraint Programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

[9] M. Bruynooghe and J. Gallagher. Inferring Polymorphic Types from Logic Programs. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*. Preproceedings, July 2004.

[10] 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.

[11] F. Bueno, M. Carro, M. V. Hermenegildo, P. López-García, , and J.F. Morales (Eds.). The Ciao System. Ref. Manual (v1.16). Technical report, July 2017. Available at `http://ciao-lang.org`.

[12] 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.

[13] F. Bueno, M. García de la Banda, and M. V. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[14] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[15] 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.

[16] D. Cabeza, M. V. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

[17] Mats Carlsson and Per Mildner. SICStus Prolog – the First 25 Years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[18] Robert Cartwright and Mike Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI 1991)*, pages 278–292, New York, NY, USA, 1991. ACM.

[19] W. Chen. A theory of modules based on second-order logic. In *Proc. 4th IEEE Int'l. Symposium on Logic Programming*, pages 24–33, San Francisco, 1987.

[20] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[21] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[22] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[23] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[24] 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.

[25] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.

[26] 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.

[27] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[28] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based Generation of Verification Conditions via Program Specialization. *Science of Computer Programming*, 147(Supplement C):78–108, 2017. Selected and Extended papers from the 17th International Symposium on Principles and Practice of Declarative Programming 2015.

[29] S. Dietrich. *Extension Tables for Recursive Query Evaluation*. PhD thesis, Departament of Computer Science, State University of New York, 1987.

[30] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(5):1–29, November 2011.

[31] Manuel Fähndrich and Francesco 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.

[32] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 48–59. ACM, 2002.

[33] J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming (ICLP'94)*, pages 599–613. MIT Press, 1994.

[34] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Horn Clauses as an Intermediate Representation for Program Analysis and Transformation. *TPLP*, 15:526–542, 2015.

[35] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proc. International Conference on Rewriting Techniques and Applications (RTA)*, pages 210–220, Aachen, Germany, 2004.

[36] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[37] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[38] Rémy Haemmerlé and François Fages. Modules for Prolog Revisited. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP'06*, volume 4079 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2006.

[39] 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.

[40] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. 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. http://arxiv.org/abs/1102.5497.

[41] M. V. Hermenegildo and The CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The Ciao System Documentation Series–TR CLIP5/97.3, Facultad de Informática, UPM, August 1997. Online at `http://ciao-lang.org`.

[42] 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.

[43] 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.

[44] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[45] 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.

[46] International Organization for Standardization, 1, rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland. *PROLOG. ISO/IEC DIS 13211-2 — Part 2: Modules*, 2000.

[47] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement Types for Ruby. In Isil Dillig and Jens Palsberg, editors, *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'18, pages 269–290, Cham, 2018. Springer International Publishing.

[48] M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo. An Approach to Static Performance Guarantees for Programs with Run-time Checks. Technical Report CLIP-1/2018.0, The CLIP Lab, IMDEA Software Institute and T.U. Madrid, April 2018.

[49] M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo. Towards Static Performance Guarantees for Programs with Run-time Checks. In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, OpenAccess Series in Informatics (OASIcs), July 2018. (Extended Abstract).

[50] Emmanouil Koukoutos and Viktor Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 263–268. Springer International Publishing, 2014.

[51] Claude Laï. Assertions with Constraints for CLP Debugging. In Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2000.

[52] Leslie Lamport and Lawrence C. Paulson. Should Your Specification Language be Typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[53] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[54] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.

[55] 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. Van Eekelen and U. Dal 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.

[56] 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 Gopal Gupta and Ricardo 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.

[57] Francesco Logozzo et al. Clousot. http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

[58] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In Chandra Krintz and Emery 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.

[59] M. Méndez-Lojo, O. Lhoták, and M. V. Hermenegildo. Efficient Set Sharing using ZBDDs. In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, volume 5335 of *LNCS*, pages 94–108. Springer-Verlag, August 2008.

[60] 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.

[61] E. Mera, P. López-García, and M. V. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.

[62] E. Mera, T. Trigo, P. López-García, and M. V. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.

[63] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, pages 79–108, 1989.

[64] MSR. Code contracts. http://research.microsoft.com/en-us/projects/contracts/.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[70] Gopalan Nadathur and Dale Miller. Higher–Order Logic Programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press, 1998.

[71] Lee Naish. Higher-order Logic Programming. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, feb 1996. URL: http://www.cs.mu.oz.au/~lee/papers/ho/.

[72] Lee Naish, P. W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In Antonio Porto, editor, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536. MIT Press, June 1989.

[73] 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.

[74] 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.

[75] 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.

[76] Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 446–456. ACM, 2015.

[77] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 139–152, New York, NY, USA, 2014. ACM.

[78] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher-order Symbolic Execution for Contract Verification and Refutation. *Journal of Functional Programming*, 27(3), January 2017.

[79] 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.

[80] Prolog Development Center. Visual Prolog. Available from http://www.visual-prolog.com/.

[81] 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.

[82] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.

[83] G. Puebla, F. Bueno, and M. V. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] G. Puebla and M. V. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[89] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015.

[90] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[91] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. Case Studies and Tools for Contract Specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 596–607, New York, NY, USA, 2014. ACM.

[92] Joachim Schimpf and Kish Shen. ECLiPSe – from LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, January 2012.

[93] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards Typed Prolog. In Enrico Pontelli and María M. García de la Banda, editors, *International Conference on Logic Programming*, number 5366 in LNCS, pages 693–697. Springer Verlag, December 2008.

[94] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *12th International Symposium Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2005.

[95] A. Serrano, P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). In T. Swift and E. Lamma, editors, *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, volume 13, pages 1–14. Cambridge U. Press, August 2013.

[96] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[97] 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.

[98] H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[99] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pages 225–235. ACM Press, September 2014.

[100] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Towards Assertion-based Debugging of Higher-Order (C)LP Programs (Extended Abstract). In M. Leuschel and T. Schrijvers, editors, *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue, On-line Supplement*, volume 14, pages 209–210. Cambridge U. Press, July 2014.

[101] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue*, 15(04-05):726–741, September 2015. http://arxiv.org/abs/1507.05986.

[102] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

[103] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Towards Run-time Checks Simplification via Term Hiding. In Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei, editors, *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, volume 58 of *OpenAccess Series in Informatics (OASIcs)*, pages 91–93, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Extended Abstract).

[104] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Exploiting Term Hiding to Reduce Run-time Checking Overhead. In Francesco Calimeri, Kevin Hamlen, and Nicola Leone, editors, *20th International Symposium on Practical Aspects of Declarative Languages (PADL 2018)*, number 10702 in LNCS, pages 99–115. Springer-Verlag, January 2018.

[105] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Some Tradeoffs 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.

[106] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

[107] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[108] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 456–468. ACM, 2016.

[109] H. Tamaki and M. Sato. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.

[110] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In George C. Necula and Philip 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 395–406. ACM, 2008.

[111] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 537–554. ACM, 2012.

[112] 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.

[113] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM.

[114] D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111, 1992.

[115] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[116] D.S. Warren and W. Chen. Formal semantics of a theory of modules. Technical report 87/11, SUNY at Stony Brook, 1987.

[117] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[118] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[119] Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *TPLP*, 12(4-5):547–563, 2012.

A

LIST OF SYMBOLS

This appendix includes notation symbols used in the dissertation.

COMMON SYMBOLS

$\_$	anonymous variable
VS	set of variable symbols
FS	set of function symbols
PS	set of predicate symbols
TS	set of all terms
$\mathsf{ar}(p)$	arity of a symbol p
θ	constraint
$\sigma(\_)$	variable renaming
$\theta_1 \models \theta_2$	constraint entailment
$\mathsf{cls}(L)$	definition of an atom L
$\langle \_ \mid \_ \rangle$	program state
$S_1 \rightsquigarrow S_2$	single reduction from a state S_1 to a state S_2
$S_1 \rightsquigarrow^* S_2$	a series of reductions from a state S_1 to a state S_2
$D_{[-1]}$	current state on a derivation D
$D_{[n]}$	n-th state on a derivation D
Q	query
$\mathcal{Q}$	set of queries
$\mathsf{derivs}(Q)$	set of derivations from a query Q
$\square$	empty goal
$\mathsf{answers}(Q)$	set of answers to a query Q
$\theta \Mapsto L$	literal L *succeeds trivially* for the constraint θ
$\mathcal{A}$	set of all assertions in the program
$\mathcal{A}\langle L \rangle$	set of all assertions for a literal L
$\mathcal{A}_C$	set of all assertion conditions in the program
$\mathcal{A}_C\langle L \rangle$	set of all assertion conditions for a literal L
c	assertion condition identifier
$\mathsf{id}(\_)$	mapping between assertion conditions and their identifiers

$\mathsf{err}(c)$	error literal
$\mathsf{check}(c)$	check literal
$S_1 \rightsquigarrow_{\mathcal{A}} S_2$	single reduction from a state S_1 to a state S_2 (with assertions)
$S_1 \rightsquigarrow_{\mathcal{A}}^{*} S_2$	a series of reductions from a state S_1 to a state S_2 (with assertions)
$\mathsf{derivs}_{\mathcal{A}}(Q)$	set of derivations from a query Q (with assertions)
$\_^{\circ}$	error-erasing syntactic rewriting

NEW SYMBOLS INTRODUCED IN CHAPTER 3

$\mathbb{M}$	cache store
$\langle \_ \mid \_ \mid \_ \rangle$	extended program state with cache store
$\theta \overset{\mathbb{M}}{\Rightarrow} L$	literal L *succeeds trivially* for the constraint θ and the cache $\mathbb{M}$
$\mathsf{upd}(\theta, \mathbb{M}, \_)$	cache update

NEW SYMBOLS INTRODUCED IN CHAPTER 4

$\mathsf{status}(c, \_)$	status of an assertion condition with the identifier c
ppt	program point identifier
L^{ppt}	literal L located at the program point ppt
$\mathsf{status}^{\mathsf{ppt}}(c, \_)$	status of an assertion condition with the identifier c at the program point ppt

NEW SYMBOLS INTRODUCED IN CHAPTER 5

MS	set of module symbols
$\mathsf{mod}(L)$	mapping for a module of a symbol L
$\mathsf{def}(m)$	set of symbols defined in a module m
$\mathsf{exp}(m)$	predicate symbols export list of a module m
$\mathsf{imp}(m)$	predicate symbols import list of a module m
$\mathsf{ret}(\_)$	clause end literal
$\mathsf{ret}(\_, \_)$	extended clause return literal
$S_1 \rightsquigarrow_{\mathsf{rtc}} S_2$	single reduction from a state S_1 to a state S_2 (with assertions and explicit module resolution)
$\mathsf{rtc\text{-}derivs}(\_)$	set of derivations for a modular program (with assertions and explicit module resolution)

$\mathsf{esc}_m(\_)$	escaping terms of a module m
$\mathsf{vis}_m(\_)$	visible terms of a module m
$\mathsf{usr}(\_)$	set of all public user terms

NEW SYMBOLS INTRODUCED IN CHAPTER 6

LS	set of literals
$S_1 \not\hookrightarrow S_2$	single reduction from a state S_1 to a state S_2 (higher order derivations)
$\bar{c}$	negated assertion condition identifier
$\mathcal{E}$	store for violated assertion conditions and assertion dependencies
$\langle \_ \mid \_ \mid \_ \rangle$	extended program state
$S_1 \not\hookrightarrow_{\mathcal{A}} S_2$	single reduction from a state S_1 to a state S_2 (higher order derivations, first order assertions)
$S_1 \not\hookrightarrow^*_{\mathcal{A}} S_2$	a series of reductions from a state S_1 to a state S_2 (higher order derivations, first order assertions)
$\_^{\diamond}$	error-erasing syntactic rewriting (higher-order case)
$H\langle X \rangle$	a hypothetical assertion condition with a free variable X instead of a predicate symbol
$\mathcal{A}_H\langle pp \rangle\langle X \rangle$	set of *anonymous assertion conditions* for the predprop $pp(X)$
$\mathcal{A}_C\langle pp \rangle\langle q \rangle$	set of non-anonymous assertion conditions generated for the predprop pp and a $q \in \mathsf{PS}$
h	hypothetical assertion condition identifier
$\bar{h}$	negated hypothetical assertion condition identifier
$S_1 \not\hookrightarrow_{\mathcal{HA}_\mathsf{s}} S_2$	single reduction from a state S_1 to a state S_2 (higher order derivations, higher order assertions, static predprop checking)
$S_1 \not\hookrightarrow_{\mathcal{HA}_\mathsf{d}} S_2$	single reduction from a state S_1 to a state S_2 (higher order derivations, higher order assertions, dynamic predprop checking)
$\mathsf{derivs}_{\mathcal{HA}_\mathsf{d}}(\_)$	set of higher-order derivations (higher order assertions, dynamic predprop checking)

B

ADDITIONAL PLOTS

B.1 ADDITIONAL PLOTS FOR CHAPTER 3

This section includes plots of the run-time checking overhead observed in the set of 7 benchmarks for different cache replacement policies. There are four groups of plots:

- overhead ratio plots, where overhead ratio curves are grouped by cache size and check depth limit (Figures B.1 and B.8);
- overhead ratio plots, where overhead ratio curves are grouped by benchmark and check depth limit (Figures B.2 and B.9);
- maximal regtype check depth reached plots, where check depth curves are grouped by benchmark and cache size (Figures B.4 and B.11);
- absolute and relative benchmark execution time plots for benchmarks without rtchecks, with rtchecks and with both rtchecks and caching (Figures B.6 and B.13).

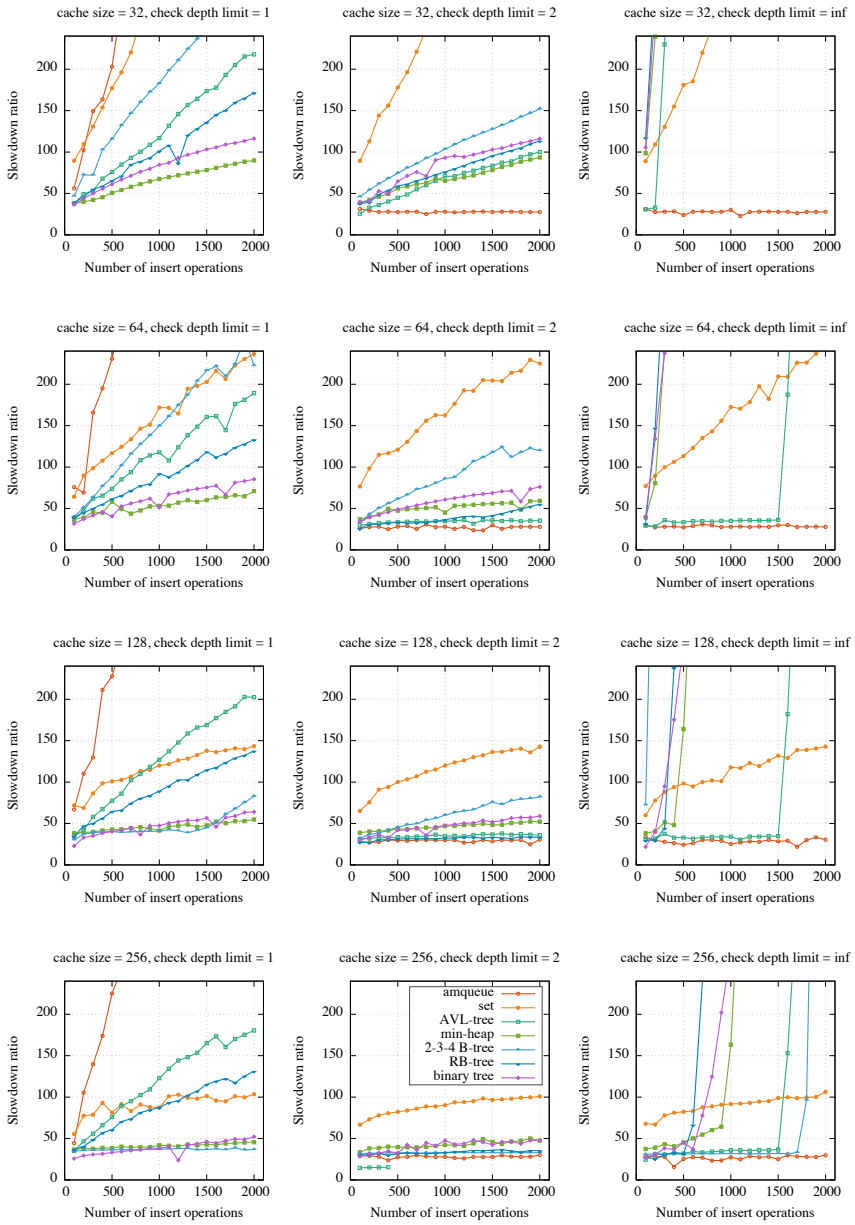


Figure B.1: Overhead ratios for all benchmarks, check depth limits 1, 2 and ∞, LRU caching policy.

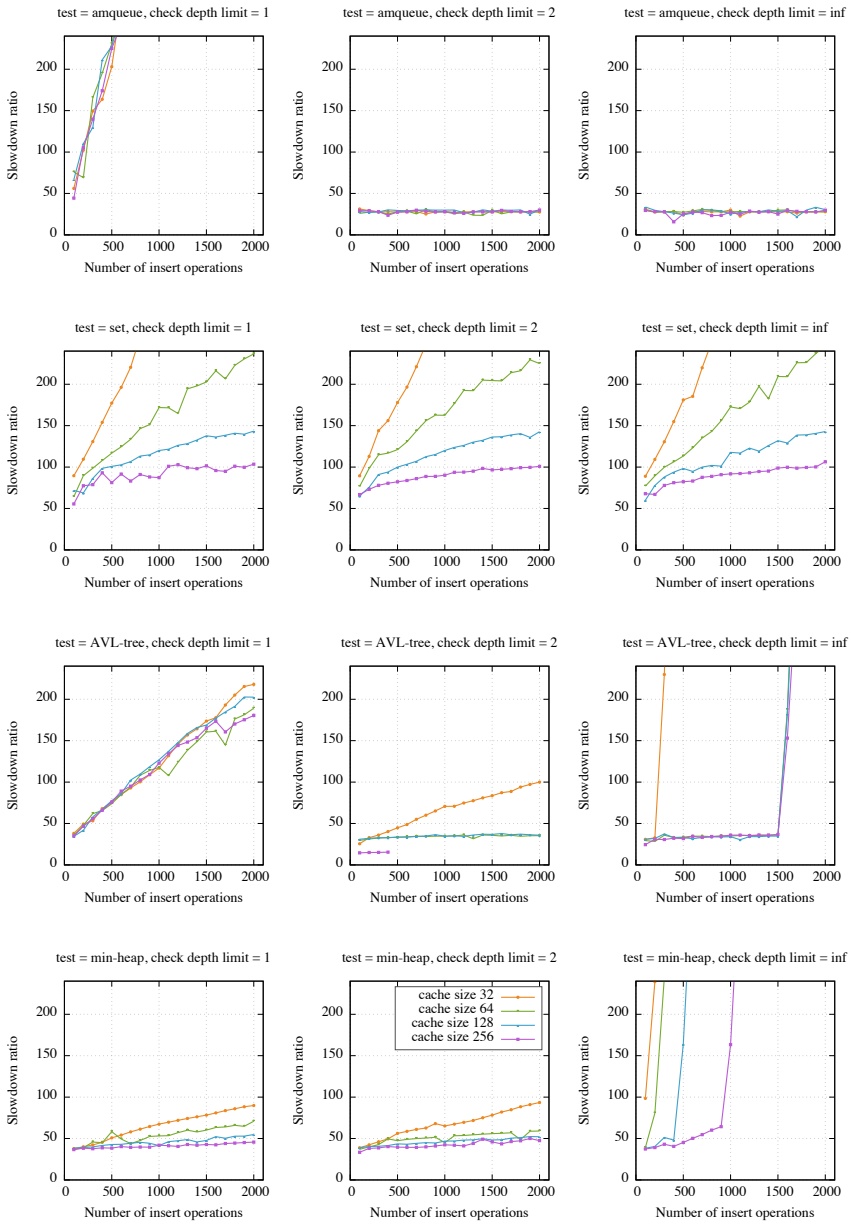


Figure B.2: Overhead ratios for each benchmark, check depth limits 1, 2 and ∞, LRU caching policy.

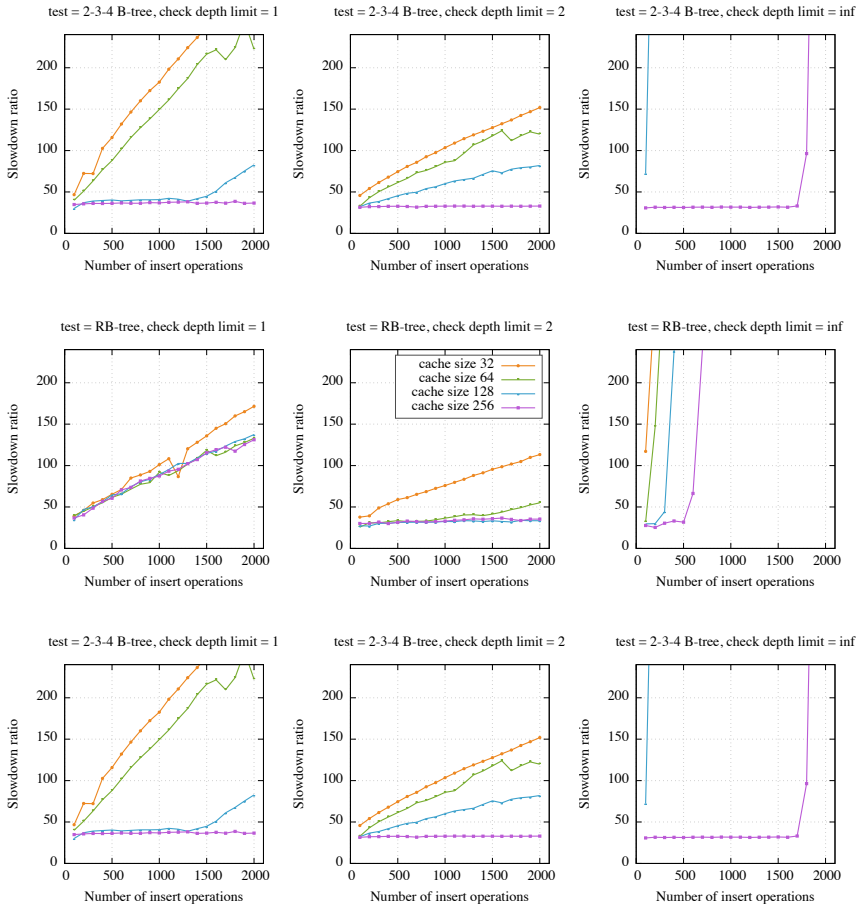


Figure B.3: Overhead ratios for each benchmark, check depth limits 1, 2 and ∞, LRU caching policy (contd).

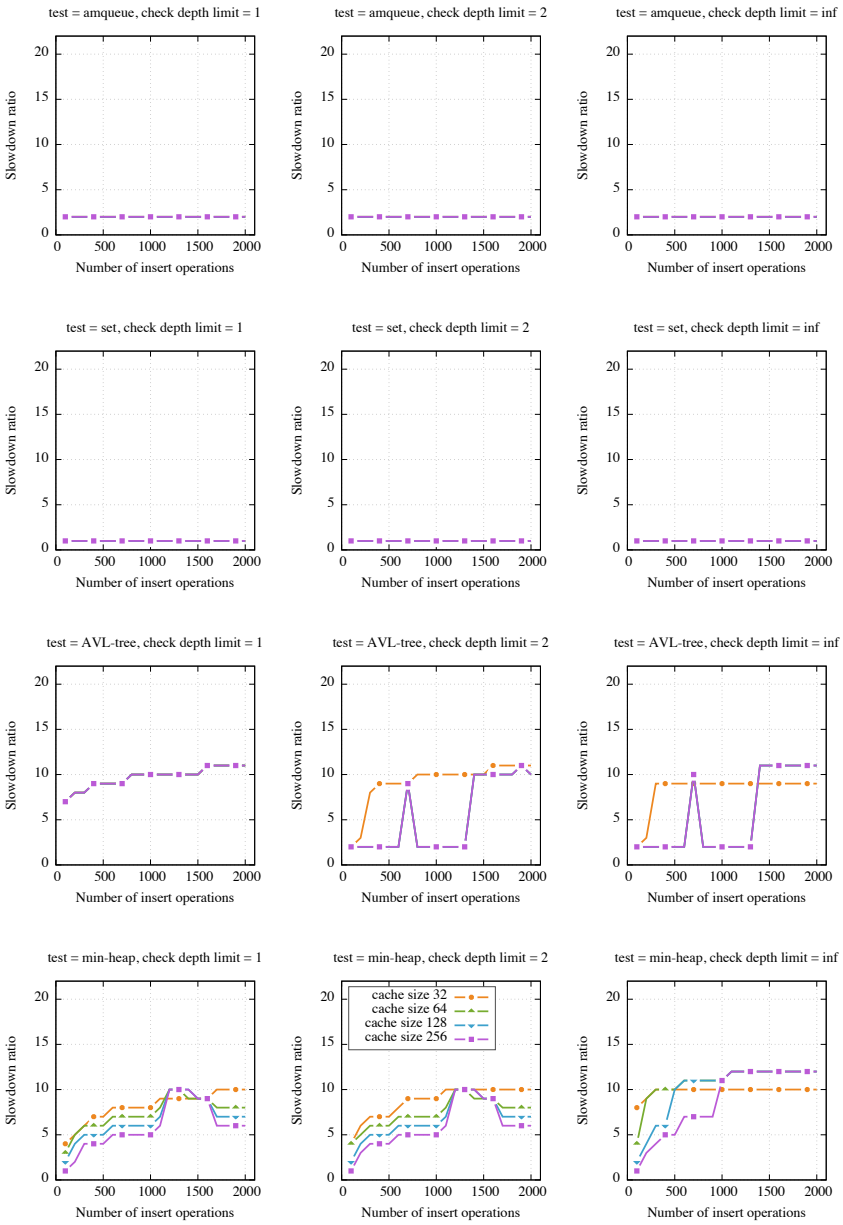


Figure B.4: Max regtype check depth for each benchmark, check depth limits 1, 2 and ∞, LRU caching policy.

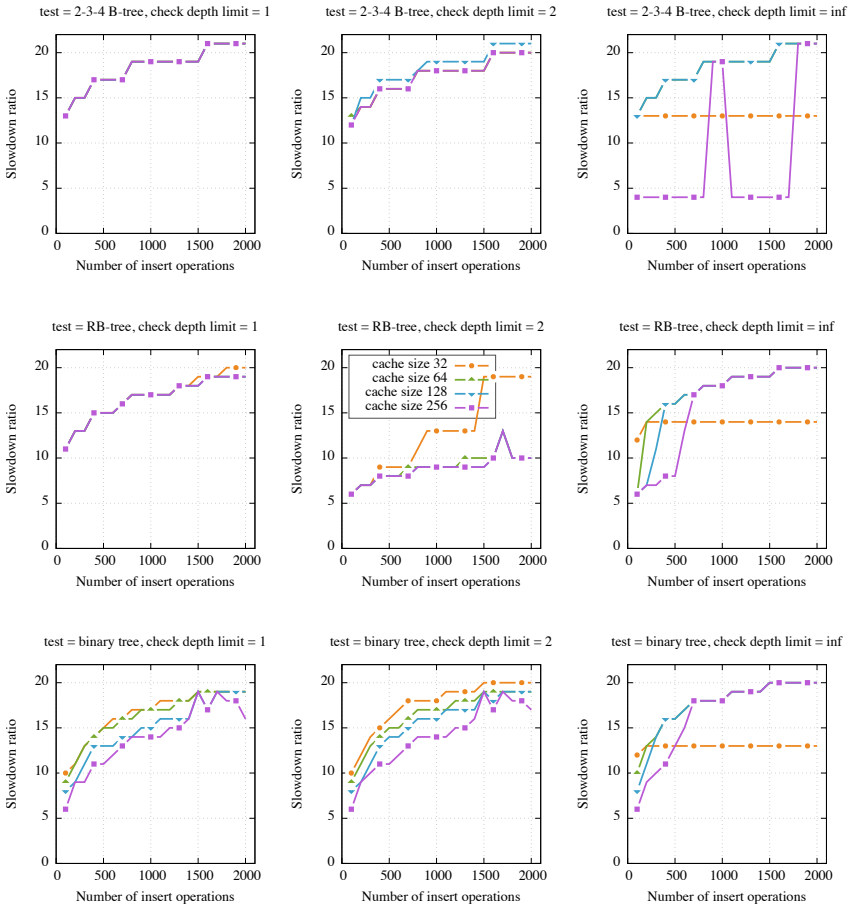


Figure B.5: Max regtype check depth for each benchmark, check depth limits 1, 2 and ∞, LRU caching policy (contd.).

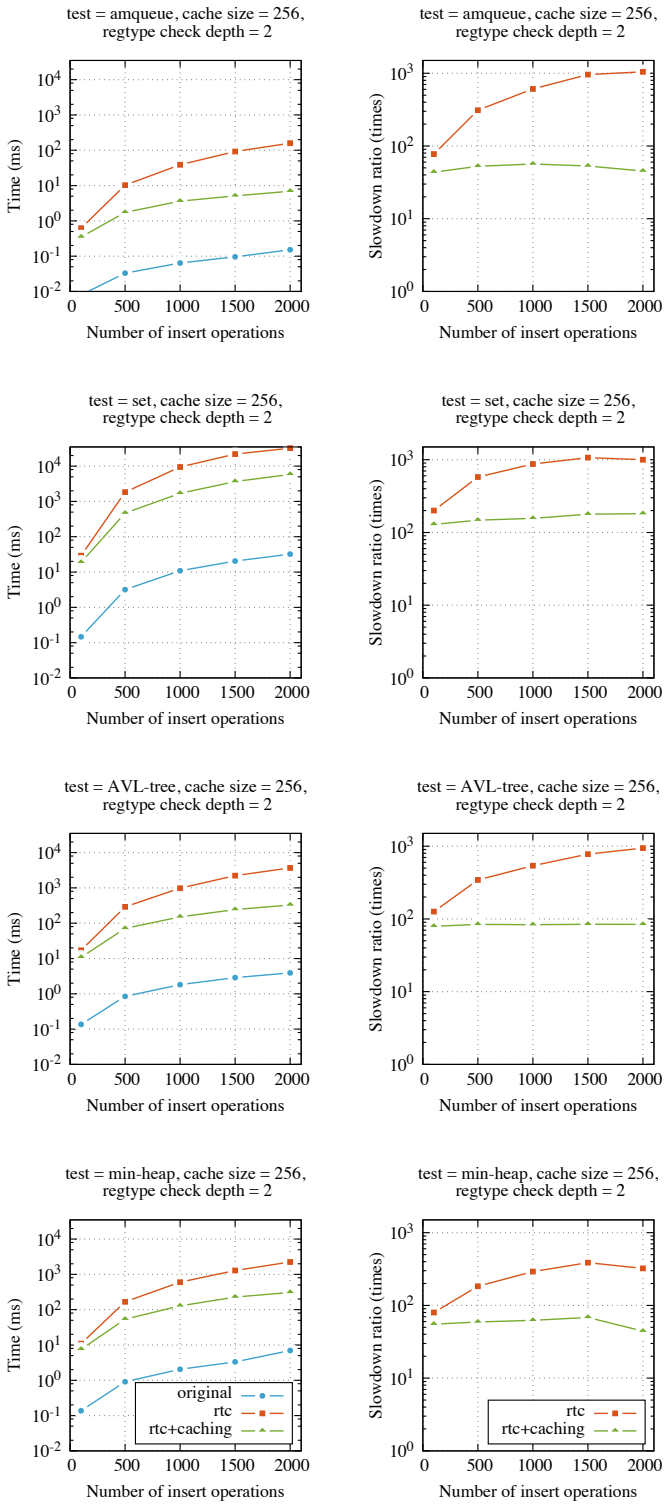


Figure B.6: Absolute and relative benchmark running times, cache size 256 elements, check depth limit 2, LRU caching policy.

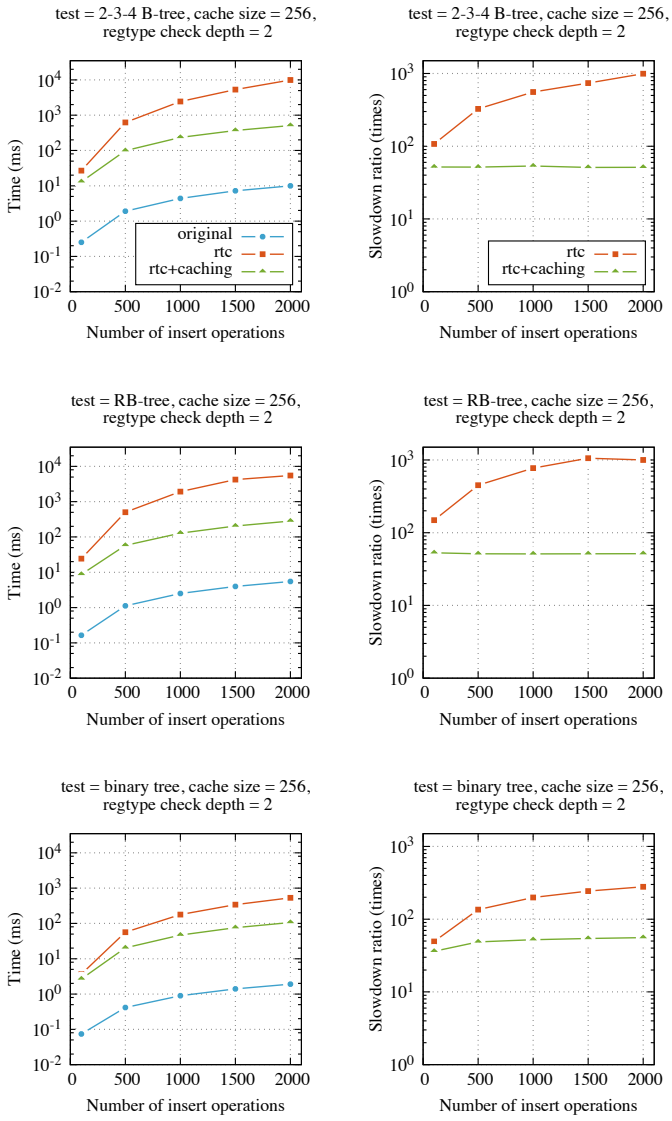


Figure B.7: Absolute and relative benchmark running times, cache size 256 elements, check depth limit 2, LRU caching policy.

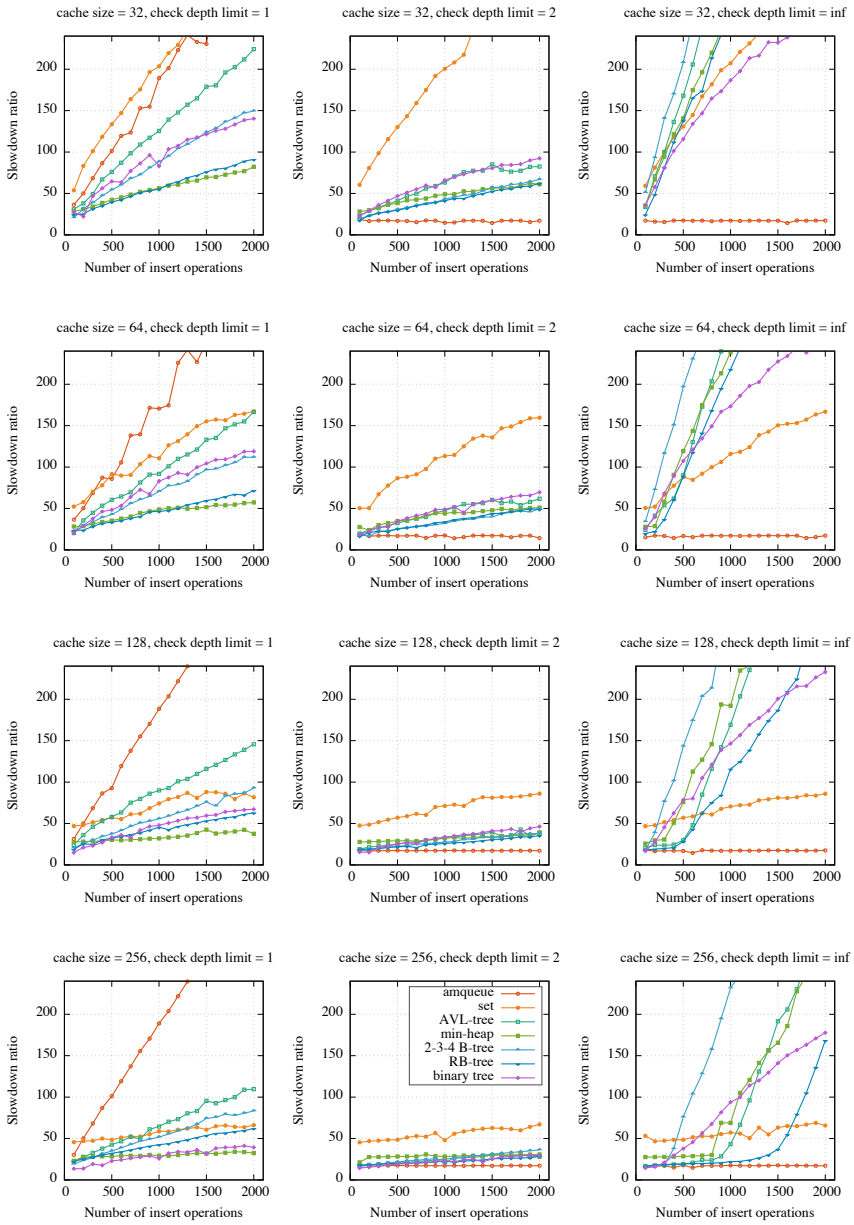


Figure B.8: Overhead ratios for all benchmarks, check depth limits 1, 2 and ∞, DM caching policy.

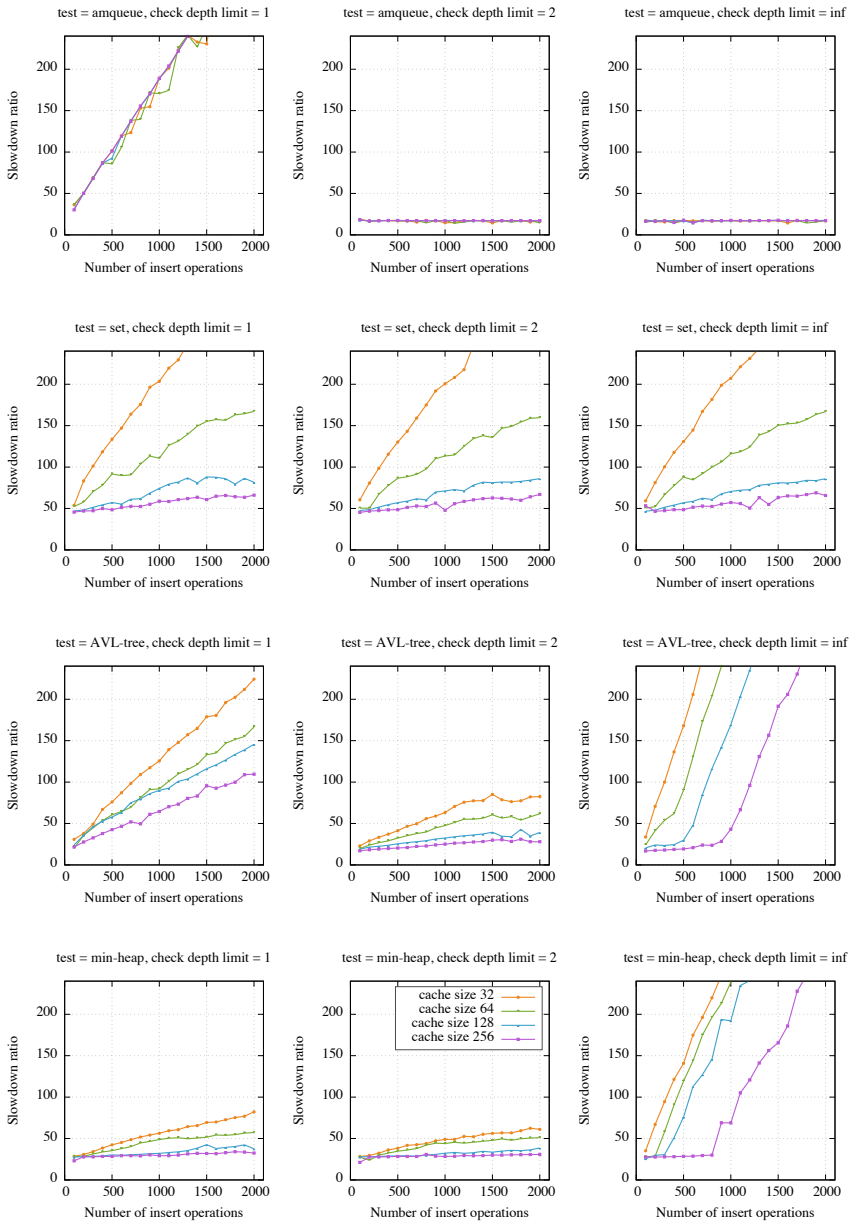


Figure B.9: Overhead ratios for each benchmark, check depth limits 1, 2 and ∞, DM caching policy.

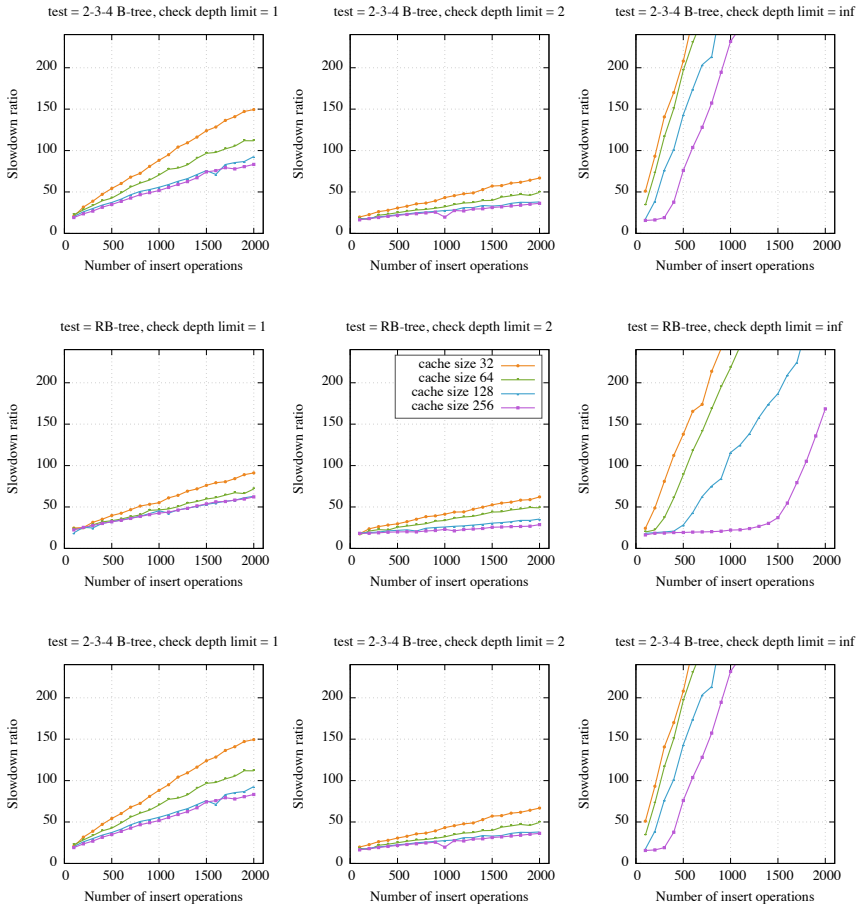


Figure B.10: Overhead ratios for each benchmark, check depth limits 1, 2 and ∞, DM caching policy (contd).

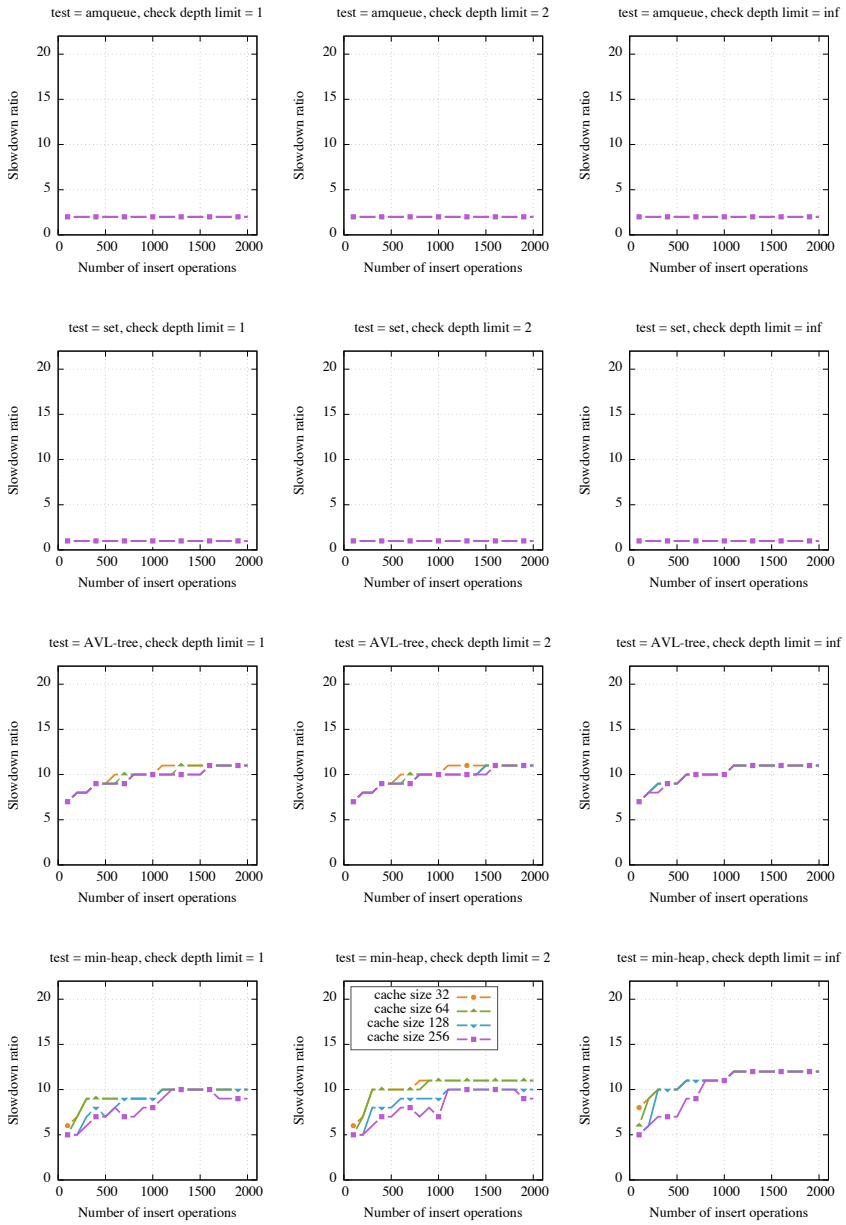


Figure B.11: Max regtype check depth for each benchmark, check depth limits 1, 2 and ∞, DM caching policy.

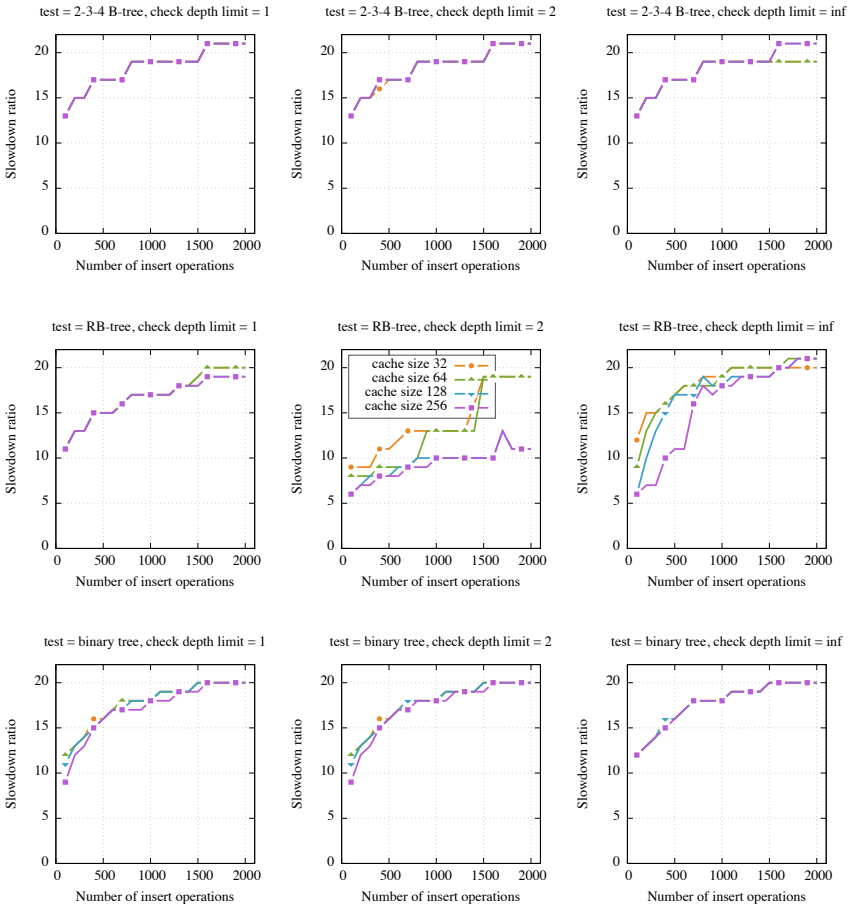


Figure B.12: Max regtype check depth for each benchmark, check depth limits 1, 2 and ∞, DM caching policy (contd.).

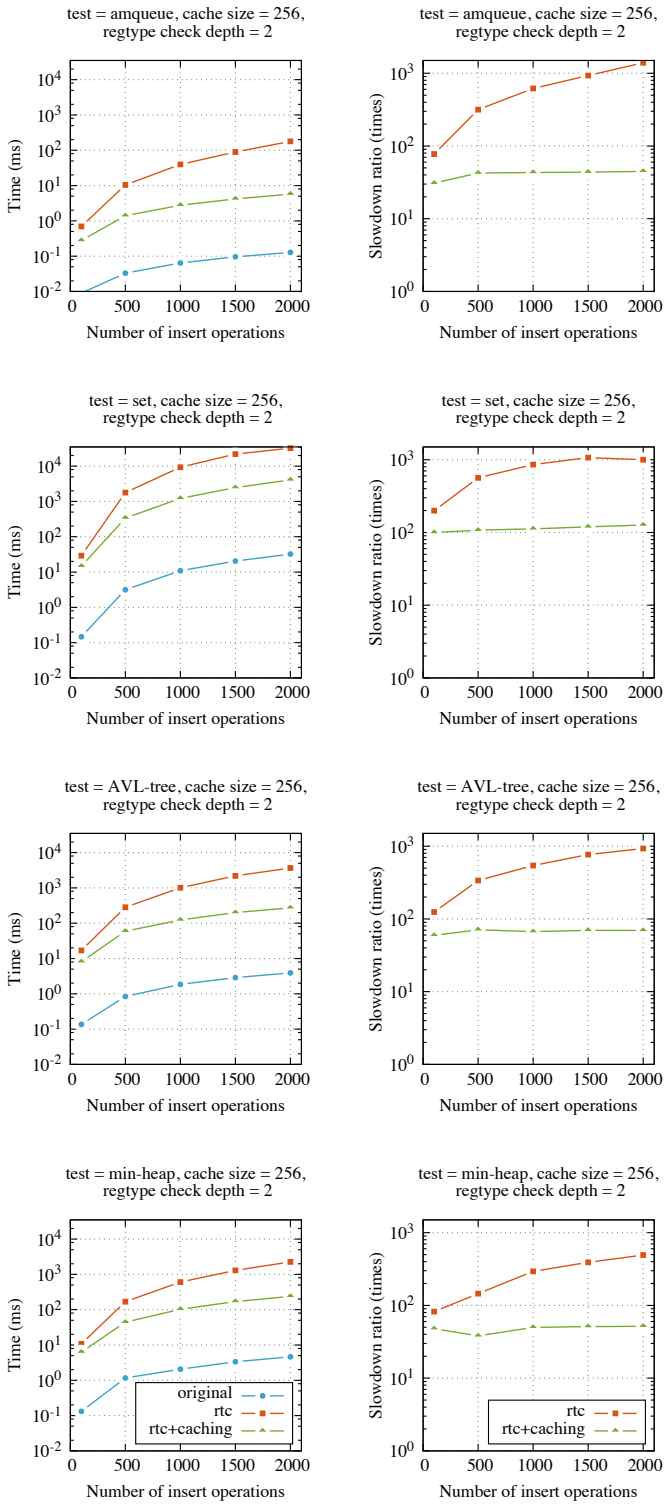


Figure B.13: Absolute and relative benchmark running times, cache size 256 elements, check depth limit 2, DM caching policy.

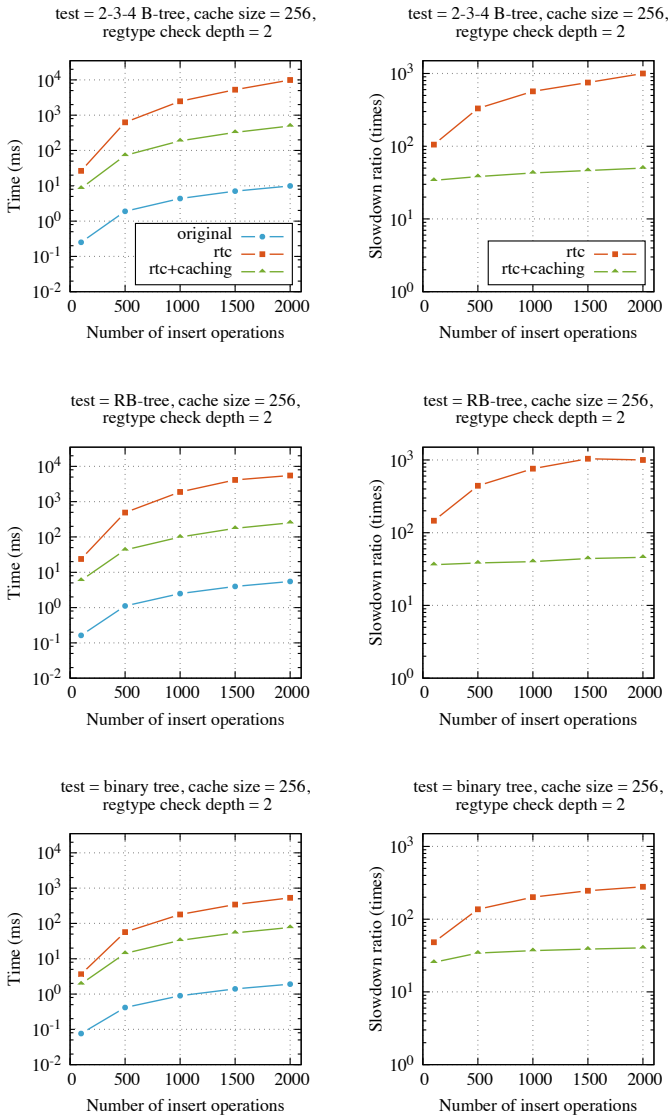


Figure B.14: Absolute and relative benchmark running times, cache size 256 elements, check depth limit 2, DM caching policy (contd.).

B.2 ADDITIONAL PLOTS FOR CHAPTER 5

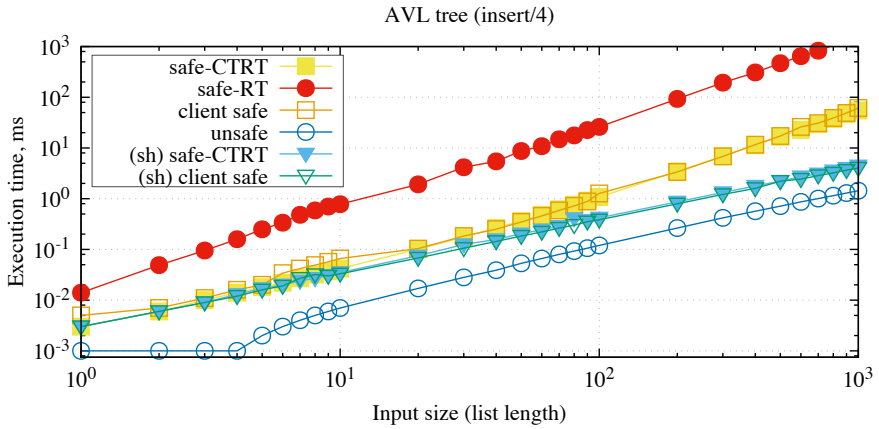


Figure B.15: Run times for the `AVL-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

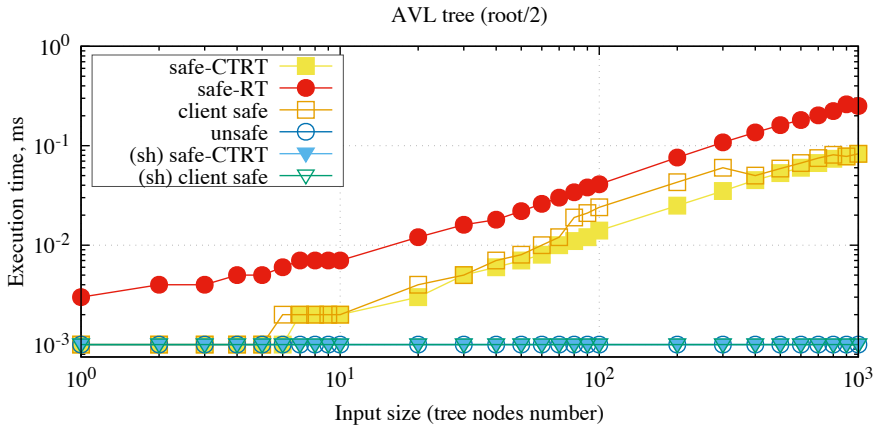


Figure B.16: Run times for the `AVL-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

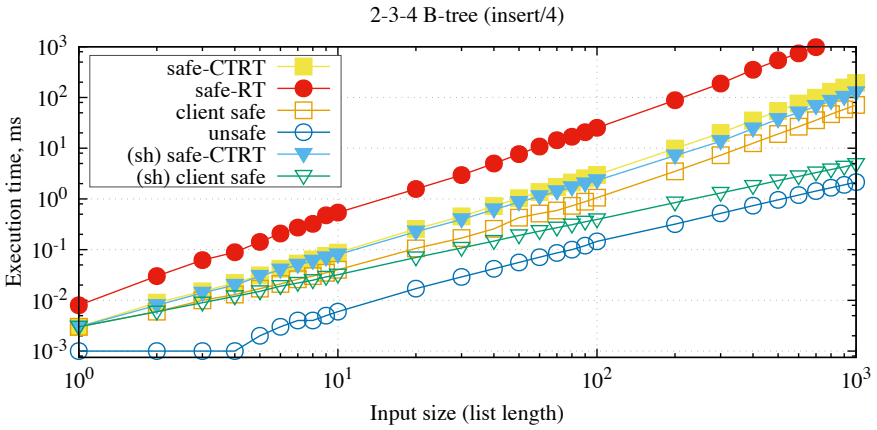


Figure B.17: Run times for the `2-3-4 B-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

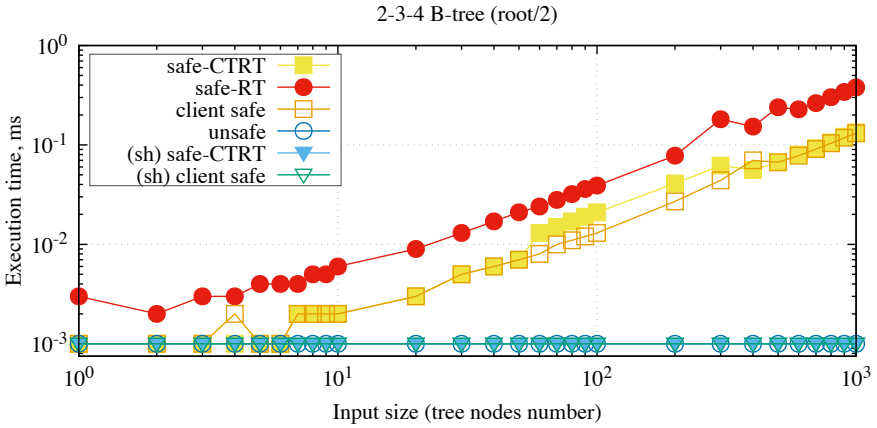


Figure B.18: Run times for the `2-3-4 B-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

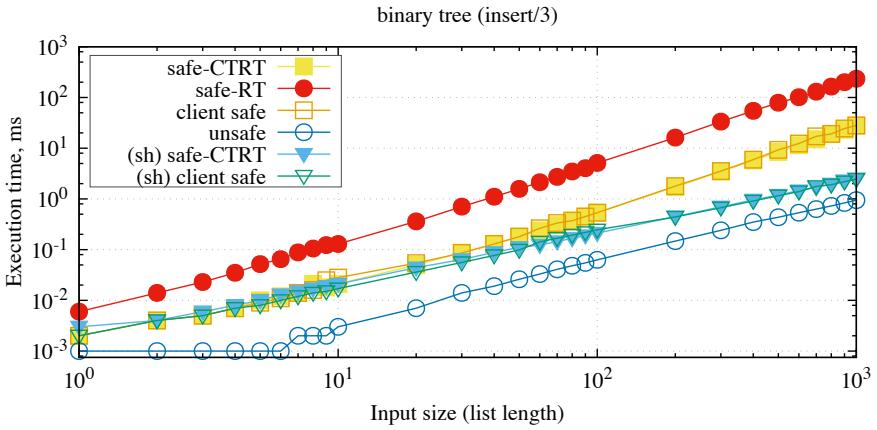


Figure B.19: Run times for the `binary tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

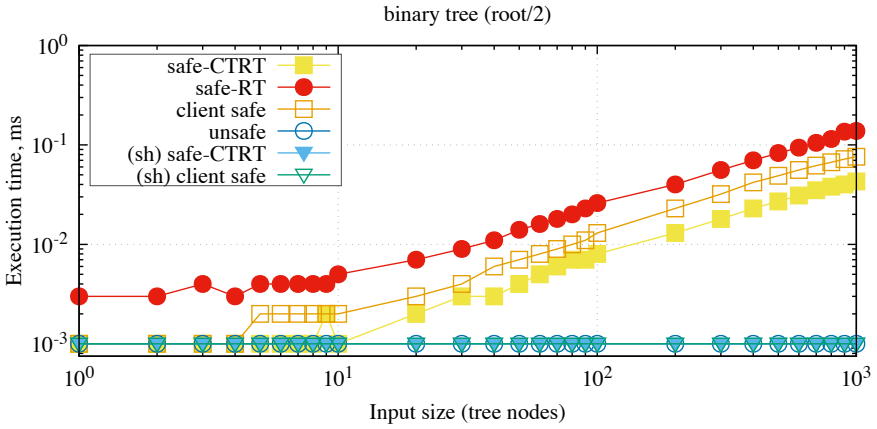


Figure B.20: Run times for the `binary tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

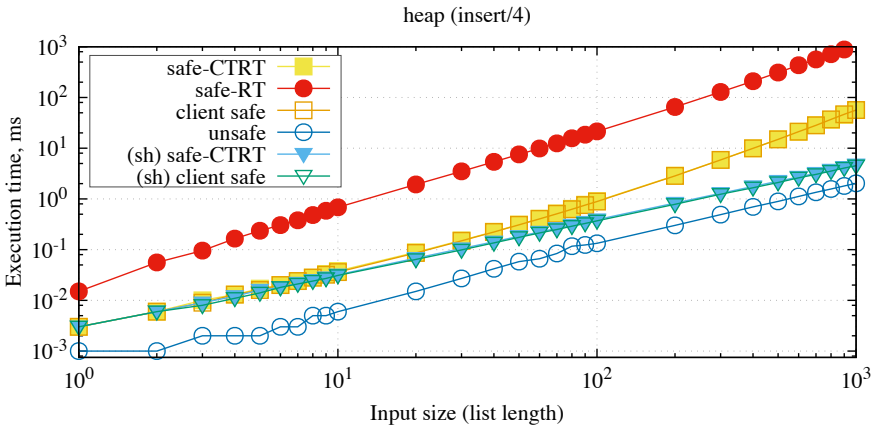


Figure B.21: Run times for the `min-heap` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

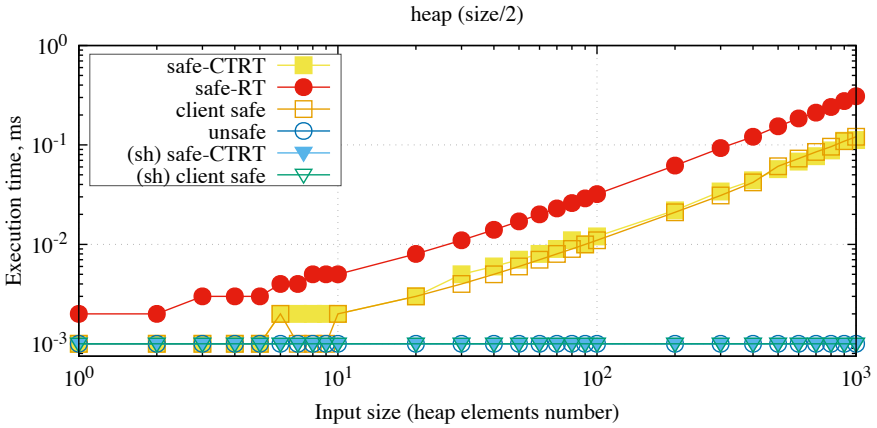


Figure B.22: Run times for the `min-heap` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

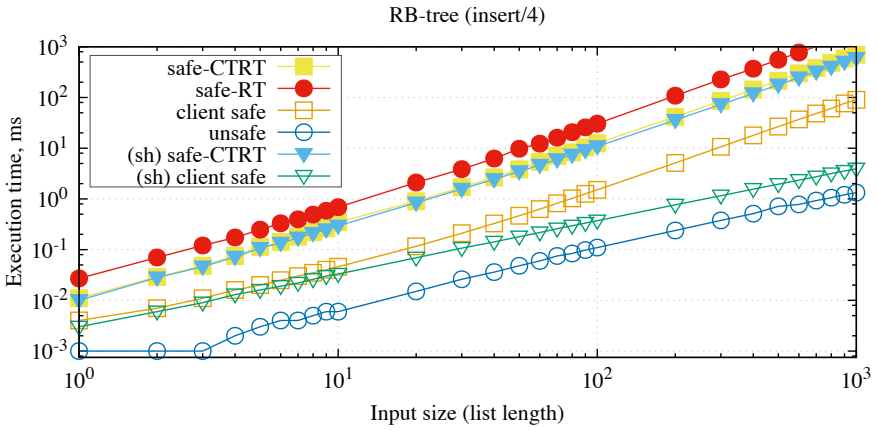


Figure B.23: Run times for the `RB-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

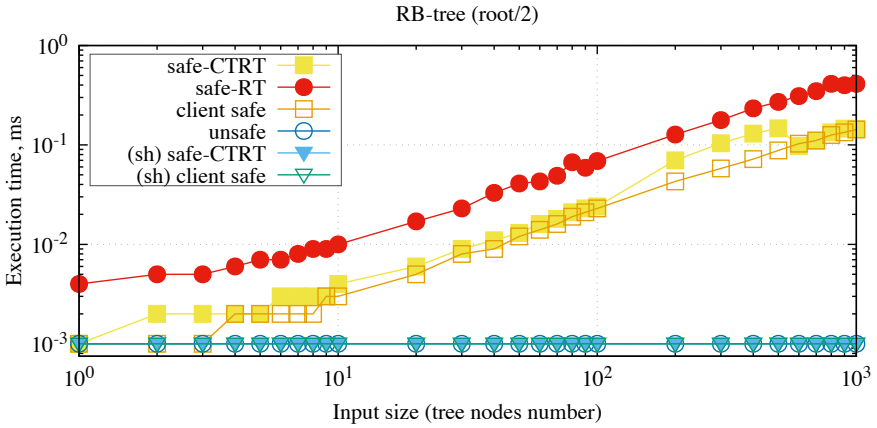


Figure B.24: Run times for the `RB-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity