BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ADVANCED STATIC PERFORMANCE ANALYSIS USING META INFER

POKROČILÁ STATICKÁ ANALÝZA VÝKONNOSTI V NÁSTROJI META INFER

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ONDŘEJ PAVELA**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

148637

Institut:	Department of Intelligent Systems (UITS)
Student:	**Pavela Ondřej, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Graphics and Interaction
Title:	**Pokročilá statická analýza výkonnosti v nástroji Meta Infer**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Study limitations of the static performance analyser Looper developed in your bachelor thesis as well as the latest developments concerning the Meta Infer framework.
2. Propose ways of significantly improving precision and/or scalability of the analysis with a stress on inter-procedural analysis.
3. Implement a new version of Looper including the proposed improvements.
4. Evaluate the new version of Looper on suitable benchmarks, including also some real-life code.
5. Describe and discuss the achieved results and their further possible improvements.

Literature:

- Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, 2020.
- Sinn, M.: Automated Complexity Analysis for Imperative Programs, PhD thesis, Vienna University of Technology, 2016.
- Bygde, S.: Static WCET Analysis Based on Abstract Interpretation and Counting of Elements, Licentiate's thesis, Mälardalen University, 2010.
- Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling Static Analyses at Facebook. Commun. ACM, 62(8):62-70, ACM, 2019.
- Çiçek, E., Bouaziz, M., Cho, S., Distefano, D.: Static Resource Analysis at Scale, In: Proc. of SAS'20, LNCS 12389, Springer, 2020.
- Çiçek, E.: Cost: Runtime Complexity Analysis. Available online at https://fbinfer.com/docs/next/checker-cost. [Checked on 3/10/2022.]

Requirements for the semestral defence:
The first point and at least some work on the second and third points.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	3.11.2022

Abstract

Looper is a static *complexity analysis* tool for inference of *tight upper bounds* on the *execution cost* of programs. It is based on the previously existing Loopus tool which used *abstract program model of difference constraints* (inequalities of the form $x' \leq y + c$), which allows for natural abstraction of common loop counter updates $x = x + c \longmapsto x' \leq x + c$ and $x = y \longmapsto x' \leq y + 0$. Looper was initially proposed and implemented in author's bachelor's thesis as a *checker* for the *Meta Infer framework* but the tool failed to meet the expectations when tested on *real-world* code. This master's thesis proposes a new improved version of Looper that aims at solving the main limitations of the original tool, namely through introduction of *interprocedural analysis*. Additionally, various extensions targeting improved precision of the intraprocedural analysis, such as new abstraction algorithm, handling of compound loop conditions and more, were implemented. Moreover, logging, issue reporting and collection of results has been significantly improved. Finally, through extensive experiments with the new Looper version, the ability to analyze *real-world* code in a more general, scalable and precise way was shown.

Abstrakt

Statický analyzátor složitosti *Looper* slouží pro odvozování *přesných horních mezí* ceny vykonání programů. Jako teoretický základ byl využit dříve existující nástroj Loopus a jeho *abstraktní programový model* využívající tzv. *difference constraints* (nerovnosti typu $x' \leq y + c$), které umožňují přirozeným způsobem modelovat typické modifikace počítadel cyklů $x = x + c \longmapsto x' \leq x + c$ a $x = y \longmapsto x' \leq y + 0$. Looper byl původně navržen a implementován v rámci autorovy bakalářské práce jako zásuvný modul aplikačního rámce *Meta Infer*. Výsledný nástroj nicméně nenaplnil očekávání při pokusech o jeho nasazení na *reálné programy*. Tato diplomová práce představuje návrh nové verze, která si dává za cíl odstranit hlavní limitace původního nástroje Looper, zejména díky nově podporované *interprocedurální analýze*. Dále byla implementována řada rozšíření, které cílily na zvýšení přesnosti intraprocedurální analýzy, jako např. nový abstrakční algoritmus, podpora pro složené podmínky v hlavičkách smyček a další. Kromě toho bylo také výrazně vylepšeno logování, hlášení chyb a sběr výsledků analýzy. Na závěr byla skrze skrze rozsáhlé experimenty demonstrována schopnost nové verze nástroje Looper analyzovat *reálný kód* obecnějším, škálovatelnějším a přesnějším způsobem.

Keywords

Meta Infer, Static analysis, Bound analysis, Complexity analysis, Amortized analysis, Cost Analysis, Difference constraints, Incremental analysis, Modular analysis, Differential analysis, Complexity degradation, Compositional analysis, Interprocedural analysis, Scalability, Looper, Loopus, Imperative programs

Klíčová slova

Meta Infer, Statická analýza, Analýza mezí, Analýza složitosti, Amortizovaná analýza, Analýza ceny, Inkrementální analýza, Modulární analýza, Rozdílová analýza, Degradace složitosti, Kompoziční analýza, Interprocedurální analýza, Škálovatelnost, Looper, Loopus, Imperativní programy

Reference

PAVELA, Ondřej. *Advanced Static Performance Analysis Using Meta Infer*. Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Zákeřné chyby ukrývající se na nečekaných místech a způsobující závažné škody jsou bohužel neodmyslitelnou součástí vývoje softwaru již od nepaměti. V reakci na tento problém se výzkumníci v několika posledních desetiletích zabývali vývojem nových nástrojů, které by — když už ne eliminovali — tak alespoň omezily vznik nových chyb v kritickém softwaru. Většina pozornosti se ovšem v minulosti upínala zejména k vývoji nástrojů pro odhalování tzv. funkčních chyb, které mohou přímo ovlivnit schopnost programu vykonávat jeho zamýšlenou funkci.

Výkonnostní chyby byly až donedávna vnímány jako méně kritické a obdobné nástroje pro odhalování těchto chyb byly proto rozvíjeny pomaleji. To vedlo k nedostatku spolehlivých nástrojů ve chvíli, kdy se začalo ukazovat, že závažnost výkonnostních chyb je srovnatelná s chybami funkčními. V extrémních případech mohou tyto chyby vést prakticky k nepoužitelnosti programů, zejména při práci s větším objemem (a nebo jiným typem) dat, než bylo očekáváno. Takové chování je nepřijatelné, zejména dnes, kdy se klade velký důraz na dobrou uživatelskou zkušenost. V současnosti se pro odhalování výkonnostních chyb nejčastěji využívají dynamické profilační nástroje, které jsou v mnoha případech dostačující. Tyto nástroje nicméně ze své podstaty mohou zanechat spoustu výkonnostních problémů neodhalených. Alternativou jsou poté statické analyzátory jako např. Coverity nebo CodeSonar, které mohou nabídnout komplementární přístup, taktéž se svými problémy — jako např. potenciální hlášení *falešných chyb*, tzv. "*false alarms*". Vetšinou jsou tyto nástroje ovšem proprietární a není možné je jednoduše vyhodnotit nebo rozšířit o vlastní analýzy.

V reakci na současnou situaci proto společnost Meta nedávno představila vlastní řešení nazvané *Meta Infer*: nástroj s otevřeným zdrojovým kódem pro vývoj kompozičních, inkrementálních a interprocedurálních statických analyzátorů, které jsou v důsledku tedy i vysoce škálovatelné. Nástroj Meta Infer v posledních letech zažil období rychlého vývoje a je v současnosti stále rozvíjen mnoha týmy po celém světě. Kromě toho je aktivně využíván pro odhalování chyb nejen samotnou společností Meta, ale mnoha dalšími velkými společnostmi jako např. Spotify, Uber, Mozilla nebo Amazon. Nástroj Meta Infer v současné době disponuje rozmanitou řadou analýz pro odhalování široké škály softwarových chyb, jako např. přístupy mimo meze polí ("buffer overruns"), uváznutí ("deadlock") a stárnutí ("starvation") ve vícevláknových programech, dereference nulových ukazatelů ("null pointer dereference"), úniky paměti ("memory leak") a mnoho dalších chyb souvisejících s bezpečnou prací s pamětí ("memory safety"). Zároveň nástroj Meta Infer představuje aplikační rámec pro rychlý a jednoduchý vývoj nových analyzátorů.

Nicméně Infer v současnosti stále zaostává v oblasti výkonnostních chyb přestože nabízí poměrně pokročilý analyzátor Cost — jediný dostupný výkonnostně zaměřený analyzátor. Tento analyzátor implementuje upravenou verzi tzv. *worst-case execution time* (WCET) analýzy založené na disertační práci Stefana Bygdeho [8]. Tento typ analýzy ovšem poskytuje pouze těžko interpretovatelnou a často (v případě složitějších algoritmů zahrnujících *amortizovanou složitost*) poměrně nepřesnou numerickou mez na čas potřebný k vykonání programu. Kromě toho se Cost zaměřuje zejména na programy napsané ve vyšších programovacích jazycích jako např. Java a jeho hlavním cílem není nutně odvozování přesných mezí, ale spíše rychlé odhalování výkonnostních degradací na základě rozdílové analýzy ("differential analysis") mezi více verzemi jednoho programu.

V rámci bakalářské práce autora (kdy analyzátor Cost ještě neexistoval) byl proto navržen a implementován nástroj *Looper* — statický analyzátor pro automatické odvozování mezí složitosti programů, který byl současně implementován jako zásuvný modul aplikačního rámce Meta Infer. Princip nástroje Looper byl založen na již dříve existujícím nástroji Loo-

pus [32], který v té době (dle našeho nejlepšího vědomí) byl jediným nástrojem schopným analyzovat amortizovanou složitost u široké škály programů. První verze nástroje Looper implementovala veškeré stěžejní algoritmy originálního nástroje Loopus a byla schopna analyzovat i komplikované, uměle vytvořené programy. I přesto však měla první verze několik kritických nedostatků. V první řadě Looper nebyl schopen (primárně kvůli technickým nedostatkům) analyzovat prakticky žádný reálný kód kromě uměle vytvořených příkladů. Zároveň Looper neobsahoval žádné mechanismy zotavení z chyb a v případě selhání analýzy jedné funkce došlo k pádu celého nástroje Infer, což prakticky znemožňovalo analýzu jakéhokoliv většího programu. Druhým hlavním nedostatkem byla chybějící podpora pro interprocedurální analýzu, jelikož původní nástroj Loopus byl pouze intraprocedurální a volání funkcí nebral v potaz (v rámci testování autoři manuálně vkládali kód volaných funkcí).

V rámci této diplomové práce byl nástroj Looper výrazně vylepšen a rozšířen. Hlavní snahou bylo nástroj vylepšit tak, aby bylo možné analyzovat reálný kód. Konkrétně byl proto od základu přepracován algoritmus pro konstrukci grafů založených na abstraktním modelů nazvaném *difference constraint program* (DCP). Dále byla výrazně vylepšena interpretace instrukcí Infer SIL mezikódu, což umožnilo základní podporu pro práci s datovými strukturami a ukazateli, které jsou masivně využívány v nízkoúrovňovém C kódu. Kromě toho byla také implementována široká řada vylepšení za účelem zpřesnění intraprocedurální analýzy, např. podpora složených podmínek v hlavičkách smyček nebo rozšíření původní omezené formy tzv. *difference constraints* (DC) nerovností. Ty původně podporovaly pouze výrazy typu $x \leq y + \mathtt{c}$, které nebyly schopny zachytit dekrementace pomocí jiných operátorů jako např. dělení nebo pravých bitových posunů. Hlavním přínosem nicméně zůstává implementace rozšíření pro interprocedurální analýzu, která umožnila otestování nástroje Looper na rozsáhlém skutečném kódu a zvýšila jeho šance pro jeho budoucí nasazení v praktický podmínkách.

V závěru byla veškerá navržená a implementovaná rozšíření nástroje Looper úspěšně otestována a experimentálně vyhodnocena nejen na ručně vytvořených, ale také na reálných programech. V porovnání s první verzí nástroje došlo k výraznému posunu, zejména ve schopnosti analyzovat skutečný kód, což bylo dříve prakticky nemožné. Zároveň se potvrdilo, že nástroj Looper skutečně je schopen dobře škálovat na rozsáhlém a komplikovaném kódu i při podpoře interprocedurální analýzy. Evaluace na rozsáhlých reálných programech poté dále odhalila, že Looper je výrazně rychlejší než aktuální verze analyzátoru Cost za cenu nižší přesnosti. Analýza výsledků ovšem naznačila, že nižší přesnost je v mnoha případech zaviněna pouhými technickými problémy a nikoliv inherentní limitací použitého algoritmu. V současnosti je proto možné přesnost dále zlepšovat mnohými způsoby, ať již odstraňováním technických nedostatků, či implementací dodatečných rozšíření z původního nástroje Loopus [32]. V neposlední řadě, implementace rozdílové analýzy a podpory pro další programovací jazyky představuje další způsob jak nástroj Looper v budoucnu vylepšit.

Vývoj původní verze nástroje Looper byl zahájen v rámci projektů H2020 ECSEL AQUAS a Arrowhead Tools. Vývoj nové verze je podporován projekty H2020 ECSEL VALU3S, GAČR (Czech Science Foundation) Snappy 20-07487S a AIDE 23-06506S. Práce na projektu byla v počátcích taktéž diskutována s vývojáři nástroje Meta Infer.

Advanced Static Performance Analysis Using Meta Infer

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of professor Tomáš Vojnar. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Pavela
May 24, 2023

Acknowledgements

I would like to express my sincere and utmost gratitude to my supervisor Prof. Ing. Tomáš Vojnar, Ph.D. for his supervision, consultations, and his expert advice over the course of this work. A special thanks goes to my brother Ing. Jiří Pavela for his invaluable writing advice as well as words of encouragement. Last but not least, a sincere and special gratitude goes to my girlfriend, family, and friends who had to put up with my rants, for their support and everlasting patience during the period I invested all my time and energy into this work.

Contents

Chapter 1

Introduction

Subtle bugs hiding in unexpected places and causing significant damage when triggered are an inherent part of software ever since the inception of the programming discipline. In response to this problem, researchers in the last few decades focused their attention on developing new tools with the primary goal of reducing the number of bugs or even proving their absence in critical software. However, most of the attention was drawn towards the field of the so-called functional bugs which can directly affect the ability of a program to perform the intended function.

Until recently, *performance bugs* were not regarded as critical and remained at the sidelines of research, resulting in a lack of reliable tools when it became apparent that the severity of performance bugs is comparable to the severity of functional ones. In extreme cases, these bugs can turn otherwise correct programs into unusable pieces of software when met with an unexpected amount and/or pattern of input data. This behaviour is unacceptable especially with today's emphasis on great user experience.

The current widespread approach is to employ extensive automated testing and leverage *dynamic analysis* tools such as *profilers* in order to catch bugs early in the development process. However, despite their undisputed usefulness, the capabilities of automated testing are directly tied to the quality of manually written tests, and profilers are able to provide performance characteristics related to specific input data only. Unfortunately, performance bugs tend to manifest in later development stages or upon deployment due to a previously unanticipated workload. In conclusion, approaches based on dynamic analysis are sufficient in many cases but can sometimes still miss critical errors and cannot provide any conclusive claims about certain properties of a program.

Static analysis offers an alternative solution which usually does not require any additional user input and can be easily employed in early development stages as it does not rely on the executability of a program. However, even static analysis has its own shortcomings such as a traditionally high rate of *false alarms*, and, most notably, a prevailing problem with *scalability* which plagues most of the current tools and renders them unusable for large and quickly changing codebases, at least when more involved classes of bugs — such as those related to performance — are of concern.

As a response, Meta has recently proposed its own solution for efficient static analysis and bug finding called *Meta Infer* — a *compositional*, *incremental*, and consequently highly *scalable* static analysis framework suitable for quick integration of new *inter-procedural* analyses. Since then, it has been successfully deployed in many large companies such as Spotify, Uber, Mozilla, and of course Meta. It currently offers a wide range of analyses, e.g.,

for detection of buffer overruns, deadlocks, data races, and many other issues. Notably, it also offers the Cost checker which performs a *worst-case execution time* analysis.

Looper In author's previous work [27], a static analyzer for automated complexity analysis (based on the previously existing tool Loopus [32]) called Looper was proposed and later implemented as an individual analysis module in the Meta Infer framework. The key distinctive feature of Looper which set it apart from other state-of-the-art tools and the existing Infer Cost checker is the ability to perform *amortized complexity analysis*. Often times, it allowed Looper to infer significantly more precise upper bounds on loop complexities compared to the Cost checker as was demonstrated in the experimental evaluation in [27].

Unfortunately, the first version of Looper suffered from many limitations, the most crucial one being the lack of support for *interprocedural* analysis. It not only affected the precision of Looper but also prevented it from analyzing certain programs as using side-effects or function return values to modify loop counter values is fairly common in real-world code. Further, it did not perform well outside of simple code due to insufficient abstraction algorithm. Moreover, *differential analysis* in the spirit of Cost checker was not implemented yet, and thus, the first version of Looper had no way of detecting complexity degradations.

New Looper Enhancements Within this thesis, multiple significant improvements to Looper were proposed and implemented. In particular, *interprocedural* analysis using Infer summaries was proposed and implemented. Furthermore, the inadequate abstraction algorithm, originally implemented in the Infer abstract interpretation framework, has been completely rewritten and the program expression abstraction process was significantly improved as well. Moreover, several other proposed improvement ideas such as better logging, issue reporting, handling of compound loop conditions, and many more smaller enhancements were implemented.

Outline of the Thesis The rest of this work is structured as follows: Chapter 2 introduces the fundamental theory behind program analysis and discusses several important concepts such as *scalability*, *soundness*, and *completeness*. It also provides a concise overview of several existing approaches to program analysis. Special attention is given to the *abstract interpretation* technique as its currently one of the most commonly used approaches and also because it is used at the core of the Infer framework which is also briefly discussed. Chapter 3 discusses both the core concepts and main limitations of the first version of Looper. Subsequently, Chapter 4 presents all of the proposed extensions and improvements. Furthermore, Chapter 5 covers the implementation of these extensions. The experimental evaluation of these extensions and new features on hand-crafted examples as well as read-world code is covered in Chapter 6 along with a brief discussion of future work. Finally, Chapter 7 concludes this thesis. Additionally, Appendix A specifies the content of the attached memory media and Appendix B provides a short installation and user manual. Also, note that parts of the thesis concerning the preliminaries and the description of the first version of Looper are partially taken from the previous work [27].

Acknowledgement The development of the first version of Looper was launched under the H2020 ECSEL projects AQUAS and Arrowhead Tools. The development of the new

version has been supported by the H2020 ECSEL project VALU3S, GAČR (Czech Science Foundation) Snappy 20-07487S, and AIDE 23-06506S. The project was also discussed with the developers of Meta Infer in its initial development phase; we hereby thank for the received support.

Chapter 2

Preliminaries

This chapter introduces the core concepts and notions that this work builds upon and which are necessary for clear understanding of the following chapters. It should be noted that parts of this chapter were partially taken from the author's bachelor's thesis [27] and the previously published student conference paper [28].

The rest of this chapter is structured as follows: the first Section 2.1 provides simple and concise explanation of various terms frequently used in this work and other literature related to program analysis. The goal of this section is to make the rest of this work reasonably self-contained. The following Section 2.2 will provide a high-level overview of the widely used static analysis technique called Abstract Interpretation (AI). It is important to note that since the original version of Looper was presented, the author has decided to abandon the use of AI for various reasons which will be discussed in Chapter 3.2. Despite this fact, *Looper* is still built upon the Meta Infer infrastructure which contains general AI framework used by most of the analyses. Thus, it was deemed appropriate to include at least a gentle introduction to the theory behind this static analysis technique. The last section of this chapter focuses on the Meta Infer tool itself and gives insight into its architecture and AI framework.

2.1 Fundamentals of Program Analysis

This section provides an overview of the fundamental concepts and questions related to program analysis. The goal of this section is to build an intuition of

- *what* is program analysis,
- *what* are the fundamental limitations,
- *what* are the main questions we ask about the analyzed program and,
- *how* do we try to answer them with respect to the computational limits.

Additionally, basic principles, limitations and trade-offs of various widely used approaches to program analysis will be discussed at the end of this section. This section is based on the introductory chapters from [29] and the overview paper [20].

2.1.1 Analysis Goal

Arguably the most important question to answer before anything else is: *what do we analyze?* More precisely, the question is twofold: *what programs* to analyze and *what properties*

are we trying to determine. The answers to these questions constitute basic characterization criterion for program analyses.

Classification based on Analyzed Programs

The first and most obvious way to characterize analyzed programs is by the programming language they have been written in. However, with the onset of modern analysis frameworks such as Meta Infer, the importance of specific language (within one programming paradigm) for characterization of analyses is diminished. These frameworks [31, 10] are usually built upon the LLVM compiler infrastructure [22] which allows them to work with one language-independent intermediate representation (IR). LLVM currently supports many major programming languages, including several modern ones such as Rust or Go. All of these languages can be compiled into one IR which can then serve as a basis for generating various different custom program models for further analysis. With this approach, at least in theory, developers can easily adapt their analysis tool for multiple different languages. Unfortunately, supporting features unique to specific languages more often than not still requires considerable effort, contrary to what theory tells us. Nevertheless, this approach is still less resource intensive than creating multiple analysis tools tailored specifically for each target language.

Issues usually arise when it comes to supporting multiple languages across different paradigms, such as imperative and functional. These paradigms differ vastly in the philosophy behind language features and in their compilation or interpretation pipelines which detracts engineers from the effort of creating such general frameworks. Moreover, bugs that users of these languages have to face on daily basis also differ vastly. To name one such example: the prominent and ever-present *off-by-one* buffer overrun bug encountered by novice and seasoned software engineers alike. Such bugs are extremely prevalent in imperative and object oriented languages but virtually non-existent in *type-safe* functional languages such as *OCaml* or *Haskell*[1]. Consequently, developing general analysis frameworks for languages across different paradigms is not only hardly feasible in practice but arguably meaningless in the first place.

A more meaningful way of characterizing analyses is whether they are *domain-specific* or not.

- **Domain-specific**: certain programs are written with a very specific purpose and requirements based on the target *domain*. These common characteristics divide programs into families and analysis designer can therefore leverage assumptions about certain family to create a more efficient and possibly more precise analysis.

  An example can be a comparison between embedded software and a common web application. The former will be of comparatively small scale (few thousands of lines), likely written in C language, using limited set of language features (no dynamic allocations or recursion), and possibly deployed in *safety-critical* environment such as controller on a plane.

  The latter can span from hundreds to millions of lines of code and might be written in numerous different high-level programming languages such as JavaScript, Java, Python or Haskell to name a few. These applications will likely involve use of class

[1] However, contrary to popular belief, *Haskell* is not as *type-safe* as it might seem [34] and one can encounter infamous `Segmentation fault` errors common to languages conventionally perceived as *unsafe* such as C.

inheritance and virtual functions among many more available complex language features. Moreover, in most instances, safety will be of little concern as the worst that can happen is a temporary unavailability of the service.

Clearly, these are two completely different types of programs with different sets of requirements and goals for potential analysis tool. Focusing on a specific domain is thus a pragmatic way to achieve more precision and lower cost by sacrificing the generality.

- **Non-domain-specific**: other analyses go in the opposite direction and are designed to be used on all kinds of programs written in a specific language. These analyses are frequently used inside compilers and their usual primary objective is to collect more information about the compiled programs which can then be leveraged for example in the optimization phase. It can also help developers to discover errors common to all kinds of programs such as buffer overruns. The main requirement is to have acceptable performance for wide range of input programs and as such the usual trade-off is lower precision across the board.

Another common way how to characterize analyses is by examining how they perform the analysis over the input program. The options are to either perform the analysis directly over program's source code, i.e., mimicking the work of compiler or creating some sort of semantic model which can then be used as an input for the analysis tool:

- **Program-level analyses**: these analyses take the source code of a program which can be written any of the conventional languages such as C or Java but it is also possible to analyse code describing hardware, i.e., VHDL or Verilog. In any case, the source code is first processed by some kind of compiler-like front-end (or it can *really* be a compiler such as LLVM) into *abstract syntax tree* which can later be used to generate IR or even other representations such as *control-flow graphs*. Moreover, these front-ends can also generate other useful data structures like *call-graphs*. Apart from source code, *program-level* analysis can also be run on executable binary of a program.

- **Model-level analyses**: another option is to create a program *model* which tries to capture semantics relevant to the analysis and then use this model as an input. The primary motivation behind this approach is the option to abstract away some of the complexity and implementation details of the source code during the *modeling* phase which should make subsequent analysis easier and possibly more efficient. The downside of this method is the need to create the *model* in the first place and prove its semantic correspondence with the original program which might be difficult. The creation process can be either manual (potentially very tedious) or automatic via specialized tools which extract a model out of the source code. Either way this introduces a middle step that might be a source of imprecision or bugs in the following analysis. Few examples of possible models are automata, Petri nets, Markov chains and other more specialized models such as *difference constraint model* which will be discussed later.

Classification based on Target Property

Based on the answer to the question "*what do we ask about the analyzed program?*" it is possible to characterize analyses into several different families. In other words, what *seman-*

tic property is the tool trying to compute divides programs into certain groups. Originally introduced by Leslie Lamport in late 1970s, the two most common target property classes are *safety* and *liveness*. However, the classification of semantic properties became more complicated over the years with the introduction of new classes and due to refinement of existing ones which resulted in the creation of various subclasses [9]. To name a few examples, other property classes involve *information flow*, *reachability*, *fairness* (special case of *liveness*) and others.

Informally, *safety* property [33] states that "*something bad never happens*" or in other words that program will never reach an erroneous state. More precisely, a safety property imposes a requirement upon the program which should be maintained throughout its entire run until the termination point [21]. I.e., this requirement must not be violated in any of the program's states in a finite run. Consequently, if property might be violated by a program it will *always* be possible to observe a counter-example in a finite run. One of the most typical examples of a safety property in the computer science domain is mutual exclusion. If this property holds then it guarantees that "*no two or more distinct concurrent processes will enter their shared critical section simultaneously*". Another very common safety property is that *buffer overflow* will never occur. A real world example of safety property with precondition could be "*as long as the key is not in the ignition position, the car won't start*" [9].

When it comes to *liveness* property, giving few real world examples from [9] can help build basic intuition:

1. "*by keeping on trying, one will eventually succeed*"
2. "*if we call on the elevator, it is bound to arrive eventually*"
3. "*the traffic light will turn green*"

In other words, the point of examining *liveness* properties is to figure out if under certain conditions "*something good will eventually happen*". Again, to be more precise, it imposes a requirement whose eventual (and possibly repeated) fulfillment has to be guaranteed but in contrast to *safety* properties it does not need to hold continuously. Additionally, compared to *safety* properties it might not be as obvious that it is *not possible* to observe a counter-example in any finite run since it can *always* be extended by a state which fulfills the requirement [21][2]. Few examples are freedom from starvation and live-lock or, perhaps confusingly, termination which guarantees that all program runs will terminate.

One might be inclined to think that *liveness* property can be expressed as *reachability* property by reformulating the statement. However, this is not the case as can be observed for example with the statement 3. The difference is that reachability only guarantees the *possibility to turn green* but it does not guarantee that the light *will eventually* turn green. Moreover, virtually in all practical use-cases the *liveness* property of interest is also *bounded*, meaning the observed event has to happen with a certain time limit[3].

2.1.2 Difference between Static and Dynamic Analysis

One of the most important distinctive features of each analysis technique is *when* it is performed. The two main commonly used options are are *before* and *during* the program execution.

[2]The prerequisite for liveness is reachability. If fulfillment state is reachable then we can extend any program run to reach it and fulfill the liveness requirement.

[3]The notion of time can be defined in many different ways when it comes to software.

The first option is to run the analysis at run-time of a program in which case the analysis is called *dynamic*. This approach gathers information along the execution of one or more program runs and can be implemented in many different ways, for example by running the program on a virtual processor or by *code instrumentation* which is a technique involving injection of additional instructions into the program's executable binary. Few notable examples of dynamic analysis tools are *Valgrind* mostly used for detection of memory leaks, Google's suite of *Sanitizers* (Thread, Address, Memory, ...) and *Intel Inspector* which helps to discover dynamic memory and threading errors. Various kinds of *profilers* for measuring of program performance or memory usage are also considered to be dynamic analysis tools.

The second possible option is to perform the analysis *before* running the program in which case we call such analysis *static*. The name comes from the fact that such analyses are run independently from any program execution and thus there are no "moving parts". Consequently, analyzing the same unchanging code repeatedly will always yield the same results (if the tool is deterministic as it should) which is not necessarily true for *dynamic* analysis. Few notable examples of *static* tools are *Coverity* and *Astrée* which are proprietary or *Frama-C* and *Meta Infer* which are Open-Source.

This distinction is more significant than might be immediately obvious as the choice between *static* or *dynamic* approach comes with different advantages and drawbacks or even fundamental limitations. The first notable trade-off is between the ease of design and implementation, and performance. Dynamic tools are typically easier to design and implement when compared to, for example, development of a fully-fledged abstract interpretation framework from scratch. Unfortunately, this benefit in time and effort domain is in some cases heavily outweighed by the tool performance and in some edge cases a tool can even be rendered unusable in certain applications due to the performance penalty. For example, it is virtually impossible to employ the previously mentioned *Valgrind* tool for memory leak detection in advanced real-time 3D graphics rendering applications which require user input (e.g. video games) because the impact on FPS (frames per second) is too significant. In comparison *static* analysis techniques do not incur any run-time performance cost but they are not free either as their deployment can have non-negligible impact on the development iteration time which is one of the many aspects that can hinder the willingness of developers to adopt a certain tool [14].

One possible advantage of *dynamic* approaches is the access to concrete run-time state information which can lead to less *false positives*[4] on average. Additionally, such information can be leveraged by the tool to provide the developers with more precise and detailed issue reports. As has been empirically tested [14], minimizing the *false positives* rate and increasing the quality of issue reports is of great importance for smooth adoption of analysis tools by developers in practice, regardless of the *static* or *dynamic* aspect.

Last but not least, the choice between these two approaches has major implications on the program properties that can be checked. Most notably, *termination* is one of the properties that is *impossible* to determine dynamically for *any* input program due to fundamental limitations of *dynamic* analysis. Simply put, the potential tool would have to run infinitely to determine *non-termination* of an input program.

2.1.3 Automation and Human Input

One of the crucial aspects of any kind of analysis is the amount of human input that is needed. The ideal which every analysis should strive for is full automation, meaning the

[4]The issue of *false negatives* and *false positives* will be explored in Section 2.1.5 in more detail.

analysis is fully functional and computes the same outputs regardless of whether any human help was provided or not, i.e., no human help is needed to obtain the most precise possible results.

Most analyses decide to give up some amount of precision in order to remain fully automated which is usually preferred by the end users in the non-safety critical domains. Unfortunately, analysis of certain properties is impossible without any human intervention due to the implications of the *Halting Problem (HP) undecidability* and related *Rice's theorem* [29]. In short, this theorem states that any *non-trivial semantic property* is not computable because it can be *reduced* to the *HP* and consequently proven *undecidable*. Moreover, property is considered to be *non-trivial* if there is at least one program which satisfies it and one program which does not satisfy it, i.e. it is not either *true* or *false* for all programs. Clearly, by this definition almost all properties are *non-trivial* and thus not computable.

In those cases analysis designers have to give up on full automation and depend on users to provide some sort of input, usually in form of local or global *invariants*. Very common are for example local *loop invariants* because complex loops typically present a challenge for any kind of analysis. This partly shifts the responsibility of computing the analysis result on the user which becomes increasingly more problematic as the programs grow in size and complexity. Thus, it is not uncommon to hire a professional with mathematical expertise, who configures and provides input to such analyses but even so, the need for human input is most of the time error prone and can compromise the correctness of potential results.

2.1.4 Scalability: Compositional and Incremental Analysis

Somewhat related to automation, *scalability* is another aspect of consideration when designing an analysis. Even when human element is completely eliminated and tool is fully automatic, it might still be unsuitable for use in real large-scale software with millions of lines of code.

The degree to which an analysis is able to handle such codebases and provide results in reasonable time is called *scalability*. It is determined by the time and memory complexity as well as by other properties of the used analysis algorithm. Namely, whether it is *compositional* and possibly *incremental* or not is especially crucial for *scalability* on real production software. As [10] states: "*A compositional analysis is one in which the analysis result of a composite program is computed from the results of its parts. As a consequence, compositional analyses can run on incomplete programs (they are not whole-program analyses), are by their nature incremental, scale well, and tolerate imprecision on parts of code that are difficult to analyse.*"

Indeed, without the ability to compose the final result from results of its parts (very common is granularity at the level of program functions), the analysis might soon reach its limits because of extensive time or memory costs even if it's otherwise very efficient on small scale programs. Furthermore, *incrementality* might not have a direct impact on the *scalability* in case of *whole-program* analysis but it is equally as important in fast paced development environments where performing costly *whole-program* analysis on each and every code change is out of the question. In such settings, the ability to run the analysis only on a single code *diff* (code change, typically contained in a single *commit*, submitted by a developer for code review) is essential for usability of a tool and can be considered as another dimension of *scalability*.

2.1.5 Approximating results: Soundness and Completeness

As discussed in Section 2.1.3, full automation is often one of the top priorities when designing an analysis for several reasons and as such it is very common to compromise on other qualities instead of relaxing the full automation requirement. The popular analysis design trade-off is to sacrifice some amount of accuracy but maintain the correctness of the results in order to preserve the automation. This can be achieved through different ways. One possibility is for the analysis to answer "*yes*" or "*no*" only in cases when it is conclusive and then introduce a third fallback option of "*don't know*". Clearly, this would allow the analysis to remain correct with its answers but the usefulness might be compromised with frequent choice of the fallback option.

Another option is *conservative approximation* with two analysis properties of dual nature that describe the form of approximation used. To express these notions, following notation will be used: let $\mathcal{L}$ be a *Turing-complete language* and π be a *non-trivial semantic* property examined by an analysis tool T targeting programs in $\mathcal{L}$. A perfect analysis tool without any kind of inaccuracies could then be described by the following equation:

$$\forall p \in \mathcal{L} : \text{analysis}_T(p) = \mathit{true} \iff p \models \pi.$$

Unfortunately, as discussed prior, such analyses are impossible to create in practice but decomposing the logical equivalence in the equation above into a pair of implications reveals two approximation possibilities:

$$\begin{cases} \forall p \in \mathcal{L} : \text{analysis}_T(p) = \mathit{true} \implies p \models \pi \\ \forall p \in \mathcal{L} : \text{analysis}_T(p) = \mathit{true} \impliedby p \models \pi. \end{cases}$$

By dropping either one of these two implications, a partially accurate tool with different kind of approximation can be achieved.

Soundness

The first option is to design a tool which satisfies the first implication but does not care about the satisfaction of the second one. Such tool is called *sound* and can be defined using the first implication as follows:

Definition 2.1.1 (**Soundness** [29])**.** The program analyzer T is **sound** with respect to property π whenever,

$$\forall p \in \mathcal{L}.\ \text{analysis}_T(p) = \mathit{true} \implies p \models \pi.$$

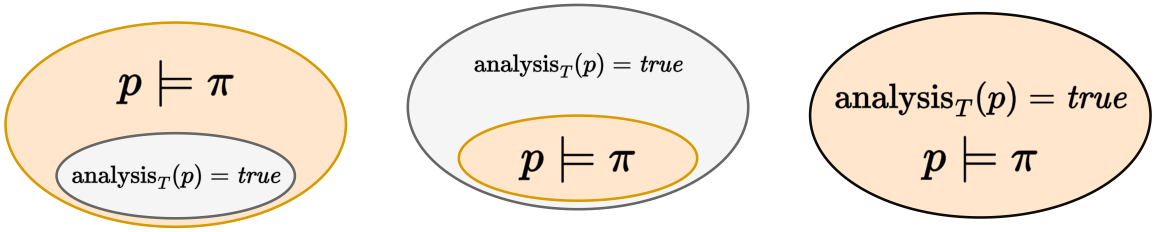
Informally, when a *sound* tool concludes that program p satisfies property π then it truly satisfies it. I.e., a *sound* analysis will *never* claim that program p satisfies π when in reality it doesn't. The concept of *soundness* is visually represented in Figure 2.1a. Consequently, a trivially *sound* but practically useless analysis can be achieved by always answering *false*.

Completeness

In contrast, a *complete* tool will disregard the fulfillment of the first implication but requires satisfaction of the second one.

Definition 2.1.2 (**Completeness** [29])**.** The program analyzer T is **complete** with respect to property π whenever,

$$\forall p \in \mathcal{L}.\ p \models \pi \implies \text{analysis}_T(p) = \mathit{true}.$$



(a) *Sound* but *incomplete* (b) *Complete* but *unsound* (c) *Sound* and *complete*

Figure 2.1: Graphical representation of *soundness* and *completeness*

Intuitively, if a program satisfies the property π then a *complete* analysis *must* conclude that it does. I.e., it will *never* reach a conclusion that a program does not satisfy property π when it in fact does. The visual representation of this concept can be observed in Figure 2.1b and as before a *complete* tool can be trivially constructed by always answering *true*.

Soundness and Completeness

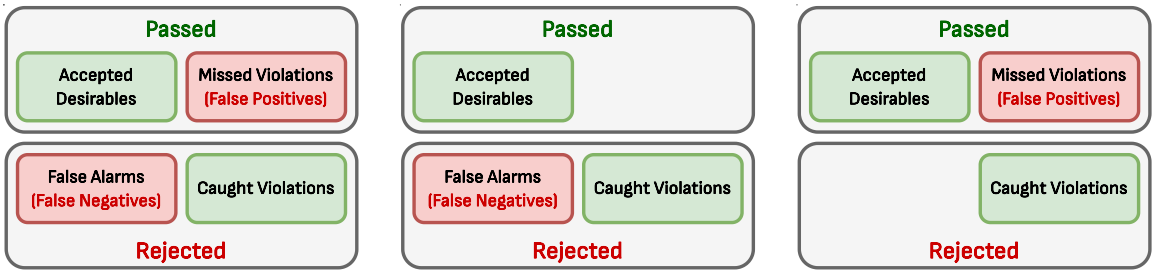
To expand on the previous discussion about computational limits, it is futile to hope for a *fully automatic*, *sound* and *complete* tool which computes a *non-trivial semantic property* of a *Turing-complete language*. Most tools opt for automatic but either *unsound* or *incomplete* analysis. There has been a lot of confusion about what *soundness* and *completeness* means in terms of specific analysis over the years, especially when it comes to the notions of *false positives* and *false negatives* borrowed from the *binary classification theory*. This confusion is mainly caused by the dual notion of these terms as well as the fact that the meaning is relative in respect to the goal of the analysis [24].

A good example to illustrate this relativity is termination analysis. Is a tool performing the analysis *unsound* or *incomplete* if it concludes that a program does not terminate when in reality it does? To answer this question it is necessary to establish the goal of the tool first. Let us consider two cases:

- *Input-Output programs*: These programs are expected to terminate in finite time and produce a valid result. As such, non-termination is **violation** and the goal of the analysis is to find these cases. Thus, if an analysis incorrectly flags a **desirable** terminating program as non-terminating it is a **false alarm** which makes the analysis *incomplete*. Conversely, if it misses a **violating** program and concludes that it terminates, then it is *unsound*.

- *HTTP Webserver*: Availability of a website is dependant on its webserver which should be running constantly in order to serve incoming user requests. Clearly, it is **desirable** for a webserver to keep running and not terminate (non-trivial *liveness* property) which would be a **violation**. Therefore, if an analysis concludes that a program does not terminate when in reality it does then it is **missed violation** and the tool is *unsound*.

As can be seen, these notions are relative and thinking in terms of **desirables** or **violations** and consequently **false alarms** or **missed violations** in the context of specific analysis dismisses any possible confusion. Figure 2.2 illustrates these notions alongside the standard *false positive* / *false negative* terminology.



(a) *Unsound* and *incomplete*: both **false alarms** and **missed violations** can occur.

(b) *Sound* but *incomplete*: only **false alarms** can occur but analysis will never miss a **violation**

(c) *Unsound* but *complete*: **violations** might be missed but **false alarms** cannot occur.

Figure 2.2: *Soundness* and *completeness* illustrated in terms of **false alarms** and **missed violations**. An analysis can either *accept* a program (possibly missing a **violation** of examined property) or *reject* it (might be correct but deemed incorrect due to a **false alarm**).

2.1.6 Analysis Techniques: An Overview

There are several heavily researched mainstream approaches to program analysis, each of them with numerous more or less distinctive variants. This section attempts to provide a concise overview of main existing techniques without delving into too much detail because comprehensive discussion about each technique is out of the scope of this thesis. As such, references to other sources with more information will be provided.

Since *Looper* is a static analysis tool, the primary focus of this thesis is given but it is beneficial to have a basic understanding of the other existing methods with their strong points and drawbacks. Moreover, nowadays there are tools which are built around interplay between multiple different approaches, e.g., combining static and dynamic analysis to get the best from both worlds. This section is mostly based on [20, 29].

Automated or Machine-Assisted Deductive Formal Verification

This static-only approach, also-called *theorem proving*, is most of the times *semi-automated* way of formally proving certain program property in a *sound* (w.r.t. to the model of the program semantics) and usually also *complete* (up to the abilities of proof assistant) way. There are three essential elements to this approach:

- *Logical theory* and its *general theorems*: the choice of a logical theory is based on the property of interest and it forms the abstraction basis. Common examples are *Integer* or *Real arithmetic*, *Boolean operations*, *Sets*, *Maps* [5], *Bitvectors* [18], *Strings* and many others.
- *Logical facts*: these facts are either part of the used logical theory, provided by the user or deduced by the system. The need to provide facts manually is one of the reasons why some of these methods are not fully automated.
- *Inference system*: provides *inference rules* that drive the deduction process. New logical facts and theorems are deduced by automatically or manually applying valid *inference rules*.

Clearly, this approach bears a strong resemblance to classical mathematical reasoning. Indeed, the principle is the same but the difference is that it is performed by software tools (*theorem provers*) which eliminate as much tedious and often error-prone human labor as possible. Unfortunately, these *machine-assisted* techniques also have their computational limits and in many cases have to rely on human input to reach any conclusion. For example, users might have to provide information which is otherwise difficult to deduce automatically such as aforementioned *loop invariants* or function *pre/post-conditions* which form "boundaries" for deducing. Apart from providing data, users might have to lead the inference process itself by deciding which rules should be used with which facts and when. Needless to say, this often requires high level of mathematical expertise and even then can be source of errors or the reason for inability to prove a property. However, despite these limitations, the verification process still remains mostly fully automatic. Few examples of these *interactive theorem provers* are PVS, Isabelle, ACL2, Coq and others.

Most of these tools have one important aspect in common: they employ so-called *satisfiability solvers* for various logical theories. These tools are essential building blocks which most higher-level verification methods rely on since every *NP-Complete* logical problem (properties of interest usually represent a *NP-Complete* problem) can be reduced to classical *Boolean satisfiability (SAT) problem*. The term *satisfiability solvers* encompasses various different solvers with most important being *SAT-solvers* solving the aforementioned *SAT* problems. Few examples of modern powerful *SAT-solvers* are IntelSAT, RLNT, Glucose and many more. Unfortunately, as many real world problems are not easily or intuitively expressible in plain Boolean logic, a new category of *SMT-solvers* (*Satisfiability Modulo Theories*) has emerged. These solvers are built for problems expressible in *first-order logic* and its various *logical theories* such as *Linear (Real) Arithmetic* theory or theory of Arrays and Lists. Many of these important theories are sadly *undecidable* [3] (as is *first-order logic* in general), but significant advances in recent years made it possible to efficiently solve a wide range of problems nevertheless. Few notable examples of modern and actively developed *SMT-solvers* are CVC, Yices, veriT or Z3, some of which are indirectly used by Looper itself.

Last notable category is comprised of high-level tools that leverage the infrastructure of other existing tools to streamline the verification process. One example is a language/verifier Dafny [23] which allows users to write imperative and sequential programs with support of generic classes and dynamic allocation. The syntax of the Dafny language resembles modern object oriented languages such as C++ but also incorporates formal specification constructs such as function *pre/post-conditions* or *termination metrics* [23]. To perform the verification, programs written in Dafny are first translated to intermediate verification language *Boogie* and then passed to the Z3 *SMT-solver* which verifies that program meets its formal specification. Finally, a correct program can be compiled into several widely used languages such as C#, Java, JavaScript or Go. Moreover, Dafny ecosystem provides plugins for certain IDEs such as Visual Studio which continuously perform the verification as user writes Dafny code.

Second, and for this thesis important, example of high-level tool is Why3 [6] *verification platform*. This tool comes with its own *first-order logic* based specification language called *WhyML* which can be used to create program models. Alternatively, these models can also be created using the OCaml language bindings that expose the Why3 API. The strength of Why3 lies in the ability to transform these formal models into several standardized formats such as `SMT-LIB`. These formats are supported by various *SMT-solvers* or *theorem provers* such as Vampire which was used by Looper at one point. It provides a front-end that

allows end users to easily employ multiple supported solvers or provers at once based on the problem at hand. This is crucial because each tool excels in different logical fragments. On the other hand, it requires a certain level of expertise and knowledge of each used solver to know which one to use for which logical problem. This is further complicated by the fact that modern solvers are mostly black boxes with not always intuitive and possibly surprising behaviour, especially when it comes to the ability to solve a given problem. The use of Why3 by Looper will be discussed in more detail in Chapter 5.

In conclusion, the methods in the field of deductive formal verification are all static and tend to be *semi-automated* but fully automated tools exist as well. Their use often necessitates mathematical knowledge and a certain level of expertise in formal verification techniques and associated tools. Most of them are sound by design and usually also complete.

Testing: Dynamically Checking Program Executions

One approach that often comes to mind first is to run a program and observe its execution, usually many times with different inputs. This approach of dynamically checking whether a program behaves as expected for a wide range of possible inputs is called *testing*. However, apart from special cases it is not feasible to observe all possible executions even if their number is *technically* finitely bounded. Moreover, non-terminating programs pose a problem as those cannot be tested. Thus, program testing can only check *finite set* of *finite executions*. Based on this fact and the famous quote from Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!", it becomes clear that in most cases testing is *unsound*. On the other hand, traditional testing approaches are considered *complete* as they cannot produce **false alarms** and usually provide user with a counter-example in case of a failed testing run.

Apart from the basic variations such as *random testing*, there are many more advanced techniques, each of them trying to address some limitation of rudimentary testing. For example, the most prominent issue of low *code coverage* is tackled by the *concolic* or *search-based* techniques which both attempt to synthesize more test data, albeit each in a different manner. Concurrent software posed another challenge to testing as certain types of bugs (deadlock, data races) rarely manifest under normal process/thread scheduling circumstances. Thus, techniques like *noise-injection*, which disrupt the normal scheduling by injecting timing noise, were developed. Other advanced techniques try to increase the chances of catching elusive concurrency bugs by performing a post-analysis and extrapolating possible erroneous states based on what has been seen in real executions. On the flip side, this technique can compromise *completeness* of testing and lead to **false alarms** as some of these extrapolated states might not be reachable in reality. There have been multitudes of tools targeting concurrency errors over the years but few prominent examples that implemented these ideas are Eraser, FastTrack or more recently ThreadSanitizer developed by Google.

Regardless of used analysis technique, any kind of testing can be performed on several levels or development stages during the software life cycle as is usually the case in the industry. Most common are so-called *unit tests* (testing of smallest testable parts in isolation), *integration tests* (individually tested modules are integrated together and tested as a group) and *system tests* (the final product is tested as a whole).

In conclusion, testing is generally easy to automate and modern cloud repository hosting services also provide infrastructure to easily employ so-called *continous integration* which

further automates testing processes. It is, apart from exceptions, *unsound* and reaching high *test coverage* requires a lot of effort and resources. On the other hand it is traditionally *complete* unless combined with other static analysis techniques which can potentially compromise *completeness*. Moreover, failed test runs produce counter-examples that often provide precise information on how to fix the program. Unfortunately, the usefulness of these counter-examples is diminished in non-deterministic programs where specific executions cannot be easily reproduced. One significant advantage of testing is the ability to perform it in the conditions of the target platform (hardware, drivers and operating system) which might reveal bugs that would not otherwise manifest. Lastly, testing inherently cannot be used to determine certain classes of properties such as program termination.

Finite-State Model Checking

The technique of *model checking* aims at addressing the *unsoundness* problem of traditional *testing* which can only check *finite set of finite executions* out of possibly infinite set of possible executions. Instead, this approach focuses on finite systems with the goal of exhaustively checking all possible executions of a system in order to *soundly* and *completely* determine whether it satisfies a property or not. To make it possible, *model checking* uses various heuristics to perform a *systematic exploration of the state space* of a constructed model. Nevertheless, the infeasibility of *sound* testing is also reflected in *model checking* which suffers from so called *state space explosion problem* [20]. To overcome this issue, various efficient data structures such as *binary decision diagrams* or *compact state space storage* are being used in order to reduce the search space, sometimes at the cost of *soundness* as is the case in *bit-state hashing method* [20]. Additionally, it is possible to apply certain abstractions or perform so-called *bounded model checking* which bounds the state space by restricting certain property, e.g., the depth of the search or the number of context switches when analyzing concurrent program. This sadly also leads to *unsoundness* in most cases.

There are two important caveats to *model checking* which have to be mentioned. Firstly, it is often presented as *automatic* technique which is true with respect to the analysis itself. However, the design and creation of model which precedes the analysis is mostly manual, tedious and resource intensive process. Moreover, it can also be a source of inaccuracies or errors which relates to the second caveat: *model checking* in its original form indeed is *sound* and *complete* but only with *respect to the model* as it is performed at model level and not at the program level. Thus, if the possible inaccuracy of the model and the relation between the program and its model is taken into account, *model checking* might be rendered *unsound* and/or *incomplete* with respect to the program. Some techniques incorporate automatic model refinements to alleviate this issue but then problems with non-termination might arise [29]. Despite this, most *model checking* tools in practice are often conservative and thus *sound* and *incomplete* w.r.t. to the modeled program.

In conclusion, model checking has been successfully employed in many software and hardware areas alike. There have been many influential commercial applications especially in the field of hardware verification. Few examples include RuleBase from IBM, Incisive Verifier from Cadence or NuSMV. However, there have also been many software *model checking* tools for *concurrent and distributed systems* as well as for *real-time* and *probabilistic systems*. More examples of existing tools and detailed discussion on *temporal logics* such as CTL or LTL which are used for specification of properties can be found in [20].

Static Analysis

The most relevant in the context of this thesis are the *static analysis* techniques (difference between *static* and *dynamic* approach was previously discussed in Section 2.1.2) since Looper tool belongs to this category. According to [29], the term is defined as follows: "*static analysis relies on other techniques to compute conservative descriptions of program behaviors using finite resources. The core idea is to finitely over-approximate the set of all program behaviors using a specific set of properties, the computation of which can be automated* ". This rather vague and general definition does not necessarily help one to understand the term in practical sense. The reason is that the field of *static analysis* is very broad and thus providing clear cut precise definition is no easy feat. The most crucial part of this definition is the *conservative over-approximation* of program behaviors, i.e., any *static analysis* method should be *sound* by design. Relaxing the *soundness* requirement yields a so-called *bug finding* method which will be discussed shortly.

The meaning of *conservative over-approximation* depends on the goal of specific analysis. For example, if the focus of an analysis is to detect array buffer overruns, then it is *conservative* to consider the worst possible minimum and maximum values of all indexing expressions while at the same time expecting the array to be as small as possible at any given access time. Doing so will ensure *soundness*, however the true difficulty lies in the *sound* computation of the aforementioned values. For this thesis more relevant example is a general resource (time, memory, open files, etc.) bound analysis which should always return a bound that is *greater or equal* than the real bound in order to remain *sound*. How to be *conservative* during the analysis then depends on the target resource.

Another, maybe more intuitive, description from [25] states: "*it is the art of reasoning about the behavior of computer programs without actually running them*" (at least not under their original semantics). As a consequence of not having to run a program to analyze it, no input data is needed which is a significant advantage over testing or dynamic analysis in general. In short, reasoning about the behavior of programs in the most rudimentary way can be done by observing certain *syntactic patterns* in the source code, which was pioneered by the famous *Lint* [19] tool. However, over the years increasingly more advanced tools started to take a different approach in reasoning by using some sort of program abstraction. An abstraction bears a certain resemblance to a model used in *model checking*. However, an abstraction has to be designed only once and then it can be applied automatically to any program, whereas a model has to be crafted specifically for each analyzed program.

Historically, certain forms of *static analysis* have been part of compilers ever since their inception, either to check correctness (type systems), extract useful information (*control-flow graph*, *call graph*, *alias analysis*, etc.), or for optimization purposes (*dead code elimination*, *invariant analysis*) and the set of compiler analyses has been growing over the years. More recently, due to the ever increasing need for parallel programs, additional analyses such as *cross-loop data dependence analysis* or *memory synchronization analysis* became part of compilers for programs written using OpenMP, OpenCL or CUDA [30].

As time went by, *static analysis* tools developed beyond compilers and became more advanced. Nowadays, it is being used to automatically find errors or even verify correctness w.r.t. to some program property. It has been heavily used to design program verifiers and other supporting tools (sometimes integrated in IDEs) that help programmers to understand code. Some advanced use-cases chosen, but not exclusively, from [25]:

- *Alias Analysis*: Are pointers p and q pointing to *disjoint data structures* in memory?

- *Buffer Overrun Analysis*: Are arrays always accessed within their bounds?
- *Termination Analysis*: Does the program terminate on every input?
- *Concurrency Issues Analysis*: Are data races possible? Can the program (or parts of the program) deadlock?
- *Worst-Case Execution Time Analysis*: What is the upper bound on the execution time for a piece of code when taking hardware characteristics of a specific platform (timings, cache, ...) into account?
- **Complexity Analysis**: What are the asymptotic complexities of specific program parts such as loops or functions?

Clearly, the field of *static analysis* truly is broad and encompasses a wide range of analyses as supported by these examples and the cited definition. However, one can still identify several traditional approaches [20]:

- *Linters* — named after the famous *Lint*, these tools search for so-called syntactic *anti-patterns* indicating possible bugs. Apart from standalone tools like *Cppcheck*, linters are used in virtually all modern compilers and IDEs like Visual Studio (Code) or CLion (built-in or as a plugin).
- *Data-flow analysis* — this type of analysis tracks how properties of interest called *data flow facts* [20] propagate through program locations. It typically operates over a *Control-Flow Graph* (CFG) which is a graph representation of a program. It can be performed in *forward* or *backward* manner and within single function (*interprocedural*) or across function boundaries (*interprocedural*). It can be either *may* (which facts **could be true** at each location) or *must* (which facts are **definitely true** at each location) analysis [7]. This type of analysis was originally mostly used in optimizing compilers but it is nowadays heavily used by many commercial as well as open-source tools, either as a data collection *pre-analysis* or standalone bug finding analysis. Two typical examples are *Live-variables analysis* and *Reaching Definitions analysis*. Examples of influential tools include Coverity, CodeSonar, PhASAR [31], or FindBugs (newly SpotBugs), which was one of the first widely used (Google and other major companies) open-source tools [2].
- *Constraint-based analysis* — this analysis works in two phases. First it derives a set of constraints which forms an abstraction of the analyzed program. These constraints are mathematical equations and their form is determined by the property of interest. As an example, a two completely different types of constraints will be needed in order to derive *linear invariants* and *linear ranking functions*. Common types of constraints are *conditional set constraints*, *linear arithmetic constraints*, *polynomial arithmetic constraints* and more.

  The real difficulty lies in the second phase which is about solving the set of constraint as the number of constraints can be in hundreds or even thousands. However, these analyses typically leverage the modern and powerful SAT and SMT *constraints-solvers* to find the solutions. The main inherent advantage is the ability to find all possible solutions and not just one (all loop invariants for example). Furthermore, solvers are constantly improving which in turn leads to advances in this field. Two examples of typical use-cases are points-to analysis and derivation of loop invariants.

- *Type-based analysis* — general term that encompasses all analyses that in some way leverage type information either as basis for the analysis or to improve existing analysis in terms of precision and efficiency. Two common approaches to *type-based analysis* are *Type and Effect* and *Types-as-Discriminators* [26].
- *Abstract Interpretation* — probably the most common approach to *static analysis* in recent years. Brief introduction is presented in Section 2.2. Current state-of-the-art tools include Astrée, Polyspace or Sparrow. Other tools like the commercial Coverity are not *per se* abstract interpretation tools but they include analyses based on this technique [16]. Advances in the category of open-source frameworks are led by Frama-C and Meta Infer which is discussed in Section 2.3. These frameworks offer both *sound* and *unsound* analyses.

Last important remark belongs to the so-called *bug finding* (also-called *bug hunting*) approach. Traditional *conservative static analysis* is fully automatic, *sound* and *incomplete* while also scalable when considering recent advances of modern tools. However, the *soundness* requirement often times goes against the trends in the industry [4], where the rate of *false positives* can determine the success of a tool. Many tools thus decide to drop the *soundness* property with the main motivation being faster and easier design process and implementation. Due to their *unsoundness*, such tools cannot be classified as *conservative static analysis* in the conventional sense even if their techniques are commonly based on those used in model checking or static analysis. The primary aim of *bug finding* tools is to quickly find as many errors while minimizing false positives and user adoption friction. They are commonly used in *non-safety critical* domains to improve the quality of program at a low cost [29]. Examples of prominent *bug finding* tools include aforementioned Coverity, CodeSonar or CBMC (C Bounded Model Checker).

Table 2.1 summarizes characteristics of analysis techniques discussed in this section.

Technique	**Automatic**	**Sound**	**Complete**	**Object**	**Approach**
Deductive verification	No	Yes	Yes/No	Model/Program	Static
Testing	Yes	No	Yes	Program	Dynamic
Model checking	Yes	Yes	Yes/No	Model/Program	Static
Conservative static analysis	Yes	Yes	No	Program	Static
Bug finding	Yes	No	No	Program	Static

Table 2.1: An overview of discussed program analysis techniques based on the information available in [29, 20]. It summarizes the main characteristics of each method.

2.2 Abstract Interpretation

Since the publication of author's bachelor's thesis, *abstract interpretation* technique is no longer used by the Looper tool for any purpose. The previous use-case and the reason why it was not desirable to continue using it will be discussed in Chapter 3.2. Despite this, abstract interpretation still remains an essential part of the *Meta Infer* framework (see Section 2.3) which was used to develop Looper and as such it was considered necessary to have at least a basic understanding of this approach. Note that this section was taken from

previously published author's work [27] and then appropriately updated and shortened to fit the scope of this thesis.

Abstract interpretation was originally formalized by a married couple of French computer scientists Patrick and Radhia Cousot in the late 1970s [13]. The theory of *abstract interpretation* provides a general framework which can be utilized in the process of creating specific static analyses. New analyses can be obtained by instantiating of the necessary components to the general framework.

Even though *abstract interpretation* falls into the domain of static analysis of programs, it actually executes instructions of analyzed program in a sense. The key difference is how are these executed instructions interpreted: each *concrete instruction* is assigned with certain *abstract semantics* that specify what effect it has when executed over a so-called *abstract domain*. This abstract domain has to be tailored for the specific needs of the analysis and its area of focus. *Abstract semantics* of an instruction are then applied to the abstract context which is used to represent a program state at a certain location. The actual physical execution of the program instructions is thus completely avoided which means that the AI preserves all the advantageous properties of static analysis. A state space of a program can subsequently be reduced significantly just by choosing the appropriate level of abstraction for the problem at hand and devising corresponding *abstract domain* and *abstract transformers*.

Components of the Abstract Interpretation

When instantiating the theory of abstract interpretation, every analysis has to provide few essential components required by the framework. These components describe the semantics of the analysis:

- **Abstract domain**: a set of abstract states. An abstract state represents a program state at a certain program location. The definition and contents of an abstract state depends on the type of analysis being designed. Simple interval domain for tracking safe lower and upper bounds of integer program variables presents a trivial example. I.e., each abstract state of this domain will contain an interval of possible values for each program variable in this form: $[a, b]$ where $a \in \mathbb{Z} \cup \{-\infty\}$, $b \in \mathbb{Z} \cup \{\infty\}$, $\top = (-\infty, \infty)$ and $\bot = (a, b)$ for $a = b$. The $\top$ symbol denotes the top element of the underlying lattice as all existing intervals are contained in the $(-\infty, \infty)$ interval. The $\bot$ symbol denotes the bottom element of the underlying lattice which is an empty interval. The integer sets for lower and upper bounds are extended by infinities because it is not always possible to determine precise bounds and interval over-approximation is necessary if we aim for a *sound* static analysis.

- **Abstract transformers**: each instruction from program's source code has assigned transformer which transforms the original semantics of an instruction to abstract semantics which can be applied to an abstract state. For example, it would be necessary to transform the integer arithmetic of a concrete program to the interval arithmetic applicable in the previously introduced interval domain. E.g., increment to variable i represented by the $[a, b]$ interval would lead to new interval $[a + 1, b + 1]$ and the assignment $i = 0$ would lead to $[0, 0]$.

- **Join operator**: accumulates two input abstract states into a new output state. Join operator is used at program junctions where several program paths meet, e.g. after

if-else construct. One possible definition of the join operator for the interval domain is as follows:

$$[a, b] \circ [c, d] = [\min(a, c), \max(b, d)]$$

.

- **Widening**: applied on a sequence of abstract contexts at a certain program location (for example loop headers) in order to accelerate fixpoint calculation. However, accelerated fixpoint computation by means of widening usually has a trade-off in a form of precision loss. Widening in the interval domain can be defined as:

$$[a_i, b_i] \bigtriangledown [a_{i+1}, b_{i+1}] = [\mathit{if}\ a_{i+1} < a_i\ \mathit{then}\ -\infty\ \mathit{else}\ a_i,\ \mathit{if}\ b_{i+1} < b_i\ \mathit{then}\ \infty\ \mathit{else}\ b_i],$$

  where intervals $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ correspond to the values of two abstract states accumulated at a loop header after two consecutive iterations i and $i + 1$.

- **Narrowing**: can be used in order to refine the result of widening operation. Some analyses do not require to define the narrowing operation. Narrowing in the interval domain can be defined as:

$$[a_0, b_0]\ \bigtriangleup\ [a_1, b_1] = [\mathit{if}\ a_0 = -\infty\ \mathit{then}\ a_1\ \mathit{else}\ a_0,\ \mathit{if}\ b_0 = \infty\ \mathit{then}\ b_1\ \mathit{else}\ b_0].$$

Importantly, there should be a correspondence between the *abstract* and *concrete* domains and semantics. The correspondence between these domains is traditionally assured by a pair of monotone functions which are called *abstraction* and *concretisation* functions, usually denoted by α and γ. These functions should form a so called *Galois connection* but the more general definition of abstract interpretation, which was later formulated by Cousot lifts this requirement. Further, program instructions are assigned with so-called *abstract* and *concrete transformers*, which are monotone functions applied on the objects of *abstract* and *concrete* domains respectively.

The last necessary component to perform the analysis is some traversal algorithm which will visit *control flow graph* nodes of the analyzed procedure in a certain order. The order in which the nodes are traversed matters especially when an *abstract domain* requires *widening*. In such cases it is desirable to *widen* at as few nodes as possible to increase precision and efficiency [15]. Examples of possible node orderings include *reverse post-order* or *weak topological order*. Finally, the analysis is performed by visiting CFG nodes in the chosen order and applying *abstract transformers* of node instructions along the way. When *abstract states* from multiple paths meet, *join* operator is used and possibly followed by the *widen* operator if located at a loop head. Optionally if defined, *narrowing* operator can be used to refine the *widening* result [20].

More detailed explanation of *abstract interpretation* with formal definitions of all mentioned concepts including *Galois connection*, *fixpoints* and their approximation can be found in [20, 13, 12, 29].

2.3 Meta Infer — Static Analysis Framework

Infer is an open-source static analysis framework developed by the *Meta Infer* team and implemented mainly in *OCaml*. Its main advantage over the most of the other existing tools is the ability to discover interprocedural bugs in a scalable manner through the use of the so-called function *summaries*.

Infer was originally a standalone analyser focused on finding of memory safety violations such as the dereferencing of null pointers or memory leaks. It has made its breakthrough thanks to the influential paper [11] presenting logical concept called *bi-abduction* which composes the static analysis in a scalable manner. Bi-abduction is a form of logical inference mainly for separation logic which is a novel kind of mathematical logic. Separation logic itself made a huge impact on a way how one can reason about computer memory and was one of the key reasons why the original shape analysis could scale.

Since then, Infer has evolved into a general abstract interpretation framework that can be used to quickly develop new kinds of modular interprocedural analyses. At the core of each interprocedural analysis stands an intraprocedural analysis that computes a summary for a single procedure. Abstract interpretation framework can then leverage those summaries at the call sites of previously analysed functions and use them to lift the analysis to the interprocedural and compositional level. As a consequence of compositionality it is also incremental which means that it can be run only on code changes instead of entire codebase. This property is especially critical for analyses that will be run on large codebases where complete re-analysis on each code change would be unfeasible for real world application which is what Infer aims for.

Infer currently consists of three main parts: *AI*, *AL* and *SL*. The AI refers to the aforementioned abstract interpretation framework, AL is a framework for basic syntax linters and SL refers to the original separation logic based analysis. The AI framework currently supports analysis of C, C++, Objective-C and Java programs and provides a wide range of analyses each focusing on different bug types. List of more mature analyses includes for example *Inferbo* (buffer overrun checker), *RacerD* (data races) or *Starvation* (concurrency starvation and some types of deadlocks).

Abstract Interpretation Framework Architecture

Infer.AI is an abstract interpretation framework implemented inside the Infer tool. It provides basic infrastructure as well as great number of facilities that simplify the development process of new analyzers such as automatic HTML logging and formatting or various OCaml modules for easier expression parsing and pattern matching. Infer.AI can be used to implement simple intraprocedural analyses which can be converted to interprocedural analyses just by adding some boilerplate code that enables usage of function summaries.

The framework architecture consists of three main components. The first main component is the frontend. Its job is to leverage the underlying LLVM compiler infrastructure to compile analyzed program from its source language to so-called *Smallfoot Intermediate Language* (`SIL`): the low-level intermediate language used by *Infer.AI* framework during the analysis.

Frontend provides an output in form of a CFG for each analyzed procedure and also another higher level interprocedural CFG for each source file, i.e., a file specific *call graph*. Frontend is able to generate variety of different procedure CFG types such as *normal*, *exceptional* with exceptional flow for languages with exceptions or *backward* (reversed direction). This approach is more flexible and gives the developer more options to choose from based on the needs of specific analysis.

Each node of the procedure CFG contains a list of `SIL` instructions that will be interpreted by abstract interpreter implemented in the framework. We can list four main instructions:

- `LOAD` — loads value from an address denoted by an expression into a temporary identifier. Address expression can be either a program variable or, e.g., more complex expression that includes array indexing,
- `STORE` — stores value of an expression into a place denoted by an address expression (same as with `LOAD` instruction). Value expression consists of constants and temporary identifiers created by previous `LOAD` instructions,
- `CALL` — represents a function call. Creates a new temporary identifier for a possible return value and provides information about return type, types of parameters and call flags. Note, that indirect function calls are handled by a combination of `LOAD` and `CALL` instructions,
- `PRUNE` — splits the control flow into two new branches based on possible results of a boolean expression. This instruction is interpreted *after* the split which means it is interpreted twice, once for the true branch and once for the false branch.

Infer also supports analysis over another higher level intermediate language called `HIL` which is built on top of `SIL`. Even though `HIL` is simpler than `SIL` and has only three instructions it is sufficient for the needs of the most of the analyses. However it is not suitable for analyses that focus on memory bugs and work with pointers on regular basis, contrary to `SIL` which is more appropriate.

The frontend module and the use of the intermediate language allows us to write new analyses with minimal language specific logic and in turn we can run one analysis on programs written in multiple programming languages.

The second main component of the architecture is scheduler which determines the suitable order of analysis of each procedure based on a *call graph*. Scheduler is especially important for interprocedural analysis where order in which procedures are analysed really matters. We will explain this problem in more detail in the Section 2.3. A procedure is analysed once it is chosen by the scheduler and returns a *summary* which is stored in the results database. This way, a procedure *summary* can be retrieved from a database and instantiated repeatedly at different call sites. Moreover, the use of a database storage allows Infer to be incremental. Scheduler is also able to determine which procedures are independent and, hence, can be analysed concurrently. Infer can then be run in a heavily parallelized manner — one of the reasons for its high scalability.

The last main component is the parameterized abstract interpreter which must be instantiated by every analyser and performs the actual analysis of each procedure. New instance of abstract interpreter must be provided with an aforementioned type of procedure CFG and a module implementing custom transfer functions for each `SIL` instruction. Effect of these transfer functions is applied to abstract states for a custom abstract domain. Infer does not impose any restrictions on the contents of an abstract domain and the only requirement is that it must provide implementation for join and widen operations and a comparator for abstract states which creates an ordering. In addition it must also define a data structure representing abstract state.

Intraprocedural Analysis

Intraprocedural analysis is an analysis that ignores the nested calls of other procedures. It focuses on a single procedure at a time and out of context of its call sites. As a result it has quite limited ability to reason about the program as a whole and can only provide

a knowledge about its procedures limited to their scope. For example, it is not possible to provide additional preconditions based on the context of specific call site and at the other end postconditions are of no value to the caller.

When performing intraprocedural analysis, the previously introduced abstract interpreter analyses a single procedure using two main components: the *command interpreter* and the *control interpreter*. The command interpreter interprets `SIL` or `HIL` instructions over input abstract states and produces new output states. The interpretation is a process of applying the corresponding transfer function to the input state which produces a new output state. The control interpreter receives this updated state and continues with next instruction based on the procedure CFG. Both components together form the main analysis loop which repeats until it processes all instructions or finds a fixpoint in case of a program loop. These parts of the abstract interpreter have access to transfer functions and a valid domain implementing necessary operations and defining abstract state.

Modularity of the AI framework is ensured by the parametric command interpreter which changes behaviour based on the currently plugged set of transfer functions. This approach makes the process of creating new analyses easy as there is no need to change command interpreter every time we decide to add new analysis. Hence, we can create new intraprocedural analysis in three steps: (1) we choose type of procedure CFG, (2) we design abstract domain, and finally (3) we implement transfer functions. Individual parts are passed to the new abstract interpreter instance that stitches everything together and exposes various functions that perform different tasks related to the analysis.

Interprocedural Analysis

Unlike intraprocedural analysis, interprocedural analysis can discover bugs caused by interactions between procedures and does take call site context into account. Postcondition of a called procedure changes based on its preconditions w.r.t. the current state of a program at specific call site. But in interprocedural analysis postconditions can also affect state of the caller via return value or pointer parameters.

Infer uses two different approaches to achieve interprocedurality. The first is based on *bi-abduction* theory and is employed in the original separation-logic based analyser. *Bi-abduction* allows Infer to break one large memory analysis of a whole program into smaller independent analyses of individual procedures. In general, it is a technique that allows Infer to automatically deduce preconditions and postconditions for a procedure by symbolic execution of its code. It is one of the reasons why the original analyser scales so well.

The second approach to interprocedural analysis is based on the notion of *summaries* and is employed in the AI framework. Summary as a general concept is a data structure that stores relevant information about the analysed procedure. In most cases, it contains collection of conditions over the formal parameters of a procedure. Subsequent violation of those conditions at specific call site with concrete arguments can then be considered as a bug. However, summary does not necessarily have to be a collection of conditions. Instead it can contain general context-independent postcondition for each formal parameter or a formula describing relation between argument values and return value. Additionally, it can also contain information about side effects of the procedure.

The AI framework does not impose any restrictions in regards to the content of a summary. As a result, it can contain any type of data and it is solely on the programmer which data he chooses to store and how he leverages them at call sites. The summary concept

allows Infer to analyse each procedure only once and then reuse stored procedure summaries as many times as needed by instantiating them at call sites. Summary instantiation is basically a substitution of general parts of a summary for concrete values at a call site.

Conversion of intraprocedural analysis into modular interprocedural analysis in the AI framework is straightforward. First we define the summary data type along with boilerplate code implementing interface exposed to the framework so that it can store and read the summary. Finally, we add logic that uses summaries in the transfer functions.

Order in which procedures are analysed during interprocedural analysis does matter, because the analyser needs to have a valid summary for each function that is called by the currently analysed procedure. The scheduler implemented in the AI framework uses a *call graph* to handle this issue and ensures that procedures are analysed in suitable order. Call graph is an oriented graph describing dependencies between procedures, i.e. which procedures can be called by a one specific procedure. Example of one such call graph can be seen in Figure 2.3.

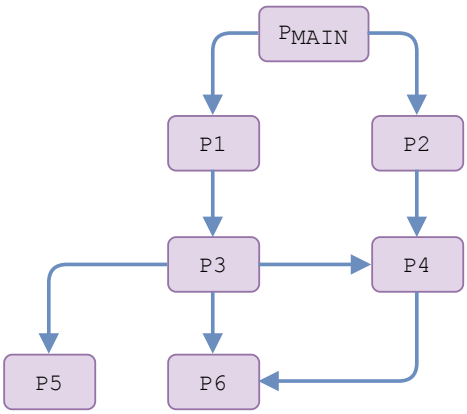


Figure 2.3: A call graph describing call dependencies of each procedure represented by a node. Outgoing edge signifies the possibility of a call to other procedure.

In the example, Infer would first analyse procedures `P5` and `P6` as they are *sink vertices*, i.e., vertices that have no outgoing edges. These procedures do not call any other user defined procedures but they might still call built-in or library procedures with defined models that do not need to be analysed. Infer would then continue in similar fashion towards *source vertices* with no incoming edges, i.e., $\texttt{P}_{\texttt{main}}$ in this case. As stated before, Infer can also analyse multiple procedures concurrently and uses call graphs to ensure that no dependencies are violated when it selects a set of procedures that could be analysed simultaneously.

This example also illustrates the incremental property of Infer that allows it to scale extremely well especially in rapidly changing code bases where conventional batch analysis is unfeasible. Incremental analysis only needs to re-analyse procedures directly affected by a code change and all procedures up the call chain as the summaries must have changed and therefore their updated versions should be propagated to all call sites. For example if procedure `P5` was changed, Infer would also have to re-analyse procedures `P3`, `P1` and $\texttt{P}_{\texttt{main}}$. However, if `P2` was changed, only $\texttt{P}_{\texttt{main}}$ would have to be re-analysed on top of it.

Chapter 3

Looper — A Worst Case Cost Analyser

As was previously mentioned, *Looper* is a *static complexity analysis* tool for automatic inference of *tight upper bounds* on program *execution cost*. Based on the previously existing LOOPUS tool, the core concepts of the original Looper tool were extensively studied within author's previously published bachelor's thesis [27], which also described the proposal and implementation of the tool in great detail. The main intention of this chapter is to revisit the original work and provide a concise summary of the most important points in Section 3.1, naturally based on the aforementioned [27]. The last Section 3.2 of this chapter is dedicated to a brief discussion about the current main limitations of the Looper tool, some of which were previously formulated in [27].

3.1 Core Concepts of Looper

The introduction of this chapter stated that the main focus of Looper is automatic inference of *tight upper bounds* on program *execution cost*. First, it is necessary to establish what exactly *cost* of a program is and the *cost model* it is based on. Traditionally, resource bound analyses define their own cost model (or multiple models) which assigns a cost to each instruction in the language of interest. This cost model captures the essence of the analysis and determines what kind of resource will be under scrutiny. For example, a trivial cost model for a memory usage bound analysis of C programs could assign *positive cost* of `N` to every `p = malloc(N)` call, *negative cost* of `N` to every `free(p)` call and cost 0 to all other instructions for the sake of simplicity. Intuitively it follows that by summing up costs of all executed instructions in a concrete program run the *total cost* is obtained. However, a *static analyser* is typically concerned with obtaining the *upper bound* of a program cost and not the total cost of one program execution (that is typically the domain of *dynamic analysers* such as VALGRIND). As such, static analysers have to consider *all possible executions* of a program and determine a bound that holds true for all executions if their goal is to remain *sound*. Moreover, these bounds should also be as precise as possible in order to be useful. Certainly, a bound of $+\infty$ is sound in all circumstances but it has no information value for the end user. The final computed bound is then typically a symbolic expression over a program or function parameters.

Back-Edge Metric

Looper uses a so-called *back-edge metric* as its cost model. This model assigns the cost 1 to every *back jump instruction* and the cost 0 to all other instructions. It might not be immediately obvious what the back jump instruction is due to its implicit nature in commonly used languages. It refers to the action of *control flow jump* which occurs at the end of each loop iteration and which is typically caused by an explicit `JMP` instruction in low-level *assembly languages*. In control flow graphs, these back jumps are represented by oriented *back-edges* that point back to *loop header* nodes with loop conditions. As a final note, the back-edge metric is interesting because it reflects the *asymptotic time complexity* of a program: the final bound corresponds to the number of loop iterations and asymptotic complexity can be obtained by disregarding the constants.

Loop Bounds

To demonstrate this concept and others, a running example from [32] presented in Figure 3.1 will be used throughout this chapter. In essence, the so-called *loop bound* represents an

```
1 l_b : void tarjan(int n) {
2         int i~= n; (processed elements)
3         int j = 0; (current stack size)
4 l_1 :   while (i~> 0) {
5             i--;
6             j++; (push)
7 l_2 :       while (j > 0 && *)
8                 j--; (pop)
9 l_e : }
```

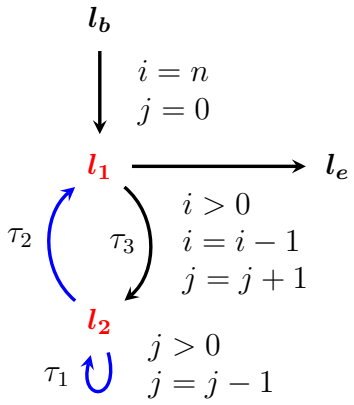


Figure 3.1: Example *tarjan* [32] models a stack which processes the total number of n elements. I.e., there are n pushes and possibly n pops due to non-determinism. The loop bounds for l_1 and l_2 are both n as it is possible to push and pop only n elements in total. The corresponding *labeled transition system* is on the right with τ_1 and τ_2 being the back-edges of interest. Note that assignments without effect such as $i = i$ are omitted here.

upper bound on how many times the control flow can return back to the loop header of an analyzed loop via any back-edge. More specifically, the loop bound for the `while` loop l_1 in Figure 3.1 corresponds with the maximum amount of times the back-edge transition τ_2 (the only back-edge for l_1) can be possibly taken to return back to the loop header. Clearly, by summing up all loop bounds of one function, the total upper bound for the execution cost is obtained.

Needless to say, the real difficulty lies in obtaining these loop bounds in the first place and the analysis that computes them has several phases. First, an abstraction algorithm is applied to a so-called *labeled transition system* (LTS) which is a type of control flow graph. The abstraction algorithm takes an LTS representation of a program as input and transforms it into a different type of control flow graph called *difference constraint program* (DCP) with *guards* over integers $\mathbb{Z}$. These *guards* are then removed in the second phase when *DCP with guards* is further abstracted to a regular *DCP* over natural numbers $\mathbb{N}$.

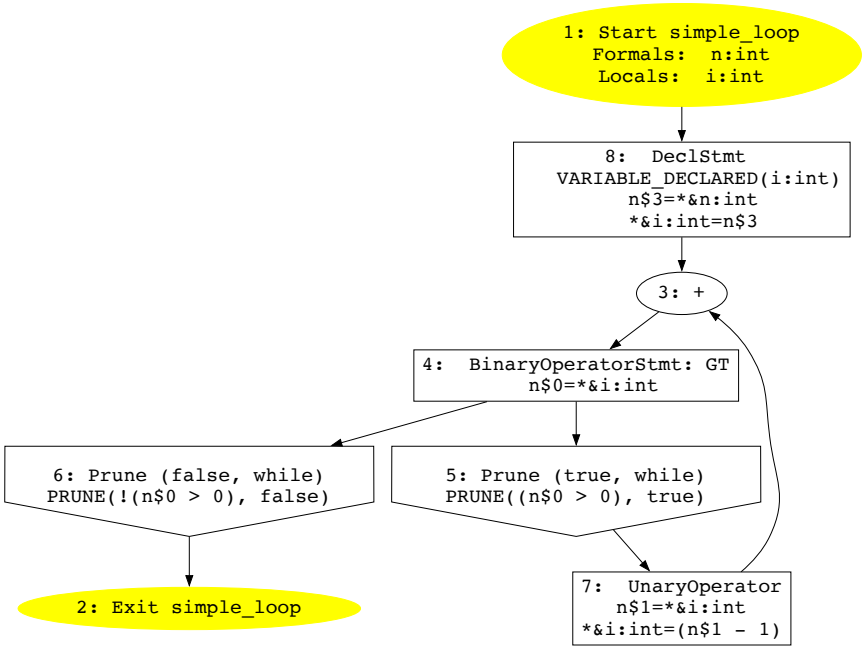


Figure 3.2: The native Infer CFG of a trivial `simple_loop` example with single while loop that decrements variable $i = n$ until zero. Infer CFG *statement* nodes contain lists of low-level SIL instructions (see Section 2.3) and edges do not contain any data. Meta-instructions were omitted for the sake of clarity

Finally, in the last phase, the *DCP* over $\mathbb{N}$ is used as an input for the *bound algorithm* that computes *loop bounds*. The *bound algorithm* itself comprises of few steps which will be briefly described in 3.1.8. This abstraction procedure is the same as the one that the original LOOPUS tool performs but Looper has to perform one more additional preprocessing step at the beginning to obtain the initial LTS graph from the Infer CFG. The following sections will cover individual abstraction stages in more detail.

3.1.1 Construction of Labeled Transition System

The first step is to transform the the native Infer CFG to the LTS program used as starting point for the abstraction algorithm. The Infer CFG is defined as follows:

Definition 3.1.1 (**Infer Control Flow Graph**). Let $\mathcal{C} = (N_\mathcal{C}, E_\mathcal{C}, n_s, n_e)$ be Infer CFG (directed labeled graph), where $N_\mathcal{C}$ is a finite set of nodes, n_s and n_e are the start and exit nodes and $E_\mathcal{C} \subseteq N_\mathcal{C} \times N_\mathcal{C}$ is a finite set of edges. We write $n_1 \to n_2$ to denote an edge $(n_1, n_2) \in E_\mathcal{C}$. Let $type(n)\colon N_\mathcal{C} \to \{start, exit, prune, join, statement\}$ be a function which maps the node $n \in N_\mathcal{C}$ to its node type. Additionally, let $\mathcal{I}_\mathtt{S}$ be a set of all `SIL` instructions and finally, let $instr(n)\colon N_\mathcal{C} \to 2^{\mathcal{I}_\mathtt{S}}$ be a function which maps the node n to a set of `SIL` instructions contained in this node.

A *start* node represents the entry point of a function, *exit* node is the exit point of a function, *prune* node is the first node in every program branch and contains the information about branching type, condition and whether it is *true* or *false* branch. Furthermore, *join* node merges two or more program paths (two CFG edges) together and finally the

statement node is a regular node without any special structural semantics. Note that contrary to the initial intuition, instructions can be contained not only in statement nodes but also in nodes of any type. An example of Infer CFG obtained from a trivial code with a single `while` loop can be seen in Figure 3.2.

These Infer CFGs serve as a starting point for the analysis in Looper which transforms them into LTS graphs that in turn served as the starting point of the original Loopus tool that inspired Looper. Loopus programs, visually represented by LTS graphs, are formally defined as follows:

Definition 3.1.2 (**Program** [32])**.** Let Σ be a set of states. A program over Σ is a directed labeled graph $\mathcal{P} = (L, T, l_b, l_e)$, where L is a finite set of locations, $l_b \in L$ is the entry location, $l_e \in L$ is the exit location and $T \subseteq L \times 2^{\Sigma\times\Sigma} \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{\lambda} l_2$ to denote a transition $(l_1, \lambda, l_2) \in T$. We call $\lambda \in 2^{\Sigma\times\Sigma}$ a transition relation.

Informally, a *run* of $\mathcal{P}$ is a sequence of transitions that starts at l_b and ends at l_e in case of *complete run*. Note that it is important to distinguish between transitions $\tau = l_1 \xrightarrow{\lambda} l_2 \in T$ and transition relations $\lambda = (\sigma_1, \sigma_2) \in 2^{\Sigma\times\Sigma}$, where l_1, l_2 are program locations (see Figure 3.1) and σ_1, σ_2 are program execution states containing values of integer variables at a specific program location. For example, the τ_3 transition from Figure 3.1 is labeled by

$$i > 0 \ \wedge \ i = i - 1 \ \wedge \ j = j - 1,$$

which encodes the following transition relation:

$$\lambda = \{(\sigma, \sigma') \in 2^{\Sigma\times\Sigma} \mid \quad \sigma(i) > 0 \ \wedge \ \sigma'(i) = \sigma(i) - 1 \ \wedge \ \sigma'(j) = \sigma(j) - 1\},$$

where $\sigma(i)$ and $\sigma'(i)$ denote the value of variable i in state σ before and after assignment respectively.

The main difference between the Infer CFG and LTS graphs is *where* and *how* they encode program instructions. Comparing the graphs in figures 3.1 and 3.2, it is immediately noticeable that Infer stores program instructions in nodes whereas LTS stores them in edges with a very different encoding and the only information stored in LTS nodes is the corresponding source code location. Moreover, due to the inherently lower level abstraction that Infer uses, the Infer CFGs are comparably more verbose. This generally results in graphs with a higher number of nodes (containing more instructions) leading to a more complicated structure overall.

Thus, the two main goals of the transformation algorithm is to construct an LTS with a less complex structure out of Infer CFG and at the same time interpret the low-level SIL instructions to construct assignments and conditions which label LTS edges. The abstraction algorithm implemented in Looper piggybacks off of the Infer *abstract interpretation* framework to achieve both goals. In particular, the framework visits all CFG nodes in an order (for example WTO, see Section 3.2) that typically minimizes the number of widening points which leads to higher precision and efficiency. Moreover, instructions of all visited nodes are interpreted (possibly several times) by the framework. The general idea of the algorithm is to take advantage of this traversal to observe visited nodes and create LTS nodes when needed. More specifically, a *start* node is created at the beginning and remembered as a last created node. When either a *prune* or *join* CFG node is encountered, new LTS node is created, connected by an edge with the previously created one and remembered as new last node. All *statement* nodes between these two points are eliminated in the LTS graph

and their contained instructions are transformed into assignments and conditions that will label the new edge. This process repeats itself until the LTS graph is complete.

This is a very high level overview of the algorithm that skips over many important details and optimizations such as how redundant edges are eliminated, how *back-edges* are detected or how multiple *join* nodes are merged together in LTS graphs to simplify the resulting structure as much as possible. As such, it is recommended to see [27] for more details and better understanding. The main issue with this approach is the nature of abstract interpretation itself, where many CFG nodes are visited repeatedly, joins and widens are performed and the interpretation continues until a *fixpoint* is reached. Moreover, by Infer design the analysis has no access to the underlying graph structure and can only access the current node and its instructions without any knowledge of graph predecessors or successors. This further complicates the transformation algorithm and led to many hacks being implemented in the end.

3.1.2 Construction of guarded DCP

Assuming that a valid LTS graph was obtained in the previous step, the first abstraction phase yielding a *guarded DCP* graph is performed next. Edges in DCP graphs are labeled by so-called *difference constraints* that are defined as follows:

Definition 3.1.3 (**Difference Constraints** [32])**.** A difference constraint over $\mathcal{A}$ is an inequality of form $x' \leq y + c$ with $x \in \mathcal{V}$, $y \in \mathcal{A}$ and $c \in \mathbb{Z}$. By $\mathcal{DC}(\mathcal{A})$ we denote the set of all difference constraints over $\mathcal{A}$.

The $\mathcal{V}$, $\mathcal{C}$ and $\mathcal{A} = \mathcal{V} \cup \mathcal{C}$ all denote finite sets of *variables*, *symbolic constants* and *atoms* respectively. Informally, all function parameters and expressions over parameters (possibly including constants $\mathtt{c} \in \mathbb{Z}$) are considered to be *symbolic constants*. Perhaps unintuitively, the $\mathcal{V}$ set in reality not only includes atomic program variables but also *variable expressions*, meaning that expressions such as $n - i$, where $n \in \mathcal{C}$ and $i \in \mathcal{V}$, belong to the $\mathcal{V}$ set because they contain at least one variable. Defined this way, *difference constraints* clearly allow *increments*, *decrements* ($x' \leq x \pm 1$) and *resets* ($x' \leq n$) to be expressed. With this in mind, the *guarded DCP* programs obtained at the end of this abstraction step are defined as follows:

Definition 3.1.4 (**Guarded Difference Constraint Program** [32])**.** A *guarded difference constraint program* (guarded DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta\mathcal{P}_G = (L, E, l_b, l_e)$, where L is a finite set of vertices, $l_b \in L$ and $l_e \in L$ and $E \subseteq L \times 2^{\mathcal{V}} \times 2^{\mathcal{DC}(\mathcal{A})} \times L$ is a finite set of edges. We write $l_1 \xrightarrow{g,u} l_2$ to denote an edge $(l_1, g, u, l_2) \in E$ labeled by a set of difference constraints $u \in 2^{\mathcal{DC}(\mathcal{A})}$ and guards $g \in 2^{\mathcal{V}}$. We use the notation $l_1 \to l_2$ to denote an edge labeled by an empty set of difference constraints and no guards. $\Delta\mathcal{P}_G$ is fan-in-free, if for every edge $l_1 \xrightarrow{g,u} l_2 \in E$ and every $\mathtt{v} \in \mathcal{V}$ there is at most one $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$ such that $\mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u$.

Informally, edges in *guarded DCP* programs are labeled solely by conjunctions of *difference constraints* and *guards* instead of conjunctions of assignments and conditions. Moreover, *guards* are subsets of $\mathcal{V}$ and their semantics are simple: the transition $l_1 \xrightarrow{g,u} l_2 \in E$ can be executed only if $\forall \mathtt{v} \in g.\ \sigma_1(\mathtt{v}) > 0$, i.e., if values of all guards are greater than zero in the program state σ_1 at the associated location l_1. Intuitively, the need for guards is due to the fact that variables in regular *DCP* programs without guards can only take values over natural numbers $\mathbb{N}$ whereas variables in guarded *DCP* programs can take any value

from $\mathbb{Z}$. As such, guards are exploited by the abstraction algorithm in the next phase to obtain a *DCP* over natural numbers. The *guarded DCP* of *tarjan* example can be seen in Figure 3.3

A brief note on the transition semantics: when $l_1 \xrightarrow{g,u} l_2 \in E$ of $\Delta\mathcal{P}_G$ is executed, $\forall(x' \leq y + c) \in u.\ \sigma_2(x) < \sigma_1(y) + c$, i.e., the value of variable x in program state σ_2 at location l_2 after executing the transition is bounded by the value of the expression $y + c$ in the program state σ_1 at the location l_1. The scope of this work does not allow for a more rigorous definition of the syntax and semantics of *difference constraints*, *guards* and *DCP* programs so please refer to [32] for more details.

The main goal of this abstraction step is to transform the assignments and conditions labeling each edge in the LTS program to *difference constraints* and *guards*. This is achieved

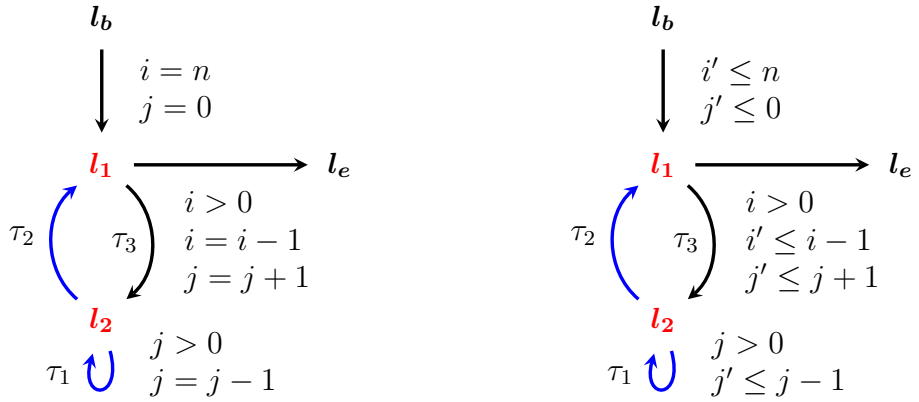


Figure 3.3: Comparison between the LTS and *guarded DCP* graphs of *tarjan* [32] example. The assignments and conditions labeling edges of LTS graph are used to derive *difference constraints* and *guards* labeling edges of DCP graph. Graphs are structurally equivalent.

in several steps using the concept of so-called *norms*. Norm is an integer valued expression that can be used to symbolically bound the number of iterations of a loop. By observing how the value of certain norms changes during the execution of a program, it is possible to determine the overall bound on the complexity.

Obtaining Initial Set of Norms N

The first challenge is how to obtain an initial set of norms which will lead the subsequent abstraction process. The [32] describes a general outline of the idea but doesn't delve into details. However, the main idea is to focus on loop conditions and extract an initial set of norms from them because complexity in imperative programs stems from iterating loops[1]. The first version of Looper implemented a rudimentary algorithm which solves this problem for a very limited subset of example programs presented in [32]. It is important to note that regardless of a specific implementation, the process of choosing the initial set of norms is inherently based on heuristics and the initial choice affects the outcome of the bound algorithm itself.

Looper tries to construct the initial set by extracting potential norms from boolean conditions of each `Prune` instruction. It considers only conditions of form $a > b$ or $a \geq b$ which can be transformed into equivalent conditions $a - b > 0$ and $a - b + 1 > 0$ respectively.

[1] Looper does not support analysis of recursive programs.

By omitting the condition part "> 0" and making it implicit, norms $a - b$ and $a - b + 1$ are obtained. Intuitively, the value of a norm thus semantically expresses the distance from zero. When Looper encounters a loop header condition of such form, it extracts a norm and places it directly in the set of initial norms[2]. Norms extracted from branching conditions along a loop path that involve some loop counter variable are first placed into a set of potential norms. A potential norm can be confirmed by a `Store` instruction that either increases or decreases the value of such norm. The first version of Looper detected only trivial increments or decrements of form $e = e \pm c$, where $e \in \mathcal{V}$ and $c \in \mathbb{Z}$. This allowed Looper to analyze all of the examples from [32] but greatly limited its usability in more realistic settings. Note that Looper builds the initial set of norms as a byproduct of the LTS construction when it interprets `SIL` instructions contained in function's CFG nodes.

Abstracting Transitions

The basic idea is to symbolically execute transitions of an LTS graph and construct difference constraints as follows:

$$\forall l_1 \xrightarrow{\lambda} l_2 \in T.\ \forall e_1 \in N.\ e_1' \leq e_2 + \mathtt{c},$$

where e_2 is obtained from e_1 by simple substitution of the assignments in λ. E.g., if $e_1 = x + y$ and $x = x + 1 \in \lambda$ then $e_2 = (x + 1) + y$. Clearly, new norms can be created during this process and the whole process has to repeat until the set N stabilizes. The goal is to keep the final number of norms at minimum while not negatively affecting the outcome of the bound algorithm. As such, new norms are only created if it is not possible to reuse some already existing ones when deriving a difference constraint. This is not only a form of optimization but in most cases a necessity due to possible non-termination of this algorithm if new norms are constantly generated. Consider the previously shown simple example of a transition with assignment $x = x + 1$ and the norm $e_1 = x + y$. A new norm $e_2 = (x + 1) + y$ is obtained through substitution and subsequently used again yielding yet another norm $e_3 = ((x + 1) + 1) + y$. Clearly this can repeat *ad infinitum* resulting in non-termination unless norms are reused.

For simplicity, original Looper implemented support for few specific types of norms and assignments because writing a sufficiently general substitution algorithm with ability to detect sub-expression norms proved to be non-trivial. The main difficulty lies in detecting sub-expression norms with regards to the associativity and commutativity properties. To circumvent this difficulty, the handling logic for specific norm types such as x, $x - y$, $\mathtt{c} - x$ and similar was hard-coded. This logic would transform each norm after substitution into a canonical form $e + \mathtt{c}$, where $e \in \mathcal{A}$ is a symbolic expression and $\mathtt{c} \in \mathbb{Z}$ is separated constant part. Such canonical form made it easy to detect if a part of an expression is equal to some already existing norm.

More detailed description of the abstraction procedure including pseudo-code algorithms can be found in [27]. It shows how the complete sets of difference constraints are constructed for each DCP edge and the main processing loop which deals with newly generated norms until the final set of norms stabilizes.

[2] During the abstraction phase Looper optimistically assumes that loops terminate and as such it expects that the value of extracted norm *will be* updated in the loop body.

Derivation and Propagation of Guards

Guards are derived after the derivation of difference constraints in the previous step. Informally, guards of a transition are norms that are guaranteed to have value greater than zero when performing the transition. Looper used Z3 *smt-solver* to derive guards for each individual DCP transition locally, i.e., it is not used for any form of a whole program analysis. Z3 is simply used to determine whether the conditions present on an LTS edge imply that value of a norm e will remain greater than zero after taking the transition, i.e.,

$$\forall l_1 \xrightarrow{\lambda} l_2 \in T.\, \forall e \in N.\, (\bigwedge_{c \in C_\lambda} c) \implies e > 0,$$

where C_λ is a set of conditions present on the transition λ.

After the initial derivation of guards, additional post-processing guard propagation step is performed. The main idea is to create a guard set intersection for all incoming edges of a location $l \in L$:

$$G = \bigcap_{\forall l_1 \xrightarrow{g,u} l \in E} g$$

Additionally, the intersection set G is pruned to not contain any guard e which is decremented on any of the incoming edges

$$g_p = \{e \in G \mid \nexists\, l_1 \xrightarrow{g,u} l.\, e' \leq e - \mathsf{c} \ \in \ u\}$$

Finally, the propagated guard set g_p is added to the initially derived set of guards g for all outgoing edges. Intuitively, if the same guard holds for all incoming edges then it must hold for an outgoing edge too unless there is a possibility that it might be decreased on one of the paths.

Looper implemented the propagation algorithm through iterative *DFS* traversal of the DCP graph which terminates as soon as the guard sets of all transitions stabilize and no new guards are being propagated. The algorithm is described through pseudo-code in more detail in [27].

3.1.3 Construction of regular DCP

The last abstraction step consists of using the obtained guard sets to transform the *guarded DCP* into a regular *DCP* defined as follows:

Definition 3.1.5 (**Difference Constraint Program** [32])**.** A *difference constraint program* (DCP) is a *guarded DCP* with the finite set of edges E redefined as follows:

$$E \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L,$$

where $u \in 2^{\mathcal{DC}(\mathcal{A})}$ is a set of difference constraints with valuation over natural numbers $\mathbb{N}$.

This transformation consists of constraining the valuation range of difference constraints to natural numbers $\mathbb{Z}$ by leveraging the guard sets and eliminating them in the process. The process is straightforward: every difference constraint $e_1' \leq e_2 + \mathsf{c}$ is transformed either to $[e_1]' \leq [e_2] + \mathsf{c}$ if $\mathsf{c} > 0$, or $[e_1]' \leq [e_2] - 1$ if $\mathsf{c} < 0$ and $e_2 \in g$ or $[e_1]' \leq [e_2]$ if $e_2 \notin g$ where $[x] = \max(x, 0)$. As a consequence, both sides of any difference constraint are guaranteed to have values greater or equal to zero even if the value of a norm is decremented on a certain edge. Loop bounds derived in later stages are obtained through syntactic manipulation of

expressions which originate from difference constraints and it is thus necessary to ensure the valuation over $\mathbb{Z}$ as a bound with negative value obviously would not be sound.

The implementation of this abstraction step in Looper is trivial and closely follows the described algorithm. An example of *guarded DCP* for the *tarjan* running example can be seen in Figure 3.4.

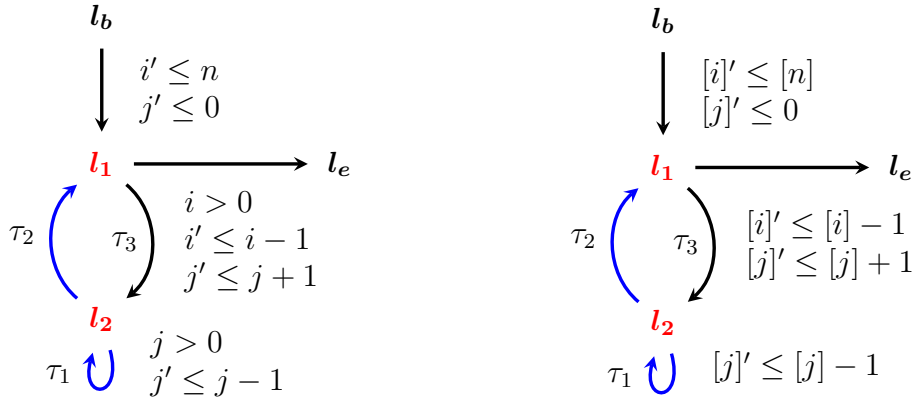


Figure 3.4: Comparison between the *guarded DCP* and DCP over $\mathbb{Z}$ graphs of *tarjan* [32] example. Regular DCP does not contain any guards and the valuation of all difference constraints is over $\mathbb{Z}$ with the use of $[x] = \max(x, 0)$ operator.

3.1.4 Bound Analysis Preliminaries

The author of [32] first presented a basic version of the bound algorithm and then proceeded to slowly build upon the main ideas in order to obtain a more powerful bound algorithm. More specifically, a bound algorithm for the special case of syntactically restricted *vector addition systems*[3] (*VASS*) was presented first, followed by an extended algorithm for *DCPs* with constant resets[4] and finally the complete algorithm for full DCPs was introduced at the end. Additionally, various precision improvement techniques such as *chained resets* or *flow-sensitive* reasoning were discussed in the subsequent sections.

Author's motivation behind this intentionally more didactic approach was to slowly build intuition starting from the simplest possible version of the algorithm in order to fully understand the inner workings before introducing a new concept. Unfortunately, the scope of this works does not allow for such approach and it is thus recommended to refer to [32] for more intuition behind the development of each concept. As such, this chapter will instead first introduce the foundational building blocks of the complete bound algorithm without focusing too much on the underlying intuition which will become apparent after gradual piece-wise introduction of the algorithm itself.

[3]VASS are strict syntactic sub-class of DCPs that allow only *monotone difference constraints* of form $x \leq x + \mathtt{c}$. Non-monotone constraints are allowed only on single initial transition from location l_b.

[4]DCPs with constant resets additionally allow *non-monotone difference constraints* $x \leq y + \mathtt{c}$ to be present on any edge, however, if such constraint is present then $y \in \mathcal{C}$, i.e., y must be a symbolic constant. As mentioned before, only expressions built over formal parameters and constants $\mathtt{c} \in \mathbb{Z}$ are considered as symbolic constants. The bound algorithm from [32] operates under the assumption that the value of formal parameters does not change during the execution of a function.

Basic Definitions

At its core, the bound algorithm syntactically differentiates between two different types of *norm* updates which are encoded in difference constraint: *increments* and *resets*.

Definition 3.1.6 (**Increments and Resets** [32])**.** Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP over* $\mathcal{A}$. Let $\mathtt{v} \in \mathcal{V}$. We define the resets $\mathcal{R}(\mathtt{v})$ and increments $\mathcal{I}(\mathtt{v})$ of $\mathtt{v}$ as follows:

$$\mathcal{R}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in E \times \mathcal{A} \times \mathbb{Z} \mid \mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u,\ \mathtt{a} \neq \mathtt{v}\}$$
$$\mathcal{I}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in E \times \mathbb{N} \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u,\ \mathtt{c} > 0\}$$

Intuitively, all difference constraints of form $x \leq y + \mathtt{c}$, where $x \neq y$ are considered to be *resets* because we're updating the value of the norm x to the value of different norm y, regardless of the constant part `c`. This type of difference constraints corresponds with our natural idea of assignments that set the value of a variable to the value of different variable such as `x = y`. Similarly, *increments* also correspond with our natural idea of assignments that increment the value of a variable such as `x = x + 5`. This assignment is represented by the *increment* difference constraints $x \leq x+5$. Technically, both resets $\mathcal{R}(\mathtt{v})$ and increments $\mathcal{I}(\mathtt{v})$ of $\mathtt{v}$ are sets of edges where the variable is either reset or incremented.

Both *resets* and *increments* are required during the computation of a bound for a certain transition. Looper heavily leverages the common practice of *caching* from *dynamic programming* and computes these sets for individual variables *on-demand* to avoid unnecessary computational overhead. The cache is defined as following partial function:

$$\chi\colon \mathcal{V} \to (E \times \mathcal{A} \times \mathbb{Z},\ E \times \mathbb{N}),$$

which maps a variable $\mathtt{v}$ to its reset and increment sets. An example of such cache is this:

$$\chi(\mathtt{v}) = \begin{cases} (\{\}, \{(l_1 \xrightarrow{u} l_3, 5)\}) & \text{if } \mathtt{v} = x \\ (\{(l_2 \xrightarrow{u} l_3, z, 2)\},\ \{\}) & \text{if } \mathtt{v} = y \end{cases}$$

When the resets or increments of a variable x are demanded, Looper first checks whether the cached results already exist and if not then it iterates over all *DCP* edges and for each edge over all difference constraints. Simple syntactic check is performed for each difference constraint to determine whether it is a reset or increment and new element is added to $\mathcal{R}(x)$ or $\mathcal{I}(x)$ respectively if so. Finally, a new mapping of $x \to (\mathcal{R}(x),\ \mathcal{I}(x))$ is added to the cache.

Definition 3.1.7 (**Counter Notation I** [32])**.** Let $\mathcal{P} = (L, T, l_b, l_e)$ be a program over Σ. Let $\tau \in T$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $\mathcal{P}$. By $\#(\tau, \rho)$ we denote the number of times that τ occurs on ρ.

Definition 3.1.8 (**Counter Notation II** [32])**.** Let $\mathcal{P} = (L, T, l_b, l_e)$ be a program over Σ. Let $\tau \in T$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $\mathcal{P}$. Let $e\colon \Sigma \to \mathbb{Z}$. By $\downarrow(e, \rho)$ we denote the number of times that the value of e decreases on ρ, i.e.,
$\downarrow(e, \rho) = |\{i \mid e(\rho_{i+1}) < e(\rho_i)\}|$

Definitions 3.1.7 and 3.1.8 establish the notions of transition execution counter and norm decrease counter. Note that 3.1.7 counts only the number of times that *one specific* transition τ is executed over the run ρ. On the contrary, the norm decrease counter 3.1.8 counts the *total number* of times that the value of e decreases on any transition of program $\mathcal{P}$. The notion of these counters is necessary to define the concept of *local bound*:

Definition 3.1.9 (Local Bound [32]). Let $\mathcal{P} = (L, T, l_b, l_e)$ be a program over Σ. Let $\tau \in T$. Let $e \colon \Sigma \to \mathbb{N}$ be a *norm* that takes values in the natural numbers. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $\mathcal{P}$. e is a *local bound* for τ on ρ if it holds that $\#(\tau, \rho) \leq \downarrow (e, \rho)$.

Informally, norm e over natural numbers is a local bound for τ on a run ρ if τ is executed less times than the number of times the value of e decreases. As was mentioned previously, it is important to keep in mind that $\#(\tau, \rho)$ refers to the total number of executions of τ, whereas $\downarrow (e, \rho)$ refers to the number of decrements of e, regardless of transition. Intuitively, the symbolic expression of a local bound norm limits the number of executions of one transition locally in isolation. Meaning, the local bound e is *only* valid under the assumption that no transition that increases the value of e is executed in the meantime.

Example 3.1.1. Consider the previously seen *DCP* graph of the running example *tarjan*:

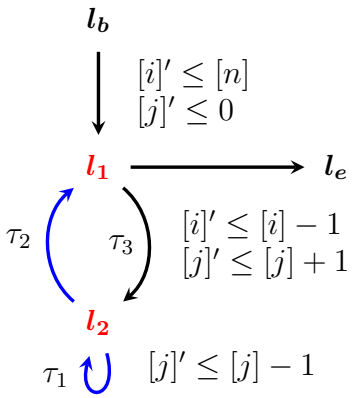


Figure 3.5

Following definition 3.1.9, norm $[j]$ represents the *local bound* for the transition τ_1 because it limits the number of executions of τ_1 as long as the transition τ_3 which increments the value of $[j]$ is not executed. Alternatively, the norm$[j]$ decreases with each iteration of loop l_2 which can be repeated only if $j > 0$. Meaning, the number of consecutive iterations of loop l_2 is limited by the norm $[j]$. However, norm $[j]$ does not limit the total number of executions of l_2 because it can be incremented on transition τ_3 and therefore it is not an overall *transition bound*. Instead, $[j]$ is merely a *local bound*.

The concept of local bounds is crucial as it stands at the core of the bound algorithm which simply tries to reason what is the overall amount by which the value of a local bound e can increase over the execution of a program. It does so by observing how many times and by how much each time can the value increase.

The main idea of the bound algorithm can be intuitively understood as a calculation of cumulative potential for the local bound e which represents the overall *transition bound*. Note that all decrements of form $[x] \leq [x] + \mathsf{c}$, where $\mathsf{c} < 0$, are transformed into $[x] \leq [x] - 1$ during the abstraction process. I.e., only decrements by one[5] can be present in *DCP*

[5] This is of course a sound approximation which only leads to less precise upper bounds.

programs. This implies that the total potential n of a local bound can be decreased n times by 1 before it reaches zero which is why it corresponds with the total *transition bound*.

Definition 3.1.10 (**Local Bound Mapping** [32])**.** Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ be a run of $\Delta\mathcal{P}$. We call a function $\zeta \colon E \to Expr(\mathcal{A})$ a local bound mapping for ρ if for all $\tau \in E$ it holds that either

1. $\zeta(\tau) \in Expr(\mathcal{C})$ and $[\![\zeta(\tau)]\!](\sigma_0)$ is a **bound** for τ on ρ or
2. $\zeta(\tau) \in \mathcal{V}$ and $[\![\zeta(\tau)]\!]$ is a **local bound** for τ on ρ.

We say that ζ is a local bound mapping for $\Delta\mathcal{P}$ if ζ is a local bound mapping for all runs of $\Delta\mathcal{P}$.

Informally, *local bound mapping* is a function that maps *non-loop* transitions to their *transition bounds* and *loop path* transitions to their *local bounds*. Intuitively, a *non-loop* transition τ can be executed exactly once, therefore $\zeta(\tau) = 1$. Note that $1 \in Expr(\mathcal{C})$. The construction of a local bound mapping is performed before the bound analysis itself which needs to have the knowledge of local bounds for all *DCP* transitions before it can be run. As discussed previously, local bounds lie at the heart of the bound analysis so a local bound mapping represents a starting point.

3.1.5 Finding Local Bounds

The previous local bound definitions did not concern themselves with the question of *how to construct a local bound mapping* and how to implement an algorithm solving this problem. The three-step algorithm presented in [32] uses the concept of *strongly connected components*[6] (SCC) to determine the initial local bound mapping.

Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP*. Let S be a set of all SCCs of P. The algorithm assumes that all SCCs for the input *DCP* graph have been previously computed. Looper uses the *ocamlgraph* library to perform this step. Finally, the algorithm steps are following:

1. $\forall \tau \in E : \nexists s \in S.\, \tau \in s \implies \zeta(\tau) = 1$. Informally, all transitions that do not belong to any SCC from S (i.e., non-loop transitions) have their bound set to 1 as they can be executed exactly once. Looper simply iterates over all *DCP* transitions $\tau \in E$ and for each edge checks whether it is part of any SCC from S using the *ocamlgraph* library.

2. According to [32]: Let $\mathtt{v} \in \mathcal{V}$. We define $\xi(\mathtt{v}) \subseteq E$ to be the set of all transitions $\tau = l_1 \xrightarrow{u} l_2 \in E$ such that $\mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u$ for some $\mathtt{c} < 0$. For all $\tau \in \xi(\mathtt{v})$ we set $\zeta(\tau) = \mathtt{v}$. Informally, if a transition τ on a loop path decreases the value of a norm $\mathtt{v}$, then $\mathtt{v}$ is the local bound for τ.

   The actual implementation follows this formal description closely: Set $D = \{\ l_1 \xrightarrow{u} l_2 \in E \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u,\ \mathtt{c} < 0\}$ is computed for every norm $\mathtt{v} \in N$ by iterating over all *DCP* transitions and performing a simple syntactic check over all difference constraints. Next, $\xi(\mathtt{v}) \to D$ is implemented by adding a *key-value* pair into a `Map` data structure. Finally, $\forall \tau \in D : \zeta(\tau) = \mathtt{v}$ which is done as a side-effect during the construction of every D. Note that Looper does not use a dedicated `Map` data structure to hold the local bound mapping, instead it stores the local bound information in each individual *DCP* edge data structure.

[6] A strongly connected component of a directed graph is a sub-graph in which every vertex is reachable from every other vertex, i.e., any two vertices of a SCC are connected by a path.

3. Lastly, according to [32]: Let $\mathtt{v} \in \mathcal{V}$ and $\tau \in E$. Assume τ was not yet assigned a local bound in previous steps. We set $\zeta(\tau) = \mathtt{v}$, if τ does not belong to any SCC of the directed graph (L, E') where $E' = E \setminus \xi(\mathtt{v})$ which is the CFG of $\Delta\mathcal{P}$ where the transitions $\xi(\mathtt{v})$ (computed in the previous step) were removed.

   Intuitively, any remaining transition τ that has not been assigned with a local bound yet must be a part of some SCC, i.e., its execution depends on the uninterrupted loop path of loop l which corresponds with such SCC. Further, if l ceases to be a loop when transitions $\xi(\mathtt{v})$ are removed, then it implies that $\mathtt{v}$ is local bound of loop l. Then it follows that $\mathtt{v}$ must also be local bound for τ due to the execution dependency of τ on l.

   The implementation first removes $\xi(\mathtt{v})$ transitions from the original $\Delta\mathcal{P}$ for selected $\mathtt{v} \in N$, then recalculates the SCCs of such modified graph and finally observes which transitions cease to be part of any SCC. Those transitions are assigned with the local bound $\mathtt{v}$. Looper systematically repeats this process for each $\mathtt{v} \in N$ until all transitions have a local bound.

As [32] notes, it is possible that more than one local bound might exist for a single transition with this algorithm. Looper adopts the same greedy approach as [32] and chooses the first viable option. Unfortunately, this approach can lead to not only lower precision but also a failure of the subsequent bound analysis if a wrong local bound is chosen. It would be more suitable to adopt a more systematic approach and backtrack in case of failure.

3.1.6 Variable Flow Graphs

Without any preprocessing done to the input *DCP* graph, the bound algorithm is flow-insensitive, which means it does not take into account that in many cases the value of a variable cannot *flow* from one program location to another. This imprecision frequently occurs when a variable is incremented or reset to a value at a later program location which affects the analysis of a loop bound at an earlier program location even though value cannot *flow* backwards. Consider the simple example in Figure 3.6a:

Clearly, the value of z at location l_2 can never flow back into z at l_1 but the bound algorithm has no knowledge of this fact as it is not encoded in regular *DCPs* in any way. To resolve this issue, Looper uses so called *variable flow graphs* (VFG) introduced in [32]:

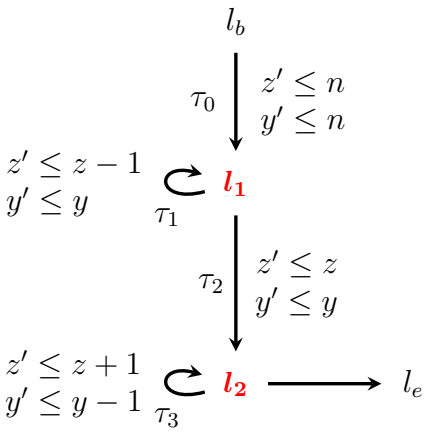
Definition 3.1.11 (**Variable Flow Graph** [32])**.** Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. We call the graph with node set $\mathcal{V} \times L$ and edge set

$$\{(y, l_1) \to (x, l_2) \mid l_1 \xrightarrow{u} l_2 \in E \wedge x' \leq y + \mathtt{c} \in u \text{ with } x, y \in \mathcal{V}\}$$

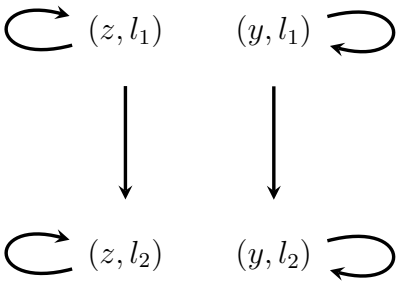
the *variable flow graph*.

An example of such VFG can be seen in Figure 3.6b. Intuitively, it shows the paths along which the values of variables can flow in the program. The formal set-builder notation directly translates to the actual implementation in Looper: a loop over all *DCP* transitions $l_1 \xrightarrow{u} l_2$ and difference constraints $x \leq y + \mathtt{c} \in u$ is performed and every difference constraint is checked whether $x, y \in \mathcal{V}$[7] If so, new nodes (y, l_1), (x, l_2) and transition between them are added to the VFG unless they already exist.

[7] As was outlined in 3.1.2, norm e is considered to be variable if the AST of expression e contains at least one program variable which is not a formal parameter.



(a) Example of a *DCP* program where *flow-insensitive* bound algorithm causes imprecision because it assumes that values from location l_2 can flow back into l_1.



(b) Example of *variable flow graph* that corresponds with the *DCP* on the left. It is used to perform preprocessing that renames program variables which produces flow-sensitive *DCP*.

Figure 3.6: An example of a regular *flow-insensitive DCP* graph 3.1.5 and its corresponding *variable flow graph*. Regular *DCPs* do not encode the variable flow information in any way which results in coarse over-approximations of the bound algorithm leading to a less precise transition bounds.

Once a VFG graph is obtained, it can be used to rename variables in the difference constraints of the original *DCP* graph. The resulting graph resembles programs in so-called *static single assignment* (SSA) form[8] which also relates to functional programming [1]. This transformation is done in two steps:

1. A transformation mapping $\varsigma \colon \mathcal{V} \times L \to \mathcal{V}$ is constructed using the VFG
2. A modified *DCP* graph $\Delta\mathcal{P}'(L, E', l_b, l_e)$ is constructed from $\Delta\mathcal{P}$. New transition set E' is generated by transforming each transition $l_1 \xrightarrow{u} l_2 \in E$ into $l_1 \xrightarrow{u'} l_2 \in E'$, where:

$$x' \leq y + \mathtt{c} \in u \implies \varsigma(x, l_2)' \leq \varsigma(y, l_1) + \mathtt{c} \in u',$$

   i.e., every variable norm $e \in \mathcal{V}$ that appears in any difference constraint of $\Delta\mathcal{P}$ is renamed using the transformation mapping ς, yielding $\Delta\mathcal{P}'$.

The ς mapping is constructed as follows: let $\{\mathtt{SCC_1}, \mathtt{SCC_2}, \ldots, \mathtt{SCC_n}\}$ be SCCs of previously constructed VFG. Then, $\mathtt{SCC_i} \to \mathtt{v}_i$, i.e., every node $(\mathtt{v}, l)$ of each $\mathtt{SCC_i}$ is mapped to an auxiliary fresh variable $\mathtt{v}_i$ as such $\varsigma(\mathtt{v}, l) = \mathtt{v}_i$. In total, n auxiliary fresh variables are generated, one for each SCC. Looper uses Infer `Pvar` module to generate these auxiliary variables.

The second step consists of looping over all transitions and difference constraints of the original *DCP* and applying the mapping. Note that the mapping function is partial as it is defined only for variable norms $\mathcal{V}$. Constant norms are thus not renamed which would be unnecessary in the first place as no updates or resets are allowed. Looper uses `Map` data structure to store the ς function. The resulting *flow-sensitive DCP* obtained via transformation of the original *DCP* from Figure 3.6a can be seen in Figure 3.7.

[8]Static single assignment form mandates that all variables are assigned only once at the moment of their definition, i.e., no updates to the variable value are permitted after the initial assignment.

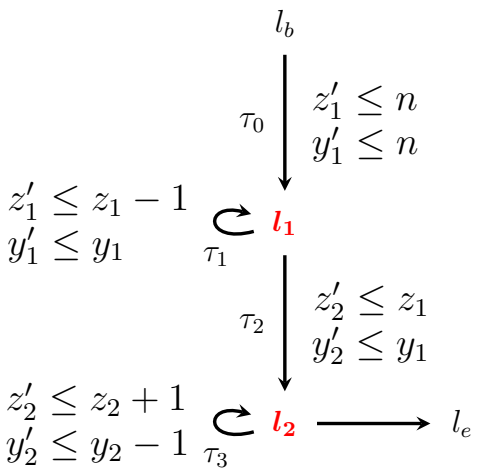


Figure 3.7: An example of a *flow-sensitive DCP* that encodes the flow information via variable renaming using a *VFG*. It resembles programs in the *single static assignment* form and does not allow backward flow of values, i.e., variable updates at later program locations do not affect the variable at earlier locations.

3.1.7 Reset Chain Graphs

The basic bound algorithm presented in [32] uses the concept of variable resets $\mathcal{R}(\mathtt{v})$ to reason about the possible bounds for values of variables. However, reasoning solely based on these resets without any additional context is another source of coarse over-approximation which often leads to transition bounds with higher *asymptotic class*. Reasoning about possible *sequences of resets* instead of isolated resets greatly improves the precision of the algorithm. For example, two assignments $y = x$ and $z = y$ form the $x \to y \to z$ reset sequence. To be able to systematically reason about all possible reset sequences, Looper uses the concept of *reset chain graphs* (RG) defined as follows:

Definition 3.1.12 (Reset Chain Graph [32]**).** Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. The *reset chain graph* or *reset graph* of $\Delta\mathcal{P}$ is the directed graph $\mathcal{G}$ with node set $\mathcal{A}$ and edges

$$\mathcal{E} = \{(y, \tau, \mathtt{c}, x) \mid (\tau, y, \mathtt{c}) \in \mathcal{R}(x)\} \subseteq \mathcal{A} \times E \times \mathbb{Z} \times \mathcal{V},$$

i.e., each edge has a label in $E \times \mathbb{Z}$. We call $\mathcal{G}(\mathcal{A}, \mathcal{E})$ a *reset chain DAG* or *reset DAG* if $\mathcal{G}(\mathcal{A}, \mathcal{E})$ is acyclic. We call $\mathcal{G}(\mathcal{A}, \mathcal{E})$ a *reset chain forest* or *reset forest* if the sub-graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a forest. We call a finite path

$$\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \dots \mathtt{a}_0$$

in $\mathcal{G}$ with $n > 0$ a *reset chain* of $\Delta\mathcal{P}$. We say that κ is a reset chain from $\mathtt{a}_n$ to $\mathtt{a}_0$. Let $n \geq i \geq j \geq 0$. By $\kappa_{[i,j]}$ we denote the sub-path of κ that starts at $\mathtt{a}_i$ and ends at $\mathtt{a}_j$. We

define several helpful functions as follows:

$$
\begin{aligned}
in(\kappa) &= \mathtt{a}_n, \\
c(\kappa) &= \sum_{i=1}^{n} c_i, \\
trn(\kappa) &= \{\tau_n, \tau_{n-1}, \ldots, \tau_1\}, \\
atm(\kappa) &= \mathtt{a}_{n-1}, \ldots, \mathtt{a}_0 \\
atm_1(\kappa) &= \{\mathtt{a} \in atm(\kappa) \mid |P(\mathtt{a}, \mathtt{v})| \leq 1\} \\
atm_2(\kappa) &= \{\mathtt{a} \in atm(\kappa) \mid |P(\mathtt{a}, \mathtt{v})| > 1\},
\end{aligned}
$$

where $P(\mathtt{a}, \mathtt{v})$ denotes the set of paths from $\mathtt{a}$ to $\mathtt{v}$ in the *reset graph*. Reset chain κ is sound if for all $1 \leq i < n$ it holds that $\mathtt{a}_i$ is reset on all paths from the target location of τ_1 to the source location of τ_i in $\Delta\mathcal{P}$. κ is optimal if κ is sound and there is no sound reset chain $\varkappa$ of length $n+1$ s.t. $\varkappa_{[n,0]} = \kappa$. Let $\mathtt{v} \in \mathcal{V}$, by $\mathfrak{R}(\mathtt{v})$ we denote the set of optimal reset chains ending in $\mathtt{v}$.

Intuitively, a *reset graph* contains a pair of nodes and an edge between them for each existing reset $(\tau, y, \mathtt{c}) \in \mathcal{R}(x)$ from the previously defined resets set 3.1.6. The original $\mathcal{R}(\mathtt{v})$ set can be seen as a set of *context-free* resets of $\mathtt{v}$ which is equivalent to a reset chain of zero length. Reset graphs thus expand the concept of resets to capture more context. Note that Definition 3.1.12 distinguishes between *reset graphs* and *reset DAGs* (and also forests) but all *reset graphs* derived from *flow-sensitive DCPs* using the *variable flow graph* are naturally *acyclic* by construction so when a reset graph or forest is mentioned it will be implicitly assumed that it is already acyclic.

An example of a *reset graph* can be seen in Figure 3.8. The enhanced bound algorithm which leverages this reset graph is able to derive the correct linear complexity for the *DCP* program on the left. The basic reasoning which leverages only *context-free* resets from $\mathcal{R}(\mathtt{v})$ is able to derive quadratic complexity. Finally, the bound algorithm can only consider *sound* reset chains in order to derive *sound* transition bounds. When considering Figure 3.8, both reset chains

$$
\begin{aligned}
n &\xrightarrow{\tau_0} r \xrightarrow{\tau_2} p, \text{ and} \\
0 &\xrightarrow{\tau_0} r \xrightarrow{\tau_2} p
\end{aligned}
$$

are *sound* according to Definition 3.1.12 because the only atom between the beginning and the end is r and it is reset on all possible *DCP* paths from l_3 (target location of τ_2) to l_2 (source location of τ_2). Intuitively, if some variable along a reset sequence is not always reset (for example due to conditional execution) then it invalidates the whole reset chain because the value might not propagate along the entire path. *Optimal* reset chains are the longest possible sound reset chains. The bound algorithm only considers *optimal* reset chains, i.e., the reset chain $r \xrightarrow{\tau_2} p$ is not considered because the aforementioned longer sound reset chains exist.

By nature, the construction of reset graphs from the reset sets $\mathcal{R}(\mathtt{v})$ is trivial in both theory and implementation. On the contrary, the algorithm for determining optimal reset chains is non-trivial as it has to use both *DCP* and *reset graphs*. Looper implements a non-optimal two phase algorithm which leverages the *ocamlgraph* library. The first phase is about finding all possible maximal reset chains by traversing the reset graph backwards starting from the origin node (reset graphs use variable norms as nodes). This translates

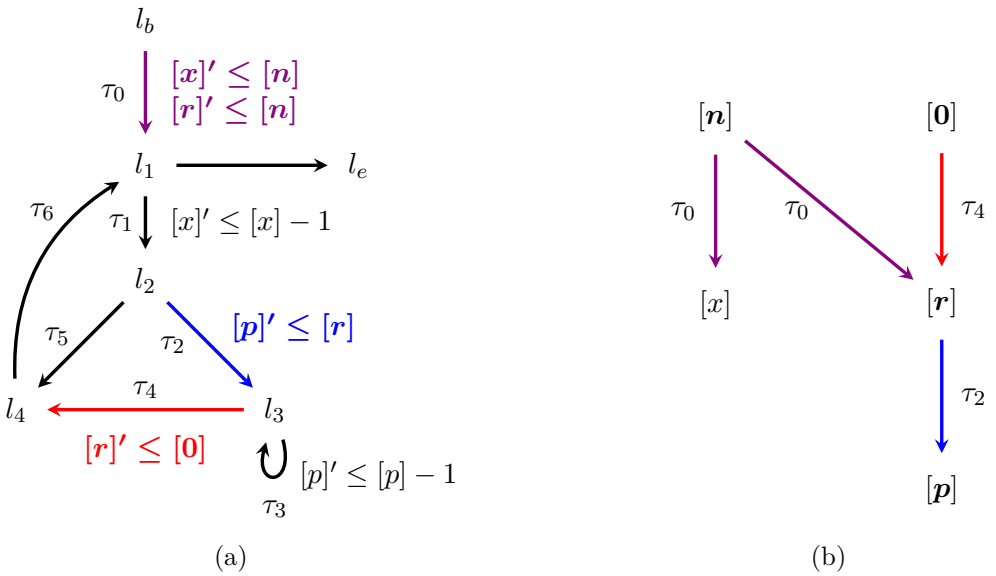


Figure 3.8: An example of a *DCP* graph that requires *reset chain* reasoning to obtain the correct linear complexity. The variable p is reset to the value of the formal parameter n through the variable r on the first execution of the transition τ_2. However, r is set to zero on τ_4 afterwards which means p will be set to zero during any subsequent execution of τ_2, leading to linear complexity. Looper uses the *reset graph* on the right to systematically reason about these reset sequences.

to performing a backwards *DFS* search and creating a new reset chain whenever a node without any predecessor is reached. This approach yields a set of reset chains with maximal lengths. Note that all reset graphs are guaranteed to be acyclic so the *DFS* algorithm is trivial in this case. Each individual chain is then systematically shortened in the second phase until it becomes sound. This ensures that the resulting set will only contain *optimal* (longest possible sound) reset chains. The idea is to start from the end of the chain and check whether each atom $\mathtt{a}_i$ is being reset on all paths between the two *DCP* locations specified in Definition 3.1.12. I.e., all paths between the source location of τ_i and target location of τ_1 are searched for resets of $\mathtt{a}_i$ using *DFS* approach.

3.1.8 Bound Analysis using Reset Chains

First to shortly summarize, the bound algorithm presented in this section can be applied to a *flow-sensitive DCP* which is constructed using the *variable flow graph* discussed in Section 3.1.6. Furthermore, all *optimal reset chains* must be found (using the *reset graphs* discussed in Section 3.1.7) prior running the analysis. Finally, the *local bound mapping* (see Definition 3.1.10) which serves as the analysis starting point has to be established.

Considering these assumptions, the previously described main idea of the bound algorithm still holds true, albeit the actual definition is considerably more complex due to the concept of reset chains which supersede the simple *context-free* resets. I.e., the main idea is to start with the local bound $\zeta(\tau)$ of a transition τ and then try to calculate how much the value of $\zeta(\tau)$ can grow in total over the course of a program run. In simple terms, this is achieved by reasoning *how many times* and *by how much each time* can the value of a local bound be incremented. The exact formal definition of the bound algorithm is as follows:

Definition 3.1.13 (Bound Algorithm [32]). Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. Let $\zeta\colon E \to Expr(\mathcal{A})$ be a *local bound mapping* for $\Delta\mathcal{P}$. We define $V\mathcal{B}\colon \mathcal{A} \to Expr(\mathcal{A})$ and $T\mathcal{B}\colon E \to Expr(\mathcal{A})$ as:

$$V\mathcal{B}(\mathtt{a}) = \mathtt{a}, \text{ if } \mathtt{a} \in \mathcal{C}, \text{ else}$$
$$V\mathcal{B}(\mathtt{v}) = \mathtt{Incr}(\mathtt{v}) + \max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})} (V\mathcal{B}(\mathtt{a}) + \mathtt{c})$$

$$T\mathcal{B}(\tau) = \zeta(\tau), \text{ if } \zeta(\tau) \notin \mathcal{V}, \text{ else}$$
$$T\mathcal{B}(\tau) = \mathtt{Incr}\left(\bigcup_{\kappa\in\mathfrak{R}(\zeta(\tau))} atm_1(\kappa)\right) + \sum_{\kappa\in\mathfrak{R}(\zeta(\tau))} T\mathcal{B}\Big(trn(\kappa)\Big) \times \max\Big(V\mathcal{B}(in(\kappa)) + c(\kappa),\ 0\Big) + \mathtt{Incr}\Big(atm_2(\kappa)\Big)$$

where $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1\leq i\ \leq n} T\mathcal{B}(\tau_i)$ and

$$\mathtt{Incr}(\mathtt{v}) = \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} T\mathcal{B}(\tau) \times \mathtt{c}, \quad \text{(we set } \mathtt{Incr}(\mathtt{v}) = 0 \text{ for } \mathcal{I}(\mathtt{v}) = \emptyset)$$

$$\mathtt{Incr}(\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n) = \sum_{1\leq i\ \leq n} \mathtt{Incr}(\mathtt{a}_i) \text{ with } \mathtt{Incr}(\emptyset) = 0$$

On the highest level, the bound algorithm is based on the interplay of two mutually recursive procedures $T\mathcal{B}$ and $V\mathcal{B}$. The first procedure $T\mathcal{B}$ calculates *transition bounds* (upper bound on the number of executions of a transition) which in most cases requires the knowledge of *variable bounds* (upper bound on the value of a variable) for certain variables. Calculation of these *variable bounds* is handled by the second procedure $V\mathcal{B}$ which in turn requires the knowledge of *transition bounds* for certain transitions, hence the mutual recursion[9].

The *variable bound* procedure $V\mathcal{B}$ will be discussed first. As mentioned before, formal parameters are considered to be constant and the upper bound for a constant is the constant itself, hence the first case $V\mathcal{B}(\mathtt{c}) = \mathtt{c}$. The computation of the *variable bound* for a variable $\mathtt{v}$ is based on the simple idea that the upper bound can be determined by adding together the highest possible initial value of $\mathtt{v}$ and the amount by which it can increase over the program run. The highest initial value for $\mathtt{v}$ can be determined by using the max operator over all possible reset values. Crucially, the $V\mathcal{B}$ procedure has to be called recursively for each operand of the max operator because the reset might contain another variable whose upper bound must be used. Finally, the $\mathtt{Incr}(\mathtt{v})$ procedure is used to calculate the total amount by which the value of $\mathtt{v}$ can increase. Intuitively, if a transition τ is executed $T\mathcal{B}(\tau)$ times and $\mathtt{v}$ is incremented by $\mathtt{c}$ each time, then their product must yield the total amount for transition τ. Obviously, this has to be done for every transition that increments $\mathtt{v}$, hence the sum.

To explain the core idea of the $T\mathcal{B}$ procedure, it is best to first consider the version using simple *context-free* resets instead of *reset chains* for a while:

$$T\mathcal{B}(\tau) = \mathtt{Incr}(\zeta(\tau)) + \sum_{(t,\mathtt{a},\mathtt{c})\in\mathcal{R}(\zeta(\tau))} T\mathcal{B}(t) \times \max\left(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0\right)$$

[9]The issue of termination and the possible causes for infinite recursion are discussed in [32] in detail.

The procedure starting point for the transition τ is its local bound $\zeta(\tau)$ whose value can be incremented over the course of the program run. However, the value of $\zeta(\tau)$ can also be reset to the value of a different variable, possibly repeatedly. The $\texttt{Incr}(\zeta(\tau))$ part computes the amount by which $\zeta(\tau)$ can increase and the remaining part deals with the possible resets of $\zeta(\tau)$. Intuitively, if a transition t contains a reset $\zeta(\tau) \leq \mathtt{a} + \mathtt{c}$ and t can be executed $T\mathcal{B}(t)$ times, then it follows that the product $T\mathcal{B}(t) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$ is equal to the total amount by which the value of $\zeta(\tau)$ can increase through resets. As before, the upper bound for the value of $\mathtt{a}$ has to be used, hence the $V\mathcal{B}(\mathtt{a})$ term.

The $T\mathcal{B}$ procedure from Definition 3.1.13 expands on this idea by replacing the *context-free* resets from $\mathcal{R}$ with *reset chains* from $\mathfrak{R}$. The basic structure of the formula stays the same but the introduction of reset chains required several changes. First, the term $\texttt{Incr}(\zeta(\tau))$ was replaced by

$$\texttt{Incr}\Big(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)\Big).$$

The reasoning behind this change is simple: if a *reset chain* with atoms $\mathtt{a}_i$, where $0 \leq i \leq n$ is considered, then it is possible that any atom $\mathtt{a}_i$ is incremented before its value is passed to the next atom $\mathtt{a}_{i-1}$ along the chain via reset $\mathtt{a}_{i-1} \leq \mathtt{a}_i$. To accommodate for this possibility, the extended term accumulates the increments of all atoms $\mathtt{a}_i$ along the chain before it finally flows into $\zeta(\tau)$ which is at the end of the chain. More precisely, it accumulates the increments only for $atm_1(\kappa)$[10] atoms of the κ chain.

Furthermore, the term

$$\sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}\Big(trn(\kappa)\Big) \times \max\Big(V\mathcal{B}(in(\kappa)) + c(\kappa),\ 0\Big)$$

also expands on the previously discussed idea about the number of times a reset can be executed by replacing the *context-free* resets with *reset chains*. Two simple observations are necessary to obtain the intuition behind these changes. First, a chain is only as strong as its weakest link, i.e., the number of executions of the entire reset chain sequence is limited by the lowest transition bound of all of its transitions. E.g., if a transition τ can be executed only once and it is part of a reset chain κ, then the entire reset sequence of κ can be realized only once. Thus, the $T\mathcal{B}$ procedure for a set of transitions is defined as the minimum of the individual bounds: $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$. Second, the upper bound for the value of a *sound* reset chain κ with atoms $\mathtt{a}_i$, where $0 \leq i \leq n$, is given by the upper bound of the last atom $in(\kappa) = \mathtt{a}_n$. Intuitively, if a chain is sound, then the value from $\mathtt{a}_n$ will eventually flow into $\zeta(\tau)$ through intermediate resets and it is thus sufficient to calculate the variable bound only for the atom $in(\kappa)$, hence the term $V\mathcal{B}(in(\kappa))$. The term $c(\kappa)$ accounts for the possible constants $\mathtt{c}_i$ in individual reset constraints $\mathtt{a}_{i-1} \leq \mathtt{a}_i + \mathtt{c}_i$.

The composition of these ideas leads to the final formula for the $T\mathcal{B}$ procedure as defined in Definition 3.1.13. The term $\texttt{Incr}(atm_2(\kappa))$ that was not discussed deals with the special case when a reset graph contains two or more paths connecting some atoms with the variable $\zeta(\tau)$. More detailed discussion of this special case (related to the concept of atm_1 and atm_2) as well as the extended explanation of the $V\mathcal{B}$ and $T\mathcal{B}$ formulas can be found [32].

Looper implements the bound algorithm through several mutually recursive OCaml procedures whose purpose and implementation will be briefly discussed now:

[10] In this case, the atm_1 function returns a set of atoms which are connected to the variable $\zeta(\tau)$ in the reset graph at maximum with one path.

- `calculate_increment_sum`: This procedure implements the

$$\texttt{Incr}(\texttt{v}) = \sum_{(\tau,\texttt{c})\in\mathcal{I}(\texttt{v})} T\mathcal{B}(\tau) \times \texttt{c}$$

formula which calculates how much the value of a variable `v` can increase in total and it recursively calls the `transition_bound` procedure. This procedure only implements the symbolic addition and multiplication of individual transition bounds with their associated increment values. I.e., the `transition_bound` procedure is called for each individual transition τ in the increment set $\mathcal{I}(\texttt{v})$ and the result is symbolically multiplied with the constant `c`. The OCaml *fold* function is then used to accumulate the total sum. In reality, Looper implements the more general

$$\texttt{Incr}(\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_n) = \sum_{1\leq i\leq n} \texttt{Incr}(\texttt{a}_i)$$

formula which accepts a set of norms and wraps the computation in one additional *fold* function that accumulates the sum of the individual atom values.

- `calculate_reset_sum`: The second procedure calculates the symbolic value of the

$$\sum_{\kappa\in\mathfrak{R}(\zeta(\tau))} T\mathcal{B}\Big(trn(\kappa)\Big) \times \max\Big(V\mathcal{B}(in(\kappa)) + c(\kappa),\ 0\Big) + \texttt{Incr}\Big(atm_2(\kappa)\Big)$$

reset sum formula. The implementation is fairly straightforward as it uses single *fold* function over all reset chains to accumulate the total value of the sum. The computation is split into three parts for each reset chain.

First, the max term is evaluated by calling the `variable_bound` procedure for the final chain atom $in(\kappa)$ and computing the value of the chain $c(\kappa)$. Reset chains are represented as lists of reset graph edges so $in(\kappa)$ trivially translates to accessing the first element of the list. Note that the *ocamlgraph* edge data structure includes both source and destination nodes which are equal to the chain atoms. On the contrary, the $c(\kappa)$ accumulates the overall reset increase value of the chain by looping through all the chain edges with another *fold* function.

Next, the transition bound of the $trn(\kappa)$ set is evaluated by calling the `transition_bound` procedure for each individual transition. The results are used as operands in the symbolic min(...) operator as per Definition 3.1.13.

Finally, the last term $\texttt{Incr}(atm_2(\kappa))$ is evaluated by calling the previously discussed `calculate_increment_sum` which handles the calculation for the set of atoms $atm_2(\kappa)$ internally.

- `variable_bound`: This procedure implements both cases of the $V\mathcal{B}$ formula:

$$V\mathcal{B}(\texttt{a}) = \texttt{a},\ \text{if } \texttt{a}\in\mathcal{C},\ \text{else}$$
$$V\mathcal{B}(\texttt{v}) = \texttt{Incr}(\texttt{v}) + \max_{(\_,\texttt{a},\texttt{c})\in\mathcal{R}(\texttt{v})} (V\mathcal{B}(\texttt{a}) + \texttt{c}).$$

First, the input norm `v` is checked whether it is a constant or not and the procedure returns the input norm itself as a bound in the case of a constant. In the other case,

the first term is handled by the `calculate_increment_sum` procedure and the max term involves folding and recursively calling itself over all $\mathcal{R}(\mathtt{v})$ resets to accumulate the arguments for the max operator.

- `transition_bound`: Finally, the implementation of the transition bound

$$\begin{aligned}
T\mathcal{B}(\tau) =& \zeta(\tau), \text{ if } \zeta(\tau) \notin \mathcal{V}, \text{ else} \\
T\mathcal{B}(\tau) =& \texttt{Incr}\left(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)\right) + \\
& \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}\Big(trn(\kappa)\Big) \times \max\Big(V\mathcal{B}(in(\kappa)) + c(\kappa),\ 0\Big) + \texttt{Incr}\Big(atm_2(\kappa)\Big)
\end{aligned}$$

formula is trivial because all the constituent parts are handled by the previous procedures so it is only matter of composing the parts together. Similarly to the variable bound procedure, the local bound $\zeta(\tau)$ of the input transition τ is first checked and $\zeta(\tau)$ itself is returned if it is a symbolic constant. The other case involves obtaining the reset chains $\mathfrak{R}(\zeta(\tau))$ for the local bound $\zeta(\tau)$ and computing the

$$\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)$$

union set which is then passed to the `calculate_increment_sum` procedure. Finally, the reset chains $\mathfrak{R}(\zeta(\tau))$ are passed as an argument to the `calculate_reset_sum` procedure and the results of both calls are symbolically added together.

Most of these formulas cannot be directly evaluated when performing intraprocedural analysis and thus the derivation of bounds is done through algebraic manipulation of symbolic expressions. Looper represents the bound expressions with `Bound` recursive data structure which extends the `Exp` Infer module with additional min and max operators and `Inf` infinity value. As these bound expressions can easily explode in size which hinders human readability, there is also an effort to simplify and possibly minimize the intermediate expressions during the computation. In fact, these algebraic optimizations are done not only during the bound analysis itself but at every step along the way with different goals. At the beginning, as part of the *DCP* abstraction phase, the difference constraint derivation algorithm tries to reorganize the symbolic expressions in order to extract the constant part and obtain constraints with the $x \leq y + \mathtt{c}$ form. This process yields more reusable norms which in turn reduces the total number of derived norms and also increases the chance that the derivation algorithm will terminate naturally without the need for forced termination[11].

In comparison, the algebraic manipulations performed during the bound analysis are more focused on eliminating unnecessary terms and overall minimizing the bound expression. More specifically, there are several steps that can be done such as unpacking and deduplicating min / max arguments, elimination of unnecessary max operators and identity elements with respect to addition or multiplication and more. For example, the bound expression $\max(\max(x, 0), x, 0)$ can be simplified to $\max(x, 0)$ and if x is an unsigned integer then the max operator can be eliminated altogether. Further, whole multiplication terms

[11] The abstraction loop can be terminated for example when a certain level of recursion is reached or when the number of norms grows past a certain threshold. However, prematurely terminating the algorithm might lead to missing some crucial norms necessary for successful bound analysis later on.

are eliminated when one of the factors is equal to zero and similarly zero terms are eliminated in addition. Note that these optimizations are heuristic in nature and do not produce minimal expressions. As such, adopting a more systematic approach based on analysis of expression ASTs or using simplification algorithms available in external provers such as Z3 would yield better results.

In summary, Looper follows the principles of *dynamic programming* and *caching* to optimize the bound analysis process. To achieve polynomial complexity of the bound algorithm, the $\mathcal{I}, \mathcal{R}, \mathfrak{R}, V\mathcal{B}$, and $T\mathcal{B}$ are all computed on demand and cached for repeated use. Finally, to obtain the total bound of a function, the $T\mathcal{B}$ procedure is called for every *DCP back-edge* and the individual results are simply added together.

3.2 Looper's Limitations

The design and implementation of the first Looper version was described in author's previous work [27]. The core ideas and algorithms from the original Loopus tool that Looper builds upon were successfully implemented and worked as intended in the first version of Looper. However, despite the progress that has been made in the context of [27], Looper struggled with performing well on *real-world* code outside of controlled environment of contrived examples. The last two chapters regarding evaluation and conclusion of [27] were already dedicated to discussing some of the current limitations of Looper, however, the major reasons why it did not perform well in *real-world* settings were neither discussed nor mentioned. This chapter is thus dedicated to the discussion about the major limitations or design issues that have been observed by the author.

Abstraction Algorithm Limitations

By far the biggest contributor to the number of analysis failure cases in the first version was the unreliability of the *Linear Transition System* construction algorithm. Looper uses the so-called *Linear Transition System* (LTS) graph abstraction to represent analyzed programs in the initial phase. This representation is subsequently transformed into a different one but that requires having correct LTS of a program in the first place. The original idea, implemented in the first version, was to leverage the Infer *abstract interpretation* framework for construction of these graphs. This solution seemed appealing at first due to its perceived simplicity but turned out to be more harmful in the long run. Common CFG traversal algorithms with node orderings such as WTO [15] or *reversed post-order* visit certain CFG nodes multiple times to apply *join* and/or *widen* operators. This is however completely unnecessary for the purpose of simple graph transformation from CFG to LTS and led to multitude of complications when trying to design a transformation algorithm in the *abstract interpretation* framework. In the end, it proved to be unnecessarily complicated and the final algorithm was too unreliable for transformation of complicated loops with complex loop header conditions (Infer CFG represents condition conjunctions or disjunctions by multiple nodes). Moreover, certain language constructs such as `switch`, `goto` or `continue` were not supported. As a consequence, most of the cases in which Looper failed were caused by inability to construct a valid LTS which is the starting point for the main analysis algorithm.

Another related issue with the first implementation was insufficient modeling of program instructions and expressions. The algorithm for transformation of integer expressions into norms and later into so-called *difference constraints* was at the time hastily implemented

and worked only for a small subset of all possible integer expressions. Only simple program expressions with addition or multiplication operators such as $i = i + 1$, $i = i * k$ and similar were supported. Support for other operators and more complex expressions was not implemented and Looper would either ignore such occurrences (the better outcome) or crash. Most affected by this were low-level C programs that heavily used bitwise operators as those have not been modeled at all. Moreover, use of data structures and pointers was not supported by the analysis at all either which significantly reduced the number of programs that was possible to analyze.

Missing Loopus Extensions

So far, Looper does not implement any of the extensions that have been proposed in [32]. These extensions either improve the abstraction procedure (such as modeling arbitrary decrements) or the analysis algorithm itself and in most cases lead to tighter bounds. However, in certain cases it helps the analysis to solve some programs that could not be solved previously (modeling boolean flags and refining control-flow). The limitations and the solutions provided by these extensions are discussed in detail in [32]. Additionally, certain inherent limitations of the *difference constraint program* (DCP) abstraction have been observed by the author when implementing the first version of Looper and that have not been addressed by any of the extensions. These limitations are caused by the restricted form of the *difference constraints* which are able to represent only following relational inequalities: $x' \leq y + c$. Obviously, this abstraction is not able to represent different inequalities involving multiplication, division or various bitwise operators in a natural way. There have been attempts to mitigate this limitation in [32] without changing the expressive power of *difference constraints*, presumably to ensure *soundness* that has been comprehensively proved for the analysis algorithms based on the previously specified form. This approach has sadly led to "hacky" extensions of sorts which seem to mostly address the symptoms and not the root cause. Moreover, some of the described extensions were actually not implemented in the latest version of the Loopus tool leaving them only in the theoretical plane. These solutions also seemed to have some gaps in logic observed by the author of this thesis when attempting to implement them in Looper.

It could be worthwhile to examine how possible changes to the expressive power of *difference constraints* could enable analysis of a wider range of programs. However, it would most likely be a non-trivial or even very challenging task to extend the existing proofs of *soundness* to these more expressive variants. Implementing these changes without proving their *soundness* in the spirit of *bug finding* approaches but warning the user in those occurrences could be an interesting compromise.

Error Handling and Recovery

Without a doubt one of the limitations impacting the practical usability in a negative way the most is non-existent error handling with recovery. This is especially apparent when analyzing large scale programs with hundreds of functions. Currently, when Looper encounters an error during the analysis of a single function, it throws an exception which is not handled in any way causing the analysis of the function to abruptly end. Moreover, if Infer catches an exception thrown during the analysis of a function then it terminates the whole currently running checker and as a consequence any remaining functions to be analyzed are skipped by the Infer scheduler. This solution is not acceptable for any kind of tool intended for practical use and it also complicates other aspects related to the tool

development such as experimental evaluation. Clearly, a reliable tool has to be resilient and be able to recover from failures which is currently one of the main improvement goals.

Reporting and Differential Analysis

Even if an analysis is able to achieve good results in its respective domain, the effort might be wasted if the results are not communicated to the user in a clear way. Unfortunately, issue reporting is one of the areas where Looper was found lacking. It is currently capable of generating log files that contain the final bound after analyzing a function but the format is not optimal. The bound expressions are not simplified or minimized in any way which leads to sometimes absurdly long expressions that are difficult to parse. Moreover, to find bound expressions for individual loops, one must search through the log file which is not user friendly. Finally, the log files do not contain any information about the worst-case asymptotic complexity $\mathcal{O}$ that could be extracted from the concrete bound. Thus, it has to be determined by the user manually if needed.

Another related major limitation is the inability to report any changes to the complexity of individual program parts upon a change when performing subsequent analysis. This so called *differential analysis* [35] which would allow Looper to track changes of complexity between different versions of analyzed program and report issues when detecting performance degradation has not been implemented yet. The existing Infer Cost checker already provides this functionality through managing several JSON files that serve as primitive database of results. Implementation of a similar differential system, either based on JSON files as well or using SQL database (Infer already uses SQL database for storing data) is another major improvement goal.

Interprocedural Analysis

Undeniably the most significant limitation of Looper in its current state is the lack of support for *interprocedural* analysis. In theory, it should not affect the ability to analyze programs apart from the obvious imprecision. Even then, such analysis could be still useful on its own in the same way that *exclusive performance profiling*[12] is useful in the field of dynamic analysis. However, it might not be immediately apparent that the inability to perform *interprocedural* analysis in fact can also severely reduce the number of loops that can be analyzed in otherwise *intraprocedural* way. The main reason are modern *range-based* for loop constructs and their variations that are prevalent in newer versions of commonly used languages such as C++, C#, Java or Python. These constructs sometimes hide the fact that there are functions being called in the background to perform the iteration and even *iterator-based* for loops in C++ involve function calls. In conclusion, the inability to perform *interprocedural* analysis affects not only the precision of the result but in many cases the analysis failure rate too.

Related issue is the missing implementation of so-called *models* that would provide summaries for library and other functions that either have unavailable source code or that cannot be analyzed for some other reason. For example, function *models* are commonly used by the Cost checker to model complexity of various container (notably Vectors, Lists and Maps) operations such as lookup or insertion in C++ or Java. They are also necessary for analysis of the aforementioned modern loop constructs that involve function calls.

[12] Exclusive profiling considers only the time spent in specific function, excluding the time spent in functions called from it

Last notable limitation of Looper that is related to the previous two issues is the support of other languages. This support should be, at least theoretically, easy to achieve thanks to the LLVM based Infer front-end module (see Chapter 2.3) that compiles all of the languages supported by Infer into a single intermediate representation. Thus, the *intraprocedural* version of Looper should in fact be able to analyze traditional loops in different languages but this remains to be tested. Nevertheless, each language will most likely still require creation of many new function *models* to be able to analyze language-specific constructs and containers.

Chapter 4

Proposal of Enhancements for Looper

The author's previous work [27] already provided a brief overview of possible future improvements of Looper but did not offer any deeper technical discussion about the specific ideas. Furthermore, the previous work did not include any systematic investigation on the limitations of the first version of Looper or even the original Loopus tool [32]. As such, the previous Section 3.2 attempted to summarize the major issues that were discovered during subsequent extensive experimentation with *real-world* code. Note that most of these limitations also apply to the original Loopus tool which is no longer actively developed. This chapter builds upon the findings summarized in the previous section and discusses the proposed solutions which mostly aim at enhancing the practical usability and precision of the analysis. Moreover, the issue of scalability is also considered. This issue was not discussed in Section 3.2 as it was not possible to evaluate the scalability of the first version of Looper due to its past inability to analyze a vast majority of *real-world* or even benchmark code. Chapter 6 then presents an experimental evaluation which demonstrates the *scalability* potential by analyzing the evaluation results of the new Looper version which is finally able to handle enough code to draw any meaningful conclusions from the data.

The remaining part of this chapter is structured as follows: Section 4.1 covers the replacement of the old and mostly hard-coded abstraction algorithm which was one of the major reasons why Looper could not cope with the vast majority of typical C code. The new abstraction algorithm attempts to alleviate this functional bottleneck which prevented most code from ever reaching the phase of bound analysis itself.

The following Section 4.2 describes a few additional extensions of the core algorithm which have not been previously implemented in Looper. Note that some of the extension ideas were inspired by the original Loopus tool [32] and adapted for the Infer framework.

By nature of the problem, any static analysis tool is limited when it comes to the code it can analyze and Looper is no exception. Unfortunately, compared to other more mature tools, Looper lacked any reliable error handling and recovery mechanisms which meant that a failure during the analysis of a single function would prevent the analysis of the remaining code. This was another major reason why it was not possible to properly benchmark, let alone deploy Looper for any practical use. Note that this chapter does not cover the proposed solution but Section 5.2 briefly addresses how these issues were solved from the implementation standpoint.

Finally, Section 4.3 provides an in-depth look at the most significant contribution over the original work — the *interprocedural* extension. The technical discussion included in that section covers the challenges and difficulties encountered when extending the original *intraprocedural* analysis as it provides useful insights into why specific design decisions have been made. Not only this extension increases the precision of the analysis but it also allows Looper to handle cases that were previously not supported, such as using the return value of a function call in a loop condition.

The implementation details for all previously mentioned sections are provided in Chapter 5.

4.1 New Abstraction Algorithm

Section 3.2 covered the limitations of the original abstraction algorithm in detail and discussed the reasons why it was necessary to replace it. There were two main deficiencies that had to be addressed:

1. The frequent inability to construct the initial *labeled transition system* (LTS) graph from the analyzed function due to functions with more complex control flow structures, which frequently posed a challenge for Looper. In most cases, the constructed LTS graph either had an incorrect structure, or the algorithm failed to produce any output altogether, resulting in a crash.

2. Constructing structurally correct LTS was not enough by itself and deriving valid LTS edge assignments out of the low-level `SIL` instructions posed another major challenge. The `SIL` intermediate language operates over memory locations denoted by identifiers `Ident.t` which can be populated with heap or stack values using the `Load` instruction. As such, it was necessary to manually keep track of the values stored in each identifier and then subsequently use these values to reconstruct the original program expressions when interpreting the `Store` instruction. Most of this handling logic was originally hard-coded, and Looper supported only basic assignments such as $i = x$ or $i = i \pm \mathtt{c}$. While this limited support was sufficient to analyze all the examples in [32], real-world code is significantly more complex and requires handling of pointers, data structures, and field accesses, among other things.

4.1.1 Manual Construction of LTS

As discussed in Section 3.2, the original Looper leveraged the Abstract Interpretation framework of Infer in order to construct the initial LTS. However, this approach was deemed not suitable for further development and had to be reimagined. Thus a special-purpose approach was adopted instead, and each LTS is constructed by directly traversing the nodes of an Infer CFG instead of relying on the traversal order of the AI framework. Hence, the main goal of transforming the Infer CFG from Definition 3.1.1 to LTS from Definition 3.1.2 remains the same but the transformation algorithm is completely new. To illustrate the proposed solution, a simple code example with the corresponding Infer CFG in Figure 4.1 and LTS in Figure 4.2 will serve as a running example. These graphs represent the input and the desired output of the new abstraction algorithm, respectively. The C code example in Listing 4.1 presents a modified version of the `remove_suffix` function, which was taken from the GNU *coreutils* repository.

```
1 static void remove_suffix(char *name, char const *suffix) {
2   char *np = name + strlen(name);
3   char const *sp = suffix + strlen(suffix);
4   while (np > name && sp > suffix) {
5     np--; sp--;
6   }
7 }
```

Listing 4.1: A simple code example extracted from the GNU *coreutils* repository. It contains a compound loop condition which produces an Infer CFG that Looper could not previously handle.

As with the first Looper version, it is important for the new abstraction algorithm to transform the Infer CFG into a valid LTS without losing the branching semantics encoded by the structure of the original Infer CFG. The most obvious difference between the two graph types is the fact that the Infer CFG encodes the program computation via node labels whereas the LTS encodes computation via edge labels.

Apart from this, the structure of both graphs is also very different. The Infer CFG uses a lot of intermediate nodes whereas the LTS attempts to *compress* the structure of the graph as much as possible, frequently merging several CFG nodes into one LTS edge. The most notable difference is the handling of compound boolean conditions in loop headers. Infer decomposes these conditions and creates one CFG branching node with `true` and `false` branches per each atomic condition as can be seen in Figure 3.2 with the `np > name && sp > suffix` compound condition. On the contrary, the LTS creates a single branching node per a loop header and stores the entire compound condition on the `true` edge which enters the loop body. The reason why reconstructing the entire compound condition out of several CFG nodes and storing it on single edge is beneficial and how it is used to improve precision of transition bounds will be discussed in Section 4.2. Note that the first version of Looper did not support compound conditions at all, and the choice of a local bound norm (these norms were generated from the individual atomic conditions) for the subsequent bound analysis was non-deterministic. The resulting transition bounds could therefore differ depending on which atomic condition was chosen as the basis for the analysis. As such, the first Looper version did not attempt to compress these individual branching nodes into a single node at all.

The new abstraction algorithm is based on explicit *depth-first search* (*DFS*) traversal of the Infer CFG which gathers necessary data along the way and creates new LTS nodes and edges on demand. Note that this section will focus solely on how the correct LTS structure is obtained, and the interpretation of `SIL` instructions will be discussed in Section 4.1.2. The construction process will be described semi-formally.

The main idea is to transform the Infer CFG $\mathcal{C} = (N_\mathcal{C}, E_\mathcal{C}, n_s, n_e)$ into the LTS $\mathcal{P} = (L, T, l_b, l_e)$ by accumulating auxiliary data during the traversal of the Infer CFG and by creating LTS nodes and edges at certain points during the traversal. The notation $out(n), in(n)$, where $n \in N_\mathcal{C}$, will be used to represent the CFG node out-degree and in-degree, respectively. Additionally, let $M \subseteq N_\mathcal{C} \times L$ be the set of associations between Infer CFG and LTS nodes. The transformation starts with initialization of the sets $L = \{l_b\}$, $T = \{\}$, and $M = \{\}$, where l_b is the LTS start node. The rest of the transformation process consists of handling special node types during the *DFS* traversal.

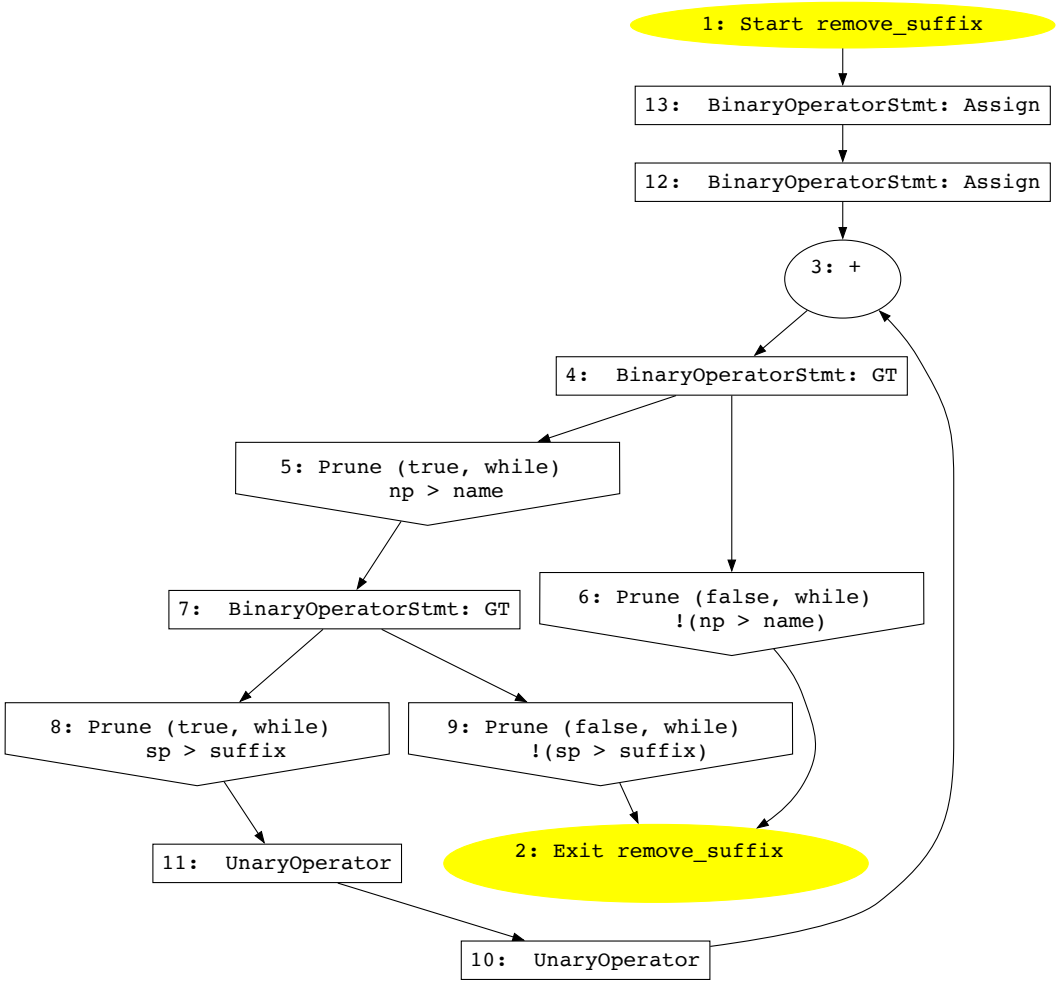


Figure 4.1: The Infer CFG obtained from the modified `remove_suffix` function presented in Listing 4.1. The `SIL` instructions contained in each node were omitted for the sake of clarity.

Handling of Branching Nodes

The first type of nodes which needs a special handling are the CFG branching nodes defined as follows: $n_p \in N_C : out(n_p) = 2$. Two examples of such nodes are the nodes with IDs 4 and 7 in Figure 4.1. There are two possible options when a branching node is encountered.

1. First, a following notation $n \in n'_p \to \cdots \to n_p$ will be used as a shorthand for $n \in \{n'_p, \ldots, n_p\}$ where $\{n'_p, \ldots, n_p\}$ is the set of nodes that appear in the path $n'_p \to \cdots \to n_p$. Assume there is a map $(n'_p, l'_p) \in M$ where n'_p is a previously encountered Infer CFG branching node with its associated LTS node l'_p. Then, let $n'_p \to \cdots \to n_p$ be the path between n'_p and n_p which was taken by the *DFS* algorithm. If it holds that $\forall n \in n'_p \to \cdots \to n_p : type(n) \neq join$, then no new LTS node is created and a new association is established only by setting $M' = M \cup \{(n_p, l'_p)\}$. I.e., instead of creating multiple LTS nodes for one loop head with a compound condition, a single

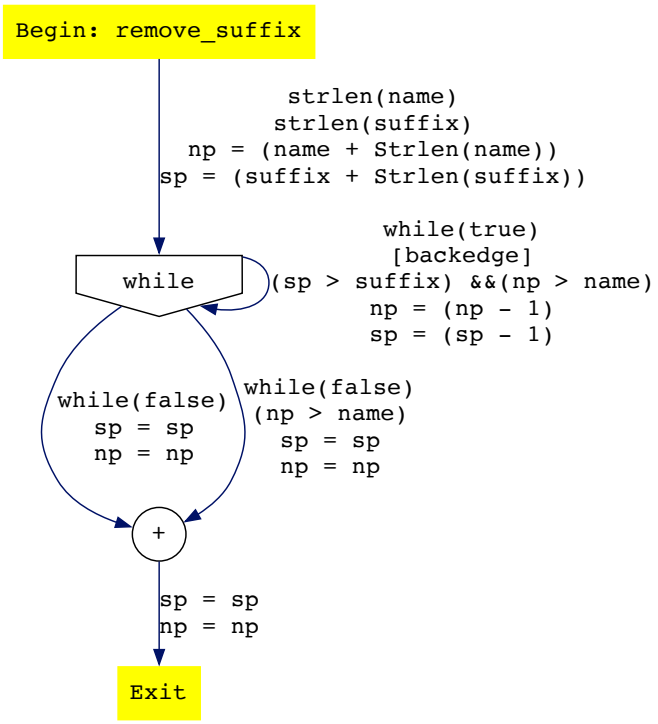


Figure 4.2: The LTS obtained from the Infer CFG in Figure 4.1 using the new abstraction algorithm.

LTS node is used to represent all of the Infer CFG branching nodes associated with the single loop head.

2. Suppose that $\exists n \in n'_p \rightarrow \cdots \rightarrow n_p$ such that $type(n) = join$, or no branching node n'_p was previously encountered. Let $l \in L$ be the last previously created LTS node on the current *DFS* path. Then a new LTS node l_p and an LTS edge $l \xrightarrow{\lambda} l_p$ are created and the corresponding sets are updated accordingly: $L' = L \cup \{l_p\}$, $T' = T \cup \{l \xrightarrow{\lambda} l_p\}$, $M' = M \cup \{(n_p, l_p)\}$. Intuitively, Infer generates only one loop-head per loop and a *join* node in this case marks a loop-head. Thus, if a *join* node appears on the path, then it means a new loop was encountered and it is not a case of a compound condition for one loop. Hence, a new LTS node for another loop is created and no merge is performed.

Handling of Merge Nodes

Any node $n_j \in N_{\mathcal{C}} \colon in(n_j) = 2 \ \wedge \ type(n_j) \neq join$ needs a special handling because the fact that $in(n_j) = 2$ implies that n_j will be visited twice by the *DFS* traversal but only one LTS node with two incoming edges should be created. Again, let $l \in L$ be the last previously created LTS node on the current *DFS* path. When the node n_j is encountered for the first time, a new LTS *join* node l_j and an LTS edge $l \xrightarrow{\lambda} l_j$ is created. Additionally, a new association (n_j, l_j) is established, and the sets are updated as follows: $L' = L \cup \{l_j\}$ and $T' = T \cup \{l \xrightarrow{\lambda} l_j\}$. When n_j is visited for the second time, the association (n_j, l_j) must

exist, and no new LTS node is created. However, a new edge is added to the set of edges: $T' = T \cup \{l' \xrightarrow{\lambda} l_j\}$ where l' is the last LTS node created on the second incoming *DFS* path.

Note that, by definition, Infer CFG *join* nodes always have two incoming edges with one of them being a loop back-edge. However, the LTS does not require a dedicated loop head *join* nodes, and hence the condition $type(n) \neq join$ as these nodes are not considered when creating LTS *join* nodes. An example of a node satisfying this condition is the exit node in Figure 4.1.

Handling of Exit Nodes

The last node type which needs special handling is the Infer CFG exit node defined as follows: $n_e \in N_C : in(n_e) \geq 1 \wedge out(n_e) = 0$. When the Infer CFG exit node is encountered, a new LTS exit node l_e is created and added to the set of nodes $L' = L \cup \{l_e\}$. A new LTS edge $l \xrightarrow{\lambda} l_e$ is also created and added to the set of edges $T' = T \cup \{l \xrightarrow{\lambda} l_e\}$ where the node l is defined as before. Even though the CFG exit node can have more than one incoming edge, the LTS exit node l_e will always have only one incoming edge. This is ensured by creating a predecessor LTS merge node l_j which merges all of the incoming edges first if $in(n_e) \geq 2$. An example of this construction can be seen in Figure 4.2.

Detecting Back-Edges

The last issue that had to be solved was the detection of *back-edges*. The fact that the Infer CFG always contains a single *join* node per loop head was leveraged, and the main idea is to detect when a *join* node is visited for the second time. Since $\forall n \in N_C : type(n) = join \implies in(n) = 2$ and since one of the incoming edges is guaranteed to be a *back-edge*, the second visit by the *DFS* forward traversal, starting from the Infer CFG start node n_s, must be via the *back-edge*. Thus, when the *DFS* traversal algorithm detects a visit of a previously visited node $n_j \in N_C : type(n_j) = join$, then a new edge $\tau_b = l \xrightarrow{\lambda} l_p$ marked as a *back-edge* is added to the set of edges $T' = T \cup \{\tau_b\}$. The node l is defined as previously, and it holds that $\exists (n_j, l_p) \in M$.

Note that this section only presented an outline of the solution and that the actual transformation process is a bit more involved, including a few additional steps which help with reducing the size of the final LTS. Chapter 5 describes how the abstraction algorithm is implemented in Looper and provides some more details.

4.1.2 Improved SIL Intepretation

Contrary to the previously described changes to the construction algorithm for the LTS structure, the `SIL` instruction interpretation core was not completely replaced but rather improved and extended. In fact, the function `exec_instr` which used to be plugged into the AI framework remained mostly the same with some changes to its parameters due to it now being called during the *DFS* traversal. This section thus focuses on how the LTS edge labels are constructed, i.e., how the assignments, conditions and norms are obtained during the interpretation.

Access Paths

The first major change regards the expressions which were allowed to appear on the left-hand side of LTS assignments. The first version of Looper used the `Pvar.t` (short for a program variable) type for the left-hand side which meant that only simple program variables such as `int x` could appear on the left. Unfortunately, *real-world* code often uses complex memory addressing methods, such as array indexing, nested structure field accesses, or pointer arithmetic. These access patterns were previously not supported at all which greatly limited the amount of code that could be analyzed. To alleviate this problem, the concept of *access paths* was employed as it was easily available in Infer which already implements it with the `AccessPath` module. The concept of *access paths*, i.e., roughly, the expressions (such as `x.y.z[10]`) used to access some value, was briefly discussed in [17] along with its limitations, especially when it comes to *aliased* accesses[1]. In short, *access paths* serve for a *syntax-based* representation of *heap* or *stack* locations. I.e., an access expression such as `x.y.z[10]` (`x` is a structure variable and `y`, `z` are its nested fields with `z` also being an array) is used to uniquely identify a memory location. Clearly, this is not precise when one considers that the same access expression can in fact point to different memory locations at different points during the execution or inversely that two different access expressions can point to the same memory location, i.e., alias. However, this is not of a much concern for this work as the new version of Looper (and Meta Infer itself) mostly aims for the *bug finding* approach discussed in Section 2.1.6. As such, the issue of aliasing was mostly ignored. Moreover, as mentioned in [17], *access paths* alone are in many cases enough to claim with a certain degree of certainty that no aliasing occurs under certain circumstances. *Access paths*, as implemented in Meta Infer, can be formally defined as follows [17]:

$$\pi \in \Pi ::= Var \times Field^*,$$

where Π is the set of all access paths, Var is the set of all program variables which can serve as a *base* of the access path, and $Field$ is the set of all existing field names (structure fields, class members, etc.). The expression `x.y.z[10]` can thus be formally written as $(x, \{y, z\})$.

Edge Expressions

Looper previously leveraged the `Exp` module provided by Infer to represent assignments and *difference constraint* expressions. Due to its limitations, namely not supporting min or max operators and ∞ values, an additional `Bound` wrapper module was implemented to extend the original `Exp` module. Using two different data types for inherently the same purpose, however, proved to be cumbersome and as such the decision to come up with a new solution was made.

The new version of Looper implements a custom `EdgeExp` module to represent all expressions used during the analysis. I.e., the same data type is used for expressions in assignments, difference constraints, and derived bounds. To partially formalize the *EdgeExp* expressions, let $\mathcal{T}$ be the set of all program types, let $\mathcal{F}$ be the set of all callable program functions, let $EdgeExp$ be the set of all expressions representable by the new

[1] Meta Infer does not implement any kind of *alias analysis* due to its cost and also because it goes against their analysis design philosophy, which was briefly discussed in Section 2.1.4. In particular, none of the existing state-of-the-art alias analysis techniques works compositionally, which is the core principle for Infer's scalability.

`EdgeExp` module, and, finally, let $\mathcal{S}$ be the set of all function *summaries*. Section 4.3 will describe how Looper leverages the concept of *summaries* for *interprocedural* analysis. The new recursive data structure then supports construction of arbitrary expressions using unary and binary operators and additionally supports the min, max, the string length function $strlen(x)\colon \Pi \to \mathbb{N}$ mapping strings accessed by access paths to their length, and the type cast function $cast(x)\colon EdgeExp \times \mathcal{T} \to EdgeExp$ which maps input expressions to new expressions based on the supplied type. Additionally, it also supports the function call $call(f, args)\colon \mathcal{F} \times 2^{EdgeExp} \to \mathcal{S}$ operator. The atomic terms of expressions can either be access paths $\pi \in \Pi$, constants $\mathtt{c} \in \mathbb{Z}$, *strlen* operator function or infinity ∞ which can be utilized as the top $\top$ value for variable and transition bounds. Conversely, the $-\infty$ can be utilized as the bottom $\bot$ value for variable bounds.

Assignment Representation

The first version of Looper used the `Exp` module for expressions in assignments α which were previously defined as follows:

$$\alpha \in \mathcal{A} ::= Pvar \times Exp$$

where $\mathcal{A}$ is the finite set of all possible assignments, $Pvar$ is the set of all program variables and Exp is the set of all expressions representable by the old `Exp` module. Intuitively, the program variable $p \in Pvar$ was set to the value of the expression $e \in Exp$. The new version of Looper redefines the old definition of assignments α to leverage the concept of *access paths* as well as the new `EdgeExp` module:

$$\alpha \in \mathcal{A} ::= \Pi \times EdgeExp \times EdgeExp.$$

For instance, the tuple $((x, \{y, z\}), 0, \infty) \in \mathcal{A}$ is an example of a valid assignment which can be informally represented by the following notation $x.y.z = [0, \infty]$. Apart from using the access expressions, the most significant change is that the right-hand side no longer consists of a simple value but instead an interval is used. The reason for this change is rooted in the added support for *interprocedural* analysis and will be discussed in more detail in Section 4.3.

Assignment Derivation

Finally, the derivation algorithm for edge assignments had to be modified to produce assignments according to the new definition. To achieve this, the interpretation of individual `SIL` instructions in the `exec_instr` function had to be significantly adjusted even though the principle of the original approach stayed the same. The semantics of each individual instruction will now be defined semi-formally:

- Load: As discussed before, the intended use of this instruction in Infer is to load *heap* or *stack* values addressed by an expression into temporary variables. Let $Ident$ be the set of all temporary variables representable by the Infer `Ident` module and let Exp be defined as before. Then, $i \leftarrow e$ denotes the instruction loading a value given by $e \in Exp$ into the identifier $i \in Ident$.

  Looper follows this convention and keeps the intended semantics. Let $\mathcal{M} ::= Ident \times EdgeExp \times EdgeExp$, then let $M \subseteq \mathcal{M}$ be a set of associations between identifiers and their associated intervals represented by two $EdgeExp$ values. E.g., $(i, e_1, e_2) \in M$

represents the association between the identifier $i \in Ident$ and the interval $[e_1, e_2]$, where $e1, e2 \in EdgeExp$. The following notation $i \mapsto [e_1, e_2]$ will be used to denote that such an association exists in M, and $M(i)$ will be used to refer to the interval associated with i. Furthermore, let $transform(x, m)\colon Exp \times \mathcal{M} \to EdgeExp \times EdgeExp$ be a function that converts an *Exp* expression into an interval of *EdgeExp* expressions using the set of associations M.

Internally, the function $transform(x, m)$ uses a recursive *bottom-up* algorithm to construct the expressions $e_1, e_2 \in EdgeExp$ from $e \in Exp$. The main goal is to replace all temporary variables $i \in Ident$, which can appear in the expression e, with their associated values from M. I.e., if $\mathcal{I}(e) \subseteq Ident$ denotes the set of all identifiers present in e, then it is transformed as follows: $\{M(i) \mid i \in \mathcal{I}(e)\}$. Apart from that, the function simply converts Infer *Exp* terms into their *EdgeExp* counterparts, e.g., an *Exp* binary operator expression $e_l + e_r$ is transformed by recursively transforming the subexpressions $e_l, e_r \in Exp$ to $e'_l, e'_r \in EdgeExp$ and finally a new *EdgeExp* expression $e'_l + e'_r$ is constructed.

When a `Load` instruction $i \leftarrow e$ is encountered, the expression e is used to construct the pair of *EdgeExp* expressions, and the set M is updated with the result. Formally, let $transform(e, M) = (e_1, e_2)$, then $M' = M \cup \{(i, e_1, e_2)\}$.

- STORE: The `Store` instruction leverages identifiers created by `Load` instructions to store the value of an expression into the *heap* or *stack*. Let $e_l, e_r \in Exp$ be expressions, then $e_l \leftarrow e_r$ denotes a store instruction which stores the value of the expression e_r into a memory location addressed by the expression e_l. Note that e_r is always an expression built solely over identifiers and constants.

  When Looper interprets a `Store` instruction, it constructs an assignment and adds it to the set of assignments $A \subseteq \mathcal{A}$ which will be used to label an LTS edge. Formally, let $(l_1, l_2) = transform(e_l, M)$ and $(r_1, r_2) = transform(e_r, M)$ such that $l_1 = l_2 \wedge l_1 \in \Pi$, then $A' = A \cup \{(l_1, r_1, r_2)\}$ where $\Pi \subset EdgeExp$. Note that Looper currently expects that both the l_1 and l_2 left-hand expressions produced by the $transform(x, M)$ function are access paths, because values can only be stored to memory locations denoted by access paths. The equality $l_1 = l_2$ enforces that only one assignment is derived since Looper currently does not support the case when $l_1 \neq l_2$ which would cause two assignments to be generated.

- CALL: Any function call is represented by a single `Call` instruction which can be formally defined as an element of the set $Ident \times Exp \times Exp^* \times \mathcal{L}$ where $\mathcal{L}$ is the set of all program locations uniquely identified by a line and column in the source code. Informally, a function call consists of an identifier for the return value, an expression which identifies the called function, a sequence of call-site arguments, and, finally the source code location of the call.

  The main purpose of the `Call` instruction is to provide a way for instantiation of a function *summary* at a call site. Looper uses summaries primarily to store the symbolic bounds for the execution cost of a function, but it also stores the lower and upper bounds for the possible return value as well as integer-typed pointer parameters which can be subject to *side-effects*. Informally, summary instantiation denotes the result of replacing formal parameters in all elements of a summary by the actual argument values at a specific call-site. Without discussing the technical details of

summaries and their instantiation (Section 4.3 covers these topics), let $B \subseteq \Pi \times EdgeExp \times EdgeExp$ be the set of instantiated bounds for formal parameters, let i be the identifier created for the return value, and let $(e_1, e_2) \in EdgeExp \times EdgeExp$ be the instantiated bounds for the return value if any exists. Then, $A' = A \cup B$ and $M' = M \cup \{(i, e_1, e_2)\}$. Furthermore, let $s \in \mathcal{S}$ be the summary obtained by instantiation of a summary of a function call at location $l \in \mathcal{L}$. Finally, let $F \subseteq \mathcal{L} \times \mathcal{S}$ be the set of associations between program locations and instantiated summaries tracked by Looper across the entire analyzed program. The set F is updated by the `Call` instruction as follows: $F' = F \cup \{(l, s)\}$.

- PRUNE: The `Prune` instruction handles the conditions which are used for program branching. It is formally defined as an element of the set $BExp \times \{true, false\} \times \mathcal{L}$ where $BExp \subset Exp$ is the set of all Boolean expressions which are used as conditions. The Boolean value $b \in \{true, false\}$ indicates whether the `Prune` instruction belongs to the *true* or *false* branch.

  Looper uses the `Prune` instruction primarily to derive new norms and to store the normalized form of the condition $c \in BExp$ on the LTS edge which is being constructed at the moment. Let $transform(c, M) = (c_1, c_2)$ be the $EdgeExp$ Boolean expressions constructed out of the condition $c \in BExp$. Looper currently uses the expression c_2 for further analysis and norm derivation only. Intuitively, the expressions c_1 and c_2 represent the "lower" and "upper" bounds of the expression c where c_2 corresponds with the originally used expression before the interprocedural analysis and lower variable bounds were introduced in this work. No suitable use-case for the lower bound value of the condition expression has been found yet and hence it is ignored.

  Let $C \subseteq EdgeExp$ be the set of conditions associated with the current LTS edge. Then, $C' = C \cup \{e_2\}$ is the updated set of conditions after interpretation of the `Prune` instruction.

Interpretation of each `SIL` instruction is in reality a bit more complicated as it involves derivation of norms and also various expression optimizations to obtain assignments and conditions in a certain canonical form. However, most of these steps are considered to be implementation details and as such were not discussed here. Furthermore, the principles of norm derivation remained mostly unchanged since the first version of Looper and thus the description provided in Section 3.1.2 is still relevant. A few minor changes to the norm derivation process caused by the introduction of *interprocedural* analysis will be discussed in Section 4.3.

4.1.3 Difference Constraint Derivation

As discussed in Section 3.1.2 and Section 3.2, the constraint derivation logic in the first version was *hard-coded* and handled a very limited subset of all possible expressions only. Instead of relying on specific types of expressions, the solution we proposed was to rewrite the derivation algorithm to make it more general. The basic concept remained the same but the derivation is now done in two steps:

1. Let $\tau = l_1 \xrightarrow{\lambda} l_2 \in T$ be an LTS edge and let $A \subseteq \mathcal{A}$ be the set of assignments encoded by the transition relation λ. Let $N \subseteq EdgeExp$ be the initial set of norms obtained during the LTS construction and let $e \in N$ be the currently processed norm. Then,

the first step is to attempt to substitute all *access path* terms $\pi \in \Pi$ of the norm e, i.e., to replace each term π by the values associated with them:

$$\pi' = \begin{cases} r, & \text{if } (\pi, r, r) \in A, \\ (r_1, r_2), & \text{else if } (\pi, r_1, r_2) \in A, \text{where } r_1 \neq r_2 \\ \pi, & \text{otherwise.} \end{cases}$$

The obvious issue is that it is not possible to simply replace a value term with an interval at a leaf node in the AST of e because the `EdgeExp` module does not support intervals as values. The solution was to split the tree into two new sub-trees r_1 and r_2 at any *access path* leaf node π. These new sub-trees are then propagated upwards until the root node of e is reached at which point two new expressions e_1 and e_2 are created. Binary and unary operators along the original AST of e are applied between pairs of nodes during the bottom-up creation of e_1 and e_2. This is somewhat similar to how interval domain operators are evaluated during abstract interpretation (e.g., $x + [0, y] = [x, x + y]$ where x would refer to a single node and $[0, y]$ would refer to a pair of nodes).

2. Once the new expressions e_1 and e_2 are obtained through substitution, the second step can be performed. It involves simplifying each expression and checking whether a new norm has to be added to N or if an existing norm can be used when deriving a difference constraint. Similarly to the changes done to the definition of assignments as discussed in Section 4.1.2, the definition of *difference constraints* had to be changed too. The original Looper defined difference constraints simply as follows:

$$\mathcal{DC} ::= Exp \times Exp \times \mathbb{Z},$$

corresponding to the $x \leq y + \mathtt{c}$ form where the operators $\leq$ and $+$ were implicit. This representation had certain limitations (see Section 3.2) even when *interprocedural* analysis was not considered. The new version redefines the difference constraints as follows:

$$\mathcal{RHS} ::= EdgeExp \times \mathcal{O} \times \mathbb{Z},$$
$$\mathcal{DC} ::= EdgeExp \times \mathcal{RHS} \times \mathcal{RHS}$$

where $\mathcal{O} = \{+, *, \div, \ll, \gg\}$ is the set of possible operators. These operators were the most commonly used in our experiments and the added support allowed Looper to consider expressions such as $x' \leq x \gg 2$ as decrements in certain cases. To add more, one would need to subsequently generalize the transition and variable bound algorithms accordingly which might be non-trivial. Note that other operators are still allowed to be used inside norm expressions but Looper will not consider them as possible increments or decrements of a loop control variable. Moreover, the "$-$" operator is not included in the set $\mathcal{O}$ because it can be trivially converted to $+(-1)$.

This new definition allows Looper to represent constraints such as $x \leq [e_1 + 1, e_2 * 5]$, i.e., constraints now also operate over intervals instead of simple values. Moreover, the $+$ operator is no longer implicit and instead any operator $\phi \in \mathcal{O}$ can be used explicitly to construct a valid *difference constraint*.

Finally, both expressions e_1 and e_2 are simplified to obtain a canonical form which makes it easier to determine whether a new norm has to be generated or not. For the

```
while (x > 0) {
    x = x >> 1;
}
```

Listing 4.2: An example of a loop pattern encountered during experimental testing. It uses the right bit-shift operator $\gg$ to decrement the value of control variable x. The new version of Looper is able to analyze such loops due to the extended definition of difference constraints.

simplification, the new version of Looper implements an algorithm which leverages the distributive, commutative and associative laws of algebra in order to simplify the input expression. Let $\mathcal{C}_\varepsilon ::= (\mathcal{O} \times \mathbb{Z}) \cup \{\varepsilon\}$ be the optional constant part of a difference constraint where ε is the empty value. Then,

$$\text{split}(x)\colon EdgeExp \to (EdgeExp \times \mathbb{N} \times \mathbb{N})^* \times \mathcal{C}_\varepsilon$$

is a function which applies various algebraic rules to simplify the expression x and splits it into an additive list of terms of the form $x_i \cdot \frac{a}{b}$ in the process. Moreover, it attempts to separate out the optional constant $\mathcal{C}_\varepsilon$. The algorithm is deterministic and thus produces the same result for any two expressions e_1 and e_2 that are possibly syntactically different but equivalent and can be transformed to the same expression through application of algebraic rules. To demonstrate how this property is leveraged, consider the norm $e_1 = (x * 5) - y$ and the edge assignment $x = x - 1$, which leads to a new norm $e_2 = ((x - 1) * 5) - y$. Unfortunately, this form makes it impossible to reuse the previously existing norm e_1 and avoid creating a new norm e_2. However, $\text{split}(e_1) = ([x \cdot \frac{5}{1}, y \cdot (-\frac{1}{1})], \varepsilon)$ and $\text{split}(e_2) = ([x \cdot \frac{5}{1}, y \cdot \frac{1}{1}], +, (-5))$ transform the expressions into a different form by applying the following algebraic transformations: $(x * 5) - y = 5x - y$ and $((x - 1) * 5) - y = (5x - 5) - y = (5x - y) - 5$, where -5 is separated as the optional constant $(+, (-5))$ and the expressions are split into addition terms. This form makes it trivial to detect if a norm can be reused. In particular, equal term lists imply that the norm e_1 can be reused. Moreover, the extended definition of difference constraints enables analysis of different loop patterns such as those in Listing 4.2 where *bit-shift* operators are used to decrement the value of the control variable.

Consequently, the definitions for the *increments*, *decrements* and *resets* from [32] had to be updated as shown below in Definition 4.1.1.

Definition 4.1.1 (Increments, Decrements and Resets). Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP over* $\mathcal{A}$. Let $l_1 \xrightarrow{u} l_2 \in E$, $\mathtt{v} \in \mathcal{V}$, $\mathtt{a} \in \mathcal{A}$, $\mathtt{c} \in \mathbb{Z}$, and $\phi \in \mathcal{O}$. We redefine the resets

$\mathcal{R}(\mathtt{v})$, increments $\mathcal{I}(\mathtt{v})$, and decrements $\mathcal{D}(\mathtt{v})$ of $\mathtt{v}$ as follows:

$$\begin{aligned}\mathcal{R}(\mathtt{v}) = &\{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \mid \mathtt{v}' \leq [\mathtt{a} \ \phi \ \mathtt{c},\ \_] \in u,\ \mathtt{a} \neq \mathtt{v}\} \cup \\ &\{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \mid \mathtt{v}' \leq [\_,\ \mathtt{a} \ \phi \ \mathtt{c}] \in u,\ \mathtt{a} \neq \mathtt{v}\}\end{aligned}$$

$$\begin{aligned}\mathcal{I}(\mathtt{v}) = &\{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \mid \mathtt{v}' \leq [\mathtt{v} + \mathtt{c},\ \_] \in u,\ \mathtt{c} > 0\} \cup \\ &\{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \mid \mathtt{v}' \leq [\_,\ \mathtt{v} + \mathtt{c}] \in u,\ \mathtt{c} > 0\}\end{aligned}$$

$$\begin{aligned}\mathcal{D}(\mathtt{v}) = &\{(l_1 \xrightarrow{u} l_2, +, \mathtt{c}) \mid (\mathtt{v}' \leq [\mathtt{v} + \mathtt{c},\ \_] \in u,\ \mathtt{c} < 0) \ \vee \ (\mathtt{v}' \leq [\_,\ \mathtt{v} + \mathtt{c}] \in u,\ \mathtt{c} < 0)\} \cup \\ &\{(l_1 \xrightarrow{u} l_2, \gg, \mathtt{c}) \mid (\mathtt{v}' \leq [\mathtt{v} \gg \mathtt{c},\ \_] \in u,\ \mathtt{c} > 0) \ \vee \ (\mathtt{v}' \leq [\_,\ \mathtt{v} \gg \mathtt{c}] \in u,\ \mathtt{c} > 0)\}\end{aligned}$$

Note that this definition follows the original notation from [32] where $\mathcal{A} = \mathit{EdgeExp}$ and $\mathcal{V} \subset \mathit{EdgeExp}$. Furthermore, the $\mathcal{I}(\mathtt{v})$ set considers the plus operator $+$ only even though constraints such as $x \leq x * 2$ could also be considered as increments with the new definition. However, considering these types of increments would require additional extensive changes to the core bound derivation algorithm which is out of the scope of this work. On the contrary, the $\mathcal{D}(\mathtt{v})$ set includes the *bit-shift* decrements precisely because it did not mandate extensive changes. Intuitively, any constraint such as $x \leq x \gg 1$ can be soundly over-approximated by $x \leq x - 1$ for the purposes of the bound algorithm, i.e., the algorithm can treat both of these decrements in the same way. Also note that the sets $\mathcal{I}(\mathtt{v})$ and $\mathcal{D}(\mathtt{v})$ simply track which transitions can increment or decrement a norm $\mathtt{v}$ and it is assumed that they are disjoint. The case when the sets are not disjoint is not supported in any way yet.

4.2 Intraprocedural Analysis Extensions

The original Loopus tool [32] proposed several extensions which aimed at improving the overall precision of the bound algorithm. However, as Section 3.2 hinted, the logic behind some of these extensions proved to be flawed when an implementation attempt was made. Moreover, when considering analysis of *real-world* code, certain major deficiencies were overlooked by all of the proposed extensions. This section presents two new extensions which aim at improving the analysis for practical use-cases.

4.2.1 Lower Variable Bound Algorithm

Even though this first major extension does not directly improve the precision of *intraprocedural* analysis in an obvious way, it serves as a building block for the *interprocedural* analysis extension subsequently covered in Section 4.3. Finally, adding a support for interprocedural analysis can, in turn, improve the overall intraprocedural precision through consideration of return values and *side-effects*.

The building block at question is the *lower variable bound analysis*. Originally, there was no need for lower bound analysis in the bound algorithm as described in Section 3.1. However, the requirement for such analysis naturally occurred in the process of designing the interprocedural analysis and more specifically when the issue of *sound summary instantiation* was considered. Note that this work does not present any formal proofs of *soundness* for any of the introduced extensions, but the author will, based on intuition, attempt to lay out the case why each extension *should be sound* (at least in some described cases).

The main idea was to modify the already existing *variable bound* procedure from Definition 3.1.13 to obtain a new procedure $LV\mathcal{B}$ which works conversely:

$$LV\mathcal{B}(\mathtt{a}) = \mathtt{a} \text{ if } \mathtt{a} \in \mathcal{C}, \text{ else}$$
$$LV\mathcal{B}(\mathtt{v}) = \mathtt{Decr}(\mathtt{v}) + \min_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(LV\mathcal{B}(\mathtt{a}) + \mathtt{c})$$

where

$$\mathtt{Decr}(\mathtt{v}) = \sum_{(\tau,+,\mathtt{c})\in\mathcal{D}(\mathtt{v})} T\mathcal{B}(\tau) \times \mathtt{c} \quad (\text{we set } \mathtt{Decr}(\mathtt{v}) = 0 \text{ for } \mathcal{D}(\mathtt{v}) = \emptyset)$$

Intuitively, the concept is the same for both $V\mathcal{B}$ and $LV\mathcal{B}$ procedures, but instead of choosing the reset with the highest possible value and calculating by how much it can increase the value, the $LV\mathcal{B}$ procedure chooses the reset with the lowest possible value and calculates by how much it can decrease the value. The intuition behind why we believe that $LV\mathcal{B}$ should be sound starts from that it leverages the already *sound* $T\mathcal{B}$ procedure for the most part. Indeed, the `Incr` and `Decr` procedures work exactly the same with the only difference being that the `Decr` procedure is using decrements $(\tau, +, \mathtt{c}) \in \mathcal{D}(\mathtt{v})$ where $\mathtt{c} < 0$. As such, multiplying the amount of times τ can be executed with the amount $\mathtt{c}$ by which $\mathtt{v}$ decreases in each execution should indeed yield the total amount by which $\mathtt{v}$ can decrease. Furthermore, the soundness of replacing the $\max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(V\mathcal{B}(\mathtt{a})+\mathtt{c})$ term with the inverse term $\min_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(LV\mathcal{B}(\mathtt{a}) + \mathtt{c})$ seems fairly obvious.

There are, however, two notable caveats. First, recall Section 3.1.3 which covered the abstraction algorithm for transformation of *guarded DCPs* into regular *DCPs*. More specifically, recall that any constant $\mathtt{c} < 0$ is transformed either to -1 or 0 depending on whether the associated norm is guarded on the given transition or not. Consequently, the proposed $LV\mathcal{B}$ procedure under-approximates the real lower bounds when evaluated over regular *DCPs* with transformed constants. To solve this, Looper also keeps the original constant for each difference constraint and uses the original values during construction of the $\mathcal{D}(\mathtt{v})$ set which is then leveraged by the $LV\mathcal{B}$ procedure.

The second issue is that only decrements $(\tau, +, \mathtt{c}) \in \mathcal{D}(\mathtt{v})$ involving the plus operator are compatible with the original idea adopted from the $V\mathcal{B}$ procedure. Contrary to the $T\mathcal{B}$ procedure, where even decrements with the right *bit-shift* operator could have been implicitly considered to obtain over-approximated transition bounds (see Section 4.1.3), the entire $LV\mathcal{B}$ procedure would have to be substantially modified in order to support the right *bit-shift* and possibly division operator. Such changes were out of the scope of this work and this issue is therefore currently one of the possible sources of *unsoundness* anytime the $LV\mathcal{B}$ procedure is leveraged.

4.2.2 Compound Local Bounds

One major overlooked issue with the original concept of *Local Bound Mapping* (see Definition 3.1.10) is that it does not take *compound loop conditions* such as $x > 0 \wedge y > 0$ into account. An extension for generalizing *local bounds* to *sets of local bounds* was already proposed in [32]. However, this extension was primarily intended for handling of *non-linear control flow* involving break statements and alike. Formally, the extension redefined the local bound mapping function from $\zeta\colon E \to Expr(\mathcal{A})$ to $\zeta\colon E \to 2^{Expr(\mathcal{A})}$.

The main motivation for our extension of the local bounds is to significantly increase the precision by considering the complete Boolean expressions in loop conditions. Originally,

any loop condition such as `x > 0 && y > 0` would generate two norms `x` and `y` but the subsequent choice of a local bound norm was *non-deterministic*. I.e., either `x` or `y` would be chosen and used to calculate the transition bound for the loop with this condition. This is clearly not precise in the case of the $\wedge$ operator and outright *unsound* when the $\vee$ operator is considered. However, the original Loopus assumed that each loop condition can contain only a single atomic condition and possibly some non-deterministic element, e.g., `x - y > 0 && *` where `*` denotes a non-deterministic condition.

This section proposes a solution to handling the compound conditions in more precise and also sound way for the $\wedge$ and $\vee$ operators, respectively. The core idea is to take the logical operators in loop conditions into account and transform them into min and max operators in the final transition bound accordingly. Namely, the loop condition $x > 0 \wedge y > 0$ should yield the transition bound $\min(T\mathcal{B}(x), T\mathcal{B}(y))$, and, conversely, the condition $x > 0 \vee y > 0$ should yield $\max(T\mathcal{B}(x), T\mathcal{B}(y))$. The reasoning is simple: a loop with the condition $e_1 > 0 \wedge e_2 > 0$ executes as long as both conditions $e_1 > 0$ and $e_2 > 0$ are true. If either one of those conditions becomes false, the loop is terminated. Therefore, the number of iterations is limited by the condition which becomes false first, hence the min operator. The intuition behind the $\vee$ operator is analogical.

To implement this reasoning, changes at several steps had to be made. First, the definition of the *local bound mapping* was changed from $\zeta\colon E \to Expr(\mathcal{A})$ to

$$\zeta\colon E \to 2^{2^{Expr(\mathcal{A})}}.$$

Each edge is now mapped to a set of sets of norms instead of a single norm which makes it possible to represent complex conditions that are in the *Disjunctive Normal Form* (DNF). E.g., the condition $(x > 0 \wedge y > 0) \vee z > 0$ would be represented as $\{\{x > 0, y > 0\}, \{z > 0\}\}$. Next, the Infer CFG to LTS abstraction process originally did not support compound conditions. The solution to this problem was proposed in Section 4.1.1 which presented a new abstraction algorithm that merges multiple branching nodes of one loop head into a single *prune* node in the LTS (see Figure 4.2). This was also leveraged to compose the individual atomic conditions of each Infer CFG path into the original compound condition present in the source code of the analyzed program. These compound conditions are now also stored as sets of sets of expressions, i.e., elements of the set $2^{2^{EdgeExp}}$. The LTS to *DCP* abstraction process then transforms these conditions into norms which can be finally used to construct the local bound mapping.

Determining Compound Local Bounds

The new algorithm for finding *compound local bounds* simply extends the original algorithm described in Section 3.1.5 by adding two new steps between the first and second step of the original algorithm. Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP*. Let S be a set of all SCCs of P and suppose that $\mathcal{V} \subset EdgeExp$. The two new steps are the following:

1. Let $\mathtt{cv} \in 2^{2^{\mathcal{V}}}$. We define $\xi(\mathtt{cv}) \subseteq E$ to be the set of all transitions $\tau = l_1 \xrightarrow{u} l_2 \in E$ such that

   $$\exists s \in \mathtt{cv}\colon \exists \mathtt{v} \in s\colon (\mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u, \mathtt{c} < 0) \ \vee \ (\mathtt{v}' \leq \mathtt{v} \gg \mathtt{c} \in u, \mathtt{c} > 0).$$

   I.e., it is sufficient if just one norm `v` from `cv` is decreased on τ to be included in the $\xi(\mathtt{cv})$ set. For all $\tau \in \xi(\mathtt{cv})$, we set $\zeta(\tau) = \mathtt{cv}$.

2. Let $\mathtt{cv} \in 2^{2^{\mathcal{V}}}$ and $\tau \in E$. Assume τ was not yet assigned a local bound. We set $\zeta(\tau) = \mathtt{cv}$ if τ does not belong to any SCC of the directed graph (L, E') where $E' = E \setminus \xi(\mathtt{cv})$ which is the CFG of $\Delta\mathcal{P}$ where the transitions $\xi(\mathtt{cv})$ were removed.

Clearly, these two new steps are simple generalizations of the steps 2 and 3 from the original algorithm with the only difference being that instead of the norm $\mathtt{v}$, the set of sets of norms $\mathtt{cv}$ is used. Therefore, the intuition behind these two steps as discussed in Section 3.1.5 also applies here.

Changes to the $T\mathcal{B}$ procedure

Finally, the $T\mathcal{B}$ procedure also had to be generalized to reflect the changes to the *local bound mapping* definition. Assuming the $T\mathcal{B}_{old}(\tau)$ procedure for norms as defined in Definition 3.1.13, the new generalized procedure is defined as follows:

$$T\mathcal{B}_{old}(\mathtt{a}) = \mathtt{a}, \text{ if } \mathtt{a} \notin \mathcal{V}, \text{ else}$$

$$T\mathcal{B}_{old}(\mathtt{v}) = \mathtt{Incr}\left(\bigcup_{\kappa \in \mathfrak{K}(\mathtt{v})} atm_1(\kappa)\right) + \sum_{\kappa \in \mathfrak{K}(\mathtt{v})} T\mathcal{B}\Big(trn(\kappa)\Big) \times \max\Big(V\mathcal{B}(in(\kappa)) + c(\kappa),\ 0\Big) + \mathtt{Incr}\Big(atm_2(\kappa)\Big)$$

$$T\mathcal{B}_{new}(\tau) = \max_{S \in \zeta(\tau)} \Big(\min_{\mathtt{v} \in S} (T\mathcal{B}_{old}(\mathtt{v})) \Big)$$

The new definition simply used the previous definition of $T\mathcal{B}_{old}(\tau)$ from Definition 3.1.13 and changed the parameter from transition τ to a norm $\mathtt{v}$ which required only minor changes as the original $T\mathcal{B}_{old}(\tau)$ procedure was already using the τ parameter only to retrieve the norm $\mathtt{v}$ via the $\zeta(\tau)$ mapping. The main contribution is in the new $T\mathcal{B}_{new}(\tau)$ term which wraps the old procedure and implements the reasoning discussed at the beginning of this section.

4.3 Interprocedural Analysis

The main contribution of this work is the *interprocedural* analysis extension of the Loopus tool which was originally only *intraprocedural*. The issue of adding support for interprocedural analysis had two main parts: construction and instantiation of summaries. The first step was to design an appropriate summary data structure which would hold enough information to make the subsequent instantiation sufficiently precise. The original simple but naive idea was to summarize a function with a single bound expression:

$$e = \sum_{\tau \in E_B} T\mathcal{B}(\tau),$$

where $E_B \subset E$ is the set of all back-edges of a *DCP* program $\Delta\mathcal{P} = (L, E, l_b, l_e)$. Note that the program $\Delta\mathcal{P}$ represents a single analyzed function. However, this design led to an overly coarse instantiation process with a significant loss of precision in many cases. A more granular design was needed, not only to increase the precision but also to isolate the points of failure to make the analysis more robust overall.

4.3.1 Function Summaries and Summary Trees

Since the original idea was insufficient for practical use, a new hierarchical and recursive summary design was adopted. The idea was to represent each analyzed function as a set of transitions, each with their associated transition bounds. With this more granular approach, a function summary is instantiated at the level of individual transitions which is not only generally more *sound* as discussed in Section 4.3.4 but can also be leveraged to more precisely isolate the parts of a program for which the analysis fails. Moreover, this approach also enables a more detailed transition-level issue reporting for the end user. Figure 4.3 demonstrates the hierarchical and recursive nature of this design.

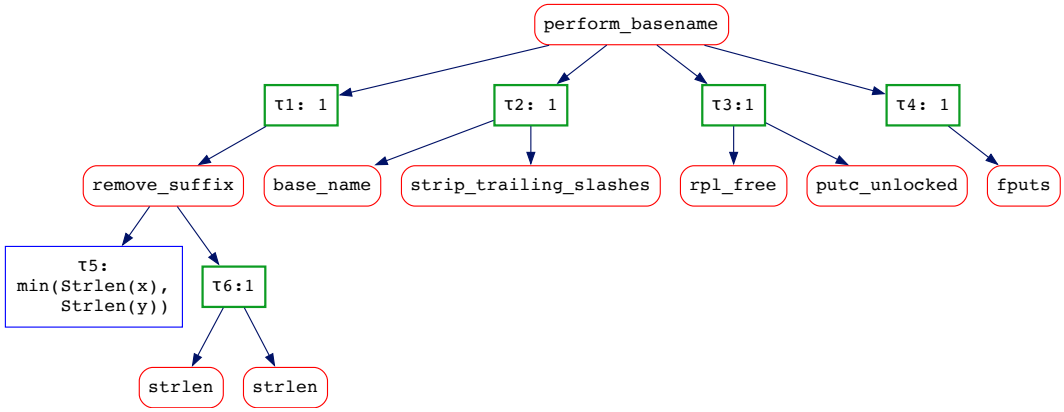


Figure 4.3: An example of a *summary tree* which visualizes the hierarchical structure of the *function summary* constructed during interprocedural analysis of the `perform_basename` function. Each function summary consists of *transition summaries* (green and blue nodes) and *call summaries* (red nodes) which are mutually recursive. Note that transition identifiers and associated source code locations were omitted from the figure for the sake of clarity.

The figure presents a so-called *summary tree* of a *function summary* obtained during the analysis of the `perform_basename` function from the GNU *coreutils* repository. The summary consists of individual *transition summaries* (green and blue nodes in the summary tree) for each *back-edge* or transition containing at least one function call. Specifically, in this case, only the transition τ_5 is a *back-edge* and the remaining transitions each contain at least one function call. Each *transition summary* of $\tau \in E$ stores the calculated transition bound $T\mathcal{B}(\tau)$, which is shown behind the semicolon ":", and a set of so-called *call summaries* (red nodes) which in turn contain a set of *transition summaries*. A leaf node of the summary tree is either a transition summary with no call summaries or conversely a call summary with no transition summaries.

Summary Formalization

Before presenting the formal definitions of *call*, *transition*, and *function* summaries, a few essential domains will be defined first. Let $\Omega = \{\leq, \geq, \sim\}$ be a set of possible function monotonicities, with $\leq$ denoting a *non-decreasing* function, $\geq$ denoting a *non-increasing*

function, and lastly $\sim$ denoting a *non-monotonic* function. Then,

$$\begin{aligned}\mathcal{M} &::= \Pi \times \Omega,\\ \mathcal{F} &::= Var \times \mathbb{N},\\ \mathcal{P}_\varepsilon &::= (EdgeExp \times EdgeExp) \cup \{\varepsilon\}, \text{ and}\\ \mathcal{B} &::= EdgeExp \times (EdgeExp \times EdgeExp)\end{aligned}$$

where $\mathcal{M}$ denotes the domain of *monotonicity maps* that associate *access paths* with function monotonicities. $\mathcal{F}$ denotes the domain of so-called *formal maps* that associate formal parameters (represented by Var variables) with their *positional indices*. $\mathcal{P}_\varepsilon$ denotes the domain of optional $EdgeExp$ pairs where ε represents the neutral value. Finally, $\mathcal{B}$ denotes the domain of *formal bound maps* which associate formal parameters with their lower and upper bounds represented by a pair of $EdgeExp$ values. Assuming these domains, the definitions of individual summary types can be presented.

Definition 4.3.1 (**Call Summary**)**.** Let $Procname$ be the domain of valid function names and let Loc be the domain of source code locations. A *call summary* is a tuple

$$(n, l, S_T) \in \mathbb{S}_C ::= Procname \times Loc \times 2^{\mathbb{S}_T},$$

where n is the name of the called function, l is the source code location of the function call, and $S_T \subset \mathbb{S}_T$ is a set of transition summaries associated with this call summary.

Definition 4.3.2 (**Transition Summary**)**.** A transition summary of a *DCP* program $\Delta\mathcal{P}$ is a tuple

$$(\tau, e, M, S_C) \in \mathbb{S}_T ::= E \times EdgeExp \times 2^{\mathcal{M}} \times 2^{\mathbb{S}_C}$$

where τ is a *DCP* transition, e is the transition bound $T\mathcal{B}(\tau)$ for τ, $M \subseteq \mathcal{M}$ is a *monotonicity map* for the transition bound e, and, lastly, $S_C \subset \mathbb{S}_C$ is a set of summaries for the function calls present on τ.

Definition 4.3.3 (**Function Summary**)**.** A function summary of a *DCP* program $\Delta\mathcal{P}$ (i.e., a single function) is a tuple

$$(F, S_T, M_R, M_F, B_F, b_R) \in \mathbb{S}_{\Delta\mathcal{P}} ::= 2^{\mathcal{F}} \times 2^{\mathbb{S}_T} \times 2^{\mathcal{M}\times\mathcal{M}} \times 2^{\mathcal{M}\times\mathcal{M}} \times 2^{\mathcal{B}} \times \mathcal{P}_\varepsilon$$

where F is a *formal map* for the formal parameters of the function $\Delta\mathcal{P}$, S_T is a set of *transition summaries* for transitions of the function $\Delta\mathcal{P}$, M_R and M_F are pairs of *monotonicity maps* for lower and upper bounds of the return value and formal parameters respectively. Furthermore, B_F is a *formal bound map* for the formal parameters of the function $\Delta\mathcal{P}$, and, finally, b_R is an optional pair of lower and upper return value bounds.

The definitions of *call* and *transition* summaries are mutually recursive which forms the tree structure of a function summary. This design leverages the function analysis order adopted by Infer, i.e., *leaf functions* of the call graph which do not call any other function are analyzed first before any other function calling them can be analyzed, moving up the call graph. Note that the depth of the function summaries grows with the depth of the call graph of a program but this did not seem to be a performance problem during our experimental testing. Alternatively, it could be beneficial to introduce a way in which a user could limit the maximum depth of function summaries.

```
1 void foo(int x, int y) {
2     while ((x - y) > 0) {
3         y++;
4     }
5 }
```

Listing 4.3: A simple code snippet which motivates the need for the analysis of function monotoncities.

4.3.2 Determining Function Monotonicities

The previous section mentioned the concept of *monotonicity maps* which needs further explanation. Fundamentally, the issue of determining function monotonicities relates both to the construction and instantiation of summaries as will be shown later. However, the process is same for both cases, albeit the results are used for a different purpose in each case. To motivate the need for this concept in the first place, consider the trivial example from Listing 4.3.

The analysis of this function will yield the expression $e = (x - y)$ as the bound for the *back-edge* of the while loop. However, in the *interprocedural* settings, it is also necessary to consider how the value of e changes when the value of either x or y changes. Indeed, the expression e is an upper bound for the while loop but only in the *intraprocedural* sense which assumes that the values of all formal parameters are fixed. This is, however, no longer true for *interprocedural* analysis which instantiates the loop bound with certain function arguments at the call site, effectively changing the values of x and y. This in turn affects the value of e which represents the loop bound. The question at hand is thus: *how to obtain the upper bound of e during instantiation?* Intuitively, the value of $(x - y)$ increases as x increases and conversely decreases as y increases. Assuming $f(x, y) = x - y$, this can be written formally as

$$\begin{aligned}(\forall x_1, x_2 \in \mathbb{Z} \times \mathbb{Z} \colon x_1 \leq x_2 \implies f(x_1, y) \leq f(x_2, y)) \land \\ (\forall y_1, y_2 \in \mathbb{Z} \times \mathbb{Z} \colon y_1 \leq y_2 \implies f(x, y_1) \geq f(x, y_2)).\end{aligned}$$

Hence, to obtain the maximum of $(x - y)$, the upper bound for x and the lower bound for y should be used. This is fundamentally a problem of *function monotonicity*: is a function monotonically *non-decreasing*, *non-increasing*, or *not monotonic* at all with respect to a certain parameter?

This problem is, however, generally complex in nature, and the imperfect solution presented in the remaining part of this section reflects that. Our first attempt at solving this issue involved using the Z3 *SMT* solver (see Section 2.1.6), which was previously leveraged for the derivation of guards covered in Section 3.1.2. Suppose an n-ary function $f(a_1, \ldots, a_n) = e$ where e is a bound expression involving the variables $a_1, \ldots, a_n$. The idea was to use the previously presented implication formula with quantifiers and check the monotonicity of f for each parameter a_i in isolation by checking if either one of the following formulas is satisfiable or not

$$\exists a_i^1, a_i^2 \colon (a_i^1 \leq a_i^2 \implies f(a_1, \ldots, a_i^1, \ldots, a_n) > f(a_1, \ldots, a_i^2, \ldots, a_n)),$$

$$\exists a_i^1, a_i^2 \colon (a_i^1 \leq a_i^2 \implies f(a_1, \ldots, a_i^1, \ldots, a_n) < f(a_1, \ldots, a_i^2, \ldots, a_n)).$$

If the first or second formula was unsatisfiable, then f was assumed to be *non-decreasing* or *non-increasing* in terms of a_i, respectively. If both formulas were satisfiable, then f was assumed to be *non-monotonic* in terms of a_i. Note that Looper handles instantiation of *non-monotonic* bound expressions by using the ∞ value, which is sound but not very useful. Moreover, checking the monotonicity of f in terms of individual isolated parameters a_i is likely not enough in general. Testing all possible permutations of increasing/decreasing parameter values might be a more accurate way to determine the monotonicities. These claims are, however, purely based on author's intuition without providing any proof in this work. Nevertheless, the bound expressions still reflect loop complexities which most often take a specific form of multivariate polynomials, and the first approach might be sufficient for those.

We have, however, not explored the above possibility further because of the following issue. Namely, the issue with this approach which manifested during experiments is that *non-linear integer arithmetic*, which is often needed in real-life code, is *undecidable* in general, and Z3 would frequently fail when trying to find solution to these formulas. The issue was somewhat mitigated by trying to solve bounds for individual transitions instead of the sum of all transition bounds, but even then there were many cases where Z3 failed to find a solution. An attempt to use *real arithmetic* which is *decidable* instead of integer arithmetic was also made, but that did not yield much of an improvement either. A final idea was to try different SMT solvers or theorem provers instead of Z3. This idea led to the decision to use the Why3 platform for solvers discussed in Section 2.1.6. After implementing the support for Why3 (see Chapter 5 for specifics), a set of different provers such as Vampire, Alt-Ergo, or CVC4 were tested on the same tasks, but the results were either the same or worse when compared to Z3. These results motivated a different approach used in the new version of Looper and presented below.

Partial Derivatives

The task of proving monotonicities directly with SMT solvers turned out to be infeasible in practice which led to the idea of transforming the monotonicity problem into a problem of finding function roots. Namely, checking whether a function derivative has any roots determines whether the function is monotonic or not. If the derivative has no roots, then the precise type of monotonicity has to be further analyzed by checking the original function. Assuming the n-ary function $f(a_1, \ldots, a_n)$ as before, the task given to Z3 indirectly via Why3 reduces to

$$\exists x_i \in \mathbb{R} : \frac{\partial f}{\partial a_i}(a_1, \ldots, x_i, \ldots, a_n) = 0.$$

This formula is checked for satisfiability, and if no root exists, then a further heuristic check is performed to figure out whether the function is *non-decreasing* or *non-increasing* with respect to a_i by plugging two increasing values $x_1, x_2 : x_1 \leq x_2$ to the parameter a_i of the original function f. This approach turned out to be much more feasible, and Z3 has been able to solve these formulas for any expression tested during the experimental evaluation. Nevertheless, it shares some of the same deficiencies as the first approach, i.e., it is unclear whether it is correct for every type of expression and hence possibly leads to an *unsound* analysis. The challenge here is how to handle functions with multiple parameters that all appear together in the bounds. This seems to be a research question going far beyond the scope of this thesis. However, our experiments have shown that this is not much of an issue in practical settings.

The process of computing partial derivatives also deserves a brief discussion. We had to propose and implement our own procedure to calculate partial derivatives in Looper because the main requirement was to perform *symbolic differentiation*, and there is currently no OCaml library available for that. Moreover, Looper uses a custom *EdgeExp* expression data structure which supports various additional operators such as min or max over arbitrary number of operands. Differentiation of these terms is generally complicated, but writing a custom algorithm made it possible to apply certain heuristics based on the knowledge that the derivative will be subsequently used for analysis of monotonicity. The algorithm itself is recursive and leverages mostly the chain rule, product rule, and quotient rule in order to compute partial derivatives. A more detailed explanation will be presented in Chapter 5.

Finally, to construct the *monotonicity map* for a single multivariate expression e with variables $A = \{a_i \in \Pi \,|\, 1 \leq i \leq n\}$, suppose that $\omega(e, a)\colon \mathit{EdgeExp} \times \Pi \to \Omega$ is a function that determines the monotonicity of e with respect to the variable a via the previously described principle. Note that Looper represents variables a_i as *access paths*. The *monotonicity map* M is constructed as follows

$$M = \{(a_i, m) \in \mathcal{M} \,|\, a_i \in A,\, m = \omega(e, a_i)\} \subset \mathcal{M}$$

For convenience, let $\mathcal{M}(e)\colon \mathit{EdgeExp} \to 2^{\mathcal{M}}$ be a function that constructs the set $M \subset \mathcal{M}$ for an expression e.

4.3.3 Construction of Function Summaries

With the individual summary types as defined above, the construction of a function summary is fairly straightforward as most of the components were already existing before the introduction of *interprocedural* analysis. For the most part, the construction is merely a process of collecting data during the intraprocedural analysis and populating appropriate data structures. To formalize, let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a *DCP* program representing a function, and let $E_B \subseteq E$ be the set of all edges of $\Delta\mathcal{P}$ which are either *back-edges* or contain at least one function call of a different function. The first step is to create a set of transition summaries for each transition $\tau \in E_B$, which involves instantiation of function calls on τ, calculating the transition bound $T\mathcal{B}(\tau)$, and, finally, the construction of a *monotonicity map* for the $T\mathcal{B}(\tau)$ bound expression. Formally written:

$$S_T = \{(\tau, T\mathcal{B}(\tau), M, S_C) \,|\, \tau \in E_B,\, M = \mathcal{M}(T\mathcal{B}(\tau)),\, S_C = \mathcal{I}_C(\tau)\} \subset \mathbb{S}_T$$

where $\mathcal{I}_C(\tau)\colon E \to 2^{\mathbb{S}_C}$ is a function that maps a *DCP* edge to a set of call summaries, i.e., the instantiation function for call summaries on the edge τ. Due to the mutually recursive nature, the instantiated function calls are transformed into call summaries and used in the summary of the caller. The $\mathcal{I}_C(\tau)$ function will be discussed in Section 4.3.4 as it is part of the instantiation process.

Once the set of transition summaries is constructed, the construction of the function summary itself can proceed. Let $E_F \subset \mathit{EdgeExp}$ be the set of formal parameters of $\Delta\mathcal{P}$, and let $\mathit{return}_\varepsilon \in (\mathit{EdgeExp} \cup \{\varepsilon\})$ be the optional return variable of $\Delta\mathcal{P}$. Additionally, let $F \subset \mathcal{F}$ be the *formal map* for $\Delta\mathcal{P}$ provided by the Infer framework. Then, the function summary $S_{\Delta\mathcal{P}} \in \mathbb{S}_{\Delta\mathcal{P}}$ is constructed as follows

$$S_{\Delta\mathcal{P}} = (F, S_T, M_R, M_F, B_F, b_R)$$

where the set of transition summaries S_T was constructed before,

$$b_R = \begin{cases} \varepsilon & \text{if } return_\varepsilon = \varepsilon, \\ (LV\mathcal{B}(return_\varepsilon),\, V\mathcal{B}(return_\varepsilon)) & \text{otherwise,} \end{cases}$$

and the *formal bound map* B_F is constructed as

$$B_F = \{(f, (LV\mathcal{B}(f), V\mathcal{B}(f)) \mid f \in E_F\}.$$

4.3.4 Instantiation of Function Summaries

The instantiation process had to be split in two parts due to execution dependency. The first phase is performed during the construction of the LTS graph as described in Section 3.1.1. More specifically, the SIL `Call` instruction retrieves the summary of a called function and partially instantiates it. A complete instantiation is not possible at that moment because lower and upper bounds of call arguments are required to properly instantiate the summary and the $LV\mathcal{B}$ and $V\mathcal{B}$ procedures cannot be called before a valid *DCP* graph of the function is constructed. However, it is still necessary to instantiate the return bound and formal parameter bounds to account for possible *side-effects*. Thus, a compromise was made, and these bounds are instantiated without computing any lower or upper bounds which is possibly *unsound*. We were unable to show the correctness of this step but, as a heuristic, it worked reasonably well in our experiments and a proper solution was out of the scope of this work.

Instantiation of Formal Parameter Bounds

The formal bounds practically represent the *side-effects* caused by updates of pointer arguments (parameters passed by reference). To apply these side-effects at the call site, Looper constructs a new edge assignment for each modified formal parameter using the formal bounds and extends the set of edge assignments. Let

$$S_f = (F, S_T, M_R, M_F, B_F, b_R) \in \mathbb{S}_{\Delta\mathcal{P}}$$

be the retrieved summary of a function f called on an LTS transition $\tau = l_1 \xrightarrow{A} l_2$, which is being constructed. Furthermore, assume that $A \subset \mathcal{A}$ is the set of assignments (see Section 4.1.2) on τ, and let $args \subset EdgeExp \times \mathbb{N}$ be the set of arguments passed to f. After processing the summary, the set A is updated as

$$A' = A \cup \{(\sigma(f, args, F),\, \sigma(lb, args, F),\, \sigma(ub, args, F)) \mid (f, (lb, ub)) \in B_F\},$$

where

$$\sigma(e, args, F)\colon EdgeExp \times 2^{EdgeExp \times \mathbb{N}} \times \mathcal{F} \to EdgeExp$$

is a function which substitutes the function arguments into the expression e based on the formal map F. Suppose that $p_1, \ldots, p_n \in Var$ are the formal parameter variables used in e. Then σ substitutes the formal parameter variable p_i with the argument a_i such that $(a_i, i) \in args$ where $(f_i, i) \in F$. Informally, when a formal parameter p_i of e is encountered, the formal map F is used to retrieve its positional index which is then in turn used to retrieve the corresponding argument expression from the set of arguments.

Instantiation of Return Bound

The return bound b_R is instantiated in a very similar fashion as the formal parameter bounds. Suppose that $M \subset Ident \times EdgeExp \times EdgeExp$ is the set of identifier associations as defined in Section 4.1.2, and let $id_R \in Ident$ be the identifier created by the `Call` instruction to store the return value. Then the set M is updated as follows:

$$M' = \begin{cases} M & \text{if } b_R = \varepsilon \\ M \cup \{(id_R, \sigma(lb, args, F), \sigma(ub, args, F))\} & \text{if } b_R = (lb, ub) \end{cases}$$

I.e., if the function f has a return value, then the formal parameters in both lower and upper bound expressions have to be substituted with the arguments from the call site. These substituted bounds are then used to create a new association with the identifier id_R and the association is added to M. The need for this intermediate identifier id_R arises because return values are not necessarily always used in practice and thus may be discarded. The identifier id_R is simply not used by any subsequent SIL instruction in such cases.

Instantiation of Transition Bounds

As mentioned above, the transition summaries cannot be instantiated before the complete *DCP* graph of the analyzed function is constructed. Looper thus stores references to call expressions for delayed instantiation. Let $Calls \subset EdgeExp$ be the set of all possible function call expressions, and let Loc be the domain of source code locations. As before, suppose that

$$S_f = (F, S_T, M_R, M_F, B_F, b_R) \in \mathbb{S}_{\Delta\mathcal{P}}$$

is the retrieved summary of a function $f \in Calls$ called on an LTS edge $\tau = l_1 \xrightarrow{A,C} l_2$[2] where $C \subset Calls$ is the set of function expressions stored on τ. Finally, let $LS \subset Loc \times \mathbb{S}_{\Delta\mathcal{P}}$ be the set of associations between source code locations and function summaries. When a SIL `Call` instruction at a location $l \in Loc$ is interpreted, the summary S_f is partially instantiated and then the sets LS and C are updated as follows:

$$C' = C \cup \{f\},$$
$$LS' = LS \cup \{(l, S_f)\}.$$

This is a new association between the call location l and the function summary S_f. Note that the location l is also part of the function expression f and will be used later to retrieve the summary S_f from LS.

The second instantiation phase happens at the end of the analysis of $\Delta\mathcal{P} = (L, E, l_b, l_e)$, after the transition bound $T\mathcal{B}(\tau)$ has been computed for all transitions $\tau \in E_B$ where $E_B \subseteq E$ is the set of edges defined in Section 4.3.3. The second phase works as follows:

1. For each $\tau = l_1 \xrightarrow{C} l_2 \in E_B$ (the set C is copied from the original LTS edge) and for each call expression $f \in C$ with an associated location $l \in Loc$, retrieve the function summary $S_f = (F, S_T, M_R, M_F, B_F, b_R) \colon (l, S_f) \in LS$. Also, as previously, let $p_1, \ldots, p_n \in Var$ be the formal parameters of f and let $args \subset EdgeExp \times \mathbb{N}$ be the set of arguments passed to f.

[2] In reality, LTS edges store sets of assignments and function calls among other data.

2. For each transition summary $s_T = (\tau, e, M, S_C) \in S_T$, instantiate the transition bound e. Similarly to the function $\sigma(e, args, F)$, suppose that

$$\varsigma(e, args, F, M)\colon EdgeExp \times 2^{EdgeExp \times \mathbb{N}} \times \mathcal{F} \times 2^{\mathcal{M}} \to EdgeExp$$

is a function which for each argument a_i of the function f substitutes either the *lower* or *upper* bound based on the computed monotonicity map M. Note that the formal map F is used to retrieve the corresponding argument a_i for the formal parameter p_i. Each parameter p_i is substituted with a value b_i as follows:

$$b_i = \begin{cases} LV\mathcal{B}(a_i) & \text{if } m_i = (\geq), \\ V\mathcal{B}(a_i), & \text{else if } m_i = (\leq), , \\ \infty & \text{otherwise} \end{cases}$$

where $(a_i, i) \in args$ for $(p_i, i) \in F$ and $(p_i, m_i) \in M$. The function ς works the same way as the previously described function σ with the only difference being that instead of a_i, either the *lower bound* $LV\mathcal{B}(a_i)$ or the *upper bound* $V\mathcal{B}(a_i)$ is used. Intuitively,

$$m_i = (\geq) \implies \big(\forall x_1, x_2 \in \mathbb{R}\colon x_1 \leq x_2 \implies e[p_i/x_1] \geq e_2[p_i/x_2]\big),$$

i.e., the value of e is *non-increasing* with respect to p_i, and thus

$$m_i = (\geq) \implies e[p_i/V\mathcal{B}(a_i)] \leq e[p_i/LV\mathcal{B}(a_i)].$$

In other words, the value of the transition bound expression e for transition τ is *non-increasing* w.r.t. the parameter p_i (the value of e decreases with the increasing value of p_i) and thus the lower variable bound $LV\mathcal{B}(a_i)$ of the argument a_i has to be substituted to *soundly* instantiate the transition bound e for the transition τ. In this case, substitution of the lower variable bound $LV\mathcal{B}(a_i)$ would lead to a lower overall value of the instantiated transition bound e' which could possibly be lower than the real bound and thus *unsound*. Note that the case of $m_i = (\leq)$ works analogically.

Finally,

$$s'_T = (\tau, e', M', \mathcal{I}_C(S_C))$$

is the instantiated transition summary where $e' = \varsigma(e, args, F, M)$ is the instantiated bound e of s_T and $M' = \mathcal{M}(e')$ is the new *monotonicity map* recomputed for e'. The instantiation of function summaries by $\mathcal{I}_C(S_C)$ is discussed in Step 3.

3. Assuming the transition summary $s_T = (\tau, e, M, S_C) \in S_T$ from the previous step, instantiate each call summary $s_C = (n, l, T) \in S_C$ where $T \subset \mathbb{S}_T$ is a set of transition summaries. To instantiate s_C, recursively repeat the second step for the set T instead of S_T. Let T' be the instantiated set of transition summaries, then $s'_C = (n, l, T')$ is the instantiated call summary. Note that the mutual recursion of Steps 2 and 3 ends when a transition with no call summaries is reached.

Intuitively, the second phase produces a set of transition summaries S_T^τ for each $\tau \in E_B$ by recursively instantiating transition summaries and call summaries. Let

$$S'_T = \bigcup_{\tau \in E_B} S_T^\tau$$

be the union of these sets. The final step is to instantiate the function summary as

$$S'_{\Delta\mathcal{P}} = (F', S'_T, M_R, M_F, B_F, b_R)$$

where F' is the *formal map* of $\Delta\mathcal{P}$ provided by Infer. The remaining components were discussed in Section 4.3.3 as the construction and instantiation of summaries overlap. Figure 4.3 from the beginning of this section shows the constructed summary $S_{\Delta\mathcal{P}}$ for the function `perform_basename` and the sub-tree of the `remove_suffix` function represents the instantiated call summary where `x` and `y` are formal parameters of `perform_basename`. The remaining functions represented by the red leaf nodes (for example `base_name` or `rpl_free`) either did not contain any loops and further function calls, or the their function summary was missing and thus could not be instantiated. A missing function summary can be caused by unavailable source code or if the analyzer previously failed to analyze the called function and did not construct any summary. For example, in this case, the function `strlen` is a library function with missing summary which can be remedied by providing a custom function *model*. In any case, the recursive instantiation process terminates when such a function is encountered.

Chapter 5

Implementation of Proposed Enhancements

This chapter presents an overview of the previously proposed enhancements of Looper from Chapter 4. It holistically discusses the implementation of the new version by covering both the implementation of the original version from [27] (see Chapter 3), as well as the new extensions proposed in the previous chapter which remain the primary focus of this chapter. The parts dedicated to the implementation of the original version are, however, mostly focused on the discussion of necessary changes that had to be made due to internal changes in the Infer framework. In fact, Infer has been under active development with daily updates for the past several years which means that the framework itself has changed considerably since the original version of Looper was published. Moreover, the added support for new features, namely the interprocedural analysis, mandated additional significant changes and heavy refactoring. Consequently, the new version bears almost no structural resemblance with the original version, albeit the implementation of the core bound algorithms based on [32] remained mostly the same.

Even though Looper is implemented as an extension of the modular Meta Infer framework introduced in Section 2.3, it does not leverage the architecture of the *abstract interpretation* framework to perform the analysis as most of the other analyzers do. The first version of Looper leveraged the framework only to construct the initial LTS graph which eventually proved to be cumbersome and the approach has been abandoned since then for a different solution discussed in Chapter 4. The current state of implementation thus relies only on the general analysis framework of Infer which provides hooks to plug-in custom analyzers into the architecture for interprocedural analysis. As with the first version, the main implementation language of Looper as well as Infer is OCaml which will be used to present simplified code snippets of important parts. Most of these code examples are written in the functional paradigm as OCaml is primarily a functional language, but it also allows developers to write code in the imperative (and even object-oriented) paradigm which is leveraged for example by the *ocamlgraph* library and few algorithms in Looper. Furthermore, similarly to [27], an intuitive pseudocode will be used to present any relevant algorithms.

The remaining part of this chapter is structured as follows. Section 5.1 describes how the code is organized into files and how each file relates to the theoretical concepts presented in Chapter 3 and in Chapter 4. It covers the basic architecture of the analyzer. The following Section 5.2 covers the entry point function of the analyzer and how it controls the whole

analysis process. Section 5.3 is dedicated to discussion of the implementation changes made in the *intraprocedural* part of Looper, more specifically the changes of the abstraction algorithm. Furthermore, Section 5.4 discusses the implementation of the extension which allows Looper to handle compound local bounds. The implementation of the newly introduced *lower variable bound* procedure is covered in Section 5.5. Lastly, Section 5.6 discusses all of the technical details related to the implementation of the *interprocedural* extension — the main contribution of this work.

As a final note, the scope of this work does not allow for an extended discussion of implementation details but the source code of both the first and the new version of Looper is publicly available at GitHub[1] so readers are encouraged to study the code itself if interested. Additionally, the Appendix B contains useful information about installation from source code and usage of Looper. Moreover, the attached memory media (see Appendix A) and the GitHub Pages[2] of the project repository also contain useful information along with some usage examples.

5.1 Code Organization and Architecture

As mentioned at the beginning of this chapter, the new version is vastly different not only in terms of the architecture but also the code itself. The code of Looper is currently divided into multitude of files in the `infer/src/looper` directory, each with its clearly defined responsibility. Moreover, the entire analyzer follows the Infer module interface which allows it to be integrated into the main analysis framework via the `infer/src/base/Checker.ml[i]` and `infer/src/backend/registerCheckers.ml` files. These files provide an easy way to register a custom checker by extending the existing variant OCaml type `Checker.t` with new analyzer entry which has to specify analysis name, command-line options, supported languages and more. Additionally, the `registerCheckers.ml` file has to be extended too, to specify the analysis callbacks (i.e., this is the hook which triggers the analysis) and also the *payload* type, which refers to the custom *summary* type. Note that Infer does not enforce any rules about the structure of a summary type which gives developers great freedom to design any type of analysis and not only *abstract interpretation* based analyses. The modular design also allows Infer users to run individual analyses independently by using specific command-line options, i.e., it is possible to run only Looper. The remaining part of this section briefly introduces each individual file and how it relates to the theoretical concepts.

- `LooperAnalysis.ml[i]`: This file contains the `analyze_procedure` entry point function of the entire analysis. It executes the high-level logic of the analysis, i.e., constructs the graphs, derives constraints, guards, initial norms and local bounds. It is also responsible for computing the transition bounds of all relevant *DCP* edges and construction of function summaries at the end of the analysis. Furthermore, it contains the implementation of `Incr`, `Decr`, $LV\mathcal{B}$, $V\mathcal{B}$ and $T\mathcal{B}$ procedures.

- `GraphConstructor.ml[i]`: The code responsible for construction of LTS graphs is present here. The manual construction of the LTS graph structure as described in

[1]The open-source repository of Looper is available at: `https://github.com/paveon/Looper`. The source code of the first as well as the new version along with an executable is available as a release at: `https://github.com/paveon/Looper/releases`.

[2]The repository GitHub Pages with additional information are available at: `https://github.com/paveon/Looper/wiki/Looper:-A-Worst-Case-Cost-Analyser`

Section 4.1.1 is performed by the `traverseCFG` function which traverses the input Infer CFG using the DFS algorithm. The interpretation of SIL instructions and construction of edge assignments as well as conditions is handled by the `exec_instr` function. This function was taken from the first version of Looper where it implemented the *abstract transformers* (see Section 2.2) in the AI framework. Since then, the AI framework is no longer used by Looper but the function was repurposed.

- `LabeledTransitionSystem.ml[i]`: These files implement the concept of LTS graphs $\mathcal{P}$ from Definition 3.1.2. The implementation file defines the `Node.t` and `EdgeData.t` types which are used by the *ocamlgraph* library to construct the LTS data type. The edge module defines several convenient functions such as for derivation of difference constraints and guards which are used during the construction of a DCP. It also defines several functions which are required in order to output an LTS graph to a file in the `DOT` format.

- `DifferenceConstraintProgram.ml[i]`: These files implement the concept of *DCP* graphs $\Delta\mathcal{P} = (L, E, l_b, l_e)$ from Definition 3.1.5. As it was possible to reuse the `Node.t` module from the LTS implementation, only the `EdgeData.t` had to be redefined and then used to construct the *DCP* graph data type. I.e., the LTS and *DCP* node data types are identical. Again, it also defines several functions which are used during the construction and output of a *DCP* graph.

- `VariableFlowGraph.ml[i]`: The concept of *variable flow graphs* with the $\mathcal{V} \times L$ node set from Definition 3.1.11 is implemented by these files. Note that $\mathcal{V} =$ `EdgeExp.t` and $L =$ `Node.t` in the actual implementation. It only defines the data types for nodes and edges with no additional functions.

- `ResetGraph.ml[i]`: The concept of *reset chain graphs* $\mathcal{G}(\mathcal{A}, \mathcal{E})$ from Definition 3.1.12 is implemented by these files. Assuming that $E =$ `Node.t` $\times$ `DCP.EdgeData.t` $\times$ `Node.t`, then $\mathcal{A} =$ `EdgeExp.t` and $\mathcal{E} = E \times$ `Binop.t` $\times$ `IntLit.t` are the actual modules used to implement the node and edge sets. The `Binop` and `IntLit` modules are provided by Infer and they are used to represent binary operators and integer literals. Moreover, these files also define the `Chain` module which implements the *reset chains* concept, likewise from Definition 3.1.12. This module is able to represent chains such as $\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \ldots \mathtt{a}_0$ and implements function `get_reset_chains` which constructs all optimal reset chains (see Section 3.1.7) of $\mathcal{G}$. Furthermore, it implements the helper functions $in(\kappa)$, $c(\kappa)$, $trn(\kappa)$ and $atm(\kappa)$.

- `LooperSummary.ml[i]`: The custom *summary* data structure which enables interprocedural analysis is defined in these files. Based on Section 4.3.1, the `t`, `transition` and `call` data types implement the function $\mathbb{S}_{\Delta\mathcal{P}}$, transition $\mathbb{S}_T$ and call $\mathbb{S}_C$ summaries respectively. It also defines a `cache` data type which is used throughout the entire analysis to cache various data to leverage the technique of dynamic programming. Furthermore, the process of *summary instantiation* (see Section 4.3.4 is implemented by this module via a `instantiate` function and additional `total_bound` function is used to compute the overall bound of a specific summary in the form of single `EdgeExp.t` expression. Lastly, a definition of a `TreeGraph` module can be found here. This module is responsible for construction of *summary trees* out of summaries and for output to files. Figure 4.3 presents an example.

- `LooperCostModels.ml`: An experimental module inspired by the existing *Cost* analyzer. It is intended for representation of function *models* for functions without available source code such as built-in or library functions, e.g., `malloc`,`std::find` or container methods from C++ or Java. These functions cannot be analyzed and thus do not have any *real* summaries for instantiation. As such, it is instead possible to create artificial summaries for these functions, i.e., *models*. The current state of implementation of this module and of function models in general is purely experimental and in active development.

- `EdgeExp.ml[i]`: The `EdgeExp` module implemented by these files is the most essential building block of the entire analyzer. It implements the theoretical domain of *EdgeExp* expression used throughout Chapter 4. This module is used to represent all expressions used by Looper in the various graph data structures and during the analysis itself. It implements a wide variety of functions ranging from convenient utility functions and operators to complex procedures for partial differentiation, monotonicity checking or expression simplification. Moreover, it provides a function for construction of `EdgeExp` expressions out of Infer `Exp` module expressions. It is also possible to convert the `EdgeExp` expressions into Why3 expressions used by the Why3 prover framework. Lastly, it defines `ValuePair` and `CallPair` wrapper modules which implement a type that can either hold a single value or a pair of values for expressions or specifically only call expressions. The need for these types arose with the introduction of interprocedural analysis and the need for both the lower and the upper bounds of values.

- `DifferenceConstraint.ml[i]`: The concept of *difference constraints* originally defined in Definition 3.1.3 and later redefined in Section 4.1.3 is implemented here. The previously introduced mathematical domain is implemented by actual OCaml modules as follows:

$$\mathcal{RHS} ::= \texttt{EdgeExp.t} \times \texttt{Binop.t} \times \texttt{IntLit.t}$$
$$\mathcal{DC} ::= \texttt{EdgeExp.t} \times \mathcal{RHS} \times \mathcal{RHS},$$

  The module also implements few convenient and simple functions, for example to check whether a difference constraint is increasing, decreasing or a reset.

- `LooperUtils.ml[i]`: A collection of several useful utility types used by other files.

- `Provers.ml`: A file that defines several data types related to the use of Why3 framework. It also specifies a list of provers supported both by Why3 and Looper. However, the main purpose of this module is to load the necessary theories and initialize the external provers for Why3 using custom *driver files* [6]. The Why3 framework is then used to derive *DCP* guards and importantly to determine monotonicity of expressions as discussed in Section 4.3.2.

5.2 Analysis Entry Point Function

As mentioned before, the `analyze_procedure` function registered in the `registerCheckers` module serves as an analyzer entry point that gets invoked once by the Infer backend for each procedure in a program. Infer does not pose any restrictions upon the function

```
let analyze_procedure analysis_data =
    (* ... Setup logging, Why3 and create output directory structure... *)
    ...
    (* Construct the initial LTS graph and set of norms *)
    let graph_data = GraphConstructor.construct analysis_data in
    ...
    (* Derive constraints and compute the final set of norms in the process *)
    let unprocessed_norms = graph_data.norms
    let final_norms = compute_norm_set unprocessed_norms EdgeExp.Set.empty in

    (* Derive guards, propagate them and construct guarded DCP  *)
    List.iter edge_pairs ~f:(fun (lts_edge, dcp_edge) ->
        let guards = LTS.EdgeData.derive_guards lts_edge final_norms in
        DCP.EdgeData.add_guards dcp_edge guards) ;
    propagate_guards guarded_nodes ;
    to_natural_numbers dcp ; (* Transform guarded DCP into regular DCP *)
    ...
    (* Create VFG graph, VFG mapping, apply mapping and construct Reset Graph *)
    determine_local_bounds dcp ;
    ...
    let edge_set, cache = DCP.fold_edges_e
        (fun ((_, edge_data, _) as edge) (edge_set, cache) ->
          if DCP.EdgeData.is_backedge edge_data then
            let _, cache = transition_bound edge cache in
            (DCP.EdgeSet.add edge edge_set, cache)
          else if not (EdgeExp.CallPair.Set.is_empty edge_data.calls) then
            (DCP.EdgeSet.add edge edge_set, cache)
          else (edge_set, cache) )
        dcp (DCP.EdgeSet.empty, LooperSummary.empty_cache)
    in
    (* Execution cost must be computed after transitions bounds
     * to avoid computation cycles *)
    let bounds, cache = DCP.EdgeSet.fold
        (fun ((src_node, edge_data, dst_node) as edge) (bounds, cache) ->
          let instantiated_calls, cache = instantiate_function_calls edge_data cache in
          let bound, cache = transition_bound edge cache in
          let monotony_map =
            if EdgeExp.is_const bound then AccessExpressionMap.empty
            else EdgeExp.determine_monotonicity bound tenv active_prover
          in
          let transition_summary =
            {src_node; dst_node; bound; monotony_map; calls= instantiated_calls}
          in
          (transition :: bounds, cache) )
        edge_set ([], cache)
    in
    ...
    (* Calculate LVB and UVB for formal parameters and return value *)
    ...
    let summary : LooperSummary.t = {
        formal_map= FormalMap.make (Procdesc.get_attributes proc_desc)
      ; bounds ; return_bound ; formal_bounds }
    in
    Some summary
```

Listing 5.1: The entry point of Looper which is invoked by Infer for every analyzed procedure. It implements the high-level logic of the analysis and is responsible for the computation of transition bounds and construction of summaries at the end of analysis.

apart from the fact that it has to optionally return a summary of the analyzed procedure. A greatly simplified implementation is presented in Listing 5.1. The presented code snippet is primarily intended to show how the analysis is split into several steps without focus on the details. As such, many steps were either left out and replaced by simple comments or greatly simplified. For example, the implementation of bound algorithms is left out but the functions are still used in the snippet. At the beginning, before the LTS graph is constructed at line 5, it is necessary to initialize logging and create a log file. Moreover, the Why3 library has to be initialized through the `Provers` module as it is used later during the analysis. However, contrary to the logging setup, the Why3 initialization happens only once when a first function is being analyzed. To avoid repeating the costly initialization, imperatively accessed cache is being used. The imperative access paradigm is facilitated by the built-in `ref` pointer type in OCaml.

The `construct` function from the `GraphConstructor` module implements the algorithm described in Section 4.1.1 and Section 4.1.2. It constructs the LTS graph and the initial set of norms, both contained in the `graph_data` data structure. It leverages the previously mentioned `traverseCFG` and `exec_instr` functions which build the LTS graph structure and edge labels respectively. Note that the `traverseCFG` function uses the Infer CFG as is, i.e., it does not filter the graph nodes or edges. Other *abstract interpretation* based analyzers typically use one of the CFG types provide by Infer, such as `ProcCfg.Normal` or `ProcCfg.Exceptional`. However, because Looper currently does not support other languages than C, there was no need to worry about specific CFG types as there is no exception handling in C. The implementation of the `traverseCFG` and `exec_instr` functions will be briefly covered in Section 5.3.

The `compute_norm_set` function on line 9 simultaneously derives difference constraints, extracts the initial set of norms and constructs a guarded *DCP* without any guards. Guards are then derived a propagated on lines 12–15 and finally the line 16 is responsible for the transformation to a regular *DCP*. The implementation of these steps was already summarized in Section 3.1.2 and Section 3.1.3 or in more detail in [27], as it is mostly identical with the first version of Looper.

Then the *flow-sensitivity* transformation (see Section 3.1.6) is performed and the *reset graph* (see Section 3.1.7) is constructed as hinted by the comment on the line 18. The last step before the bound computation is the `determine_local_bounds` function which implements the enhanced version of the algorithm as proposed in Section 4.2.2. The implementation of the enhanced version will be described in Section 5.4.

The lines 21–46 are responsible for the main part of the analysis, i.e., the computation of transition bounds $T\mathcal{B}(\tau)$ for all relevant edges $\tau \in E_B$, where E_B is the previously defined edge set. The second half instantiates the call summaries $\mathbb{S}_C$ (line 35) and uses them to construct the transition summaries $\mathbb{S}_T$ (lines 41–43). Additionally, the *monotony map* $\mathcal{M}$ is computed for each $T\mathcal{B}(\tau)$, where the `determine_monotonicity` function corresponds with the $\mathcal{M}(e)$ function defined in Section 4.3.2. The implementation of the `determine_monotonicity` function will be covered in more detail, including the algorithm for partial differentiation, in Section 5.6.1. Furthermore, the line 48 corresponds with a code section that computes the lower $LV\mathcal{B}(\mathtt{v})$ and upper $V\mathcal{B}(\mathtt{v})$ variable bounds for formal parameters $f_1, \ldots, f_n$ as well as for the `return` shadow variable of the analyzed function. Note that only parameters f_i of *pointer to integer* types are considered. The implementation of the new $LV\mathcal{B}$ procedure introduced in Section 4.2.1 will be covered in Section 5.5.

Finally, the remaining lines 50–54 construct the function summary $\mathbb{S}_{\Delta\mathcal{P}}$ of the currently analyzed function and return it to the Infer backend which stores it for subsequent instan-

```
1  let analyze_procedure analysis_data =
2      (* ... Setup logging, Why3 and create output directory structure... *)
3      try
4          let graph_data = GraphConstructor.construct analysis_data in
5          ...
6          Some summary
7      with error ->
8          let stack = Printexc.get_backtrace () in
9          report_issue IssueType.looper_infinite_cost
10             "cannot be computed due to thrown exception" ;
11         Utils.close_outf log_file ;
12         debug_fmt := List.tl_exn !debug_fmt ;
13         None
```

Listing 5.2: Exception handling of the entry point function from Listing 5.1.

tiation. The `formal_map` value (line 51) which corresponds with the *formal map* $\mathcal{F}$ domain from Section 4.3.1 is provided by Infer infrastructure.

On a final note, the `analyze_procedure` function in reality also contains exception handling for everything except the initialization of Why3 and logging at the beginning. Listing 5.2 shows how the code from Listing 5.1 is wrapped in *try-with* blocks. Due to the complexity of Looper as well as the complexity of *real-world* code that is being analyzed, there are always unforeseen issues and code patterns that might cause the perpetually incomplete analyzer to crash. This coarse exception handling is intended to prevent these errors from crashing the entire analysis loop of Infer and instead gracefully handle the issue so that the analysis can continue, albeit with missing summary.

5.3 Construction of LTS

As mentioned before, the `traverseCFG` function is responsible for construction of LTS graphs. Listing 5.3 shows the implementation of OCaml modules used to represent the LTS nodes and edges. The `Node` module only holds the information about node type and the `EdgeData` module primarily contains edge assignments, conditions and calls among other useful data. These modules are used during the construction of the LTS graph structure in Algorithm 1. Suppose that $Edge ::= \texttt{Node.t} \times \texttt{EdgeData.t} \times \texttt{Node.t}$ is the type of LTS edge.

The algorithm follows a fairly standard DFS pattern and primarily serves as a replacement for the original AI framework which was visiting nodes in a predefined order, such as WTO (see Section 3.2). It visits each node at least once where the node is processed by the `process_cfg_node` function on the first visit (line 6) and by the `process_visited_cfg_node` function on any subsequent visit (line 3). Section 4.1.1 briefly discussed the ideas behind these functions and the implementation details will not be covered here.

Furthermore, `process_cfg_node` function indirectly calls the `exec_instr` function which implements the logic described at the end of Section 4.1.2. Extended discussion about implementation is not provided here because it closely follows the previously outlined logic which does not involve any complicated algorithms that would require further discussion. Moreover, the code was also already reviewed in [27] in greater detail. Nevertheless, the implementation of `Call` instruction is worth revisiting as it performs the instantiation pro-

```
module Node : sig
    type t =
        | Prune of (Sil.if_kind * Location.t * Procdesc.Node.id)
        | Start of (Procname.t * Location.t)
        | Join of (Location.t * Procdesc.Node.id)
        | Exit
    ...
end
module EdgeData : sig
    type t = {
        backedge: bool
        ; conditions: EdgeExp.Set.t list
        ; condition_norms: EdgeExp.Set.t list
        ; assignments: (HilExp.access_expression * EdgeExp.ValuePair.t) list
        ; branch_info: (Sil.if_kind * bool * Location.t) option
        ; calls: EdgeExp.CallPair.Set.t }
    ...
    val add_invariants : t -> AccessExpressionSet.t AccessPath.BaseMap.t -> t
end
```

Listing 5.3: Implementation of LTS `Node` and `EdgeData` modules.

cess for real function summaries or models. Listing 5.4 shows an outline of the `exec_instr` function with focus on the call instruction. It follows the two-step instantiation scheme described in Section 4.3.4 where the `Call` instruction only substitutes the formal parameters in the bound expressions (return and formal bounds) with actual arguments but does not compute any lower or upper value bounds. More specifically, lines 15–18 substitute the arguments and then lines 19–20 create the summary mapping for delayed instantiation. Lines 24–27 deal with the experimental support for function models of built-in or library functions as discussed in Section 5.1. Implementation is not discussed due to its experimental nature. Finally, lines 29–31 are responsible for updating the appropriate summary components, where `ident_map` (defined in Section 4.1.2 as $\mathcal{M} ::= Ident \times EdgeExp \times EdgeExp$) holds the associations between identifiers and expression pairs.

The remaining `Load` `Store` and `Prune` instructions are responsible for creation of the other LTS node types and also connecting edges as described in Section 4.1.1.

5.4 Compound Local Bounds

Implementation of the *compound local bounds* feature was fairly straightforward. The first step was to extend the existing `EdgeData` modules of both LTS and *DCP* graphs to store the `condition_norms: EdgeExp.Set.t list` which are copied over from LTS to *DCP* during construction. The next step was to implement logic for detection of *decreasing edges* where *compound norm* norm is considered to be decreasing if at least one atomic norm is being decreased on an edge. Listing 5.5 shows the code that computes the $\xi(\texttt{cv}) \subseteq E$ set (see Section 4.2.2) of decreasing edges for compound norms instead of atomic norms. Note that the `scc_edges` set contains all edges which can be executed more than once, i.e., loop edges.

Algorithm 2 shows how the individual decreasing edge sets $\xi(\texttt{cv})$ are used to determine the *compound local bounds* and closely follows the outline from Section 4.2.2. Additionally, Algorithm 3 outlines the implementation of the $T\mathcal{B}(\tau)$ procedure which had to be gener-

```
let exec_instr = fun (graph_data, proc_data) instr -> match instr with
| Prune (cond, loc, branch, kind) -> ...
| Store {e1= lhs; typ; e2= rhs; loc} -> ...
| Load {id; e; typ; loc} -> ...
| Call ((ret_id, ret_typ), Exp.Const (Const.Cfun callee_pname), args, loc, _) ->
    (* process arguments and extract norms out of them.
     * lb_args and ub_args are used during substitution *)
    let lb_args, ub_args = ... in
    let call_pair, arg_norms = ... in
    let summaries, return_bound, edge_data =
    match analyze_dependency callee_pname with
    | Some payload ->
        (* substitute optional return bound with actual arguments *)
        let subst_ret_bound_opt = ... in
        (* instantiate payload.formal bounds and create edge assignments
         * function ~f(...) uses lb_args and ub_args *)
        let edge_data = EdgeExp.Map.fold formal_bounds edge_data ~f:(...) in
        let summary = {payload with return_bound= subst_ret_bound_opt} in
        let summaries = Location.Map.add loc summary summaries in
        let return_bound_pair = ...
        in
        (summaries, return_bound_pair, edge_data)
    | None ->
        match LooperCostModels.Call.get_model callee_pname args_pairs with
        | Some call_model -> (* instantiate and return model summary *)
        | None -> (* no model, ignore / report issue *)
    in
    ident_map= Ident.Map.add ret_id return_bound ident_map in
    calls= EdgeExp.CallPair.Set.add call_pair calls in
    norms= EdgeExp.Set.union norms arg_norms in
    ...
```

Listing 5.4: Interpretation of SIL `Call` instructions during the construction of LTS graph. The code shows an outline of the partial instantiation of function summaries in the first phase. It includes the experimental instantiation of function models.

```
let condition_norms_decreasing norms (edge_data : DCP.EdgeData.t) =
  List.exists norms ~f:(fun norm_set ->
      (* It is enough if at least one norm from the compound norm is decreased *)
      EdgeExp.Set.exists
        (fun condition_norm ->
          match DC.get_dc condition_norm edge_data.constraints with
          | Some dc ->
              DC.same_norms dc && DC.is_decreasing dc
          | _ ->
              false )
        norm_set )
in
let get_decreased_edge_set condition_norms = DCP.EdgeSet.filter
    (fun (_, edge_data, _) -> condition_norms_decreasing condition_norms edge_data)
    scc_edges
in
```

Listing 5.5: Detection of decreasing compound norms and computation of sets of edges with decreasing compound norms.

Algorithm 1: Constructing the node and edge sets for LTS

Input : Infer CFG node $n \in N_{\mathcal{C}}$; set of visited nodes $N \subset N_{\mathcal{C}}$; construction data $(l_d, e_d, L_d, T_d) \in \texttt{Node.t} \times \texttt{EdgeData.t} \times 2^{\texttt{Node.t}} \times 2^{Edge}$, where l_d is the last created LTS node, e_d is the data of the LTS edge which is currently being constructed and L_d and T_d are already created nodes and transitions respectively.

Output:

```
1  def traverseCFG(n, N, (l_d, e_d, L_d, T_d)):
2      if n ∈ N then
3          process_visited_cfg_node(n, data)
4      else
5          N ← N ∪ {n};
6          process_cfg_node(n, data);
7          if type(n) = exit then
8              l_e ← Node.Exit;
9              L_d ← L_d ∪ {l_e};
10             T_d ← T_d ∪ {(l_d, EdgeData.add_invariants e_d, l_e)};
11             return (l_e, EdgeData.empty, L_d, T_d);
12         else
               /* store a copy of the original data for each DFS path    */
13             (l, e) ← copy((l_d, e_d));
14             for n_s ∈ successors(n) do
15                 traverseCFG(n_s, N, (l, e, L_d, T_d));
```

alized for compound norms. The function `process_norm` implements the original $T\mathcal{B}(\tau)$ procedure which was modified in Section 4.2.2 to take norms instead of transitions as input. The presented algorithm does not include error checking, use of cache or checks for infinite recursion which can occur for example when the real transition bounds are exponential.

5.5 Lower Variable Bound

Section 4.2.1 proposed an algorithm for computing lower variable bounds which were a necessary prerequisite for implementation of *interprocedural analysis*. The section covered how the proposed algorithm was obtained through a modification of the original variable bound algorithm. Listing 5.6 shows a simplified version of the implementation which leveraged the similar nature of both procedures and simply modified the original `variable_bound` function to support computation of both lower and upper bound. Lines 2–5 choose the appropriate sub-algorithms based on the bound type which are then used on lines 9 and 18. Note that the actual implementation contains additional updates of cache and the *true* branch of the if-statement on line 10 handles a special case when the end of a reset chain is a *formal variable*.

The concept of formal variables had to be introduced in order to support analysis of pointer side-effects. The original Loopus tool considered all formal parameters of an analyzed function to be *immutable* constant values and transition bounds were built over these

Algorithm 2: Determining *compound local bounds*

Input : set of DCP loop edges `scc_edges` $\subset E$

```
1  E_r ← scc_edges;
2  for e ∈ scc_edges do
3      cv = e.condition_norms;
4      E_cv ← get_decreased_edge_set(cv);
5      for e ∈ (E_cv ∩ E_r) do
6          e.bound_norms ← cv;
7      E_r ← E_r \ E_cv;
8      remove_edges(ΔP, E_cv);
9      E_non_scc = E_r \ get_scc_edges(ΔP);
10     for e_non_scc ∈ E_non_scc do
11         e_non_scc.bound_norms ← cv;
12     add_edges(ΔP, E_cv);
13     E_r ← E_r \ E_non_scc;
14     if E_r = {} then
15         break;
```

Algorithm 3: Computing $T\mathcal{B}(\tau)$ for edges with *compound local bounds*

Input : DCP edge $e \in E$

Output: bound expression `EdgeExp.Max`

```
1  def transition_bound(e):
2      A_max ⊂ EdgeExp ← {};
3      for N ∈ e.bound_norms do
4          A_min ⊂ EdgeExp ← {};
5          for n ∈ N do
6              A_min ← A_min ∪ { process_norm(n)};
7          A_max ← A_max ∪ {EdgeExp.Min(A_min)};
8      return EdgeExp.Max(A_max);
```

parameters. However, in most cases, side-effects imply that the value of a parameter is changed by the function through pointer access. Thus, the original assumption about constant parameter values no longer holds and a temporary "*hacky*" solution was implemented. The idea is to treat formal parameter norms of *pointer to integer* type as *variables* $\mathcal{V}$ when deriving difference constraints but as *symbolic constants* $\mathcal{C}$ when the parameter appears at the end of a reset chain. This approach is not optimal but allows for basic analysis of side-effects. A proper solution would probably involve introduction of so-called *shadow variables*, i.e., generated auxiliary variables initialized with the parameter values at the beginning of the analyzed function.

```
let variable_bound ~bound_type norm cache =
    let sum_function, min_max_constructor = match bound_type with
    | BoundType.Upper -> (calculate_increment_sum, max_constructor)
    | BoundType.Lower -> (calculate_decrement_sum, min_constructor)
    in
    match EdgeExp.Map.find_opt norm bound_cache with
    | Some bound -> (bound, cache) (* retrieve from cache *)
    | None -> if is_terminal_norm norm then (norm, cache) else
        let sum_part, cache = sum_function (EdgeExp.Set.singleton norm) cache in
        if Resets.is_empty norm_updates.resets then ( ... ) else (
            let min_max_reset_args, cache =
            Resets.fold (fun (_, norm, const) (args, cache) ->
                let var_bound, cache = variable_bound ~bound_type norm cache in
                let min_max_arg = ... in
                (EdgeExp.Set.add min_max_arg args, cache)
            )
            norm_updates.resets (EdgeExp.Set.empty, cache) in
            let min_max_reset = min_max_constructor min_max_reset_args in
            (EdgeExp.add min_max_reset sum_part, cache)
        )
```

Listing 5.6: A generalized version of the original `variable_bound` procedure which can compute either $LV\mathcal{B}(\mathtt{v})$ or $V\mathcal{B}(\mathtt{v})$.

5.6 Interprocedural Analysis

The `LooperSummary.ml[i]` files contain the definition of summary types as proposed in Section 4.3.1 and the implementation can be seen in Listing 5.7 Lines 12–16 define the main function summary $\mathbb{S}_{\Delta\mathcal{P}}$ with $\mathcal{F}$ as `formal_map`, $2^{\mathbb{S}_T}$ as `bounds`, $\mathcal{P}_\varepsilon$ as `return_bound` and finally $2^{\mathcal{B}}$ as `formal_bounds`. Lines 5–10 define the transition summary $\mathbb{S}_T$ where `src_node` and `dst_node` correspond with the transition τ, $e \in EdgeExp$ as `bound`, $M \in 2^{\mathcal{M}}$ as *monotony_map* and lastly the set of call summaries $S_C \in 2^{\mathbb{S}_C}$ as `calls`. Furthermore, line 1 defines the type for the call summary $\mathbb{S}_C$ where $n \in Procname$ is represented by `name`, $l \in Loc$ by `loc` and the set of transition summaries $S_T \in 2^{\mathbb{S}_T}$ by `bounds`. The call summary type is additionally wrapped in a variant type `call` which can hold summaries of both real and model calls. The `model_call` type is omitted due to its experimental nature.

5.6.1 Monotonicity and Partial Differentiation

Section 4.3.2 proposed a solution for the problem of sound instantiation of function summaries. The issue was whether a lower or upper bound of an argument a_i should be substituted into summary bounds during instantiation. The idea was to analyze the monotonicities of individual transition bounds with respect to each function parameter and then substitute either $LV\mathcal{B}(a_i)$ or $V\mathcal{B}(a_i)$ based on the result. The proposed solution leveraged partial derivatives and the Why3 prover platform (mainly using Z3 prover) to determine the monotonicities. Listing 5.8 presents a simplified version of the `partial_diff` function which performs the partial differentiation using a recursive approach and leveraging the *sum*, *product* and *quotient* rules. The code is fairly straightforward and follows a standard differentiation pattern. Lines 20-22 process the leaf nodes of an expression AST and terminate the recursion. More specifically, line 20 treats an *access path* π expression as

```
type real_call = {name: Procname.t; loc: Location.t; bounds: transition list}

and call = ModelCall of model_call | RealCall of real_call

and transition =
  { src_node: LTS.Node.t
  ; dst_node: LTS.Node.t
  ; bound: EdgeExp.T.t
  ; monotony_map: Monotonicity.t AccessExpressionMap.t
  ; calls: call list }

type t =
  { formal_map: FormalMap.t
  ; bounds: transition list
  ; return_bound: EdgeExp.ValuePair.pair option
  ; formal_bounds: EdgeExp.ValuePair.pair EdgeExp.Map.t}
```

Listing 5.7: The definition of summary types with the model summary omitted.

```
let rec partial_diff exp ~diff_var = match exp with
| BinOp ((Binop.PlusA _ as op), lexp, rexp)
| BinOp ((Binop.MinusA _ as op), lexp, rexp) ->
    (* Apply Sum rule: f +/- g -> f' +/- g' *)
    ...
| BinOp (Binop.Mult kind, lexp, rexp) ->
    (* Apply Product rule: f * g -> (f'  * g) + (f * g') *)
    let lexp' = partial_diff lexp ~diff_var in
    let rexp' = partial_diff rexp ~diff_var in
    BinOp (Binop.PlusA kind, BinOp (Binop.Mult kind, lexp', rexp),
                             BinOp (Binop.Mult kind, lexp, rexp'))
| BinOp (Binop.DivI, lexp, rexp) ->
    (* Apply Quotient rule: f / g -> (f  * g - f * g') / g^2 *)
    let lexp' = partial_diff lexp ~diff_var in
    let rexp' = partial_diff rexp ~diff_var in
    let num_lhs = BinOp (Binop.Mult None, lexp', rexp) in
    let num_rhs = BinOp (Binop.Mult None, lexp, rexp') in
    BinOp (Binop.DivI, BinOp (Binop.MinusA None, num_lhs, num_rhs),
                       BinOp (Binop.Mult None, rexp, rexp))
| UnOp (Unop.Neg, exp, typ) -> UnOp (Unop.Neg, partial_diff exp ~diff_var, typ)
| Access access -> if AccessExpression.equal access diff_var then one else zero
| Strlen str -> if AccessExpression.equal str diff_var then one else zero
| Const (Const.Cint _) -> zero
| _ -> ...
```

Listing 5.8: A simplified version of the recursive `partial_diff` function for partial differentiation. It leverages the *sum*, *quotient*, and *product* rules in combination with recursion to compute the result.

```
1  let determine_monotonicity exp tenv prover_data =
2      let exp = simplify exp in
3      let why3_solve_task task = ... in
4      let derivative_variables = get_accesses exp in
5      AccessExpressionSet.fold (fun diff_var acc ->
6          let exp' = partial_diff exp ~diff_var |> simplify in
7          if is_const exp' then
8              let exp_monotonicity = incr_or_decr exp diff_var in
9              AccessExpressionMap.add diff_var exp_monotonicity acc
10         else
11             let why3_exp' = to_why3_expr exp' tenv prover_data in
12             let non_zero_exp' =
13               t_app_infer ne_symbol [why3_exp'; zero_const]
14             in
15             let free_vars = Mvs.keys (t_vars non_zero_exp') in
16             let non_zero_forall = t_forall_close_simp free_vars [] non_zero_exp' in
17             let task =
18               Task.add_prop_decl base_task Decl.Pgoal nonzero_goal non_zero_forall
19             in
20             match (why3_solve_task task).pr_answer with
21             | Call_provers.Valid ->
22                 let exp_monotonicity = incr_or_decr exp diff_var in
23                 AccessExpressionMap.add diff_var exp_monotonicity acc
24             | Call_provers.Invalid | Call_provers.Unknown _ ->
25                 AccessExpressionMap.add diff_var Monotonicity.NotMonotonic acc
26             | _ -> (* error handling *)
27     ) derivative_variables AccessExpressionMap.empty
```

Listing 5.9: A simplified version of the `determine_monotonicity` function which determines the monotonicity of the input expression for each expression variable. For brevity, handling of type conditions that can increase the success rate of the used prover was omitted.

an *variable* if it is equal to the current differentiation variable `diff_var` or as *constant* otherwise. Regardless of the current `diff_var`, constants are always derived as zero.

The snippet does not include the code for handling of special operators such as min or max which are difficult to differentiate properly, especially when there are multiple arguments. The adopted solution is thus heuristic in nature and not mathematically accurate. However, by leveraging the knowledge about the specific use-case (determining monotonicity), it was possible to come up with a solution which reasonably approximates the expected outcomes. Namely, the min and max operators are transformed to + for the purposes of the differentiation. Such transformation likely is not generally correct, however, for our practical purposes it proved to be sufficient.

The previously discussed `partial_diff` function serves as a building block used in the `determine_monotonicity` function presented in Listing 5.9. The function first finds all *access path* variables used in the input expression `exp` on line 4. Then the expression is differentiated with respect to each variable on line 6 and the derivative `exp'` is used to construct a Why3 expression and a prover task out of it on lines 11–19. Finally, the task is given to Why3 and checked whether it is valid or not on line 20. Note that the Why3 API works with *validity* instead of *satisfiability* but the task is then internally converted to a satisfiability problem before it is delegated to Z3, hence the lines 21–25 process the result in terms of validity. A valid result on line 21 implies that the derivative has no

```
1  let formal_bounds, cache = EdgeExp.Map.fold (fun vfg_norm formal (bound_map, cache) ->
2      let upper_bound, cache = variable_bound ~bound_type:BoundType.Upper vfg_norm cache in
3      let lower_bound, cache = variable_bound ~bound_type:BoundType.Lower vfg_norm cache in
4      (EdgeExp.Map.add formal (lower_bound, upper_bound) bound_map, cache)
5  ) formals_mapping (EdgeExp.Map.empty, cache)
6  in
7
8  let ret_type = Procdesc.get_ret_type proc_desc in
9  let return_bound = match ret_type.desc with
10 | Tint _ ->
11     let return_access = AccessPath.base_of_pvar (Procdesc.get_ret_var proc_desc) ret_type
12     |> HilExp.AccessExpression.base
13     in
14     let return_norm =
15         EdgeExp.T.Max (EdgeExp.Set.singleton (EdgeExp.T.Access return_access)) in
16     let ub = variable_bound ~bound_type:BoundType.Upper return_norm cache in
17     let lb = variable_bound ~bound_type:BoundType.Lower return_norm cache in
18     Some (lb, ub)
19 | _ ->
20     None
21 in
22 let summary : LooperSummary.t = { ... } in
23 Some summary
```

Listing 5.10: Construction of formal and return bounds used in the function summary at the end of analysis. It is an extension to the code from Listing 5.1 which omitted parts related to return and formal bounds.

root, i.e., the input expression `exp` is monotonic with respect to variable `diff_var`, but a further investigation of the specific monotonicity type is required. The omitted function `incr_or_decr` tries to find out by substituting two increasing constant values instead of `diff_var` into the original input expression `exp`. This approach will fail to give a decisive answer when a locally constant interval of `exp` is encountered, but such cases did not occur during the experimental testing.

5.6.2 Construction and Instantiation of Summaries

Listing 5.1 already presented part of the construction process of summaries which involved creation of transition summaries but omitted the return and formal bounds. Listing 5.10 thus expands the original code and presents a simplified version of the code that computes both of these bounds. Lines 1–6 compute the formal bounds using the `formals_mapping` which maps the auxiliary variables `vfg_norm` generated during the *flow-sensitivity* transformation to the original `formal` parameter variables. Hence, lines 2 and 3 call the `variable_bound` function for the `vfg_norm` instead of directly calling it for the `formal` variable. The remaining lines compute the lower and upper bounds for the `return` variable but only if it exists and is of integer type (line 10) at the same time.

As mentioned previously in Section 5.2, the `analyze_procedure` function from Listing 5.1 also simultaneously instantiates all function summaries stored on *DCP* edges with the `instantiate_function_calls` function on line 35. Algorithm 4 outlines the instantiation process for a single DCP edge $e = l_1 \xrightarrow{C} l_2$, where the set $C \subset Calls$ is a set of function expressions $(n, A, l) \in C$ defined in Section 4.3.4. Each function call expression

Algorithm 4: Instantiating all function calls on a DCP edge e

Input : DCP edge $e \in E$; `Location.Map.t` mapping from locations to function summaries $LS \subset \mathtt{L} \times \mathbb{S}_{\Delta\mathcal{P}}$

Output: Set of instantiated call summaries $S_C \subset \mathbb{S}_C$

```
1 def instantiate_function_calls(e, LS):
2     S_C ⊂ 𝕊_C ← {};
3     for (n, A, l) ∈ e.calls do
          /* retrieve function summary stored in LS                    */
4         (F, S_T, B_F, b_R) ← Location.Map.find(l, LS);
5         S'_T ← {};
6         for s_T ∈ S_T do
              /* instantiate all transition bounds of the retrieved summary */
7             S'_T ← S'_T ∪ { instantiate_transition_summary(s_T, A, F)};
8         S_C ← S_C ∪ {(n, l, S'_T)};
9     return S_C;
```

$(n, A, l) \in \mathtt{Procname.t} \times 2^{\mathtt{EdgeExp.t}} \times \mathtt{Loc.t}$ from the $e.calls$ list is first used on line 4 to retrieve the associated function summary which was previously stored in LS (also defined in Section 4.3.4) using the call-site location l. Each function summary contains a set of transition summaries S_T which are then instantiated with the `instantiate_transition_summary` recursive function on line 7. This function produces a modified transition summary s'_T for every input summary s_T. These summaries are used to construct a set of modified summaries S'_T and those are in turn used to construct a modified set S'_C which is returned and used in Listing 5.1 on line 42 to construct a transition summary for the currently analyzed function $\Delta\mathcal{P}$.

Algorithm 5 then shows the implementation of the `instantiate_transition_summary` function from line 7 of the previous algorithm.

The function works recursively by first instantiating the transition bound e on line 3 (constant bounds do not have to be instantiated) and recomputing the monotony map M' for the instantiated bound e'. A new monotony map is required because the bound e' is now built over the formal parameters of the currently analyzed function $\Delta\mathcal{P}$ instead of parameters of the called function itself. The final step is to instantiate the call summaries present on τ which are stored in S_C. Each function call represented by a function summary $s_c \in S_C$ also contains a set of transitions represented by summaries S_T. These transition summaries are thus recursively instantiated by the `instantiate_transition_summary(`t`)` function call on line 10 and used to construct a new set of instantiated summaries S'_T which is used to create updated set of call summaries S'_C. on line 11. Finally, an instantiated transition summary for input transition τ is returned on line 12. Note that both of these algorithms are simplified versions of the actual implementation which additionally includes error checking, management of computational cache and also handling of model call summaries which are currently in an experimental stage of development.

The final piece of the puzzle is the `instantiate_bound` function which instantiates a single input bound expression e based on the principle presented in Section 4.3.4. Algorithm 6 outlines the process where the task is to maximize the value of expression e via substitution of either upper or lower bound of each argument. More specifically, the notation $\pi \in e$ on line 2 denotes a loop over all *access path* variables π in the expression e,

Algorithm 5: Recursively instantiating transition summary $s_T \in \mathbb{S}_T$

Input : transition summary $s_T = (\tau, e, M, S_C)$; set of arguments $A \subset EdgeExp$; formal map $F \subset \mathcal{F}$

Output: instantiated transition summary (τ, e', M', S'_C)

```
1  def instantiate_transition_summary((τ, e, M, S_C), A, F):
2      if EdgeExp.is_const(e) then e' ← e
3      else e' ← instantiate_bound(e, A, M, F)
       /* create new M' for the instantiated bound e'                    */
4      if EdgeExp.is_const(e') then M' ← {}
5      else M' ← EdgeExp.determine_monotonicity(e')
6      S'_C ← {}
7      for (n, l, S_T) ∈ S_C do
8          S'_T ← {}
9          for t ∈ S_T do
               /* recursive instantiation                                */
10             S'_T ← S'_T ∪ { instantiate_transition_summary(t, A) }
11         S'_C ← S'_C ∪ {(n, l, S'_T)}
12     return (τ, e', M', S'_C)
```

i.e., it represents the visit of each π leaf node in AST of e. The formal variable π is then used on line 3 to retrieve the associated formal index i from the formal map F which is in turn used to retrieve the corresponding argument arg from the list of argument A on line 4. Furthermore, the function monotonicity m associated with the formal parameter π is retrieved from the monotonicity map M on line 7. Based on the monotonicity m, the `variable_bound` function is used to compute either a lower or upper bound for arg on lines 8–14. Moreover, the notation on line 15 denotes the substitution of expression arg into the formal parameter π in expression e. Finally, the expression e is returned on line 16 after all formal parameter variables have been substituted.

Algorithm 6: Instantiating single transition bound expression

Input : bound expression $e \in EdgeExp$; list of arguments $A \subset EdgeExp$; monotonicity map of formal parameters $M \subset \mathcal{M}$; formal map $F \subset \mathcal{F}$

Output: instantiated bound expression e

```
1  def instantiate_bound(e, A, M, F):
2      for π ∈ e do
3          i ∈ ℕ ← get_formal_index(π, F);
4          arg ← A[i];
5          if EdgeExp.is_const(arg) then return arg ;
6          else
7              m ∈ Ω ←AccessExpressionMap.find(π, M);
8              switch m do
9                  case ≤ do
10                     arg ← variable_bound(arg, BoundType.Upper);
11                 case ≥ do
12                     arg ← variable_bound(arg, BoundType.Lower);
13                 case ∼ do
14                     arg ← EdgeExp.Inf;
15             e ← e[π/arg];
16     return e;
```

Chapter 6

Experimental Evaluation of Enhanced Looper

This chapter covers the *experimental evaluation* and *testing* of the new Looper version which was proposed in Chapter 4 and subsequently implemented in Chapter 5. The proposed enhancements and extensions underwent continuous testing throughout the development process on both hand-crafted code examples and real-world code samples. Each enhancement was first tested on suitable hand-crafted examples to ensure proper basic functionality w.r.t. the proposal and then additional testing on real code was conducted. Especially the fairly complex *interprocedurality* extension was extensively tested to ensure proper functionality. The rest of this chapter is structured as follows: First, Section 6.1 revisits the old test suite used during evaluation of the original version of Looper and includes the results from the new version. Further, Section 6.2 briefly discusses the issue of scalability and precision which was evaluated in comparison with the Cost checker on the GNU *coreutils* suite of programs. Finally, Section 6.3 briefly addresses the achieved results and provides a short discussion on the potential future work.

6.1 Revisited Loopus Test-Suite

The original version of Looper was tested on several examples that were taken from [32]. These examples were mostly extracted from the SPEC CPU2006 benchmark and included complex iteration patterns which demonstrated the strengths of original Loopus and compared Looper against the Cost checker from Infer. The evaluation from [27] additionally included a small test-suite of selected Cost examples to provide a fair comparison when tested on code which should favor the Cost checker.

This test-suite was revisited and used to test the new version of Looper to ensure that the original intraprocedural functionality was not compromised after the extensive modifications proposed and implemented in this work. It also provided an insight into the improvements that have been made to the Cost checker over the years and whether it can now handle complex iteration patterns it originally could not. Table 6.1 presents the Loopus test-suite results which are mostly unchanged. The two notable differences are the drastically reduced run-time of both checkers and also few changes in the inferred bounds. The reduction of run-time is mostly attributed to the improved performance of Infer itself and possibly also the Cost checker. The new version of Looper did not introduce any major performance optimizations in regards to the original intraprocedural analysis algorithm and

	Real Bound	Inferred bound				Total Time [s]			
		Cost		**Looper**		**Cost**		**Looper**	
		v1	**v2**	**v1**	**v2**	**v1**	**v2**	**v1**	**v2**
#1	n	n^2	n^2	$2n$	$2n$	10.2	0.7	5.5	2.3
#2	$2n$	n^2	n^2	$2n$	$2n$				
#3	$4n$	∞	∞	$5n$	$5n$				
#4	$*n^2$	∞	∞	n^2	∞				
#5	$2n$	∞	$2n$	$2n$	$2n$				
#6	$*n$	∞	∞	n	n				
#7	$2n$	∞	∞	$2n$	$2n$				
#8	$2n$	∞	∞	$2n$	$2n$				

Table 6.1: The evaluation results taken from [27] and extended with newly measured data using the new versions of Cost and Looper. The origin of these examples is from from [32] which extracted them mostly from SPEC CPU2006 benchmark. *The precise real bounds of examples no. 4 and 6 were $(n+1) * max(n-1, 0) + n + 1$ and $3n + max(m1, m2)$ respectively.

the improvements are thus attributed mostly to the internal changes of Infer. The second difference are the changes to inferred bounds where Cost is now able to successfully analyze the example no. 5 which it previously could not. Additionally, Looper failed to analyze the example no. 4 which it previously could analyze. The reason lies in the inherent non-determinism of the local bound algorithm which assigns local bounds to transitions in a random order. The internal changes made to Looper affected this order, leading to a different choice of local bounds which in turn caused *cyclic* computation, i.e., infinite bound.

When revisiting the Cost checker test-suite, all of the available 71 examples were included this time compared to the 30 curated examples in [27]. Original Looper did not support interprocedural analysis or compound Boolean conditions in loop headers (and branching statements in general) among other things. Moreover, it could not handle various control flow patterns such as `goto` or `do-while`. These limitations were the reason for choosing a specific subset of examples that Looper could potentially handle. The original results of this comparison can be seen in Table 6.2.

	Precise Bounds	**Imprecise Bounds**	**Errors**	***Time [s]**
Looper	24	3	3	5.5
Cost	27	3	0	15.3

Table 6.2: Revisited results from [27]. The Cost test-suite contained 30 relevant functions from the total of 62 functions available at the time. *The total time was measured only on 27 functions which did not cause a crash of Looper.

In comparison, the new version of Looper no longer has these limitations and thus could be tested on the complete Cost test-suite which now contains 71 examples up from the previous 62. The results of this comparison can be seen in Table 6.3

	Precise Bounds	Tight Bounds	Imprecise Bounds	Time [s]
Looper	50	6	15	7.35
Cost	8	57	6	2.51

Table 6.3: The results of an experimental evaluation of Looper using the complete Cost checker test-suite with 71 functions. The test-suite contains some challenging iteration patterns including the use of `goto` to create multiple entry points of a loop. Moreover, it tests the interprocedural capabilities of checkers.

The new table does not include the column for *errors* anymore because Looper now returns ∞ bound instead of crashing when it fails to determine a bound or encounters an error. Additionally, the table now also differentiates between the *precise* and *tight* bounds where a bound is considered to be *precise* if it is equal to the real execution cost, including constants. On the other hand, a bound e is considered *tight* if the big-Θ of e is equal to the big-Θ of the real bound. Analyzing the results, all of the imprecise bounds in case of Looper are in fact equal to ∞ due to a fail during analysis. In most cases a successful analysis of these examples is within the reach of Looper and not an inherent limitation of the used algorithm. However, there are also a few cases where the algorithm in its current state cannot possibly succeed. One such case is presented in Listing 6.1.

```
1  void larger_state_FN() {
2      int i~= 0, k~= 0;
3      while (k~< 100) {
4          i++;
5          if (i~>= 10000) {
6              k++;
7              i~= 0;
8          }
9      }
10 }
```

Listing 6.1: A function from the Cost test-suite which demonstrates an inherent limitation of Looper's bound algorithm. Looper fails to determine a local bound and returns ∞.

Figure 6.1 presents the *DCP* graph derived from the code of this function and will be used to explain the limitation. The core issue is that Looper is unable to determine the local bounds for all *DCP* transitions and the reason is two fold. First, the norm $[100 - k]$ is, indeed, assigned as a local bound to the transition τ_2 but removing τ_2 from the graph does not break the strongly connected component of the while loop because the transition τ_3 still exists. Second, the inverted *if-statement* condition `i >= 10000` produces norm $e = [i - 9999]$. However, norm e is not decreased on any edge, i.e., the increment $i++$ on line 4 actually increases the value of the norm as follows $[(i + 1) - 9999] \rightarrow [i - 9999] + 1$. As a consequence, norm e cannot be used as a local bound for the transition τ_1 and in fact no other norm can be used as a local bound for τ_1. This exhausts all of the options and the remaining transitions end up with no local bounds. In general, Looper in its current state cannot handle such iteration patterns involving this type of inverted conditions, even if the *path-sensitive* reasoning extension from [32] has been implemented.

In conclusion, most of the imprecise bounds (∞) are caused by rather technical reasons than due to general limitations of the approach and only few iteration patterns present in the test-suite could pose a difficult problem to solve.

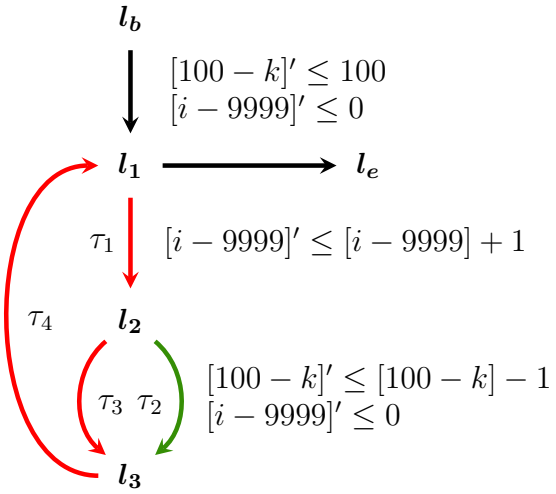


Figure 6.1: A *DCP* graph derived from the code in Listing 6.1. The norm $[100-k]$ is assigned as a local bound for τ_2 but no local bound can be determined for the remaining transitions τ_1, τ_3 and τ_4. Note that constant assignments were omitted for brevity.

6.2 Evaluation of Scalability and Precision

The next step was to evaluate the scalability and precision of the new version of Looper on real-world code to determine whether it can handle large codebases as well as its main competitor, the Infer Cost checker. The entire publicly available codebase of GNU *coreutils*[1] (totaling roughly 95,000 lines of code) was used as a suitable benchmark for the evaluation purposes. It consists of complex low-level programs that heavily rely on the use of pointers and involve complicated control flow structures such as `goto` and `switch` statements.

The experiments were performed on a 2020 Macbook Air machine with the ARM based 3.2 GHz Apple M1 processor and 16 GB of system memory, running the Ventura 13.3.1 version of the macOS operating system. Table 6.4 shows the analysis results measured on the total number of 2,143 functions.

	Success	**Infinite Bound**	**Fail Rate [%]**	**Total Time [s]**
Looper	489	1,654	77.2	161
Cost	1,508	635	29.6	1,026

Table 6.4: Evaluation results for the total number of 2,143 functions from the GNU *coreutils* codebase. It presents the *fail-rate* and *total time* of both Looper and Cost. The data is based on the assumption that the source code in reality does not contain any infinite loops.

Firstly, the assumption is that none of the functions contain an infinite loop and any inferred ∞ bound is thus considered incorrect. Moreover, due to the amount of code, it was not possible to closely examine each analyzed function to determine whether a derived non-infinity bound is actually correct or not. Considering this, the data shows that Cost is superior when it comes to the ability to analyze real-world code, successfully analyzing almost 4 times as many functions as Looper. However, it is important to note that upon

[1]The source code is available at the GitHub repository: `https://github.com/coreutils/coreutils`.

a closer inspection of few analyzed functions which produced ∞ bound, it was determined that the reason is rather technical. In most instances, the failure was caused by the inability to handle a certain language construct, namely complex expressions involving the use of pointers.

As a result, many functions which in fact do not contain any loop failed to produce a valid bound due to a failure during construction of *DCP* or evaluation of return bounds. It was thus concluded that it does not demonstrate a general limitation of the approach to handle complex iteration patterns. One major class of program loops that Looper currently fails on due to an inherent limitation are loops which process data in memory and break when a certain condition is met. For example iterating until a `NULL` pointer or a string delimiter is encountered. Also, seemingly infinite loops with `break` statements in the loop body are currently out of reach. However, even the Cost analyzer struggles with these loop patterns.

When it comes to the performance of both tools, neither Cost nor Looper experienced any timeouts and managed to complete the analysis of all functions. The results indicate that the current version of Looper is roughly $\sim$6.4 times faster than the current version of Cost when analyzing a large C codebase. Looper thus has a clear advantage on the performance front which is unfortunately outweighed by the considerably higher rate of failure as discussed above. Nevertheless, the presented results bear witness to the future potential of the approach adopted by Looper when it comes to *scalability*. As a final note, the original version of Looper is not included in the comparison due to its inability to analyze any real code in practice without causing crashes of Infer. Moreover, it would require non-trivial changes of the original source code to make it compatible with the current version of Infer.

6.3 Summary and Future Work

Section 6.1 first demonstrated that the intraprocedural part of Looper still works correctly after the implementation of all proposed enhancements by revisiting the old Loopus test-suite. Next, the complete Cost test-suite consisting of 71 small hand-crafted functions was used to validate that the implemented enhancements work correctly w.r.t. the proposal. It was primarily used to test if the new version of Looper is able to construct *DCP* graphs even for more complex control flow structures and if the major interprocedural extension works correctly w.r.t. the proposal.

The evaluation of scalability and precision in Section 6.2 reveals that the new version of Looper is now able to analyze real-world low-level C code at scale and with reasonable performance. In fact, compared to the Cost checker, Looper was proven to be significantly faster at the expense of precision. Moreover, compared to the original version of Looper which failed to analyze even trivial real-world code, the new version was able to infer a non-infinity bound for 489 functions from the GNU *coreutils* repository which marks a success, despite the comparison with Cost analyzer.

Based on these results, the potential future work will mainly focus on eliminating the current technical limitations which hinder the precision of the analysis. Namely, the testing on real-world code has shown that complex expressions involving the use of pointers are a major failure point due to the fairly incomplete implementation. As such, additional improvements to the use of newly introduced *access paths* could lead to major increase of precision. Furthermore, following the direction of the Cost checker and adding support for other higher-level languages without pointers (such as Java), could alleviate this issue and

present a more viable path for further development. This would require implementing the previously discussed support for various data *containers* which are heavily used in those languages.

Orthogonally to those issues, further improvements to the *user experience* are crucial for the usability of the tool and possible adoption by users. Namely, a better reporting of issues and support for differential analysis pose a major future work prospect. The new version of Looper already includes a few improvements in those aspects but still leaves a lot to be desired. Compared to the original version, logging was significantly improved, the analysis results are output in a form of a structured JSON file and Infer issues are reported when Looper fails to determine a bound. Nonetheless, it would be worthwhile to further improve the accuracy and verbosity of the reporting to help the end users with analysis of the results. Moreover, implementing the differential analysis in the same fashion as the Cost analyzer is currently another major future work priority apart from improving the precision.

Chapter 7

Conclusion

The introductory chapters of this thesis were dedicated to the topic of program analysis on a more theoretical level. The goal of these chapters was to provide a basic insight into the motivation behind program analysis, what the key properties of any analysis are, and finally a short overview of most common analysis techniques was provided. Special attention was given to the abstract interpretation technique as it plays a major role in the subsequently discussed Meta Infer static analysis framework. The following chapter focused on the Looper analyzer — proposed and implemented within the author's bachelor's thesis [27]. More specifically, the main limitations of the original tool that motivated further development were discussed.

The biggest pain point of the first Looper version was its inability to analyze a vast majority of *real-world* code outside of contrived examples. It was shown that the two main causes are insufficient *abstraction algorithm* and missing support for the *interprocedural analysis*. Solutions to both of these issues as well as some additional enhancements to improve the precision were proposed and implemented in this work. Namely, a completely new *abstraction algorithm* which can handle more complex control flow structures as well as more complicated program expressions involving the use of pointers and data structures was implemented. Additionally, the precision of the original intraprocedural analysis was improved by adding support for compound loop conditions. However, most importantly, the main contribution of this work — the *interprocedural analysis* extension — was successfully implemented and tested on *real-world* code.

The newly implemented features and enhancements were successfully tested and experimentally evaluated on hand-crafted examples as well as extensive *real-world* software. The evaluation confirmed that Looper is now indeed able to analyze real code and infer precise or at least tight upper bounds on the execution cost. Moreover, the evaluation has shown that Looper scales very well, even compared to the existing Infer Cost analyzer. Overall, the achieved results were deemed promising, especially when compared to the first version of Looper, and further improvements to the accuracy can be made. For instance, the precision could be significantly increased by further focusing on elimination of technical limitations. Other possible future improvements include added support for more programming languages with everything it entails, such as the analysis of code involving the use of data containers. Lastly, the implementation of *differential analysis*, which could detect the complexity degradation of functions between different versions of a program, presents an exciting future possibility.

Bibliography

[1] Appel, A. W. SSA is Functional Programming. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. April 1998, vol. 33, no. 4, p. 17–20, [cit. 2023-03-24]. DOI: 10.1145/278283.278285. ISSN 0362-1340. Available at: https://doi.org/10.1145/278283.278285.

[2] Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J. and Pugh, W. Experiences Using Static Analysis to Find Bugs. *IEEE software.* 1st ed. IEEE. 2008, vol. 25, no. 5, p. 22–29, [cit. 2023-01-28]. DOI: 10.1109/MS.2008.130. Available at: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/34339.pdf.

[3] Barrett, C. *SAT and SMT: Theory and Practice* [online]. September 2008 [cit. 2023-01-22]. Available at: https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/barret1_sat.pdf.

[4] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B. et al. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM.* 1st ed. New York, NY, USA: Association for Computing Machinery. February 2010, vol. 53, no. 2, p. 66–75. DOI: 10.1145/1646353.1646374. ISSN 0001-0782. Available at: https://doi.org/10.1145/1646353.1646374.

[5] Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G. and Paskevich, A. The Why3 platform. *LRI, CNRS & Univ. Paris-Sud & INRIA Saclay* [online]. 1st ed. 2011, vol. 2, no. 1, [cit. 2023-01-22]. Available at: https://why3.lri.fr/download/manual-0.82.pdf.

[6] Bobot, F., Filliâtre, J.-C., Marché, C. and Paskevich, A. Why3: Shepherd Your Herd of Provers. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages* [online]. Wroclaw, Poland: [b.n.], 2011, p. 53–64 [cit. 2023-01-22]. Available at: https://hal.inria.fr/hal-00790310.

[7] Bodik, R., Cheung, A., Ahmad, M., Ringer, T. and Tebbs, B. *Data Flow Analysis* [online]. University of Washington, 2016 [cit. 2023-01-28]. Available at: https://courses.cs.washington.edu/courses/cse401/16wi/sections/section8/dfa.html.

[8] Bygde, S. *Static WCET analysis based on abstract interpretation and counting of elements.* Västerås, SE, 2010. [cit. 2023-05-23]. Dissertation. Mälardalen University. Available at: https://www.diva-portal.org/smash/record.jsf?dswid=6649&pid=diva2%3A292120.

[9] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A. et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. 1st ed. Springer Berlin, Heidelberg, June 2001. ISBN 978-3-540-41523-7.

[10] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P. et al. Moving Fast with Software Verification. Springer International Publishing. April 2015, p. 3–11, [cit. 2023-01-14]. Available at: https://research.facebook.com/file/892640211665108/publication00124_download0001.pdf.

[11] Calcagno, C., Distefano, D., O'Hearn, P. and Yang, H. Compositional Shape Analysis by Means of Bi-Abduction. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2009, p. 289–300 [cit. 2023-01-22]. POPL '09. DOI: 10.1145/1480881.1480917. ISBN 9781605583792. Available at: https://doi.org/10.1145/1480881.1480917.

[12] Cousot, P. *A Tutorial on Abstract Interpretation* [online]. Paris, FR: [b.n.], January 2005 [cit. 2023-01-30]. Available at: https://homepage.cs.uiowa.edu/~tinelli/classes/seminar/Cousot--A%20Tutorial%20on%20AI.pdf.

[13] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252 [cit. 2023-01-30]. POPL '77. DOI: 10.1145/512950.512973. ISBN 9781450373500. Available at: https://doi.org/10.1145/512950.512973.

[14] Distefano, D., Fähndrich, M., Logozzo, F. and O'Hearn, P. W. Scaling static analyses at Facebook. *Communications of the ACM*. ACM New York, NY, USA. August 2019, vol. 62, no. 8, p. 62–70, [cit. 2023-01-06]. DOI: 10.1145/3338112. Available at: https://discovery.ucl.ac.uk/id/eprint/10084236/1/O%27Hearn%20AAM%20scaling-static-analysis-at-facebook.pdf.

[15] Elder, M. Bourdoncle Components. [online]. Madison, WI, USA: University of Wisconsin-Madison. June 2010, [cit. 2023-01-30]. Available at: https://pages.cs.wisc.edu/~elder/stuff/bourdoncle.pdf.

[16] Farago, D., Merz, F. and Sinz, C. Automatic Heavy-weight Static Analysis Tools for Finding Bugs in Safety-critical Embedded C/C++ Code. *Softwaretechnik-Trends*. Springer. May 2014, vol. 34, no. 3, [cit. 2023-01-29]. Available at: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_34_Heft_3/tav36.pdf.

[17] Harmim, D. *Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer*. Brno, CZ, 2021. Master's Thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/136837.

[18] Huang, R., Monroe, A., Bjørner, N., Halleux, P. de, Moura, L. de et al. *Z3 Guide: Bitvectors* [online]. Microsoft Corporation, 2023 [cit. 2023-01-22]. Available at: https://microsoft.github.io/z3guide/docs/theories/Bitvectors.

[19] Johnson, S. C. *Lint, a C Program Checker*. Bell Laboratories, July 1978 [cit. 2023-01-27]. Available at: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf.

[20] Křena, B. and Vojnar, T. Automated formal analysis and verification: An overview. *International Journal of General Systems*. Taylor & Francis. May 2013, 42:4, p. 335–365. DOI: 10.1080/03081079.2012.757437. ISSN 0308-1079.

[21] Lamport, L. Safety, Liveness, and Fairness. May 2019, [cit. 2022-12-27]. Available at: https://lamport.azurewebsites.net/tla/safety-liveness.pdf.

[22] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. USA: IEEE Computer Society, March 2004, p. 75. CGO '04. DOI: 10.5555/977395.977673. ISBN 0769521029. Available at: https://dl.acm.org/doi/10.5555/977395.977673.

[23] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E. M. and Voronkov, A., ed. *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 348–370. DOI: 10.1007/978-3-642-17511-4_20. ISBN 978-3-642-17511-4. Available at: https://citeseerx.ist.psu.edu/document?doi=6c725d2a7e88515c5f7c877936f90b0184c4fe8f.

[24] Meyer, B. *Soundness and Completeness: With Precision* [online]. Communications of the ACM, April 2019 [cit. 2023-01-16]. Available at: https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext.

[25] Møller, A. and Schwartzbach, M. I. *Static program analysis* [online]. December 2022 [cit. 2023-01-27]. Aarhus University, Department of Computer Science. Available at: https://cs.au.dk/~amoeller/spa/spa.pdf.

[26] Palsberg, J. Type-Based Analysis and Applications. In:. New York, NY, USA: Association for Computing Machinery, 2001, p. 20–27. PASTE '01. DOI: 10.1145/379605.379635. ISBN 1581134134. Available at: https://doi.org/10.1145/379605.379635.

[27] Pavela, O. *Static Analysis Using Facebook Infer Focused on Performance Analysis*. Brno, CZ, 2019. [cit. 2023-02-08]. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/21919/.

[28] Pavela, O., Harmim, D. and Marcin, V. Scalable Static Analysis Using Facebook Infer. In: *Proc. of Excel@FIT'19* [online]. 2019. Available at: http://excel.fit.vutbr.cz/submissions/2019/059/59.pdf.

[29] Rival, X. and Yi, K. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. 1st ed. Cambridge: The MIT Press, February 2020. ISBN 978-0-262-04341-0.

[30] Satoh, S., Kusano, K. and Sato, M. Compiler optimization techniques for OpenMP programs. *Scientific Programming*. IOS press. 2001, vol. 9, 2-3, p. 131–142, [cit. 2023-01-27]. Available at: https://downloads.hindawi.com/journals/sp/2001/189054.pdf.

[31] Schubert, P. D., Hermann, B. and Bodden, E. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2019, p. 393–410. DOI: 10.1007/978-3-030-17465-1_22. ISBN 978-3-030-17465-1. Available at: https://link.springer.com/chapter/10.1007/978-3-030-17465-1_22.

[32] Sinn, M. *Automated Complexity Analysis for Imperative Programs*. Vienna, AU, 2016. [cit. 2023-01-30]. Dissertation. Vienna University of Technology. Available at: https://publik.tuwien.ac.at/files/publik_257756.pdf.

[33] Urban, C. *Static analysis by abstract interpretation of functional temporal properties of programs*. Paris, FR, 2015. [cit. 2022-12-27]. Dissertation. École normale supérieure. Available at: https://theses.hal.science/tel-01176641v2/document.

[34] Zilberstein, N., Mandel, T. and Stern, M. *Unsafe Haskell* [online]. University of Pennsylvania, April 2015 [cit. 2023-01-27]. Available at: https://www.seas.upenn.edu/~cis1940/spring15/lectures/12-unsafe.html.

[35] Çiçek, E. *Cost: Complexity Analysis* [online]. Meta, 2023 [cit. 2023-01-30]. Available at: https://fbinfer.com/docs/next/checker-cost/.

Appendix A

Contents of the included storage media

This appendix lists the contents of the attached memory media. In particular, the attached memory media contains the following:

- `/xpavel34-thesis-2023.pdf`
  - The PDF file of this thesis.
- `/thesis-latex/`
  - LaTeXsource files used to produce the PDF of this thesis.
  - The included `Makefile` can be used to compile the source files into the PDF either with `pdflatex` or `bibtex` via the `make` command.
  - Note that Overleaf was used to compile the attached PDF.
- `/README.md`
  - A `README` file with instructions for installation and basic user manual.
- `/src/`
  - Contains the source code of the entire Meta Infer framework, including the code of Looper.
  - The source files of Looper specifically are located in the `infer/src/looper/` subdirectory but several other Infer files were modified in order to plug-in Looper into the Infer framework.
- `/examples/`
  - Several examples that have been used for testing during development and experimental evaluation.

Appendix B

Installation and User Manual

This appendix contains a brief installation and user manual. More detailed information (including some examples) can be found at Loopers's GitHub pages (`https://github.com/paveon/Looper/wiki/Looper:-A-Worst-Case-Cost-Analyser`) and on the attached memory media (see Appendix A). In the case of missing or corrupted files in the `/src/` directory on the attached memory media, the entire repository can be cloned from GitHub using git via SSH or HTTPS using one of the following commands:

```
git clone git@github.com:paveon/Looper.git --branch develop
    --single-branch --depth 1 --recurse-submodules src
git clone https://github.com/paveon/Looper.git --branch develop
    --single-branch --depth 1 --recurse-submodules src
```

Alternatively, it can be manually download through the GitHub repository page at the following address: `https://github.com/paveon/Looper`. However, it might be necessary to manually switch to the `develop` branch and initialize the submodules when choosing this approach.

Installation Manual

Due to certain limitations, no pre-compiled binary of Infer with Looper is available. Building Looper from the source is thus currently the only option. *Beware that building Looper from the source may be very time-consuming as it involves compilation of a customized clang compiler.*

At first, it is required to install Meta Infer's dependencies and then to compile Meta Infer with Looper. Here are the prerequisites to be able to compile Facebook Infer with Looper on Linux:

- `opam` $\geq$ 2.0.0,
- `pkg-config`,
- Java (only needed for the Java analysis),
- `gcc` $\geq$ 5.X or `clang` $\geq$ 3.4 (only needed for the C/C++ analysis),
- `autoconf` $\geq$ 2.63,
- `automake` $\geq$ 1.11.1,

- `cmake` (only needed for the C/C++ analysis).

The compilation and installation of Meta Infer with Looper can be done using the following commands:

```
cd src
./build-infer.sh
sudo make install
```

The official and up-to-date Meta Infer's installation manual (which also includes instructions for other operating systems) can be found at `https://github.com/facebook/infer/blob/main/INSTALL.md`. Meta Infer with Looper should be now installed system-wide with executable binaries located in `/src/infer/bin/`.

User Manual

This section assumes that Meta Infer with Looper is installed system-wide and executable by the command `infer`.

In general, an analysis of a C/C++ program with Facebook Infer can be done using the following command (for a single source file):

```
infer -- gcc -c source_file.c
```

Java programs can be analyzed using the following command:

```
infer -- javac source_file.java
```

Another option is to analyze the entire project with `Makefile` using the following:

```
infer -- make <target>
```

For advanced usage, see `https://fbinfer.com/docs/infer-workflow`. Many other build systems may be used; see `https://fbinfer.com/docs/analyzing-apps-or-projects`. Looper is deactivated by default but the analysis can be triggered using the following commands:

```
infer capture -- gcc -c source_file.c
infer analyze --looper-only
```

or simply

```
infer run --looper-only -- gcc -c source_file.c
```

For more information, please refer to the official Infer website or the GitHub repository.