BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS USING FACEBOOK INFER FOCUSED ON PERFORMANCE ANALYSIS

STATICKÁ ANALÝZA V NÁSTROJI FACEBOOK INFER ZAMĚŘENÁ NA ANALÝZU VÝKONNOSTI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDŘEJ PAVELA**
AUTOR PRÁCE

SUPERVISOR **Doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

21919

Student:	**Pavela Ondřej**
Program:	Informační technologie
Název:	**Statická analýza v nástroji Facebook Infer zaměřená na analýzu výkonnosti**
	Static Analysis Using Facebook Infer Focused on Performance Analysis
Kategorie:	Analýza a testování softwaru

Zadání:

1. Prostudujte principy statické analýzy založené na abstraktní interpretaci. Zvláštní pozornost věnujte přístupům zaměřeným na analýzu výkonnosti a výkonnostních chyb.
2. Seznamte se s nástrojem Facebook Infer, jeho podporou pro abstraktní interpretaci a s existujícími analyzátory vytvořenými v prostředí Faceboook Infer.
3. Navrhněte a implementujte analyzátor v prostředí Facebook Infer zaměřující se na analýzu výkonnosti.
4. Experimentálně ověřte funkčnost vytvořeného analyzátoru na vhodně zvolených netriviálních programech.
5. Shrňte dosažené výsledky a diskutujte možnosti jejich dalšího rozvoje v budoucnu.

Literatura:

- Moritz Sinn, Florian Zuleger, Helmut Veith: Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints
- Oswaldo Olivo, Isil Dillig, Calvin Lin: Static detection of asymptotic performance bugs in collection traversals
- Oficiální stránky projektu Facebook Infer: http://fbinfer.com/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	15. května 2019
Datum schválení:	1. listopadu 2018

Abstract

Static analysis has nowadays become one of the most popular ways of catching bugs early in the modern software. However, reasonably precise static analysis tools still often struggle to scale well on large and quickly changing codebases. Efficient static analysers, such as Coverity or Code Sonar, are usually proprietary and difficult to openly evaluate or extend. On the contrary, Facebook Infer offers an open source static analysis framework with the emphasis on *compositional*, *incremental* and consequently highly *scalable* inter-procedural analysis. This thesis presents *Looper* — a new performance oriented *resource bounds* analyser which extends the capabilities of Facebook Infer. We have based our implementation on an existing resource bounds analyser *Loopus* and evaluated it on two different test suites, showing encouraging results in comparison with the existing Cost analyser developed by the Infer team.

Abstrakt

Statická analýza se v současnosti dostává do popředí v oblasti technik pro odhalování chyb v moderním software. Nedostatečná škálovatelnost — především v kombinaci se zachováním potřebné přesnosti — je však přetrvávající problém u většiny současných nástrojů pro statickou analýzu, což je činí nepoužitelnými v případě rozsáhlého a často se měnícího kódu. Efektivní statické analyzátory, jako například Coverity nebo Code Sonar, jsou navíc často proprietární a není tedy možné je jednoduše rozšířit nebo srovnávat jejich výsledky. Oproti tomu Facebook Infer nabízí open source rámec s důrazem na *kompoziční*, *inkrementální*, a v důsledku i *škálovatelnou* inter-procedurální statickou analýzu. Tato práce představuje *Looper* — nový analyzátor zaměřující se na analýzu výkonnosti, přesněji na analýzu mezí, rozšiřující rámec nástroje Facebook Infer. Implementace našeho analyzátoru je založena na existujícím nástroji *Loopus*, který se zaměřuje na přesnou analýzu mezí. Výsledný prototyp jsme otestovali na dvou různých testovacích sadách a povzbudivé výsledky srovnali s existujícím analyzátorem Cost, který je vyvíjen Infer týmem.

Keywords

Facebook Infer, static analysis, abstract interpretation, performance analysis, resource bounds analysis, amortized complexity, Loopus, OCaml

Klíčová slova

Facebook Infer, statická analýza, abstraktní interpretace, výkonnostní analýza, analýza mezí, amortizovaná složitost, Loopus, OCaml

Reference

PAVELA, Ondřej. *Static Analysis Using Facebook Infer Focused on Performance Analysis.* Brno, 2019. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Mgr. Adam Rogalewicz, 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.

Pro odhalování těchto chyb v raných fázích vývoje se dnes nejčastěji používají rozsáhlé automatické testy a nástroje pro *dynamickou analýzu* — například profilaci. Přes jejich nespornou užitečnost se úspěšnost automatických testů přímo odvíjí od kvality a počtu manuálně tvořených testovacích případů, přičemž nástroje pro profilaci jsou schopny poskytnout výkonnostní charakteristiky pouze pro konkrétní použitá vstupní data. Výkonnostní chyby se ovšem dle [7] nejčastěji projevují až v pozdějších fázích vývoje, případně až při nasazování finálního produktu, např. kvůli rozdílům mezi očekávanou a skutečnou zátěží. Přístupy založené na dynamické analýze jsou tedy ve výsledku v mnoha případech dostačující, ale stále je zde riziko, že mnoho chyb zůstane neodhalených. Zároveň pak tyto techniky neposkytují záruky o jakýchkoliv vlastnostech analyzovaného programu.

Statická analýza představuje odlišný přístup, který ve většině případů nevyžaduje žádnou dodatečnou manuální obsluhu a může být použit i v počátcích vývoje, jelikož nezávisí na tom, zda je program spustitelný. Nicméně i statická analýza má své problémy, jako například tradiční vysokou míru falešných pozitiv a zejména přetrvávající problém se škálovatelností, který sužuje většinu současných nástrojů a činí je nepoužitelnými v případě rozsáhlého a často se měnícího kódu.

V reakci na tento problém Facebook nedávno představil vlastní nástroj pro efektivní odhalování chyb a verifikaci programů, nazvaný *Facebook Infer* — kompoziční, inkrementální a v důsledku tedy i vysoce škálovatelný rámec pro *abstraktní interpretaci* [15], který je uzpůsoben pro rychlou integraci nových *inter-procedurálních* analyzátorů. Zmíněný nástroj byl již nasazen a aktivně se používá ve společnosti Facebook (a v několika dalších, jako například Spotify, Uber nebo Mozilla), přestože se stále nachází ve fázi rychlého vývoje. V současnosti již disponuje rozmanitou řadou analýz, například pro verifikaci přetečení zásobníku, bezpečnosti u vícevláknových programů nebo úniku zdrojů.

Infer bohužel v současnosti stále zaostává v oblasti výkonnostních chyb. Jediný výkonnostně zaměřený analyzátor Cost (představen Infer týmem v průběhu naší práce) implementuje pouze upravenou verzi tzv. *worst-case execution time* (WCET) analýzy. 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.

Loopus [12] je analyzátor mezí, který je schopen — dle našeho nejlepšího vědomí — jako jediný analyzovat amortizovanou složitost u široké škály programů. Loopus nicméně provádí pouze *intra-procedurální* analýzu a samotný nástroj (bez inkrementálního rámce) není vhodný pro rozsáhlý a rychle se měnící kód. Tato práce proto představuje *Looper* — analyzátor mezí reimplementující Loopus v rámci nástroje Infer, což otevírá nové možnosti pro ještě efektivnější analýzu.

Stěžejním konceptem nástroje Loopus je použití abstraktního modelu nazvaného *difference constraint program* (DCP). Jedná se o graf, jehož hrany jsou popsány pomocí nerovností tvaru $x \leq y + \mathtt{c}$, kde x a y jsou celočíselné výrazy sestavené nad proměnnými programu a $\mathtt{c} \in \mathbb{Z}$ je numerická konstanta. Tyto nerovnosti, nazývané jako *difference constraints* (DC), jsou schopny modelovat velkou část imperativních programů s celočíselnými počítadly cyklů. Loopus je schopen na základě této reprezentace analyzovat meze pomocí vzájemné rekurze dvou procedur $T\mathcal{B}(\tau)$ a $V\mathcal{B}(\mathtt{a})$. Procedura $T\mathcal{B}(\tau)$ slouží k výpočtu horní meze počtu provedení konkrétního DCP přechodu τ sledováním *kolikrát* a *o kolik* se může zvýšit hodnota celočíselného výrazu, který přímo limituje počet souvislých provedení přechodu τ. Procedura $V\mathcal{B}(\mathtt{a})$ aplikuje podobný princip na výpočet horní meze hodnoty konkrétní DCP proměnné $\mathtt{a}$. Vzájemným propojením těchto dvou procedur je možné sestavit algoritmus pro výpočet horních mezí celkového počtu provedení libovolného přechodu τ. V našem případě sledujeme tzv. *zpětné hrany*, které navracejí tok programu zpět k hlavičce cyklu po ukončení iterace. Sečtením horních mezí jednotlivých provedení všech zpětných hran programu získáváme celkovou *cenu*, která přímo reflektuje jeho *asymptotickou složitost*.

Výsledný prototyp jsme otestovali na dvou různých testovacích sadách a s povzbudivými výsledky srovnali s existujícím analyzátorem Cost, který je vyvíjen Infer týmem. Navazující práce se bude zaměřovat zejména na rozšíření původního *intra-procedurálního* konceptu směrem ke škálovatelné *inter-procedurální* analýze při zachování rozumné přesnosti. Dalším cílem je implementace zbývajících rozšíření prezentovaných v [12], či návrh systému, který by byl schopen monitorovat změny výkonnostních charakteristik jednotlivých funkcí mezi různými revizemi programu.

Static Analysis Using Facebook Infer Focused on Performance Analysis

Declaration

I declare that I have prepared this Bachelor's thesis independently, under the supervision of Doc. Mgr. Adam Rogalewicz, Ph.D.. Ing. Tomáš Fiedor and Prof. Ing. Tomáš Vojnar, Ph.D. provided me with further information. I listed all of the literary sources and publications that I have used.

....
Ondřej Pavela
May 15, 2019

Acknowledgements

I would like to thank Doc. Mgr. Adam Rogalewicz, Ph.D. and Prof. Ing. Tomáš Vojnar, Ph.D. for their supervision, consultations, and their expert advice over the course of this work. A special thanks goes to Ing. Tomáš Fiedor for his expert advice, helpful consultations regarding the Loopus tool, and, especially, for his invaluable writing advice and immense help with proofreading of this work. Further, I would like to thank Sam Blackshear and Nikos Gorogiannis from Infer team at Facebook for helpful discussions regarding the development of this analyser. And finally, last but not least, a special gratitude goes to my family, 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 a specific input data only. Unfortunately, according to [7], performance bugs tend to manifest in later development stages or upon deployment due to previously unanticipated workload. In conclusion, approaches based on dynamic analysis are sufficient in many cases but can sometimes still miss too many 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 traditional high rate of *false positives*, 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.

In response, Facebook has recently proposed its own solution for efficient bug finding and program verification called *Facebook Infer* — a *compositional*, *incremental* [9], and consequently highly *scalable abstract interpretation* [15] framework suitable for quick integration of new *inter-procedural* analyses. Although still rapidly developing, it is already deployed in Facebook (and several other companies, such as Spotify, Uber, or Mozilla) and offers a wide range of analyses, e.g., for verification of buffer overflow, thread safety, or resource leakage.

Unfortunately, Infer currently lacks analyses focused on the mentioned *performance-based* bugs. The only performance-based Cost analyser (introduced by Infer in the course of our work) implements a modified *worst-case execution time* (*WCET*) analysis only. However, this analysis provides a numerical bound on the time required for the execution of a program only, which can be hard to interpret, and, above all, it is quite imprecise for more complex algorithms, e.g., requiring *amortized* reasoning.

Loopus [12] is a powerful resource bounds analyser, which, to the best of our knowledge, is the only one that can handle the *amortized complexity* for a broad range of programs. However, it is limited to *intra-procedural* analysis only, and the tool itself (without an incremental framework) is not suitable for large and quickly changing codebases. Hence, in this work we propose *Looper* — a resource bounds analyser that recasts the powerful Loopus within Infer, enabling the possibility for a more efficient analysis. The experimental evaluation shows encouraging results, when even our immature implementation inferred precise bounds for selected benchmarks.

This work was supported by H2020 ECSEL project Aquas; we hereby thank for the received support.

Structure of this work. The rest of this work is structured as follows: **Chapter** 2 introduces the theory of abstract interpretation technique in static analysis and demonstrates its key concepts on a simple example. **Chapter** 3 provides a general overview of the Facebook Infer tool with an emphasis on how Infer achieves the scalability with its compositional inter-procedural analysis. **Chapter** 4 introduces the existing Loopus tool and provides a high level overview of the key concepts and ideas without any formal definitions or proofs of soundness. The follow-up **Chapter** 5 discusses the current state of implementation which is based on the key concepts from previous chapter. Further, **Chapter** 6 presents the encouraging results of our experimental evaluation conducted on two different test suites. Finally, **Chapter** 7 discusses the possibilities for future work and concludes this thesis.

Chapter 2

Abstract Interpretation

Abstract interpretation (AI) was originally formalized by a French computer scientist Patrick Cousot and his wife Radhia Cousot in the 1970s [4]. The theory of AI 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 the AI technique falls into the domain of static analysis of programs it actually executes instructions of analyzed program in a sense. However, the key difference is how we interpret those executed instructions. We basically assign abstract semantics over an abstract domain to each concrete instruction and tailor the domain for the specific needs of our analysis based on 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.

The Facebook Infer tool provides a scalable framework for static analysis based on AI (Infer.AI). The scalability is achieved by following the principles of compositionality allowing Infer to perform incremental analysis which can be run on individual code changes.

2.1 Components of the Abstract Interpretation

Each new type of analysis has to define few essential components required by the general abstract interpretation framework. These components create the semantics of our analysis:

- **Abstract domain**: a set of abstract states. An abstract state represents a program state at a certain program location. The content of an abstract state varies depending on a specific type of analysis. A trivial example might be an interval domain tracking safe lower and upper bounds of integer program variables, i.e., $[a, b]$ 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 on an abstract state. For example, we would need to transform the integer arithmetic of a concrete program to the interval arithmetic applicable in our 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**: combines multiple abstract states into a new one. Joins are used at program junctions where several program branches merge together. Join in the interval domain can be defined as: $[a, b] \circ [c, d] = [\min(a, c), \max(b, d)]$.

- **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: $[a_0, b_0] \nabla [a_1, b_1] = [\texttt{if } a_1 < a_0 \texttt{ then } -\infty \texttt{ else } a_0, \texttt{ if } b_1 > b_0 \texttt{ then } \infty \texttt{ else } b_0]$, where both intervals collide at the same loop header after two consecutive iterations of a loop.

- **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: $[a_0, b_0] \vartriangle [a_1, b_1] = [\texttt{if } a_0 = -\infty \texttt{ then } a_1 \texttt{ else } a_0, \texttt{ if } b_0 = \infty \texttt{ then } b_1 \texttt{ else } b_0]$.

2.2 Formal Definition

The abstract interpretation is based on notion of a mathematical structure called *semilattice* [14], which is defined as follows:

- Partially ordered set $(A, \leq_A)$ is a semilattice if each non-empty, finite subset B of A has a least upper bound in A.

- Partially ordered set $(A, \leq_A)$ is a complete semilattice if each subset B of A has a least upper bound in A.

The abstract interpretation I of a program P with the instruction set `Instr` is a tuple:

$$I = (Q, \circ, \sqsubseteq, \top, \bot, \tau),$$

where

- Q is a abstract domain, i.e., domain of abstract states,

- $\top \in Q$ is the least upper bound of Q,

- $\bot \in Q$ is the greatest lower bound of Q,

- $\circ : Q \times Q \to Q$ is the binary join operator,

- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \circ y = y$ in $(Q, \circ, \top)$ which is a complete semilattice.

- $\nabla : Q \times Q \to Q$ is the binary widening operator with following properties:

  - $\forall C, D \in Q : (C \circ D) \sqsubseteq (C \nabla D)$,
  - For all infinite sequences $(C_0, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \ldots, s_n, \ldots)$ defined recursively as:

$$s_0 = C_0, \qquad s_n = s_{n-1} \nabla C_n$$

    is not strictly increasing,

- $\vartriangle$: $Q \times Q \to Q$ is the binary narrowing operator with following properties:
  - $\forall C, D \in Q : D \sqsubseteq C \Rightarrow (D \sqsubseteq (C \vartriangle D) \sqsubseteq C)$,
  - For all infinite sequences $(C_0, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \ldots, s_n, \ldots)$ defined recursively as:

$$s_0 = C_0, \qquad s_n = s_{n-1} \vartriangle C_n$$

    is not strictly decreasing.

2.3 Fixpoint Approximation

Abstract interpretation is based on the principle of program execution in the abstract domain which substantially reduces the number of possible program states. However, the underlying theory is based on the notion of *lattices* and computing of *fixpoints*, i.e., we gradually apply abstract transformers on states, until we do not produce any new ones. Fixpoint is an element $a \in A$ of a function $f :\to A$ over a semilattice $(A, \leq_A)$ with a function value equal to a itself, e.g., $f(a) = a$ holds.

Computing of precise fixpoints is generally not guaranteed to terminate in acceptable or even finite time. Those cases mostly comprise of loops where it is usually impossible to avoid some kind of fixpoint approximation in order to guarantee the termination. Widening provides solution to this problem as it can be used to over-approximate the fixpoint and thus guarantee the termination in finite time at the expense of precision. It can also be followed by a subsequent narrowing operation which might provide more precise approximation. Both operations are optional and general abstract interpretation framework does not require their presence but at least widening is convenient if we aim for efficient analysis focused on real world code. The following example with a loop demonstrates all introduced concepts:

```
$l_1$ : x = 0;
$l_2$ : while (x < n) {
$l_3$ :     x++;
$l_4$ : }
```

l_1: $x_1 = [0, 0]$	l_3: $x_2 = [1, 1]$	l_2: $x_3 = x_1 \circ x_2 = [0, 1]$
l_3: $x_4 = [1, 2]$	l_2: $x_5 = x_1 \circ x_4 = [0, 2]$	l_2: $x_6 = x_1 \nabla x_5 = [0, \infty)$
l_3: $x_7 = [1, \infty)$	l_2: $x_8 = x_1 \circ x_7 = [0, \infty)$	l_2: $x_6 = x_8 = FP$
l_4: $x_{end} = x_8 \vartriangle x_{false} = [0, \infty] \vartriangle (-\infty, n] = [0, n]$		

Table 2.1: A calculation of the $[0, \infty)$ fixpoint requiring the *widening* operation and subsequent refinement with the *narrowing* operation. Variable x_i corresponds to the ith step of the fixpoint computation. Each step is listed with location of instruction which was abstractly executed.

Chapter 3

Facebook Infer

Infer is an open-source static analysis framework developed by the *Facebook 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 [3] 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 matured analyses includes for example *Inferbo* (buffer overrun checker), *RacerD* (data races) or *Starvation* (concurrency starvation and some types of deadlocks).

3.1 Infer.AI

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 analysers 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.

3.1.1 Framework Architecture

Figure 3.1 presents a simplified block diagram of the framework architecture consisting of three main components. The first main component is the frontend. Its job is to leverage the underlying *LLVM* compiler infrastructure to compile analysed 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.

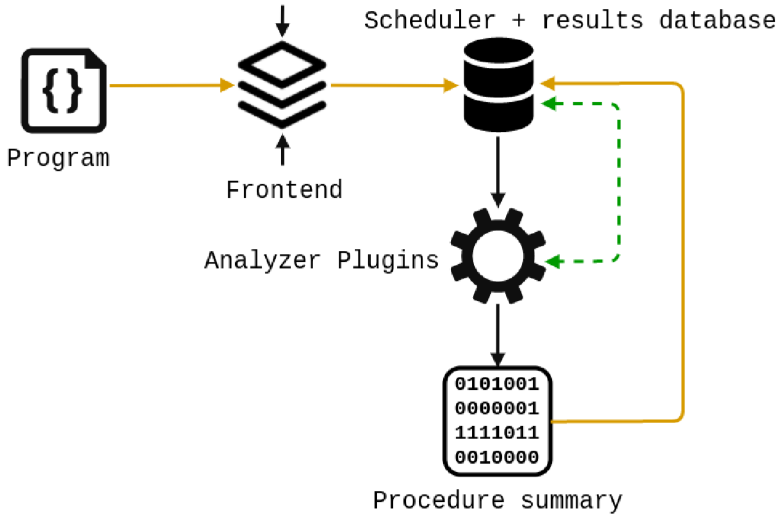


Figure 3.1: Architecture components

Frontend provides an output in form of a Control Flow Graph (CFG) for each analysed 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 3.3. 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.

3.2 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.

Figure 3.2 describes the process of the intraprocedural analysis in Infer. The 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

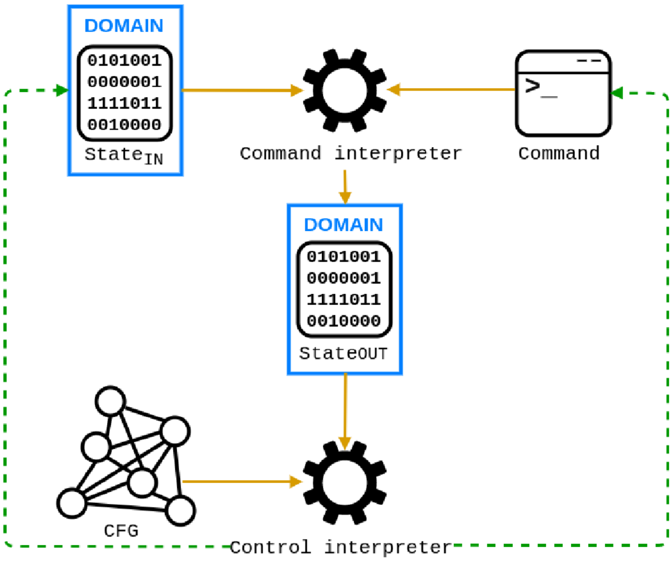


Figure 3.2: The process of intraprocedural analysis in Infer. The command interpreter applies transfer functions to input abstract states and produces output states. The control interpreter chooses the inputs based on a CFG.

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.

3.3 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 3.3.

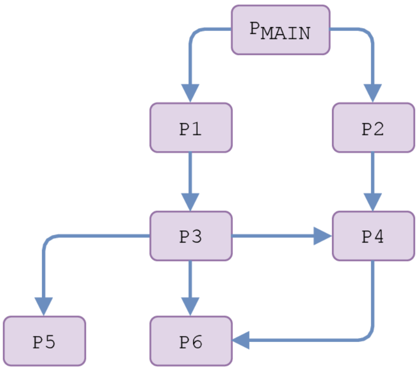


Figure 3.3: A call graph describing call dependencies of each procedure represented by a node. Outgoing edge signifies the possibility of a call to other procedure.

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.

3.4 Cost analyser

Over the course of our work, Infer introduced a new performance oriented analyser called COST — a *worst-case execution cost* analyser based on ideas of parametric *worst-case execution time* (WCET) analysis presented in [2]. The basic idea of the original tool is to generate a so called *execution count function* (ECF$_P$) and a so called *parametric calculation function* (PCF$_P$) for a program P and obtain the final *parametric* WCET (PWCET$_P$) function as a functional composition of ECF$_P$ and PCF$_P$:

$$\mathrm{PWCET}_P : \mathbb{Z}^{|\mathcal{I}_P|} \longrightarrow \mathbb{N} = \mathrm{PCF}_P \circ \mathrm{ECF}_P \,. \tag{3.1}$$

This composite function takes an integer vector of instantiated input parameters of a program P ($\mathbb{Z}^{|\mathcal{I}_P|}$) and returns a single natural number ($\mathbb{N}$) representing the final WCET. This approach takes advantage of the fact that it is often possible to obtain *tighter* upper bounds on the WCET if the values of program input variables are known beforehand which is often the case when deploying software for embedded systems.

3.4.1 Execution Count Function

The purpose of Execution Count Function is to calculate an upper bound on the number of possible executions for each program point $q \in \mathcal{Q}_P$ where $\mathcal{Q}_P$ is a set of all points in a program P. It is defined as follows:

$$\mathrm{ECF}_P : \mathbb{Z}^{|\mathcal{I}_P|} \longrightarrow \mathbb{N}^{|\mathcal{Q}_P|} \,. \tag{3.2}$$

It takes the same input as the composite PWCET$_P$ function and returns a vector of upper bounds ($\mathbb{N}^{|\mathcal{Q}_P|}$) on the number of executions for each program point. Note that the notion of a program point is not formally defined as it depends on the underlying program model, and thus it can for example be a node or an edge of a CFG. The original tool leverages abstract interpretation and generates the ECF$_P$ based on flow constraints inferred via *flow analysis* and *counting the elements in the abstract state*.

The idea of counting the elements is used to find *loop bounds* and is based on the following observation: "the number of environments associated with a program point is an upper bound of the number times the program point can be visited in any execution" [2]. An environment maps tracked program variables to specific integer values, e.g., $[n \mapsto 2, i \mapsto 0]$ is an environment which maps the variable n to 2 and the variable i to 0. Thus, if we count all associated environments, i.e., all possible combinations of assignments to tracked variables, we get an upper bound on the number of times a program point can be visited.

However, the number of environments grows with the number of tracked variables (number of possible combinations increases) and it is thus desirable to track only relevant variables if we aim for a reasonably precise analysis. The original tool tracks only so called *control variables* which directly or indirectly affect the control flow of a program, i.e., variables which affect the values of conditional expressions in CFG control nodes (similar to the notion of *loop counter* variables that will be introduced in Section 4.2). Additionally, to further reduce the set of tracked variables, an invariant analysis is employed in order to find and remove variables that are invariant in the body of a specific loop.

Finally, the abstract interpretation is used to construct countable environments for each program point. The basic idea is to perform a *value analysis* for each tracked variable with an abstract domain of choice and subsequently count all discrete points in the concrete domain. E.g., with an interval domain the counting is trivial as the total number of elements in the interval $[a, b]$ is computed as:

$$|[a,b]| = |\gamma([a,b])| = |\{n \in \mathbb{Z} | a \leq n \leq b\}| = b - a + 1\,, \tag{3.3}$$

where γ is the concretization function which maps abstract interval to concrete values it represents. Computation for empty and infinite intervals is trivial and the results are 0 and ∞, respectively. However, depending on the chosen abstract domain, the counting process might be quite difficult, especially for relational domains (such as polyhedral) which are useful as it is often desirable to express the range of possible values symbolically in terms of input parameters of a program. This way, we can count the number of associated environments for a single tracked variable and the total number of environments $|\sigma|$ for all variables combined can be computed as follows:

$$|\sigma| = \prod_{v \in \mathcal{V}} |\sigma(v)|\,, \tag{3.4}$$

where $|\sigma(v)|$ is the calculation for a single variable v from the set of tracked variables $\mathcal{V}$. The result is the desired upper bound on the number of executions of a single program point. Please refer to [2] for more details.

The Cost analyser leverages the existing Inferbo analyser and its relational symbolic interval domain [8] to perform the value analysis, count the elements, and obtain symbolic upper bounds in terms of function parameters for each program point of a function. These symbolic upper bounds can be used to calculate the final numerical execution bound when a set of initial values for the input parameters is given. Additionally, the Cost analyser assigns the bound of one to all program points with empty set of tracked variables, i.e., program points outside of loops.

3.4.2 Parametric Calculation Function

The purpose of this function is to transform a vector of upper bounds ($\mathbb{N}^{|\mathcal{Q}_P|}$) calculated by the ECF$_P$ function into a single number ($\mathbb{N}$) representing the final WCET. It is defined as follows:

$$\text{PCF}_P : \mathbb{N}^{|\mathcal{Q}_P|} \longrightarrow \mathbb{N}\,. \tag{3.5}$$

The original tool uses parametric calculation and the *Implicit Path Enumeration Technique* [11] (IPET) to obtain the PCF$_P$ function for a program P. The basic idea of IPET is to obtain an estimate of the WCET by maximising the following objective function:

$$\sum_{q \in \mathcal{Q}_P} c_q x_q\,, \tag{3.6}$$

where q is a program point, c_q is an atomic worst-case cost for a program point q obtained through a low-level analysis. The x_q represent an upper bound on the executions of a program point q which is unknown but subject to a set of flow constraints obtained through the flow analysis and also additional symbolic upper bound constraints calculated by the ECF_P function. I.e., it formulates the problem of finding the WCET as an *Integer Linear Programming* (ILP) problem of maximising the objective function given a set of constraints. This problem can be solved by any ILP solver which finds a solution for each unknown variable x_q. Additionally, to get a bounded problem which can be solved, *structural constraints* have to be introduced. E.g., the upper bounds x_i and x_e for the initial and the exit node are equal to one or the upper bound x_j for a join node is equal to the sum of upper bounds of both incoming edges x_q and x_p, i.e., $x_j = x_q + x_p$. Note that the atomic cost c_q for a program point q can refer to clock cycles, milliseconds, or any other measurement unit provided by the low-level analysis. Please refer to [2] for more details.

The Cost analyser adopts similar approach and estimates the worst-case execution cost with Equation 3.6 where the c_q factor refers to statically assigned atomic cost of Infer `SIL` instruction, e.g., the `LOAD` instruction has atomic cost of one. However, it does not employ an ILP solver and instead constructs equations based on generated structural constraints and the symbolic upper bounds from the ECF_P function. It applies various heuristics on these equations to subsequently pick the tightest upper bound from a set of possible bounds for each program point.

Chapter 4

Loopus

Loopus [12] is a scalable tool for automated complexity and resource bound analysis of integer programs. It was implemented by Moritz Sinn from Forsyte team in TU Wien.

Loopus mainly focuses on so called *tight upper bounds* on the *worst-case cost*. It uses the *back-edge metric* — an *uniform* cost model that assigns the cost of one to each *back jump instruction* (i.e., the instruction that is executed at the end of each loop iteration, causing the return of control flow to the loop header) and the cost of zero to all others. The back-edge metric is especially interesting in relation with the *asymptotic time complexity* where it specifies how many times can a specific instruction be executed during a program run. Moreover, the back-edge metric gives us the *cost* of executing *concrete* implementations contrary to general asymptotic complexity.

Loopus is built upon the LLVM intermediate representation. The input C program is first parsed into a so called *labeled transition system* (LTS) — an initial representation of the program. LTS is further transformed into a *difference constraint program* (DCP), which models difference constraints between program parameters and variables. Finally, Loopus performs intraprocedural analysis and computes symbolic polynomial bounds over program parameters for loops and consequently the complexity of non-recursive functions. Because the intraprocedural analysis is quite restrictive and could lead to imprecise results in most cases, Loopus uses function inlining. More specifically, it inlines calls to all non-recursive functions that do not contain loops. This is crucial for precision because even simple functions without loops might still modify loop counter variables through pointers or return values and thus affect the total number of loop iterations.

Loopus has certain limitations [12], but authors claim that most of them are due to technical reasons rather than due to general limitations of the adopted approach. Loopus also currently supports experimental heuristics that allow it to handle some cases of non-integer code including typical loop iteration patterns over recursive data structures. Moreover, recently a sound technique [5] was proposed and implemented in the Ranger tool [6], which transforms input heap-manipulating programs into integer representation and uses Loopus as a backend.

In this chapter we will provide a high level overview of the approach used by Loopus as well as the process of program abstraction or bound calculation. This chapter is based on [12, 6, 5].

4.1 Program Representation

The core concept behind the program abstraction in Loopus is the use of so called *difference constraints*: a natural abstraction for typical operations with counters in imperative programs. Difference constraints have been previously used for termination analysis [1] which is, to some degree, related to bound analysis and Loopus has extended their potential in this field.

Difference constraint (DC) is a relational inequality of the form $x' \leq y + \mathtt{c}$ where x and y are expressions over program variables and $\mathtt{c} \in \mathbb{Z}$ is a constant. DCs are expressive enough to be able to model large portion of real world imperative integer programs, in particular their complexity aspects. Especially convenient is their ability to model problems related to *amortized complexity* which is the main reason Loopus is able to obtain tighter upper bounds more often than most of the existing tools. Real world examples that demonstrate the need of amortized complexity analysis can be found, e.g., in parsing and string matching procedures [13].

The $x' \leq y + \mathtt{c}$ inequality can model counter *increments*, *decrements* and *resets*. The main advantage of this representation is the fact that it is easy to distinguish between increments and resets with just a syntactic check. For example if $x = y$ then we get $x' \leq x + \mathtt{c}$ which is clearly an increment for $\mathtt{c} > 0$ and decrement for $\mathtt{c} < 0$. Otherwise if $x \neq y$ then we get the standard inequality $x' \leq y + \mathtt{c}$ where $\mathtt{c}$ can be zero which leads to the simplest case of reset $x' \leq y$. Loopus takes advantage of this in its bound calculation algorithm and distinguishes between counter increments and resets to achieve better precision.

The semantics of DCs is that the value of x in the current state cannot be greater than the value of y from the previous state possibly increased by some constant value $\mathtt{c}$. So $x' \leq x + 1$ inequality can be interpreted in the following way: the value of variable x in the current state cannot be greater than the value of the same variable from previous state plus one. In other words, the value of x in the next program state will not increase by more than one compared to the previous state.

DCP is then an abstract program model which is represented by a directed labeled graph with transitions denoted by sets of DCs. The bound algorithm uses this model to calculate the back jump cost of programs.

4.1.1 Labeled Transition Systems

The frontend of Loopus represents programs by labeled transition systems (LTS) that are defined as tuples of set of program locations, set of transitions, single entry location and single exit location. LTS is an oriented graph with program locations as nodes and transitions as edges where each edge is labeled by a formula that encodes a transition relation specified by each transition in a program. Figure 4.1 shows an example of simple integer program on the left with its corresponding LTS representation on the right. Each branching point is regarded as a program location which means we get two program locations, one for each while loop. LTS contains one edge for each possible program path, where the semantics of edge formulas is straightforward. For example, the formula $i > 0 \wedge i' = i - 1 \wedge j' = j + 1$ labeling the edge τ_1 in Figure 4.1 contains one condition $i > 0$ and two assignments: $i' = i - 1$, $j' = j + 1$. Conditions restrict the possibility of transition execution and assignments specify values of assigned variables *after* the execution. Assignments such as $i' = i$ represent that the value of variable i was not changed on a transition. The asterisk symbol used in

```
lb : void tarjan(int n) {
       int i = n; (processed elements)
       int j = 0; (current stack size)
l1 :   while (i > 0) {
           i--;
           j++; (push)
l2 :       while (j > 0 && *)
               j--; (pop)
le : }
```

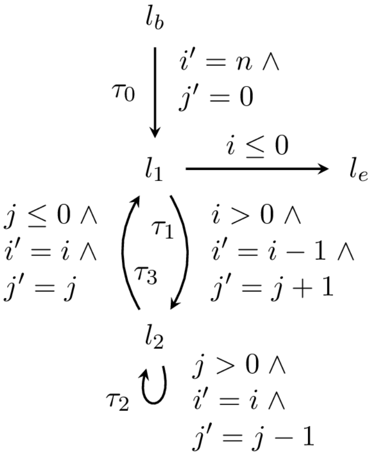


Figure 4.1: Example `tarjan` [12] models a stack which processes the total number of n elements. I.e., there are n pushes and possibly n pops due to non-determinism. Corresponding LTS representation is on the right.

program conditions denotes the non-determinism that models conditions not supported in the analysis. Consequently, these conditions are not part of edge formulas.

4.1.2 Difference Constraint Programs

DCP is an abstract program model that uses difference constraints instead of concrete assignments and conditions to specify its transition relations. It can be represented by an oriented graph, where its edge formulas consist solely of difference constraints. Example of a DCP obtained from the previous LTS graph can be seen in Figure 4.2. We will discuss the abstraction algorithm for transformation of a LTS into a DCP in the next chapter. Each edge is labeled by a set of DCs of form $[x]' \leq [y] + \mathtt{c}$ where $[\cdot]$ denotes a $max(\cdot, 0)$ function. Resource bounds algorithms are mostly based on finding so called *numerical measures* (norms): an *integer* valued expressions over the program state. In this case,

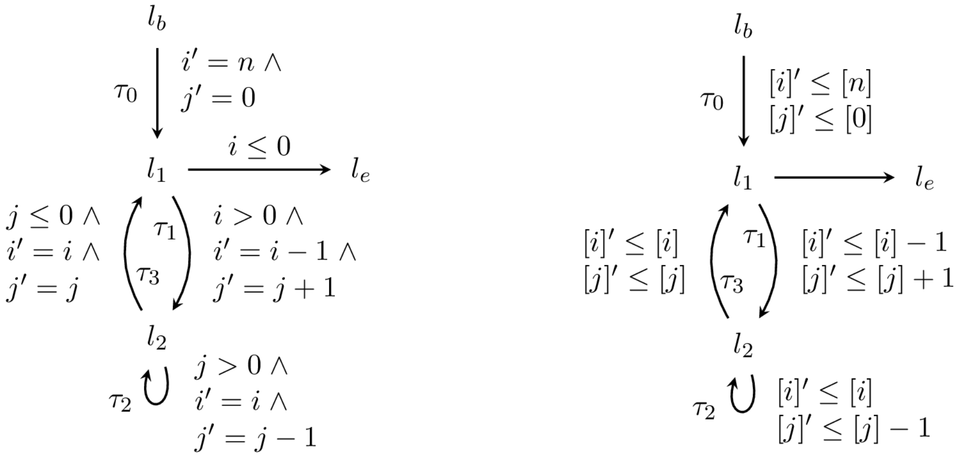


Figure 4.2: Comparison of the LTS graph and its DCP abstraction obtained from the previous `tarjan` example.

elements x and y are norms and the maximum function restricts their values to the *natural* numbers. But there are other restrictions for valid DCPs regarding sets of DCs on edges. First, there must not be any edges containing more than one DC with the same norm on the left-hand side. Second, norms that are purely built over program parameters at the left-hand side of DCs are forbidden. For example the DC $[n]' \leq [i] - 1$ is forbidden as it has norm n consisting solely of program parameters at the left-hand side. On the other hand, even though norm $(i - n)$ contains the formal parameter n, it is not purely built over formal parameters, thus it can be used as left-hand side. The intuition behind the first restriction is obvious: we cannot assign multiple values to one norm on a single edge. The second one stems from the fact that program parameters are regarded as constants, and thus all norms built purely over parameters are constant as well, i.e., it is not possible to assign a new value to a constant.

4.2 Abstraction Algorithm

The abstraction algorithm used in Loopus is a two step procedure that converts a program represented by a LTS to a DCP. The algorithm first converts a program to a so called *guarded* DCP over integers and then abstracts the guarded DCP to a DCP over *natural* numbers. The first step utilizes the concept of *guards* to extend the natural numbers domain of standard DCPs to the non-well founded domain of integers. The notion of guards is related to transitions and similar to program conditions as they also limit the possibility of transition execution. Transitions in a guarded DCP can only be executed if values of all associated guards are greater than zero. In other words, guards are simply norms that must have value greater than zero on execution of transitions that are guarded by them. For example the norm i would be the only guard of transition τ_1 from Figure 4.2 because of the condition $i > 0$ of the outer while loop.

4.2.1 Abstraction to Guarded DCP

This abstraction process consists of three steps. First, we heuristically construct the initial set of tracked norms, then we abstract transitions, and finally infer guards for each transition.

1. **Initial norm selection**: The idea is based on so called *loop counters*, i.e., the variables that are incremented or decremented inside the loop. We search for all loop headers and branching locations on loop paths and extract conditions of form $a > b$ or $a \geq b$ that involve loop counter variables. These extracted conditions are then converted to integer expressions, i.e. norms, in the following manner: conditions of form $a > b$ are transformed to $a - b$ and conditions $a \geq b$ are transformed to $a - b + 1$.

2. **Abstracting Transitions**: All transitions initially start with an empty set of DCs which is extended as follows: for each norm e from the set of norms N and each transitions τ we check if associated DC set already contains a DC with e at the left-hand side and derive a new DC if not. Note, that we try to derive new DC only if all variables used in the norm e are defined on the according edge. In order to derive new DC we symbolically execute τ and observe how the value of the norm e changes. So for example, given norm $e = (len - i)$ and τ with the assignment $i = end$ we would infer norm $(len - end)$ as well as new DC $(len - i)' \leq (len - end)$.

We try to avoid generating new norms as much as possible because every new norm is added to the set of norms N and the abstraction cycle has to repeat until the set stabilizes. For this reason, we first check if it is possible to derive new DC from tracked norms and generate new ones only if we fail. For example, given the same norm as in the previous example and an assignment $i = i + 1$, we can use the same norm to generate DC $(len - i)' \leq (len - i) - 1$.

3. **Inferring Guards**: Each transition initially has an empty set of guards and we iterate over each norm e from the final set of tracked norms N and each transition τ to determine if e is a *guard* of transition τ. Loopus uses the Z3 SMT solver to prove that the value of e must be greater than zero in order to be able to execute transition τ. For example, if τ is guarded by the condition $len > i$, then it can easily prove validity of formula $len > i \implies (len - i) > 0$ which means norm $(len - i)$ is a guard of τ.

The precision of abstraction can be further improved by means of so called *guard propagation*. If all incoming edges of a location l share common subset of guards, then we can propagate this subset to all outgoing edges from location l, provided that none of these guards are decreased on any of the incoming edges. An example of a guarded DCP can be seen in Figure 4.3.

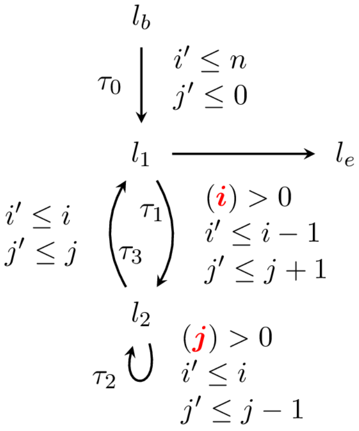


Figure 4.3: A guarded DCP obtained after the first abstraction step. Norms i and j are guards of the respective transitions τ_1 and τ_2.

4.2.2 Abstraction to DCP over Natural Numbers

The final abstraction step is straightforward. We simply remove the guards and use the previously introduced *max* function to ensure the natural number valuation range of all norms. We also have to modify the constant parts of DCs to make sure that they remain invariant over natural numbers.

Every DC $e_1' \leq e_2 + \mathtt{c}$ with norms e_1 and e_2 is transformed based on the value of $\mathtt{c}$ depending on if e_2 is a guard or not. For $\mathtt{c} \geq 0$ we infer $[e_1]' \leq [e_2] + \mathtt{c}$ which is guaranteed to *remain* invariant over natural numbers. For negative values of $\mathtt{c}$ we first check if e_2 is a guard and infer $[e_1]' \leq [e_2] - 1$ if it is. In other case we infer $[e_1]' \leq [e_2] + 0$. The reasoning is simple: if e_2 is a guard, i.e., $e_2 > 0$ before execution of a transition and $\mathtt{c} < 0$ then

$[e_1]' \leq [e_2] - 1$ remains invariant over natural numbers. However, if e_2 is not a guard, then the inequality $[e_1]' \leq [e_2] + \mathtt{c}$ does not hold and is not invariant for $[e_2] = 0$ and $\mathtt{c} < 0$.

4.3 Bound Algorithm

In this section we first present the basic version of the bound algorithm and subsequently extend it with optimizations and heuristics, eventually obtaining the version used in Loopus.

4.3.1 Local Bounds

The notion of local bounds is one of the core concepts underlying the bound calculation. Local bound is a norm e that limits the number of executions of some transition τ as long as some other transitions that might increase the value of e are not executed. For example, in function *tarjan* in Figure 4.3, we can conclude that the norm j limits the number of consecutive executions of transition τ_2. I.e., it does not limit the *total* number of iterations of the inner while loop. Thus, we call it a *local* bound instead of (total) bound as it does not take into account that the value of norm j might increase on some other transitions.

The algorithm that determines the local bound for each transition uses the concept of *strongly connected components* (SCC):

1. First, we compute SCC for given DCP and eliminate transitions that do not belong to any SCC, i.e., transitions that are not part of any cycle. Obviously, these can be executed only once and hence the local bound is equal to 1.

2. We construct the set $\xi(\mathtt{v})$ of transitions for each norm $\mathtt{v}$ that is not purely built over constants or program parameters. We search for transitions that decrease the value of norm $\mathtt{v}$, i.e., transitions that contain DC of form $\mathtt{v} \leq \mathtt{v} + \mathtt{c}$, where $\mathtt{c} < 0$. Then, each transition from set $\xi(\mathtt{v})$ is assigned a local bound of $\mathtt{v}$.

3. The last step is performed only for remaining transitions without assigned local bound. For each $\xi(\mathtt{v})$ we try to remove one of its transitions from the original DCP graph, recalculate SCCs and check if there are some transitions without a local bound that are no longer part of any SCC. Such transitions are then assigned with a local bound of $\mathtt{v}$ and we repeat the process with remaining sets $\xi(\mathtt{v})$ until all transitions have either local bound or bound assigned.

   There is also a possibility that two different sets $\xi(\mathtt{v}_1)$ and $\xi(\mathtt{v}_2)$ will share some transitions that will consequently have more than one possible local bound. In those cases we can either choose one local bound non-deterministically or perform separate bound calculation for each possible local bound and choose the one that leads to the minimal (most precise) overall bound.

4.3.2 The Basic Algorithm

In this section we present the basic bound algorithm, where we limit ourselves to a syntactic subclass of DCPs that allows use of standard DCs only on the single initial transition and use of *monotone* DCs of form $x' \leq x + \mathtt{c}$ everywhere else. In other words, we allow resets of norms only on the single initial transition and all subsequent updates must be either increments or decrements.

The core idea behind this algorithm is to reason *how often* and *by how much* might the local bound of a single transition increase during program execution. Loopus describes the result as a *transition bound* (TB), i.e., the total bound on the number of times a certain transition might be executed. The total execution cost of a program is then equal to the sum of TBs of all its back-edges. The transition bound $\mathcal{TB}(\tau)$ of a transition τ is defined as follows,

$$\mathcal{TB}(\tau) = \begin{cases} \tau_v, & \text{if } \tau_v \notin \mathcal{V} \\ \\ \texttt{IncrementSum}(\tau_v) + \texttt{ResetSum}(\tau_v), & \text{else} \end{cases} \tag{4.1}$$

where τ_v is the local bound of the transition τ. The first case returns the local bound itself if it is a constant. In the second case, the $\texttt{IncrementSum}(\tau_v)$ procedure captures *how often* and *by how much* might the local bound be increased. It is defined as follows:

$$\texttt{IncrementSum}(\tau_v) = \sum_{(\texttt{t},\texttt{c}) \in \mathcal{I}(\tau_v)} \mathcal{TB}(\texttt{t}) \times \texttt{c} \tag{4.2}$$

The $\mathcal{I}(\tau_v)$ is a set of transitions which increase the value of local bound τ_v by the constant $\texttt{c}$, i.e., those that contain a DC of form $\tau_v \leq \tau_v + \texttt{c}$, where $\texttt{c} > 0$. The `IncrementSum` procedure iterates over all these transitions, recursively calls the $\mathcal{TB}$ procedure, and multiplies the computed transition bound by the constant `c`. The resulting sum gives us the total amount by which might the local bound τ_v increase or 0 if $\mathcal{I}(\tau_v)$ is empty.

However, to get a precise bound we need to incorporate the initial value of the local bound τ_v into the equation. The `ResetSum` procedure is defined as:

$$\texttt{ResetSum}(\tau_v) = \sum_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(\tau_v)} \max(\texttt{a} + \texttt{c}, 0) \tag{4.3}$$

Similarly to the `IncrementSum`, the set $\mathcal{R}(\tau_v)$ contains transitions `t` that reset the value of local bound τ_v to `a` with constant increment or decrement `c`, i.e., transitions that contain a DC of form $\tau_v \leq \texttt{a} + \texttt{c}$. Note that in the basic version the only transitions that can reset the values of local bounds are the initial ones. Thus, the $\mathcal{R}(\tau_v)$ sets might only contain the initial transition of a program because it is the only transition that can contain non-monotone DCs. I.e., it can contain up to one $max(\texttt{a} + \texttt{c}, 0)$ element which represents the initial value of the local bound τ_v.

The combination of `IncrementSum` and `ResetSum` procedures gives us the final $\mathcal{TB}(\tau)$ procedure. However, the obtained transition bound is precise only if we assume that all counter decrements in the concrete program are by 1 as the DCP abstraction algorithm without extensions does not model arbitrary decrements. The only thing left is to apply this procedure to all back-edges of a program to obtain the final worst-case cost.

4.3.3 Extending the Procedure with Constant Resets

We can improve the basic procedure to support DCPs with constant resets. In this case, we allow resets of local bounds anywhere and not only on the initial edge. However, the right-hand side norm of a reset DC has to be purely built over *symbolic constants* (program parameters), hence *constant* resets. For example, the norm e_2 of $[e_1]' \leq [e_2] + \texttt{c}$ reset DC would have to be purely built over program parameters. The bound algorithm for such DCPs differs only in the definition of `ResetSum` procedure which is now defined in the following way:

$$\texttt{ResetSum}(\tau_v) = \sum_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(\tau_v)} T\mathcal{B}(\texttt{t}) \times \max(\texttt{a} + \texttt{c}, 0) \tag{4.4}$$

As we can see, the only difference is that we additionally multiply each reset value by the bound of the transition `t` where the reset occurs. The reasoning behind this change is simple: as constant resets might now happen on any transition, we need to take into account the fact that it might be executed multiple times. Thus, the total amount by which the value of local bound τ_v might increase is also affected by the number of times the reset happens. Example of such case can be seen in Figure 4.4. In this example we

```
void foo(int n) {
    int j = n;
    int cnt = 0;
    while(j > 0) {
        j--;
        for(int i = n - 1; i > 0; i--)
            cnt++;
    }
}
```

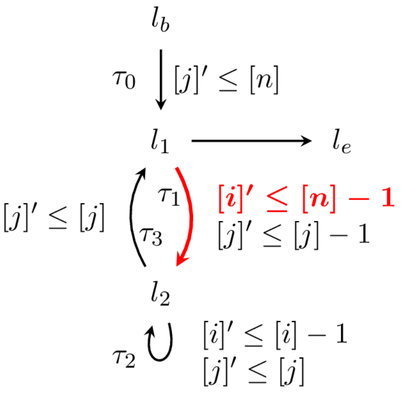


Figure 4.4: A DCP with the $[i]' \leq [n] - 1$ constant reset on the non-initial transition τ_1.

reset the value of local bound i of the transition τ_1 to the value of $[n] - 1$ in each iteration of outer loop. Thus, the total cost of function `foo` is $[n] + [n] \times \max([n] - 1, 0)$ because we have n back-jumps of the outer loop and $n - 1$ back-jumps of the inner loop for each iteration of the outer loop.

4.3.4 Extending the Procedure with Non-Constant Resets

We further extend the basic algorithm with a concept of *variable bounds*. Variable bounds are expressions over program parameters that over-approximate the value of non-constant resets. This way, Loopus reduces the problem of reasoning about non-constant resets to the problem of reasoning about constant resets. For example, if we had the $x = y$ reset assignment in a concrete program, we would transform the variable y into an expression $\texttt{expr}(params)$ over program parameters. Loopus calls this expression an *upper bound invariant* for y because the inequality $y \leq \texttt{expr}(params)$ has to be invariant. I.e., the $\texttt{expr}(params)$ *bounds* the value of y in terms of program parameters. The variable bound is just a special case of an upper bound invariant which is used in DCPs.

The variable bound concept is used to extend the bound algorithm with a procedure $V\mathcal{B}$ which is similar to the $T\mathcal{B}$ procedure and is defined as:

$$V\mathcal{B}(e) = \begin{cases} e, & \text{if } e \notin \mathcal{V} \\ \texttt{IncrementSum}(e) + \max\limits_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(e)} (V\mathcal{B}(\texttt{a}) + \texttt{c}), & \text{else} \end{cases} \tag{4.5}$$

The $V\mathcal{B}$ returns the input norm e itself it it is built purely over program parameters, i.e., it is a constant. The `IncrementSum` procedure from the second case leads to mutual recursion of procedures $T\mathcal{B}$ and $V\mathcal{B}$. The *max* function picks the maximum value of all possible resets and adds the total amount by which it might increase.

The $T\mathcal{B}$ procedure also needs to be modified by over-approximating the value of a reset of each element from the `ResetSum` by the $V\mathcal{B}$ procedure:

$$\texttt{ResetSum}(\tau_v) = \sum_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(\tau_v)} T\mathcal{B}(\texttt{t}) \times \max(V\mathcal{B}(\texttt{a}) + \texttt{c}, 0) \tag{4.6}$$

This leads to aforementioned mutual recursion between both procedures. Figure 4.5 presents such DCP:

```
void twoSCCs(int n, int m1, int m2) {
    int y = n;
    int x;
    if (*)
        x = m1;
    else
        x = m2;
    while(y > 0) {
        y--;
        x = x + 2;
    }
    int z = x;
    while (z > 0)
        z--;
}
```

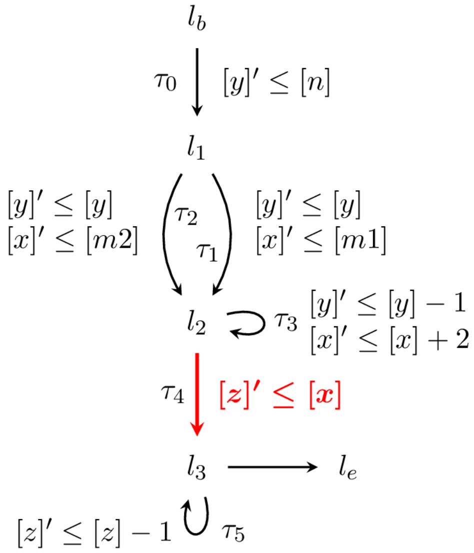


Figure 4.5: A DCP with the $[z]' \leq [x]$ non-constant reset on the transition τ_4

The key part of this example is the reset of the norm $[z]$ to the value of norm $[x]$ on the transition τ_4. Because the norm $[x]$ is not constant, we employ the newly introduced $V\mathcal{B}$ procedure to over-approximate its value by a constant expression. This way we obtain the variable bound of $max([m1], [m2]) + 2 \times [n]$ for the norm $[x]$ which in turn becomes the transition bound for transition τ_5. Arguments for the *max* function are determined by the two possible resets of $[x]$ to norm $[m1]$ or $[m2]$ and the increment of $2 \times [n]$ is caused by the n iterations of the first while loop. The full computation process is described in Table 4.1.

Even though this version of bound algorithm can solve DCPs with non-constant resets, it still has many shortcomings. Most notable one is the fact that it is *flow-insensitive* and consequently *path-insensitive* which leads to a coarse over-approximations even in simple cases.

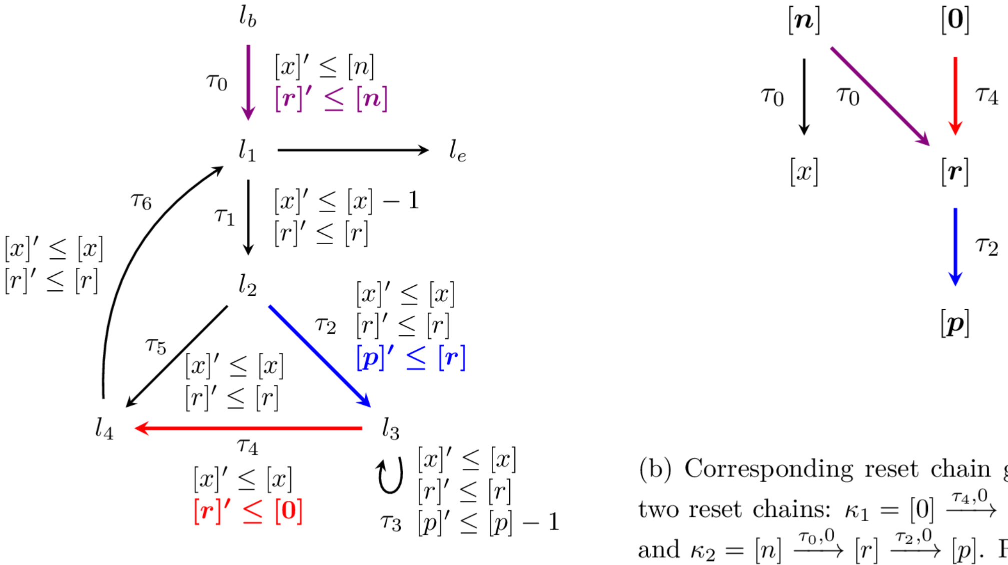
4.3.5 Improving the Bounds with Reset Chains

The introduced algorithm can be optimized by several techniques that improve the inferred resource bounds. First, we will improve the bounds with the notion of reset chains. The basic algorithm was based on the assumption that resets to the values over-approximated by the $V\mathcal{B}(\tau)$ procedure are valid in all iterations of a loop and it did not consider the possibility that some reset values are reachable through program paths which can be executed only once.

$T\mathcal{B}(\tau_5)$	$\rightarrow \texttt{Incr}([z]) + T\mathcal{B}(\tau_4) \times \max(V\mathcal{B}([x]) + 0, 0)$
	$\rightarrow 0 + 1 \times \max(2 \times [\boldsymbol{n}] + \max([\boldsymbol{m1}], [\boldsymbol{m2}]) + 0, 0)$
	$\rightarrow 2 \times [\boldsymbol{n}] + \max([\boldsymbol{m1}], [\boldsymbol{m2}])$
$V\mathcal{B}([x])$	$\rightarrow \texttt{Incr}([x]) + \max(V\mathcal{B}([\boldsymbol{m1}]) + 0, V\mathcal{B}([\boldsymbol{m2}]) + 0)$
	$\rightarrow 2 \times [\boldsymbol{n}] + \max([\boldsymbol{m1}], [\boldsymbol{m2}])$
$\texttt{Incr}([x])$	$\rightarrow T\mathcal{B}(\tau_3) \times 2 = [\boldsymbol{n}] \times 2$
$T\mathcal{B}(\tau_3)$	$\rightarrow \texttt{Incr}([y]) + T\mathcal{B}(\tau_0) \times \max(V\mathcal{B}([\boldsymbol{n}]) + 0, 0)$
	$\rightarrow 0 + 1 \times [\boldsymbol{n}] = [\boldsymbol{n}]$

Table 4.1: Computation of the transition bound for the τ_5 transition from the `twoSCCs` 4.5 example. Note that $\mathcal{I}([z]) = \mathcal{I}([y]) = \varnothing$, and consequently $\texttt{Incr}([z]) = \texttt{Incr}([y]) = 0$.

Reset chains allow reasoning about sequences of resets which might occur during execution and introduces a program path *context* for each reset. Loopus uses so called *reset chain graphs* which allow for systematical reasoning about the context of each reset and consequently provide an easy way how to find all the possible reset chains. Figure 4.6 presents a DCP (a) with the corresponding reset graph (b):



(a) A DCP requiring reset chain reasoning in order to obtain the precise linear complexity of $2n$.

(b) Corresponding reset chain graph with two reset chains: $\kappa_1 = [0] \xrightarrow{\tau_4,0} [r] \xrightarrow{\tau_2,0} [p]$ and $\kappa_2 = [n] \xrightarrow{\tau_0,0} [r] \xrightarrow{\tau_2,0} [p]$. Reset chain κ_2 is valid only in the first iteration of the loop l_1 due to the τ_0 transition.

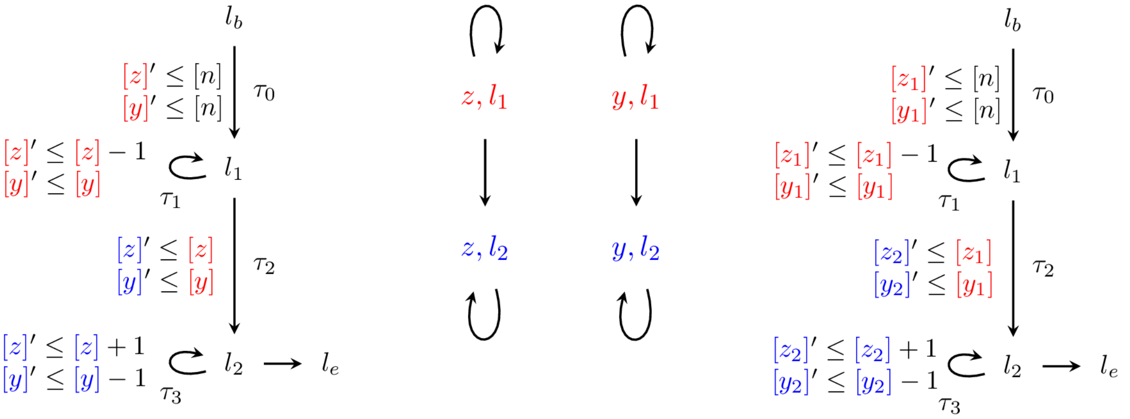
Figure 4.6

The semantics of this oriented reset graph are simple: each labeled edge corresponds to a transition of a DCP with the same label where the reset $[x]' \leq [y] + \texttt{c}$ occurs. For example, the edge from node $[0]$ to node $[r]$ represents a reset $[r]' \leq [0]$ which occurs on the transition τ_4 in the original DCP. We can also intuitively find all the *maximal* reset chains ending in $[p]$ by looking at the graph: $\kappa_1 = [0] \xrightarrow{\tau_4,0} [r] \xrightarrow{\tau_2,0} [p]$ and $\kappa_2 = [n] \xrightarrow{\tau_0,0} [r] \xrightarrow{\tau_2,0} [p]$.

These two reset chains allow Loopus to infer the linear bound n instead of n^2 for the loop l_3. The basic idea is to use a set of reset chains instead of a set of simple resets in the $\mathcal{TB}$ procedure and then, in case of our example, apply following reasoning: $[p]$ is the local bound for τ_3 which is reset on τ_2. If we execute τ_2 under context of τ_0, $[p]$ gets reset to the value of $[n]$. However, τ_2 can be executed under context of τ_0 only once because τ_0 is the initial transition. Thus, the reset chain κ_2 is valid only for the first iteration of the outer loop l_1, which means $[p]$ can be reset to $[n]$ through $[r]$ only once, leading to the aforementioned linear bound of n for the loop l_3. The reset chain κ_1 is, unlike κ_2, valid in all of the iterations of l_1 as the transition τ_4 is part of the loop. However, it does not increase the total number of iterations of l_3 because it resets $[p]$ to $[0]$ through $[r]$. Please refer to [12] for details on the modified bound algorithm incorporating reset chains.

4.3.6 Improving the Bounds with Flow-Sensitivity

The basic algorithm is *flow-insensitive* which can lead to coarse over-approximations such as the one in Figure 4.7a. Here, we infer bound $2n$ for the transition τ_1 because the initial value n of the local bound $[z]$ increases by one in each of n iterations of the loop l_2. However, this bound is imprecise because the increments to $[z]$ at the location l_2 can never *flow* back to the location l_1 and thus cannot affect the local bound of τ_1. The correct bound for τ_1 is just n.



(a) Flow-insensitive DCP (b) Variable Flow Graph (c) Flow-sensitive DCP

Figure 4.7: DCPs constructed by the abstraction algorithm from Section 4.2 are *flow-insensitive* by default. We construct a *variable flow graph* and rename the program variables to obtain a *flow-sensitive* DCP.

Loopus can support *flow-sensitivity* by a pre-processing which renames the norms of a DCP based on a *variable flow graph* (VFG). As the name suggests, a VFG shows how the values of program variables (not symbolic constants) *flow* from one location to another. Figure 4.7b shows the VFG obtained from the DCP in Figure 4.7a. The basic idea is that we find all the SCCs in a VFG and then choose a fresh artificial variable v for each SCC ζ which we assign to all program variables at program locations that are part of ζ. After that, we just rename the original left-hand side variables of DCs to v on all edges that are incoming to some location included in ζ and also rename all right-hand side variables of DCs to v on all outgoing edges from some location in ζ. For example, the VFG on Figure 4.7b contains four SCCs: $\zeta_1 = \{(z, l_1)\}$, $\zeta_2 = \{(z, l_2)\}$, $\zeta_3 = \{(y, l_1)\}$ and $\zeta_4 = \{(y, l_2)\}$. We thus create four fresh variables and assign them to original program variables at certain locations: $(z, l_1) = z_1$, $(z, l_2) = z_2$, $(y, l_1) = y_1$ and $(y, l_2) = y_2$. The only thing left to do is to rename the original variables in the flow-insensitive DCP based on those assignments and renaming rules which leads us to the flow-sensitive DCP in Figure 4.7c. For more details please refer to [12].

Chapter 5

Implementation

In this chapter we describe the implementation of *Looper* – the new performance oriented analyzer for Infer based on the original Loopus tool. Due to the scope of this work, we limited ourselves to the core abstraction and bound algorithms without additional extensions presented in [12]. However, even without the extensions, our analyzer is able to handle some challenging code examples as we will show in our experimental evaluation. There are three main parts of the implementation which we will discuss in detail in separate sections: conversion of the native Infer CFG to the LTS representation, abstraction algorithm for transformation of a LTS to a DCP, and the bound algorithm itself.

5.1 Construction of Labeled Transition System

The first implementation task was to devise an algorithm for construction of the *labeled transition system* representation. The native control flow graph used by Infer is overly complex due to its low-level nature as it builds upon the Smallfoot Intermediate Language. Thus, it contains all the details regarding the manipulation with internal temporary variables by `Load` and `Store` instructions as well as the details about their lifetime. However, Loopus works with a much higher-level of representation. Moreover, the abstraction algorithm discussed in Section 4.2 transforms a LTS to a DCP. Figure 5.1 shows an example of the native CFG generated by Infer.

We leverage the AI framework and construct a LTS after the symbolic execution of the program. During the symbolic execution we also gather additional information which is needed in the abstraction process, e.g., the initial set of norms. We use the abstract state defined by the domain to store all the necessary data, then construct the LTS and perform the bound analysis after we receive the *post-condition* state from the abstract interpreter.

5.1.1 Structure Construction

The first construction task was to implement the *transfer functions* and the join operator in such a way that we would obtain a set of nodes and a set of edges at the end of the interpretation. The construction of a graph from those two sets is handled by the *OCamlgraph* library which also implements many useful graph algorithms such as the computation of SCCs. However, in order to use the parametric `Graph` library module, we had to provide an implementation for the signature node and edge data modules.

First, we introduce the necessary `Node` module used for unique identification of each relevant program point in a LTS graph. It has the following inner variant type:

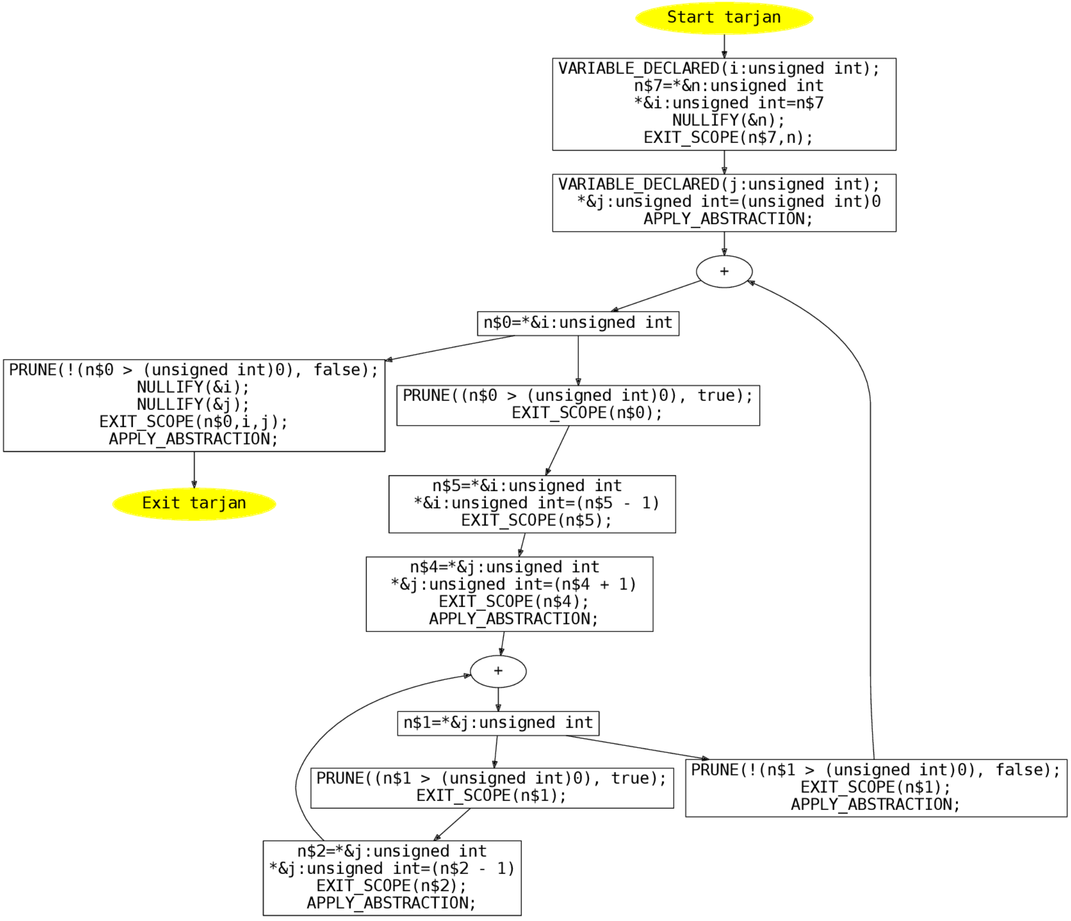


Figure 5.1: Native Control Flow Graph generated by Infer from the `tarjan` (4.1) example.

```
type t = | Start of Location.t
         | Prune of (Sil.if_kind * Location.t)
         | Join of (t * t)
         | Exit
```

This type basically extends the `Location` module provided by Infer which is used to represent *physical* (file, line and column) code locations only. However, in a LTS, we also need to be able to represent abstract join and exit locations that do not have a counterpart in the code. We thus use the `Node` module to model both physical and abstract locations. The `Prune` (naming adopted by Infer) case models control flow splits that occur for example with loops or conditional statements and is defined by the specific prune type (while loop, for loop, switch, etc.) and a program location. The `Start` case is an entry point of a procedure defined by the location of its header. The `Join` case represents an abstract location uniquely specified by the last visited locations of the two joined branches as Infer supports only binary join operator. The final `Exit` case represents an abstract exit point of a procedure which does not need to be uniquely specified by any additional data as we only have a single LTS exit node per procedure.

Next we introduce the EdgeData module which implements various methods used during the abstraction process and stores necessary data for both the LTS and DCP graphs. The inner record data type is defined as:

```
type t = {
      backedge: bool;
      conditions: Exp.Set.t;
      assignments: Exp.t PvarMap.t;
      ...
}
```

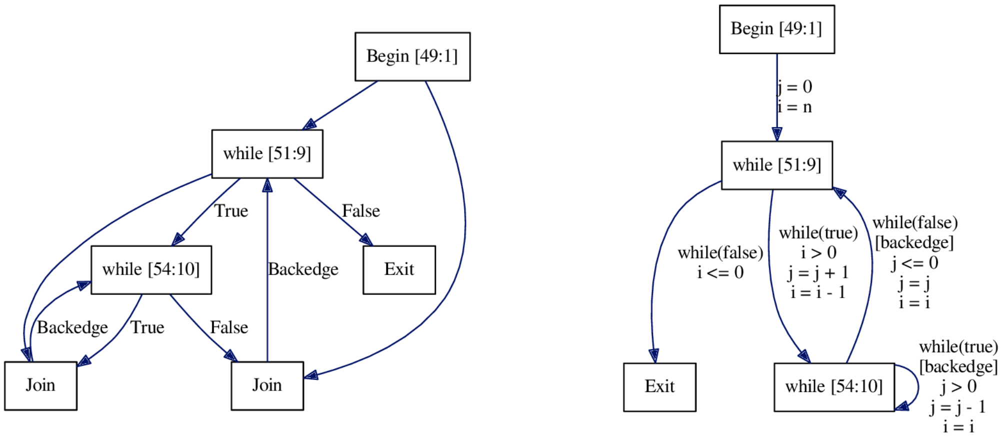
We explicitly store the information if the edge is a back-edge in the backedge boolean field so we can easily detect it during the interpretation process if we encounter a prune location and the last known location stored in the abstract state is the same or with a higher line number. The conditions field stores all the conditional expressions from the Prune SIL instructions, e.g., if we have a loop header with a $i > 0$ termination condition, we add it to the expression set on the *true* edge and then add the negated $i \leq 0$ expression to the set on the *false* edge. This information is necessary for the derivation of guards. The assignments map stores all the assignment expressions that occur between two program points connected by an edge. We use the existing Pvar (program variable) module as the key type of the map for easier look-ups when we need to check if a variable is modified on an edge and the existing Exp (expression) module as the type for the associated values. We use these assignment expressions during the abstraction process to check how the value of our initial norms changes in order to derive new ones along with the creation of the set of DCs for each transition.

With both the Node and EdgeData modules covered, we can now introduce some of the fields from the abstract state record:

```
type t = {
  last_node: DCP.Node.t;
  edge_data: DCP.EdgeData.t;
  graph_nodes: DCP.NodeSet.t;
  graph_edges: DCP.EdgeSet.t;
  ...
}
```

The last_node field stores a Node instance corresponding to the last visited program location. Once we need to create a new node, we simply add the (last_node, edge_data, new_node) tuple to the graph_edges set and replace the value stored in the last_node field with the new_node instance. Each new node is also added to the graph_nodes set and every time we add a new tuple to the graph_edges set we also create a new empty instance of the EdgeData module and store it in the edge_data field. In other words, we create a new edge that connects two subsequently visited program locations (nodes) and repeat this process until we obtain complete sets of nodes and edges.

However, this basic approach leads to a LTS with some unwanted nodes and edges due to the way abstract interpretation works and due to specifics of Infer implementation. These unwanted graph elements could break the bound analysis and we thus improved the basic concept in order to obtain a simplified graph which would resemble the LTS presented in [12] as close as possible. Figure 5.2 shows the structural comparison between the initial and final LTS graphs obtained from the previously featured tarjan (4.1) example.



(a) Initial LTS

(b) Final LTS

Figure 5.2: Structural comparison between the initial and final LTS graphs obtained from the `tarjan` 4.1 example.

As we can see, the initial version has two additional join nodes and consequently four more edges which is caused by the fact that the basic approach exactly follows the interpretation process. For example, when the interpreter reaches the end of the inner while loop, it has to perform a join between the abstract state inside the loop and the state right before the header of the inner loop which has the header of the outer loop stored as the last known location. Thus, we create a new join node and two new edges from the last known locations of both abstract states pointing to it. The interpreter then needs to analyze the inner loop again with the joined state to check if we have reached a *fixpoint* and executes the header `Prune` instruction again. As that happens, we create a new back-edge from the last known join node to the inner loop node.

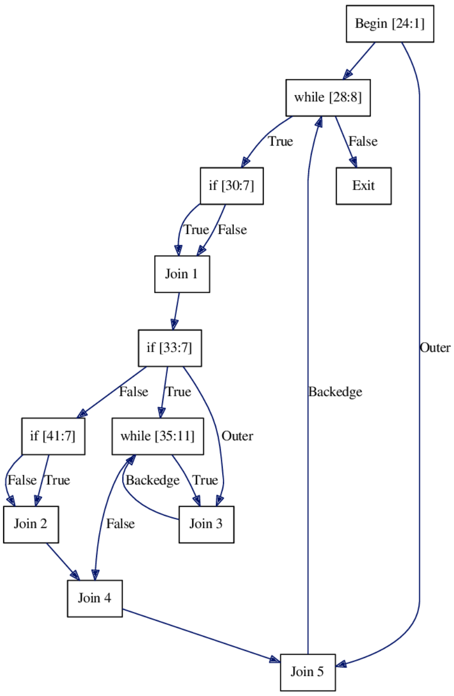
The first issue with direct back-edges such as the one at the inner loop is solved as follows: we store the locations of both abstract states on a join and check if one of them matches the location of the next encountered `Prune` after the join. If it does, we delete the join node along with the edge from the outer scope and redirect the correct edge to create a loop-back as seen in the final LTS in Figure 5.2b.

The second issue is the join node between the *false* branch of the inner loop and the abstract state from right before the the first prune which basically joins the final state from the end of the outer loop body with the outside scope. Our goal is thus to ignore all the edges from outer scopes which is a generalization of the first issue. However, unlike with the first issue we cannot easily detect which edge is from the outer scope because the locations do not match, i.e., we do not have a loop-back to the same location. To solve this, we introduce a new concept of branching path which is a stack of the following tuples: `(Sil.if_kind * bool * Location.t)`. The `Sil.if_kind` element describes the type of prune, i.e., the specific kind of loop or conditional statement. The boolean value tells us if it is the *true* or *false* branch and the last element is the location of a prune. We use the `branching_path` field in the abstract state record to store the current path for each abstract state and whenever we encounter a `Prune` instruction we push a newly constructed tuple to the top of the stack. This way we can track our current nesting level along with the information about each scope and also uniquely identify each branch of the program. We leverage this to detect edges from the outer scopes as follows: we compute a common prefix of both paths on a join and if this prefix exactly matches the path from one of the states, we ignore the edge coming from it as it must be the outer scope state. We can demonstrate this method on our `tarjan` example with two paths: $\pi_1 = [\,]$ and $\pi_2 = [(\texttt{while}, \texttt{true}, \texttt{51} : \texttt{9}), (\texttt{while}, \texttt{false}, \texttt{54} : \texttt{10})]$. The common prefix is obviously $\pi = [\,]$ and it exactly matches the first path π_1. Thus, the state with this path must be the outer scope state and we can ignore the edge. However, we still have the join node with one incoming edge, so on the next visit of the loop header we detect a join node with a single incoming edge which we redirect to the loop header and delete the useless join node.

These two adjustments lead to a valid but still more complex LTS in most cases. We can further reduce the amount of nodes and edges and consequently simplify our graphs without any alterations to the original semantics. We will demonstrate some of these techniques on Figure 5.3. We can remove the useless `Join 1` node that merges both branches of the first `if` statement together. Every time we encounter the `Prune` instruction we check the `last_node` field of the abstract state. If it is a `Join(lhs, rhs)`, where the `lhs` and `rhs` nodes match and the `edge_data` field of the abstract state is currently empty, we delete the node and redirect incoming edges of both branches to the current `Prune` node.

```
void xnu(int len) {
  int beg, end, i = 0;
  while(i < len) {
    i++;
    if (*)
      end = i;
    if (*) {
      int k = beg;
      while (k < end)
        k++;
      end = i;
      beg = end;
    } else if (*) {
      end = i;
      beg = end;
    }
  }
}
```

(a) Example `xnu` from [12]



(b) Initial LTS obtained from `xnu` 5.3a example

Figure 5.3

The graph can be further simplified by merging the consecutive `Join 2`, `Join 4`, and `Join 5` nodes together to create one N-ary join node. We first detect if a join is consecutive. The naive approach is to check if the `last_node` field of one of the joined states is already a join node and declare the current join as consecutive if it is. If this happens, we do not create a new join node and instead store the previous one in the `last_node` field of the new joined state. We also create a new edge from the last node of the other state and add it to the `incoming_edges` set field in the joined abstract state. In reality, on every join we always first store all of the new edges in the `incoming_edges` set. This way we can easily redirect new edges if need be when we encounter a next `Prune` instruction and then move them to the `graph_edges` set after that. The simplification process in this case might look like this: we first perform the join of both branches of the `if[41:7]` statement, create the `Join 2` node, and add two new edges in the `incoming_edges` set. Next we perform the join which would normally create the `Join 4` node and identify it as consecutive. We thus create a new edge from the `while[35:11]` node and add it to the `incoming_edges` set instead. Then we perform the second and last consecutive join represented by the `Join 5` node. However, in this case we do not create a new edge as it would be an edge from an outer scope. Finally, we execute the `Prune` instruction at the `while[28:8]` location, create a new back-edge, and move all of the edges stored in the `incoming_edges` set to the `graph_edges` set. The final LTS obtained with this approach can be seen in Figure 5.4.

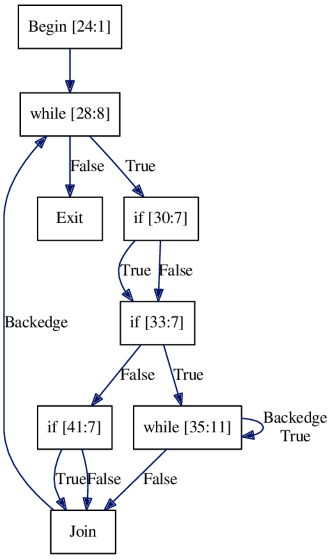
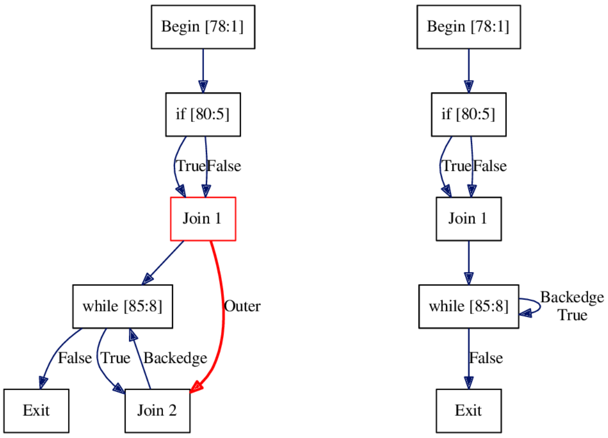


Figure 5.4: Final LTS

However, the naive approach to the detection of consecutive joins can fail in some cases such as the one in Figure 5.5. The reason why it fails in this case is the fact that we have

```
void foo(int n, int m)
{
  int x;
  if(*)
    x = 0;
  else
    x = m;
  int y = x + n;
  while(y > 0) {
    y--;
  }
}
```

(a) Example `foo`



(b) Initial LTS

(c) Final LTS

Figure 5.5

a join node right before the first while loop. Thus, when we join the loop branch with the outer scope branch which has a join node stored in the `last_node` field, we detect a consecutive join even though we do not consider it as one. This would lead to a LTS with semantics different from the original program and in order to fix this, we leverage the previously introduced `branching_path` field in a similar way once again. We compute a common prefix of both paths and check if it matches the path from the state with a join node. If it does, we do not consider it as a consecutive join because the join node must have been from an outer scope. For example, the `Join 2` node joins branches with following paths: $\pi_1 = []$ and $\pi_2 = [(\texttt{while}, \texttt{true}, 85 : 8)]$ with the common prefix $\pi = []$ which matches the π_1 path of the outer scope state. We thus consider it as a normal join and create the `Join 2` node which is subsequently deleted because it has only one incoming edge from the `while[85:8]` node (outer edge is ignored). The single edge is then redirected and we get a direct back-edge in the final graph seen in Figure 5.5c.

We have now covered the basic principle behind our construction algorithm. It is neither perfect nor final but it is not even meant to be as we are gradually improving it every time we encounter a new problem with the construction. We also did not cover some implementation details or minor hacks that were necessary due to some Infer specific alterations of the basic interpretation algorithm. It would be useless considering the rapid development of Infer which frequently renders some of our solutions obsolete or, better yet, unnecessary due to some core changes.

5.1.2 Construction of Edge Formulas

Apart from the structure, we still need to construct assignment and conditional formulas for each graph edge to get a complete LTS such as the one in Figure 5.2b. The process is quite straightforward and involves implementation of *transfer functions* for SIL `Load` and `Store` instructions. Infer always first loads all the addressable values into temporary identifiers represented by the `Ident` module with the `Load` instruction and then uses the `Store` instruction to store a new value into a program variable. The stored value is given by an expression built over those temporary identifiers. Constant values can be stored directly without previous `Load` instructions.

However, LTS edges are labeled by simple assignment or conditional expressions built purely over program variables and not temporary identifiers. We thus use the `ident_map` field in our abstract state to store the `ident` → `pvar` mapping, i.e., the field is just a map with `Ident` instances as keys and `Pvar` instances as values. The `Load` instruction is implemented in a simple way: we create a new `id` → `pvar` association and add it to the `ident_map` field if the loaded value is just a program variable. However, Infer might load one temporary identifier into another in some cases such as when it encounters a pointer dereference, i.e., we get a load assignment $\texttt{id}_2 = \texttt{id}_1$ which we cannot store in the map. To fix this, we find the existing $\texttt{id}_1 \rightarrow \texttt{pvar}$ association in the map and add a new transitive $\texttt{id}_2 \rightarrow \texttt{pvar}$ association based on the complete $\texttt{id}_2 \rightarrow \texttt{id}_1 \rightarrow \texttt{pvar}$ load chain to it.

Our implementation of the `Store` instruction then uses the `ident_map` field to substitute all identifiers with program variables in the right hand side expression of the $\texttt{pvar} = \texttt{expr}$ store assignment. We use a simple recursive function that traverses the expression tree and performs the substitution whenever it encounters an identifier. Then we traverse the updated expression once again and check if there already is an assignment stored in the `edge_data` field for each `pvar`. If there is, we substitute it with the right hand side of the stored assignment, e.g., if we had $x = z + 1$ and then $y = x$, we would get $y = z + 1$. This step is not necessary but Loopus performs it to simplify the subsequent abstraction process. Then we take the final expression and store it in the `edge_data` field in our abstract state.

Similarly to `Store`, we also substitute all identifiers in conditional expressions of each `Prune` instruction to get conditions built purely over program variables. Then we simplify the *false* branch condition because Infer wraps the *true* branch condition with the unary logical not operator, i.e., we transform expressions such as $\neg(x > 0)$ to simpler $x \leq 0$ normalized form which saves us some pattern matching later on. At the end, we store the result in the `edge_data` field.

Apart from that, we also need to generate missing constant assignments such as $\mathtt{i}' = \mathtt{i}$ on each `Prune` for all local function variables that were not modified on the current edge and are defined at its target program location. However, there is a slight problem with local variables because Infer in its current state exposes only the list of all local variables used throughout a function without the information about their scope. This might decrease the precision of the bound algorithm as we get more constant assignments and consequently derive more norms during the abstraction process. Fortunately, we are able to detect variable declarations during the interpretation with the `VariableLifetimeBegins` metadata SIL instruction or even through the `Store` instruction if it is part of a CFG node with the `DeclStmt` (Declaration Statement) type. However, even this approach has a flaw as it works only if the declared variable is also initialized at the same time, otherwise Infer neither generates the `VariableLifetimeBegins` instruction nor sets the type of the node as `DeclStmt`. We opted for this approach in spite of that because it currently is the only possible way how to reduce the precision loss with a small trade-off.

The complete generation process is quite simple. First we determine active local variables for each relevant program location, then we keep track of all variables that were modified on the current edge and finally we add new `pvar` $\rightarrow$ `pvar` association to the `assignments` map in the `edge_data` field for each unmodified variable which we obtain through `locals` $\setminus$ `edge_modified` set difference. The `locals` set of `Pvar` elements is stored in our abstract state and we add a new `Pvar` every time we detect a declaration. The join is defined as the $\mathtt{locals_a} \cap \mathtt{locals_b}$ set intersection which models variables going out of scope. Similarly, the `edge_modified` set keeps track of all the local variables that were modified on the current edge by `Store` instructions and we empty the set on each join.

These generated constant assignments are abstracted to constant DCs which are necessary for the correct functionality of the bound algorithm. For example, the flow-sensitivity transformation would not work properly without constant DCs which create the flow of variable values from one location to another. We should now have complete edge formulas including both conditions and assignments on each edge of a LTS and with that we can conclude our construction discussion.

5.1.3 Initial Set of Norms

The collection process of initial norms is discussed under the construction section even though it technically falls under the abstraction algorithm because we are collecting the norms during the interpretation as we construct a LTS graph. The basic norm derivation concept described in Section 4.2.1 is followed by our implementation: we parse each `BinOp(op, lhs, rhs)` binary expression condition represented by the `Exp` module and then rearrange it based on the specific `op` operator to obtain a new norm with the same `Exp` type.

However, the selection of relevant conditions to derive norms from is based purely on heuristics introduced in [12]: we look for conditions of form $x > y$ or $x \geq y$ found in loop headers or conditional statements inside the loop if they involve variables incremented/decremented on any loop path. The solution for loop header conditions is straightforward because we know that their values and consequently the values of involved variables must change if the loop terminates. Thus, we only need to derive a new norm and then immediately save it in the `initial_norms` set of `Exp` expressions in our abstract state.

The situation with conditional statements on loop paths is more complicated. First we need to track all variables that were modified in a loop similarly to variables modified on a single edge. We add a new `Pvar` to the `loop_modified` field in our abstract state every time we modify a variable inside a loop and we use the stored branching path to check if the current state is indeed in a loop. We also check the common prefix of branching paths on each join and empty the set if we detect that we are no longer in any loop after the join. Otherwise we union both sets. Finally, we add a new norm derived from each non-loop `Prune` to the auxiliary `potential_norms` set in our abstract state if we are currently in a loop and the variables constituting the derived norm are not present in the `loop_modified` set. The `potential_norms` set basically contains norms that first have to be confirmed by `Store` increment or decrement in the current loop. Potential norms are moved to the `initial_norms` set upon confirmation.

5.2 Implementation of the Abstraction Algorithm

Our implementation conceptually follows the abstraction algorithm as presented in Section 4.2. However, there are differences between the concept and the implementation and the mere concept does not cover the derivation of DCs in much detail. This chapter covers the implementation of each abstraction step described in Section 4.2.

5.2.1 Abstraction of Transitions

We have decided to implement the main abstraction loop in the imperative paradigm because it proved to be much easier to write and the final algorithm is more concise and readable than it would have been in the functional style. The basic idea which we have built upon is to iterate over all graph edges repeatedly, trying to construct new DCs and consequently derive new norms until the set of all norms becomes stable. Algorithm 1 presents a pseudocode of the main abstraction loop.

Algorithm 1: Abstraction loop for inference of norms and derivation of DCs.

```
Input  : Initial set of norms InitialNorms and a set of LTS edges with assignments
Output: A final set of norms and a set of edges with derived difference constraints
Unprocessed = InitialNorms;
Processed = $\emptyset$;
while $Unprocessed \neq \emptyset$ do
    Norm = Unprocessed.pop();
    Processed.insert(Norm);
    foreach $Edge \in LTS$ do
        NewNorm = Edge.deriveConstraint(Norm);
        if $NewNorm \notin Processed$ then
            Unprocessed.insert(NewNorm);
```

We start with the `Unprocessed` set of initial norms and empty `Processed` set which will contain all norms at the end. We basically pick a random norm from the `Unprocessed` set in each iteration and then move it to the `Processed` set. Then we iterate over all edges of a LTS and try to construct a new DC for each edge based on the previously selected norm. However, this might lead to derivation of a norm which needs to be added to the `Unprocessed` set if it is new in which case we have to repeat the whole process with the new norm until the `Unprocessed` set gets empty.

The entire derivation algorithm is implemented by the `derive_constraint` function of our `EdgeData` module. It creates a new DC for an edge based on the input variable norm and also forms a new norm if it is not possible to construct a new DC only by reusing the input one. We symbolically execute assignments stored in the `assignments` field of the current edge and observe how the value of the input norm changes in order to construct new DC. The whole process is trivial for `Pvar` norms: first we check if there even is an assignment for the `Pvar` and immediately return if not, otherwise we further check if the assignment is constant in which case we just construct a constant DC and return. We currently support only few but common forms of non-constant assignments for `Pvar` norms:

1. $x' = y$ or $x = \mathtt{c}$: construct $x' \leq y/\mathtt{c}$ DC and derive variable y or constant `c` norm.
2. $x' = x \pm \mathtt{c}$: construct $x' \leq x \pm \mathtt{c}$ DC and do not derive any new norm.
3. $x' = y \pm \mathtt{c}$: construct $x' \leq y \pm \mathtt{c}$ DC and derive variable y norm.

We also currently support norms of form $x - y$ with considerably more complicated processing. First we have to check if both involved variables are defined at the destination location, i.e., they are assigned at the current edge. Note that formal parameters are defined at all program locations and have imaginary constant assignments on each edge as a consequence. Again, the function immediately returns if one of the variables is not defined, otherwise it checks for constant assignments and constructs a constant DC if none of them is modified. Now there are three remaining options:

1. Variable x is modified:
   - $x' = x \pm \mathtt{c}$: the value of $x - y$ norm increases/decreases, do not derive any new norm and construct $(x - y)' \leq (x - y) \pm \mathtt{c}$ DC.
   - $x' = y$: the value of $x - y$ norm is set to 0, derive constant norm 0 and construct $(x - y)' \leq 0$ DC.
   - $x' = z$: derive new norm z and construct $(x - y)' \leq (z - y)$ DC.
2. Variable y is modified:
   - $y' = y + \mathtt{c}$: the overall value of $x - y$ norm decreases (interval shrinks), do not derive any new norm and construct $(x - y)' \leq (x - y) - \mathtt{c}$ DC.
   - $y' = y - \mathtt{c}$: the overall value of $x - y$ norm increases (interval expands), do not derive any new norm and construct $(x - y)' \leq (x - y) + \mathtt{c}$ DC.

   The remaining possibilities are symmetric to the previous option.
3. Both variables are modified:

- $x' = z \wedge y' = z$: the value of $x - y$ norm is set to 0, derive constant norm 0 and construct $(x-y)' \leq 0$ DC.

We focused on the most frequent types of norm expressions as writing processing code for all of the possibilities at once would be unfeasible in the scope of this work but we plan to gradually extend it in the future.

5.2.2 Guard Inference Algorithm

The next step is to infer guards for each transition, i.e., determine which norms from the final set of norms are guaranteed to have positive value upon execution of a transition. We employ the Z3 SMT solver to prove that a norm is also a guard on a transition based on the `Prune` conditions obtained during the LTS construction. We process each graph edge with the `derive_guards` function of the `EdgeData` module. The main purpose of this function is to parse and transform native expressions from the `Exp` module format to the `Z3.Expr` module format used by the Z3 solver. It transforms all of the conditions on the input edge and constructs following Z3 formula for each norm: $\neg(\mathit{conditions} \implies \mathit{norm} > 0)$, where *conditions* denotes logical conjunction of all transformed edge conditions. By checking the *satisfiability* of such formula, we are able to prove that a norm is also a guard if it is unsatisfiable as it means that formula $\mathit{conditions} \implies \mathit{norm} > 0$ is *valid*. All proved guards are stored in the `guards` field of the `EdgeData` module.

Algorithm 2: Guard inference algorithm

```
Input  : Set of norms $NormSet$ and a set of $LTS$ edges with conditions
Output: A set of DCP edges with $guards$ and $difference\ constraints$
foreach $Edge \in LTS$ do
    ConditionsZ3 = TransformZ3(Edge.conditions);
    foreach $Norm \in NormSet$ do
        NormZ3 = TransformZ3(Norm);
        Formula = $\neg(ConditionsZ3 \implies NormZ3 > 0)$;
        if $Z3.Check(Formula) = UNSATISFIABLE$ then
            Edge.guards.insert(Norm);
```

5.2.3 Guard Propagation Algorithm

The algorithm for propagation of guards as presented in Section 4.2.1 has one major flaw. It does not propagate guards to *false* branches at loop headers correctly because the loop back-edge is also an incoming edge and the basic recursive algorithm does not prioritize the propagation through loop body before it processes the *false* branch. It is necessary to first propagate guards through the loop body, recalculate the set of shared guards from all incoming edges including the back-edge, and finally propagate guards to the *false* branch. The modified version is presented in Algorithm 3.

First, we iterate over all DCP edges and construct a set of nodes which have at least one guarded incoming edge. This set is passed to the `PropagateGuards` recursive function which implements the propagation algorithm. The auxiliary `GetSharedGuards` function constructs a set of non-decreased guards shared among all incoming edges excluding back-edges. Similarly, the `activeGuards` method of the `EdgeData` module returns only non-decreased guards of an edge. The `partition` method separates the set of outgoing edges into a *true* and *false* edges based on the stored tuple from the top of the `branching_path` stack. Recall that the tuple contains a boolean value which specifies the type of branch.

Algorithm 3: Modified guard propagation algorithm which prioritises propagation through *true* branches at loop headers

```
Input   : Initial set of guarded DCP nodes $GuardedNodes$
Output: A DCP with propagated guards
Function PropagateGuards($GuardedNodes$):
    if $GuardedNodes \neq \varnothing$ then
        Node = GuardedNodes.pop();
        OutgoingEdges = Node.outgoingEdges;
        Guards = GetSharedGuards($Node.incomingEdges$);
        if $Node\ is\ LOOP\_HEADER$ then
            TrueEdge, FalseEdge = OutgoingEdges.partition();
            TrueEdge.guards.add(Guards);
            if $\neg TrueEdge.backedge$ then
                PropagateGuards($GuardedNodes \cup TrueEdge.dstNode$);
                Backedge = Node.incomingEdges.findBackedge();
            else
                Backedge = TrueEdge;
            Guards = Guards $\cap$ Backedge.activeGuards();
            OutgoingEdges.remove(TrueEdge);
        if $Guards \neq \varnothing$ then
            foreach $Edge \in OutgoingEdges$ do
                Edge.guards.add(Guards);
                if $\neg Edge.backedge$ then
                    GuardedNodes.add(Edge.dstNode);
        return PropagateGuards($GuardedNodes$);
    else return;
```

5.3 Implementation of the Bound Algorithm

The implementation of the bound algorithm relatively closely resembles the formulas presented in Section 4.3 due to the functional paradigm of the *OCaml* language. This chapter covers some of the more technical but nevertheless important details which were not mentioned in the previous chapters focusing on the approach used in Loopus. For example, one such detail is *caching* of results which might seem irrelevant but is in fact crucial if we want to preserve the polynomial time complexity of the bound algorithm.

Algorithm 4 presents the main bound computation loop. We iterate over all DCP edges and compute a bound for each back-edge with the `TransitionBound` procedure which also returns updated cache. If the bound computation does not terminate due to cyclic mutual recursion, we break the main loop and return the ∞ bound. This occurs if the input

Algorithm 4: Main loop of the bound algorithm

```
Input   : DCP over natural numbers without guards
Output: A sum of all back-edge cost bounds reflecting the worst-case asymptotic
        complexity of a function
Cache = EmptyCache();
BoundSum = 0;
foreach Edge $\in$ DCP do
    if Edge.backedge then
        EdgeBound, Cache = TransitionBound(Edge, Cache);
        if EdgeBound $= \infty$ then
            BoundSum $= \infty$;
            break;
        BoundSum = BoundSum + EdgeBound;
```

program does not terminate or if the precise bound is not polynomial. Otherwise we add the edge bound to the final sum and simplify the expression in the process.

We use a custom `Bound` module which extends the `Exp` module provided by Infer to represent bound expressions. The inner recursive variant type is defined as follows:

```
type t = | BinOp of Binop.t * t * t
         | Value of Exp.t
         | Max of t list
         | Min of t list
         | Inf
```

The first `BinOp` case allows us to model complex bound expressions with binary operators. The `Value` case represents individual terms, i.e., constants or formal parameters. The `Max` and `Min` cases model the *max* and *min* functions respectively and the `Max` case covers both the $max(x, 0)$ and $max(x, \dots)$ variants determined based on the size of the `t list` argument list. The *min* function is used in a modified bound algorithm which leverages *reset chains* introduced in Section 4.3.5. The last `Inf` case models previously discussed ∞ bound.

The previously mentioned caching mainly concerns the results of the `TransitionBound` and `VariableBound` procedures. However, we also cache constructed $\mathcal{I}(\tau_v)$ and $\mathcal{R}(\tau_v)$ sets and reset chains to avoid repeated computations. We use the following record data type to store the results:

```
type cache = {
    updates: (Increments.t * Resets.t) Exp.Map.t;
    variable_bounds: Bound.t Exp.Map.t;
    reset_chains: RG.Chain.Set.t Exp.Map.t;
}
```

The `updates` field is a map which associates local bound norms τ_v to the $\mathcal{I}(\tau_v)$ and $\mathcal{R}(\tau_v)$ sets represented by the `Increments` and `Resets` modules. The `Increments` set contains following tuples: (`DCP.E.t * IntLit.t`), where the `DCP.E.t` element is a DCP edge which increments the local bound τ_v by an integer constant represented by the `IntLit.t` Infer module, i.e., edge with $\tau_v \leq \tau_v + \texttt{c}$ DC. Similarly, the `Resets` set contains (`DCP.E.t *`

`Exp.t * IntLit.t`) tuples which correspond to edges with $\tau_v \leq \mathtt{a} + \mathtt{c}$ reset DCs. The `variable_bounds` map stores computed variable bounds and the `reset_chains` map stores all reset chains for each local bound norm τ_v used in a $T\mathcal{B}(\tau)$ computation. The transition bound cache is stored directly in the mutable `bound_cache` field of the `EdgeData` module.

5.3.1 Implementation of the $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\mathtt{a})$ Procedures

The $T\mathcal{B}(\tau)$ implementation is based on the modified version which incorporates reset chains discussed in Section 4.3.5. First, we will describe the generalised `IncrementSum` procedure intended for sets of norms:

$$\mathtt{IncrementSum}(\mathcal{A}) = \sum_{\mathtt{a} \in \mathcal{A}} \sum_{(\mathtt{t},\mathtt{c}) \in \mathcal{I}(\mathtt{a})} T\mathcal{B}(\mathtt{t}) \times \mathtt{c} \tag{5.1}$$

where $\mathcal{A} = \bigcup_{\kappa \, \in \, \Re(\tau_v)} atoms_1(\kappa)$. Similar to the $\mathcal{R}(\tau_v)$ set, the $\Re(\tau_v)$ is a set of all maximal reset chains for a local bound τ_v. The $atoms_1(\kappa)$ is a set of variable norm nodes along a reset chain κ that have at most one path to the local bound τ_v in a reset graph. Consider the reset graph in Figure 5.6 and assume we have two maximal reset chains $\kappa_1 = [0] \xrightarrow{\tau_0,0} [\boldsymbol{x}] \xrightarrow{\tau_2,5} [\boldsymbol{y}] \xrightarrow{\tau_4,0} [\boldsymbol{\tau_v}]$ and $\kappa_2 = [0] \xrightarrow{\tau_0,0} [\boldsymbol{x}] \xrightarrow{\tau_1,1} [\boldsymbol{z}] \xrightarrow{\tau_3,2} [\boldsymbol{\tau_v}]$. In this case, we have $atoms_1(\kappa_1) = \{[\boldsymbol{y}], [\boldsymbol{\tau_v}]\}$ and $atoms_1(\kappa_2) = \{[\boldsymbol{z}], [\boldsymbol{\tau_v}]\}$ as there is at most one path between those nodes and the $[\boldsymbol{\tau_v}]$ node in the reset graph. The $atoms_1(\kappa)$ set has a $atoms_2(\kappa)$ set complement which contains remaining nodes with more than one path, i.e., $atoms_2(\kappa_1) = atoms_2(\kappa_2) = \{[\boldsymbol{x}]\}$. We consider only variable norms, i.e., non-source nodes, because constants and formal parameters cannot be incremented. Note, that $\mathcal{A}$ is a set of unique elements, hence $\bigcup_{\kappa \, \in \, \{\kappa_1, \kappa_2\}} atoms_1(\kappa) = \{[z], [y], [\tau_v]\}$. We can rewrite the generalised `IncrementSum` procedure with the use of the original one defined in Equation 4.2 as follows:

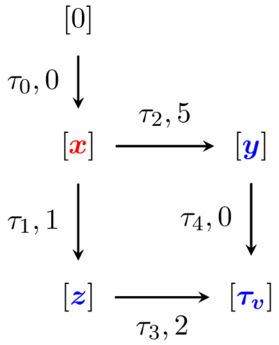


Figure 5.6: A reset graph with multiple paths between two nodes.

$$\mathtt{IncrementSum}(\mathcal{A}) = \sum_{\mathtt{a} \in \mathcal{A}} \mathtt{IncrementSum}(\mathtt{a}) \tag{5.2}$$

The modified `ResetSum` procedure is defined as follows:

$$\begin{aligned}\mathtt{ResetSum}(\tau_v) = \sum_{\kappa \, \in \, \Re(\tau_v)} & T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0) \\ & + \mathtt{IncrementSum}(atoms_2(\kappa))\end{aligned} \tag{5.3}$$

where $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$ is a generalisation of the $T\mathcal{B}$ procedure for sets of transitions. The $trn(\kappa)$ is a set of all transitions of the κ reset chain, the $in(\kappa)$ refers to the first norm of the κ reset chain, and finally the $c(\kappa)$ is a sum of all integer constants along the κ reset chain.

Algorithm 5 presents the transition bound procedure without any error or cache handling in order to remain concise. The `GetResetChains` procedure constructs a set of reset chains for the local bound norm based on a reset graph and a DCP. We discuss this procedure in more detail in Section 5.4. The rest is straightforward: we compute and add together both the `IncrementSum` and the `ResetSum` and return the final transition bound.

Algorithm 5: Simplified transition bound procedure without error handling and cache manipulation. The `IncrementSum` and `ResetSum` procedures follow Equation 5.1 and Equation 5.3, respectively.

```
Input  : DCP edge Edge with determined local bound norm
Output: A transition bound for the input edge and updated cache
Data: Atoms1 — a set of reset chain nodes (norms) with at most one path to the
      node representing the local bound norm in the corresponding reset graph
Function TransitionBound(Edge, Cache):
    if $Edge.localBound \in \mathcal{V}$ then
        ResetChains = GetResetChains(Edge.localBound, ResetGraph, DCP);
        Atoms1 = $\varnothing$;
        foreach Chain $\in$ ResetChains do
            Atoms1 = Atoms1 $\cup$ Chain.atoms1();
        return IncrementSum(Atoms1) + ResetSum(ResetChains);
    else return Edge.localBound;
```

The implementation of these two procedures as well as the $V\mathcal{B}(\mathtt{a})$ procedure closely follows the previously presented equations and involves mainly basic arithmetic over bound expressions and simplification techniques which reduce the number of terms in the final bound expression.

5.4 Construction of Reset Chains

The construction process involves generation of a reset graph which we proceed to traverse in order to find maximal reset chains. However, we also have to ensure that all reset chains are so called *optimal* which requires additional traversals of a corresponding DCP. We will cover the validation process in more detail shortly.

We will first discuss the `RG` module which represents a reset graph and the generation process. Similarly to the DCP construction, we used the parametric `Graph` module from the *OCamlgraph* library and provided an implementation for the signature `Node` and `Edge` modules. It corresponds to the reset graph nodes and edges as discussed in Section 4.3.5. Thus, the `RG.Node` module is just a wrapper for the `Exp` module which we use to represent norms. The `RG.Edge` module has a following inner record data type:

```
type t = {
    dcp_edge : DCP.E.t;
    const : IntLit.t;
}
```

The `dcp_edge` field stores a reference of the original DCP edge with the $node_{\mathtt{dst}} \leq node_{\mathtt{src}} + \mathtt{const}$ reset DC. It is necessary to keep references due to the `ResetSum` procedure which has to construct the $trn(\kappa)$ transition set for a given reset chain κ and recursively call the `TransitionBound` procedure for each of these transitions.

The straightforward process of reset graph construction is presented in Algorithm 6. We search for reset DCs of form $[x] \leq [y] + \mathtt{c}$ and use the `isReset` function to perform the syntactic inequality check of $[x]$ and $[y]$ norms. Each reset DC is transformed into a $[y] \xrightarrow{\tau, \mathtt{c}} [x]$ part of the reset graph, i.e., the $[y]$ norm becomes the source node and the $[x]$ norm becomes the destination node which signifies the assignment of y to x.

Algorithm 6: Construction of a reset graph based on edges of existing DCP. Each reset DC of form $x \leq y + \mathtt{c}$ on DCP edge e is used to create two new RG nodes where x and y represent the destination and source nodes, respectively. Both nodes are connected by an edge which stores the DCP edge e and the DC constant $\mathtt{c}$.

```
Input  : A set of DCP edges
Output: A corresponding reset graph
ResetGraph = RG.create();
foreach Edge ∈ DCP do
    foreach DC ∈ Edge.constraints do
        if DC.isReset() then
            SrcNode = ResetGraph.addNode(DC.rhsNorm);
            DstNode = ResetGraph.addNode(DC.lhsNorm);
            ResetGraph.addEdge(srcNode, {Edge, DC.const}, dstNode);
```

The subsequent construction of reset chains is however not as simple as traversing the reset graph and finding the longest possible sequences of transitions. We also have to check if a reset chain is *optimal*, i.e., maximal and *sound* at the same time and shorten the chain accordingly if it is not. A reset chain $\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \ldots \mathtt{a}_1 \xrightarrow{\tau_1, c_1} \mathtt{a}_0$ is sound, if each norm $\mathtt{a}_i$ for $1 \leq i < n$ is reset on *all* paths from the *target* location of τ_1 to the *source* location of τ_i in the corresponding DCP. The intuition is following: if we are using a reset chain κ for a norm $\mathtt{a}_0$, all the variable norms along the chain must actually be reset between any two executions of the transition τ_1, otherwise the entire sequence of resets might not occur on some execution paths of a program and therefore the chain would not be applicable on all relevant execution paths. An *optimal* reset chain is thus a sound reset chain that cannot be further extended without becoming *unsound*.

Recall the previously featured Figure 4.6. We can demonstrate that the reset chains

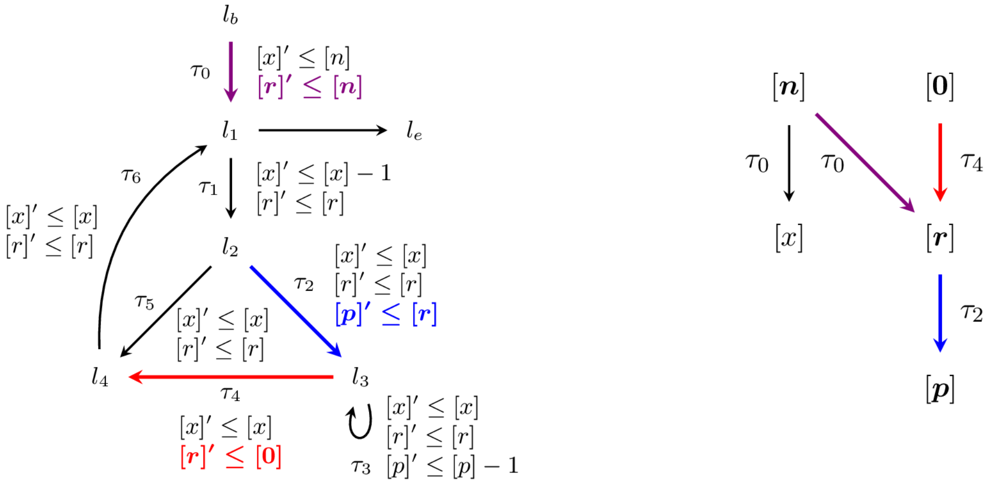


Figure 4.6: #2 (from page 25)

$\kappa_1 = [0]\xrightarrow{\tau_4,0}[r]\xrightarrow{\tau_2,0}[p]$ and $\kappa_2 = [n]\xrightarrow{\tau_0,0}[r]\xrightarrow{\tau_2,0}[p]$ are indeed *optimal*, i.e., *sound* and in this case obviously also maximal. Both of these chains have only single variable norm $[r]$ that has to be reset and $i = 1$, hence $\tau_i = \tau_1$. Therefore, $[r]$ has to be reset on all paths from the location l_3 to the location l_2, which is clearly true because the transition τ_4 with the $[r]' \leq [0]$ reset DC is executed on all possible paths. We can conclude that both reset chains are *sound* and maximal as they cannot be further extended.

The construction of optimal reset chains involves two steps. First, we find longest and possibly unsound reset chains with the recursive procedure presented in Algorithm 7. We start at the local bound node with a chain of zero length represented by an empty list and perform a depth first search in the opposite direction through the reset graph. The `incomingEdges` procedure returns a set of incoming edges in the form of a tuple (`RG.Node.t`, `RG.Edge.t`, `RG.Node.t`) where the first element is the predecessor source node and the last element is the destination node equal to the `CurrentNode`. We terminate the recursion if there are no incoming edges and return a *singleton* set with the final chain which is reversed due to the opposite direction traversal. Otherwise we create a new extended chain and recursively call the `TraverseRG` procedure for each predecessor node. Finally, we return a set of all reset chains. We do not need to maintain a set of visited nodes as with the general DFS algorithm because a reset graph of a flow-sensitive DCP is guaranteed to be *acyclic*.

Algorithm 7: Reset graph traversal and construction of *unsound* reset chains. The modified DFS algorithm traverses the input reset graph in the opposite direction, starting at the *CurrentNode*, and gradually extends the input zero length chain with each reset graph node until it reaches a source node.

```
Input   : Reset graph node CurrentNode representing the local bound norm and
          a zero length reset chain Chain
Output: A set of all longest (possibly unsound) reset chains
Data: RG — a corresponding reset graph with global scope
Function TraverseRG(CurrentNode, Chain):
    IncomingEdges = RG.incomingEdges(CurrentNode);
    if IncomingEdges = ∅ then return { Chain } ;
    else
        ChainSet = ∅;
        foreach (SrcNode, Edge, DstNode) ∈ IncomingEdges do
            ExtendedChain = Chain.append((SrcNode, Edge, DstNode));
            ChainSet = ChainSet ∪ TraverseRG(SrcNode, ExtendedChain);
        return ChainSet;
```

The second step involves validation of each obtained reset chain, i.e., we try to find the longest *sound* subsequence of each reset chain. Algorithm 8 presents a chain optimization pseudocode. We start at the first $\mathtt{a}_0$ norm of the reversed chain (local bound) and gradually extend the chain as we check for resets of each norm $\mathtt{a}_{1\leq i<n}$ on all DCP paths between locations `Origin` and `End`. The recursive `CheckPaths` procedure performs a DFS of the corresponding DCP and returns `None` if there is a path without a reset or if there are no paths at all. If this occurs, we terminate the loop and return the sound subsequence. Note,

Algorithm 8: Finding the longest *sound* subsequence of a reset chain. The algorithm gradually extends the initial zero length *sound* chain with elements from the *unsound* input chain, starting at the end and progressing towards the first element. Each norm of the unsound chain is checked for resets on all DCP paths with the `CheckPaths` procedure.

```
Input  : Unsound reset chain UnsoundChain and the origin node PathOrigin for
         the paths check, i.e., the destination node of the last transition of the
         input reset chain
Output: An optimal (longest sound) reset chain
Function OptimizeChain(UnsoundChain, PathOrigin):
    OptimalChain = [UnsoundChain[0]];
    for $i \leftarrow 1$ to UnsoundChain.length() - 1 do
        SrcNorm, Data, DstNorm = UnsoundChain[i];
        PathEnd = Data.dcpEdge.dstNode;
        PathsReset = CheckPaths(PathOrigin, $\varnothing$, DstNorm, None);
        if PathsReset = None then break ;
        else OptimalChain.prepend(UnsoundChain[i]) ;
    return OptimalChain;
```

that we *prepend* each new norm to get non-reversed final chain which starts with the $\mathtt{a}_n$ norm.

The `CheckPaths` procedure presented in Algorithm 9 performs a slightly modified traditional DFS. We start at the `PathOrigin` node and recursively traverse a DCP until we reach the `PathEnd` node or a dead end without any outgoing edges. The termination condition for a valid path contains additional check for non-empty set of visited nodes because we start at the `PathOrigin` node which can be equal to the *PathEnd* node in edge cases. The `PathReset` argument stores the information whether we already encountered a reset of the `Norm` argument on the current path. The `PathReset` argument has an optional boolean data type with three possible values in order to differentiate between three possible scenarios: a path that contains a reset, a path that does not contain a reset, and not a path. We start with a *None* value and change it to *True* when we encounter a reset of the norm. The *True* value cannot be changed and is propagated further until we reach the end of the path. The *None* value returned in cases of no path is ignored as we care only about resets on valid paths. The procedure returns *False* if it reaches the end of the path with the *None* value. However, the three possible values are encoded only to *True* or *None* at the end for more convenient checking later on. Note that it is possible that there will be no valid paths in which case the procedure naturally returns *None*. Thus, we can interpret the *None* value as *False* because it does not matter whether there were no valid paths or if there was a path without a reset. The `isReset` method also returns *None* if the `Edge` does not contain a reset of the `Norm`. We assume that `PathOrigin` and `PathEnd` variables have global scope.

5.5 Flow-sensitivity Transformation

The three step flow-sensitivity transformation follows the concept discussed in Section 4.3.6. First, we construct a VFG, then we compute SCCs and create a VFG mapping from each

Algorithm 9: Norm reset check with DFS traversal. The algorithm checks if all paths between the *PathOrigin* and *PathEnd* nodes contain a reset of the input *Norm*.

```
Input  : Path origin node Current, empty set of visited nodes Visited, and the norm
         of interest Norm.
Output: An information if the input norm was reset on all paths
Data: The PathOrigin, and PathEnd variables are inherited from the scope of
      Algorithm 8
Function CheckPaths(Current, Visited, Norm, PathReset):
    if $Current = PathEnd \land Visited \neq \varnothing$ then
        if $PathReset = None$ then return False ;
        else return PathReset ;
    else
        Outgoing = DCP.outgoingEdges(Current);
        if $Outgoing = \varnothing$ then return None ;
        else
            if $Current \neq PathOrigin$ then Visited.insert(Current) ;
            foreach $Edge \in Outgoing$ do
                if $Edge.dstNode \notin Visited \land \neg Edge.isLoopback()$ then
                    if $PathReset = None$ then PathReset = Edge.isReset(Norm) ;
                    Reset = CheckPaths(Edge.dstNode, Visited, PathReset);
                    if $Reset = True$ then PathReset = True ;
                    else if $Reset = False$ then return None ;
            return PathReset;
```

SCC to a new artificial variable, and finally we apply the mapping to variables of a DCP. Please note that all algorithms referenced in this section were moved to Appendix in order to save space.

The VFG construction process is presented in Algorithm 10. We use the *OCamlgraph* library to and provide an implementation for the signature `Node` and `Edge` modules. In this case, the inner data type of the `Node` module is the following tuple: `(Exp.t * DCP.Node.t)` where the first value is a norm and the second one is a DCP location represented by a DCP node. The `Edge` module remains empty (`unit` data type) because VFG does not store any data on edges. We iterate through all DCs of each edge and search for DCs of form $x \leq y + \mathtt{c}$, where both x and y must be non-constant norms and x can be equal to y, i.e., we process all kinds of updates involving variable norms unlike with the construction of a reset graph. The x norm is coupled with the DCP destination node to create a new destination node of the VFG and the y norm is coupled with the DCP source node to create a new source node.

Next we proceed to create the $(\mathtt{norm}, \mathtt{location}) \rightarrow \mathtt{v}$, i.e, $\mathtt{VFGNode} \rightarrow \mathtt{v}$ mapping as shown in Algorithm 11. We create a fresh artificial variable `v` for each existing SCC ζ and use the iteration index to ensure that the name of each new variable is unique. The associations between VFG nodes of each SCC ζ and fresh variables `v` are stored in the `VFGMapping` map.

Finally, we apply the VFG mapping and construct a new set of transformed DCs for each DCP edge as shown in Algorithm 12. We combine appropriate norms of each DC with nodes of the current edge and use the constructed tuples as search keys for the `VFGMapping` map. The map contains associations only for variable norms and we thus extend the mapping to constants by using the original constant norms when we fail to find an association in the map. The `LhsNorm` and `RhsNorm` variables are used to form the transformed DC which is stored in the `DCs` set. Finally, we replace the set of original constraints with the `DCs` set for each edge. Figure 5.8 shows an example of a DCP after the renaming transformation.

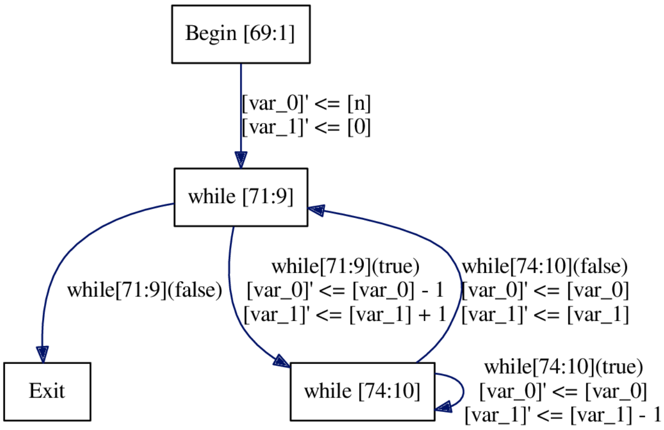


Figure 5.8: A flow-sensitive DCP obtained from the `tarjan` (4.1) example after the VFG transformation.

Chapter 6

Evaluation

This chapter experimentally evaluates our implementation of the *Looper* analyser. We first verified the correct implementation of the abstraction algorithm and the bound algorithm extended with *reset chains* and *flow-sensitivity* transformation and checked if it produced expected results on few artificial examples. Next, we evaluated *Looper* on several code examples from [12] to demonstrate its precision. Most of these examples are originally from the `SPEC CPU`®`2006` [13] benchmark and focus on real world instances of problems requiring *amortized complexity* analysis such as the aforementioned string parsing or pattern matching algorithms. Finally, we evaluated *Looper* on a test set[1] used by the Cost[2] analyser in order to get a fair comparison between both solutions and discuss limitations of either approaches.

6.1 Evaluation on the Cost Analyser Test Set

We evaluated Looper on the Cost test set to show that our analyser works well even on non-cherry-picked code examples. These examples do not require amortized reasoning and serve as a set of basic tests for our evaluation.

The complete test set contains a total of 62 functions from which we have selected 30 samples. The remaining 32 functions are mainly interprocedural tests or functions with `goto` statements which we do not currently support. Moreover, the `goto` statement was used for direct jumps into loops (or to simulate such behaviour) which we argue is uncommon as opposed to real world usage of `goto` such as for error handling purposes in Linux Kernel[3]. Additionally, four functions contained `break`, `switch` or `continue` statements which we also do not support[4]. Finally, there were 8 cases with no loops at all which we ignored.

Table 6.1 presents the comparison between results of Looper and Infer's Cost analyser for the set of 30 selected samples. Looper managed to infer the precise bound in 24 cases and imprecise bound in 3 cases. One imprecise bound was caused by insufficient *guarding* concept from Section 4.2 which failed to infer a guard necessary for determining of a local bound. Next imprecise bound was caused by a function which requires a *path sensitive* reasoning. In the last case, Looper inferred imprecise bound n instead of $\frac{n}{2}$ due to insufficient abstraction algorithm which abstracts all decreasing DCs to $x \leq x-1$. However, the original *Loopus* tool implements an extension which allows it to handle arbitrary decrements. We argue though that this imprecision is negligible.

[1] `https://github.com/facebook/infer/tree/master/infer/tests/codetoanalyze/c/performance`

[2] `https://github.com/facebook/infer/blob/master/infer/src/checkers/cost.ml`

[3] `https://koblents.com/Ches/Links/Month-Mar-2013/20-Using-Goto-in-Linux-Kernel-Code/`

[4] However, the original `Loopus` tool implements extensions which allow it to handle some of these cases

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

Table 6.1: An experimental evaluation of *Looper* on a subset of Infer's test suite and comparison with the existing COST analyser. The complete test suite contains a total of 62 functions and the subset consists of 30 selected relevant examples. *The total time was measured only on 27 functions which did not result in a crash of *Looper*.

The three remaining error cases crashed our analyser due to the complicated short-circuiting of loop conditions which generate complex native Infer CFG and our LTS construction algorithm failed to produce a valid LTS. We conclude that most of the imprecise bounds and errors were caused by the shortcomings of our immature implementation rather than by the general limitations of the adopted approach.

COST analyser managed to infer the precise bound in 27 cases and imprecise bounds in the remaining 3 cases, i.e., it was able to analyse all tests without any error. But, all imprecise bounds inferred by COST are *unsound* as it managed to infer a constant cost for a test with non-deterministic termination and for another test which does not terminate in all executions. The last unsound bound was inferred for the test which requires a *path-sensitive* reasoning. Additionally, we optimistically classified the $2 + (3 \times n) + 2 \times (2 + \max(-1, n))$ cost inferred for the test with $\frac{n}{2}$ real bound as correct due to our inability to interpret the true meaning of the $\max(-1, n)$ term. Note that COST was able to infer the correct cost of $2 + (3 \times n) + 2 \times (1 + \max(0, n))$ for a modified version with the bound of n.

The runtime comparison was conducted on a total of 27 functions which did not result in a crash of Looper. The experiment was run on a Core i5-3320M processor at 3.30 GHz running Ubuntu 16.04 with 64-bit binaries for both analysers. Infer was run in the multi-threaded batch mode which analysed all tests in one go in order to avoid the non-negligible startup overhead of Infer on individual analyses. In conclusion, our Looper was nearly 3 times faster than the current implementation of COST analyser on a subset of their test suite. The higher runtime of COST presumably mainly arises from the internal use of a relatively expensive invariant pre-analysis developed by the *Inferbo* [8] team for the purposes of their buffer overrun analysis.

6.2 Evaluation on the Loopus Test Set

The second experiment was conducted on our test suite of 8 functions which require *amortized complexity* reasoning in order to infer the precise bounds. The test suite consists of functions selected from the benchmark [12] and is publicly available at our *Bitbucket* repository[5]. Note that while this benchmark may seem small, it contains challenging examples, and hence it can accurately demonstrate the precision of one's approach.

Table 6.2 presents the results and comparison of both analysers on our test suite which was also used for evaluation of Looper in our Excel@FIT'19 [10] paper. However, we have corrected few functions where we did not include non-deterministic conditions which can affect the results of COST analyser, hence the differences in examples no. 2 and 5 (originally $5n$ and $12n$, respectively). Additionally, we have remeasured the total runtime in the batch mode and used it instead of the original per-example runtime which included

[5] https://bitbucket.org/paveon/infer-performance/src/looper-develop/examples/Loopus/

the aforementioned non-negligible startup time of Infer. This allowed us to obtain a more accurate measurement which should be closer to results in real world scenarios. Note that the real bounds of examples no. 4 and 6 are actually $n + n \times \max(n-1, 0)$ and $3n+\max(m1, m2)$, respectively. For the sake of presentation, we have simplified all bounds.

	Real Bound	Inferred bound		Total Time [s]	
		Looper	**Cost**	**Looper**	**Cost**
#1	$\boldsymbol{n}$	$\boldsymbol{2n}$	$\boldsymbol{n^2}$	5.5	10.2
#2	$\boldsymbol{2n}$	$\boldsymbol{2n}$	$\boldsymbol{n^2}$		
#3	$\boldsymbol{4n}$	$\boldsymbol{5n}$	∞		
#4	$^*\boldsymbol{n^2}$	$\boldsymbol{n^2}$	∞		
#5	$\boldsymbol{2n}$	$\boldsymbol{2n}$	∞		
#6	$^*\boldsymbol{n}$	$\boldsymbol{n}$	∞		
#7	$\boldsymbol{2n}$	$\boldsymbol{2n}$	∞		
#8	$\boldsymbol{2n}$	$\boldsymbol{2n}$	∞		

Table 6.2: An experimental evaluation of *Looper* and comparison with the existing Cost analyser on our test suite which consists of 8 functions selected from [12]. *The real bounds of examples no. 4 and 6 were simplified.

Looper managed to infer the exact precise bound in 6 cases and reasonably precise bound with the correct degree in the remaining two cases. The first example requires a path-sensitive reasoning in order to infer the precise bound n and the imprecision in example no. 3 is introduced due to the *flow-sensitivity* transformation discussed in Section 4.3.6. The transformation generates a redundant path in the corresponding reset graph and consequently a redundant reset chain which adds additional n to the total sum through the $\texttt{IncrementSum}(atoms_2(\kappa))$ term of the modified `ResetSum` procedure 5.3. However, Looper is able to infer the precise bound $4n$ with the flow-sensitivity transformation disabled.

Cost analyser failed to infer the precise bound in all cases. In 6 cases, it failed to infer at least reasonably precise bound and returned ∞. The remaining two cases produced imprecise bounds with wrong polynomial degree which were expected from an analyser without the support for amortized complexity analysis. Note that the inferred costs were simplified to presented bounds based on the *back-edge* metric used by Looper.

Chapter 7

Conclusion

In this work we extended the *scalable* abstract interpretation framework of the *Facebook Infer* tool with a new performance oriented analyser. More specifically, we wanted to focus on the somewhat lacking area of *automated complexity* and *resource bound* analysis which was not supported by Infer at the time. Our solution recasts the existing powerful intraprocedural *Loopus* [12] tool within incremental Infer.AI, allowing it to scale on a large and quickly changing codebases. The *Looper* (Loopus in Infer) is currently able to analyse both trivial programs with loops as well as moderately complex programs which require *amortized complexity* analysis. The latter is where *Looper* outperformed the existing Cost analyser currently developed by the Infer team.

Our future work will primarily focus on extending the current intra-procedural approach to scalable inter-procedural analysis which would keep a reasonable precision. Additionally, we plan to implement the remaining extensions proposed in [12], such as *path-sensitive* reasoning which should greatly improve the precision of *Looper*. Lastly, we intend to devise a mechanism similar to the one used by Cost which would track changes in the performance footprint of each function between individual program revisions.

The preliminary results of this thesis were presented at the Excel@FIT'19 [10] conference where it received an award from the expert committee.

Bibliography

[1] Ben-Amram, A. M.: Size-change Termination with Difference Constraints. *ACM Trans. Program. Lang. Syst.*. vol. 30, no. 3. May 2008: pp. 16:1–16:31. ISSN 0164-0925.

[2] Bygde, S.: *Static WCET analysis based on abstract interpretation and counting of elements*. PhD. Thesis. Mälardalen University. 2010.

[3] Calcagno, C.; Distefano, D.; O'Hearn, P.; et al.: Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of POPL'09*, vol. 44. ACM. 2009. ISBN 978-1-60558-379-2. pp. 289–300.

[4] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*. ACM. 1977. pp. 238–252.

[5] Fiedor, T.; Holík, L.; Rogalewicz, A.; et al.: From Shapes to Amortized Complexity. In *Proc. of VMCAI'18*. Springer International Publishing. 2018. ISBN 978-3-319-73721-8. pp. 205–225.

[6] Fiedor, T.; Holík, L.; Rogalewicz, A.; et al.: Ranger: A Tool for Bounds Analysis of Heap-Manipulating Programs. online. 2018.
Retrieved from:
http://www.fit.vutbr.cz/research/groups/verifit/tools/ranger/

[7] Jin, G.; Song, L.; Shi, X.; et al.: Understanding and Detecting Real-world Performance Bugs. In *Proc. of PLDI'12*. ACM. 2012. ISBN 978-1-4503-1205-9. pp. 77–88.

[8] Kwangkeun, Y.: Inferbo: Infer-based buffer overrun analyzer. online.
Retrieved from:
https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/

[9] O'Hearn, P. W.: Continuous Reasoning: Scaling the impact of formal methods. In *Proc. of LICS'18*. ACM. 2018. pp. 13–25.

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

[11] Puschner, P.; Kirner, R.; Huber, B.: Techniques to Calculate the Worst-Case Execution Time. online. 2016. Zeitanalyse von sicherheitskritischen Echtzeitsystemen. TU Wien, Institute of Computer Engineering.

Retrieved from: https://ti.tuwien.ac.at/cps/teaching/courses/wcet/slides/wcet02_static_analysis.pdf

[12] Sinn, M.: *Automated Complexity Analysis for Imperative Programs*. PhD. Thesis. Vienna University of Technology. 2016.

[13] Standard Performance Evaluation Corporation: SPEC CPU® 2006. online. Retrieved from: https://www.spec.org/cpu2006/

[14] Vojnar, T.: Lattices and Fixpoints for Symbolic Model Checking. online. 2017. Formal Analysis and Verification. BUT, Faculty of Information Technology.

[15] Vojnar, T.; Lengál, O.: Abstract Interpretation. online. 2017. Formal Analysis and Verification. BUT, Faculty of Information Technology.

Appendix A

Additional algorithms

Algorithm 10: Construction of a *Variable Flow Graph* (VFG) from a DCP. Each DC of form $x \leq y + \mathtt{c}$ where x and y are variable norms is used to create two VFG nodes represented by tuples $(\mathtt{x}, \mathtt{dst})$ and $(\mathtt{y}, \mathtt{src})$ where `src` and `dst` are source and destination nodes of the corresponding DCP edge. These nodes are connected by an empty VFG edge.

```
Input  : A set of DCP edges
Output: A corresponding VFG
VFGraph = VFG.create();
foreach Edge ∈ DCP do
    foreach DC ∈ Edge.constraints do
        if $DC.lhsNorm \in \mathcal{V} \land DC.rhsNorm \in \mathcal{V}$ then
            DstNode = VFGraph.addNode((DC.lhsNorm, Edge.dstNode));
            SrcNode = VFGraph.addNode((DC.rhsNorm, Edge.srcNode));
            VFGraph.addEdge(SrcNode, (), DstNode);
```

Algorithm 11: Construction of a VFG mapping. The input VFG is split into *strongly connected components* (SCC) and a fresh auxiliary variable `v` is created for each SCC. VFG nodes of each SCC are mapped to the corresponding fresh variable.

```
Input  : VFG VFGraph split into SCCs
Output: VFG mapping of form (norm, location) → v
SCCs = VFGraph.computeSCCs();
VFGMapping = VFG.Map.empty();
foreach $(\zeta, index) \in SCCs$ do
    FreshVariable = Exp.Lvar('var_' + index);
    foreach Node ∈ $\zeta$ do
        VFGMapping[Node] = FreshVariable;
```

Algorithm 12: DCP renaming transformation. A VFG mapping σ is used to transform all $x \leq y + \texttt{c}$ DCs on each edge e into $\sigma(x, e_{\texttt{dst}}) \leq \sigma(y, e_{\texttt{src}}) + \texttt{c}$.

```
Input  : A set of DCP edges and VFG mapping VFGMapping
Output: Flow-sensitive DCP with renamed variable norms
foreach $Edge \in DCP$ do
    DCs = $\varnothing$;
    foreach $DC \in Edge.constraints$ do
        LhsNorm = VFGMapping[(DC.lhsNorm, Edge.dstNode)];
        RhsNorm = VFGMapping[(DC.rhsNorm, Edge.srcNode)];
        if $LhsNorm = None$ then  LhsNorm = DC.lhsNorm ;
        if $RhsNorm = None$ then  RhsNorm = DC.rhsNorm ;
        DCs.insert((LhsNorm, RhsNorm, DC.const));
    Edge.constraints = DCs;
```