BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

IMPROVING PRECISION OF PROGRAM ANALYSIS IN THE 2LS FRAMEWORK

ZLEPŠENÍ PŘESNOSTI FORMÁLNÍ ANALÝZY PROGRAMŮ V NÁSTROJI 2LS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **MARTIN SMUTNÝ**
AUTOR PRÁCE

SUPERVISOR — **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2018/2019

Bachelor's Thesis Specification

Student: **Smutný Martin**
Programme: Information Technology
Title: **Improving Precision of Program Analysis in the 2LS Framework**
Category: Formal Verification

Assignment:

1. Get acquainted with the 2LS framework for formal analysis of C programs. Study existing concepts that the framework builds on, mainly template-based synthesis of invariants.
2. Study forms of invariants for several different abstract domains that are currently supported by 2LS.
3. Propose a way to analyse the computed invariants such that it is possible to determine which parts of the invariant cause an undecidability of the verification.
4. Design a method that uses the obtained information to identify variables of the original program that a successful verification depends on.
5. Implement the proposed solution in the 2LS framework such that it supports the program analysis in at least two different abstract domains.
6. Test the created solution on currently undecidable programs from the test suite of 2LS and from the International Competition on Software Verification SV-COMP 2018.

Recommended literature:

- Brain, M.; Joshi, S.; Kroening, D.; et al.: Safety Verification and Refutation by k-Invariants and k-Induction. In Proceedings of the 22nd International Static Analysis Symposium, LNCS, vol. 9291. Springer. 2015. pp. 145-161.
- Schrammel, P.; Kroening, D.: 2LS for Program Analysis - (Competition Contribution). In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 9636. Springer. 2016. pp. 905-907.
- Malík, V.: Template-based Synthesis of Heap Abstractions. Master's thesis, Brno University of Technology, Brno (2017)

Requirements for the first semester:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Malík Viktor, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 15, 2019
Approval date: November 1, 2018

Abstract

The goal of this work is to propose a way to improve precision of program analysis in the 2LS framework, based on its existing concepts, mainly template-based synthesis of invariants. 2LS is a static analysis framework for analysing C programs which relies on the use of an SMT solver and of abstract interpretation for automatic invariant inference. In a case when 2LS can not decide whether a program is correct, the proposed solution analyses the invariants computed in various abstract domains and identifies parts of the invariants that potentially cause undecidability of the verification. Using the obtained information, the designed method is able to identify variables of the original program that possibly determine whether the verification is successful. The output of our solution can be used as a feedback to indicate variables with problematic values that should be constrained. Also, it can be utilized by the 2LS developers for debugging purposes during development of new analyses. The solution has been implemented in the 2LS framework. Testing our solution on various benchmarks from the International Competition on Software Verification (SV-COMP) shows that it can identify variables that cause undecidability of the verification in more than half of the programs where the verification currently fails.

Abstrakt

Cílem této práce je navrhnout způsob vedoucí ke zvýšení přesnosti analýzy programů pomocí nástroje 2LS, založený na existujících konceptech, a to hlavně na syntézi invariant na základě šablon. 2LS je nástroj pro statickou analýzu programů napsaných v jazyce C, který využívá SMT solver a abstraktní interpretaci k automatickému odvození invariant. V případě kdy 2LS nedokáže rozhodnout zda je program správný, navrhované řešení analyzuje invarianty vypočítané v různých abstraktních doménách, a identifikuje takové části invariant, které mohou s největší pravděpodobností způsobit nejednoznačnost verifikace. Pomocí těchto získaných informací, dokáže navrhnutá metoda identifikovat proměnné původního programu, na kterých pravděpodobně závisí úspěch verifikace. Výstup tohoto řešení může posloužit jako zpětná vazba indikující proměnné, jejichž problematické hodnoty by měly být omezeny. Také může být výstup využit vývojáři 2LS pro účely debugování při vývoji nových analýz. Řešení bylo implementováno v nástroji 2LS. Na základě různých experimentů mezinárodní soutěže ve verifikaci programů SV-COMP, dokáže řešení identifikovat proměnné způsobující nejednoznačnost verifikace ve více než polovině programů, na kterých verifikace momentálně selhává.

Keywords

formal verification, program analysis, static analysis, 2LS framework, abstract interpretation, invariant, SSA form, abstract domain, template-based analysis

Klíčová slova

formální verifikace, analýza programů, statická analýza, 2LS framework, abstraktní interpretace, invariant, SSA forma, abstraktní doména, analýza na základě šablon

Reference

SMUTNÝ, Martin. *Improving Precision of Program Analysis in the 2LS Framework.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

2LS je verifikační nástroj kombinující řadu známých technik pro analýzu programů takovým způsobem, který umožuje jejich současný běh a efektivní výměnu informací. Díky tomu je 2LS schopen dokázat či vyvrátit škálu různých vlastností programu. Verifikační proces 2LS je založen na výpočtu invariant cyklů a funkcí zdrojového programu s využitím SMT solveru. Tyto invarianty jsou poté použity k dokazování různých vlastností programu.

V případě analýzy complexních programů, verifikační program často nedokáže rozhodnout zda je program správný. V kontextu 2LS je tato nejednoznačnost často způsobena nepřesnými částmi vypočtených invariant. Takové vypočtené invarianty mohou být dlouhé i několik desítek řádků a jsou obtížně čitelné, protože obsahují symbolické proměnné na místo proměnných původního programu.

Vzhledem k výše zmíněným problémům je navrhnuto řešení v nástroji 2LS, které je schopno identifikovat části verifikovaného programu, které jsou určitým způsobem problematické pro daný verifikační nástroj. K symbolickým proměnným identifikovaných nepřesných částí vypočtených invariant, je tak možno nalézt odpovídající proměnné v původním programu. Poté může uživatel upřesnit hodnoty těchto problematických proměnných (s použitím tzv. “assume” konstrukcí) v analyzovaném programu, a tímto tak dopomoci k úspěšné verifikaci programu. Zároveň, toto řešení může být užitečné i pro vývojáře nástroje 2LS s debugováním v průběhu přidávání nových funkcí.

Pro naše řešení jsou narženy dvě metody využívající přístupu abstraktní interpretace v 2LS k výpočtu invariant. K verifikaci programů využívá 2LS koncept indukčních invariant, ale protože je výpočet takových invariant výpočetně drahý používá se tzv. šablon. Šablony představují redukci tohoto problému, který je vyjádřitelný ve druhém řádu predikátové logiky, na problém který je řešitelný pomocí iterativní aplikace solveru prvního řádu. Přistup k verifikaci je takto zjednodušen na výpočet invariant cyklů za pomoci paramterizovaných šablon, kde SMT solver je využíván právě k nalezení vhodných hodnot těchto parametrů. Takto jsou všechny abstraktní domény v 2LS založeny na výpočtu invariant cyklů pomocí šablon, tzv. abstraktní domény založené na šablonách.

První navržená metoda slouží tedy k identifikaci nepřesných výrazů šablon v intervalové abstraktní doméně a heap doméně. Tyto výrazy odpovídají symbolickým proměnným v 2LS, jejíchž hodnoty jsou omezeny pomocí vypočtených invariant cyklu. Účelem je tedy, po vypočtení hodnot parametrů šablon, identifikovat v dané abstraktní doméně ty symbolické proměnné, které mají odpovídající hodnotu parametru označenou jako nepřesnou v dané abstraktní doméně. Identifikovali jsme tyto nepřesné hodnoty, odpovídající vyjádření hodnot suprema v dříve zmíněných abstraktních doménách. Tímto pomocí navržených algoritmů, specificky pro každou ze zmíněných domén, dojde k určení výrazů šablon, které mají hodnoty parametrů označené jako nepřesné v daných doménách.

Ze získaných informací předchozích metod (jedná se o jména symbolických proměnných) jsme schopni určit odpovídající zdrojové informance za pomocí vnitřích reprezentací programů používaných v 2LS. S využitím jmenných konvencí jsme schopni obdržet původní jména proměnných, jejich lokaci ve vnitřní reprezentaci a další informace, které jsou součástí zpětné vazby řešení. Za pomocí nyní získaných elementů ve vnitřní reprezentaci dokážeme zjistit lokaci elementu začátku a cyklu, alokace proměnných, atd. S pomocí uložených informací o zdrojovém programu v instrukcích control-flow grafu reprezentace programu, jsme schopni tak lokalizovat odpovídající symbolické proměnné v původním programu a určit tak řádky cyklů, na kterých identifikovaná nepřesnost nastává.

Účelem této práce byl návrh řešení sloužící k identifikaci nepřesných částí invariant vypočtených v nástroji 2LS, které mohou s největší pravděpodobností určit zda-li je veri-

fikace úspěšná. Toto řešení identifikuje proměnné původního programu, které mají nepřesné hodnoty a cykly, ve kterých tato nepřesnost nastává. V případě dynamických objektů (allokovaných na haldě) je schopno určit řádek alokace v daném kódu, a navíc případě strukturovaných objektů také původní jména jejich polí (ve struktuře), které drží nepřesné hodnoty.

Byly navrženy dvě metody vedoucí k identifikaci těchto proměnných: (1) metoda pro identifikaci nepřesných částí vypočtených invariant v různých abstraktních doménách a (2) metoda pro lokalizaci proměnných odpovídající těmto částem s použitím vnitřní reprezentace programů napsaných v jazyce C využívané v 2LS.

Řešení bylo implementováno v nástroji 2LS. Metoda pro identifikaci nepřesných částí invariant může být implementována v jakékoliv existující doméně 2LS. Momentálně je tato metoda implementována v abstraktní intervalové doméně, v heap doméně, v jejich kombinaci — heap interval doméně a nakonec v rozšíření — heap intervalové doméně se symbolickými cestami. Metoda pro lokalizaci byla integrována do modulu SSA analyzer v 2LS.

Bylo ukázáno na příkladech, že toto řešení je schopno identifikovat nepřesné proměnné invariant a lokalizovat ty části analyzovaného programu, ve kterých tyto nepřesnosti nastávájí. Navíc bylo ukázáno na příkladu, že zpětná vazba tohoto řešení je užitečná pro uživatele a napomáhá jemu ke správné verifikaci programu. Provedli jsme řadu testů z testové sady SV-COMP 2017 a ve většině z programů, na kterých verifikace selhává, je schopno toto navržené řešení lokalizovat ty části invariant, které potencionálně způsobují nejednoznačnost verifikace.

V budoucnosti bychom rádi oficiálně integrovali toto rozšíření do nástroje 2LS. Toto řešení by mohlo být hlavně užitečné pro vývojáře nástroje 2LS při vývoji nových funkcí.

Improving Precision of Program Analysis in the 2LS Framework

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Viktor Malík. The supplementary information was also provided by Ing. Viktor Malík. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Martin Smutný
May 15, 2019

Acknowledgements

I would like to thank my supervisor Viktor Malík for his patience, support, constructive revisions of this work and consultations that gave me both theoretical and practical insight into this matter.

Contents

Chapter 1

Introduction

Software verification tools are mostly designed to target a specific problem domain and focus on certain properties defined by the type of the tool. They prove to be good at their problem domain, but are necessarily limited outside their primary focus area. Most non-expert users tend to use them rather for debugging, than to formally prove correctness. Users who need absolute verification invest in customizing a tool that is more suited to their product. One of the solutions that aims to cover this demand — to offer a free wide range tool for proving various classes of program properties, even for non-experts, is the 2LS verification framework.

2LS combines a range of well-known program analysis techniques in a way that enables them to work simultaneously and efficiently exchange information. This enables 2LS to prove and refute various different classes of program properties, while being more user-friendly. The verification process of 2LS is based on computing invariants of loops and of functions of the source program by utilizing an SMT solver. These invariants are then used to reason about various program properties.

When analysing complex programs, verifiers often cannot decide whether the programs are correct. For 2LS, this undecidability is usually caused by imprecise parts of computed invariants. Such computed invariants might be even tens of lines long and are not very readable, since they consist of symbolic variables instead of the original program variables.

With respect to the above, we propose a solution in the 2LS framework, to identify parts of the verified programs that cause problems to the verifier. By identifying imprecise parts of the computed invariants, it is possible to map the symbolic variables back to the original program. Then, the user can refine values of problematic variables (e.g., using special "assume" constructions) in the analysed program, which can help the verification to succeed. At the same time, this may help the developers of the 2LS framework with debugging when e.g. adding new features. A motivation example is presented in Section 1.1 below.

There are many different literature approaches to inferring invariants, many of which are not efficient enough e.g., to infer strong loop invariants. Because of 2LS' template-based approach to computing invariants, it is not only efficient in producing all kinds of invariants but also allows us to design a viable solution for our problem. In Chapter 2 we introduce basic concepts of program verification and the current state of the art of 2LS.

Implementation of our solution in the 2LS framework requires a specifically designed algorithm to work with its computational approach of using various classes of templates over its representations of analysed programs. The designed solution is described in Chapter 3.

The proposed solution was implemented in the 2LS framework. The solution is able to identify and locate variables that cause the undecidability of the verification inside loop invariants, their lines of definition, and the loops in which the imprecision is caused. Based on various benchmarks taken from the 2LS regression testing suite and from the International Competition on Software Verification (SV-COMP), our solution can identify such variables in more than half of currently inconclusive programs.

Implementation details are presented in Chapter 4, examples of verification of sample C programs with our extension, as well as results of various tests are presented in Chapter 5.

1.1 Motivation example

In this section we present a motivational use-case behind this work using a simple example in the form of a C program given in Figure 1.1.

Even when analysing, at the first sight, a fairly trivial program, as in the case of the program in Figure 1.1, the verification of the assertion at line 8 might be inconclusive. Since 2LS computes invariants of loops and of functions to prove program properties, such undecidability might be caused by weak computed invariants. For loop invariant to be classified as weak, the values of its parts must be imprecise or unconstrained, as is the case with the program in Figure 1.1.

The dependencies between values of variables x and i, and y and i where variable i determines the number of loop iterations (lines 5-9), are not recognized by 2LS. Therefore "infinite" loop iterations are expected, resulting in a possibility of overflows for both variables x and y. The assertion at line 8 is evaluated to unknown, hence the result of the verification is inconclusive.

```
1 void main() {
2   int x = 1000;
3   int y = x-1;
4
5   for (int i = 0; i < 100; i++) {
6     x++;
7     y--;
8     assert(y <= x);
9   }
10 }
```

Figure 1.1: Inconclusive program in C

Whenever this kind of *imprecision* occurs, it would be convenient for the verifier to provide the user with a feedback indicating which variables inside the generated invariant have unconstrained values and are thus a potential cause of undecidability of the verification. Then the user himself could refine the values of such problematic variables (e.g. using special "assume" constructions) in a fairly obvious way, e.g. variable x can never be less than 1000 inside the loop between lines 5-9, thus helping the verification to succeed. In Chapter 5 we demonstrate on this example that our extension is able to identify such variables and thereby provide a viable feedback.

Chapter 2

Program Analysis in the 2LS Framework

In this chapter, the 2LS framework is introduced alongside with basic concepts of techniques it uses for program analysis. The purpose of this work is to devise a way to analyze computed invariants in order to determine parts that cause an undecidability of the verification. Therefore in this chapter we describe the concepts of 2LS that are needed to understand the solution we propose.

2LS is a free static analysis and verification tool for analyzing sequential C programs. It is built upon the *CPROVER* verification framework [1] and allows verifying memory safety, user-specified assertions, and termination properties [13, 8]. 2LS was developed by Peter Schrammel and Daniel Kroening at the University of Oxford, UK. Currently, it is maintained by Peter Schrammel and company *Diffblue Ltd*, which is a spin-off of the University of Oxford [7].

The main strength of 2LS lies in a unique combination of multiple software verification techniques together, resulting in a single unified algorithm, called *kIkI*. This algorithm combines bounded model checking, k-induction, and abstract interpretation into a scalable framework that allows these techniques to interact and reinforce each other in the process of verification [3]. In this work, we will mainly focus on the *abstract interpretation* part of 2LS which is used to compute so-called *inductive invariants* of loops and of functions of the source program. These invariants are inferred using an SMT solver-based approach and are then used to reason about various program properties. In Section 2.1 we introduce the abstract interpretation framework and in Section 2.2 we describe the main concepts of invariant inference and how inductive invariants are used to verify programs in 2LS.

2LS uses so-called *templates* and a specific form of abstract interpretation to compute invariants efficiently using an SMT solver in various *abstract domains*. We describe the approach of computing invariants over templates in Section 2.3 and in Section 2.5 we present two of the most important abstract domains currently used in 2LS.

Finally, as a simplification 2LS views the analysed programs as transition systems, but in order to be more efficient with the solver-based approach the internal representation of a program is converted to *single static assignment form* (shortly, SSA) instead. Internal representation as well as the concept of SSA and conversion into it, are described in Section 2.4.

2.1 Abstract Interpretation

Abstract interpretation is a static analysis framework used to approximate concrete program semantics in order to verify various dynamic program properties at compile time [5]. The approach of abstract interpretation is, in order to answer concrete questions about certain program property, one may sometimes answer it using a simpler abstract question. When doing an abstract evaluation of a program, computations are described in a universe of abstract objects, so that the results of abstract evaluation give some information on the actual computations. The abstract evaluation gives a summary of some facets of the actual executions of a program. Generally, this summary is easily obtained but may be inaccurate.

A *concrete domain* P denotes a set of concrete program states. An *abstract domain* Q denotes a set of *abstract values*. When doing abstract evaluation of a program, abstract values in abstract domain Q are associated with elements from the concrete domain P. The choice of abstract values depends upon the specific dynamic properties we want to analyse in a program. Provided that this choice of abstract values satisfies the general framework specifications, correctness and termination of abstract interpretation are guaranteed [4].

An abstract interpretation I of a program P with the instruction set `Instr` is a tuple [5]:

$$I = (Q, \circ, \sqsubseteq, \top, \bot, \tau) \tag{2.1}$$

where

- Q is the *abstract domain* along with well-defined *abstraction* and *concretisation functions*:
  - $\alpha : P \to Q$ is the abstraction function that defines the correspondence between a set of concrete values and an abstract value from the abstract domain,
  - $\gamma : Q \to P$ is the concretisation function that defines the correspondence between an abstract value and a concrete value of the concrete domain. Then $\gamma(q)$ denotes a concrete value represented by $q \in Q$ and $\alpha(p)$ denotes the most precise abstract value in Q whose concretisation contains $p \in P$.
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\circ : Q \times Q \to Q$ is the *join operator*, $(Q, \circ, \top)$ is a complete semilattice,
- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \circ y = y$,
- $\tau : \texttt{Instr} \times Q \to Q$ defines the *abstract transformers* for particular instructions.

Now, let us view source program as a transition system $T = (S, S_i, t)$, where S is a set of program states, S_i is the set of initial states and $t \subseteq S \times S$ is a *transition relation* between a program state (defined in the following subsection) and its possible successors. We denote S_r as the set of *all reachable program states* as the *least fixed point* of t starting from S_i after i execution steps:

$$S_r = \bigcup_{i \in N} t^i(S_i) \tag{2.2}$$

The least fixed point of the transition relation in concrete domain is generally not computable, therefore abstract interpretation computes an over-approximation of the program

reachable states by computing the fixed point of τ in the abstract domain. The fixed point of the abstract semantics provides a useful information about a program property, but it may not always hold in the concrete domain. Therefore we want the abstract interpretation to be sound — whenever a property holds for the computed abstract semantics then it always holds for the set of all reachable program states as well [6].

For abstract interpretation to be sound, an abstract value must describe at least all reachable concrete states in a given program location. This can be guaranteed using *Galois connection* between the concrete and abstract domains.

Galois connection is a quadruple $\pi = (P, \alpha, \gamma, Q)$ such that [9]:

- $P = \langle P, \leq \rangle$ and $Q = \langle Q, \sqsubseteq \rangle$ are partially ordered sets,
- $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q \tag{2.3}$$

Because abstract interpretation computes an over-approximation of the source program reachable states (concrete semantics), it must always be sound, but not complete, hence may generate, so-called *false positives* (i.e., when a property does not hold for the computed abstract semantics, but it may hold for the set of all reachable program states). In order to mitigate the number of the false positives, one may, for example, compute the abstract semantics in a more precise abstract domain.

2.2 Verification Using Inductive Invariants

In this section we describe the approach to program verification using *inductive invariants* which is a problem expressible in existential fragment of second-order logic. Due to the least fixed point being difficult to compute, as described in the previous section, the main task of abstract interpretation in 2LS is the *inference of inductive invariants*.

As in the previous section, the analysed programs are viewed as *symbolic transition systems*. A *program state* describes a logical interpretation of logic variables, which correspond to each program variable and the program counter. We can describe a set of states using a formula — states in the set are models of the formula. Given a vector of variables $\boldsymbol{x}$, we denote the initial program states as a predicate $Init(\boldsymbol{x})$. A *transition relation* between a program state and its possible successors is described using a formula $Trans(\boldsymbol{x}, \boldsymbol{x'})$. From these, a set of all reachable states can be determined as the least fixed point of the transition relation starting from the states described by the predicate $Init(\boldsymbol{x})$. Since computing such predicate that describes such a set is difficult, therefore not practical, instead *inductive invariant* is used. Predicate Inv is then an inductive invariant if it has the following property:

$$\forall \boldsymbol{x}, \boldsymbol{x'} \,.\, (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x'}) \implies Inv(\boldsymbol{x'})). \tag{2.4}$$

An inductive invariant describes a fixed point of the transition relation, but is not necessarily the least one, nor it is guaranteed to include $Init(\boldsymbol{x})$. Some invariants are not sufficient enough to be used for proving any properties, such as predicate *true* that describes the complete state space. From an inductive invariant, it is possible to find *loop invariants* and *function summaries* by projecting it on to a subset of variables $\boldsymbol{x}$.

Proving system safety can be often simplified to showing that the reachable states do not intersect with the set of error states denoted as $Err(\boldsymbol{x})$. Using existential second-order

quantification ($\exists_2$), this can be formalised as [3]:

$$\begin{aligned}\exists_2 Inv\,.\,\forall \boldsymbol{x}, \boldsymbol{x'}\,.\,(Init(\boldsymbol{x}) \implies Inv(\boldsymbol{x})) \wedge \\ (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x'}) \implies Inv(\boldsymbol{x'})) \wedge \\ (Inv(\boldsymbol{x}) \implies \neg Err(\boldsymbol{x})).\end{aligned} \tag{2.5}$$

Even though computing inductive invariants is simpler than computing the least fixed point of the transition relation, it is still a challenging problem, especially under the concrete semantics. Therefore, 2LS computes inductive invariants in various abstract domains using a rather specific form of abstract interpretation.

2.3 Template-based Synthesis of Invariants

In order for Formula 2.5 to be solved directly by a solver, it would need to handle second-order logic quantification. Since there is currently no efficient second-order solver available, the problem is thus reduced to a problem solvable by iterative application of a first-order solver. The reduction is realized by restricting the form of the inductive invariant Inv to $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$, where $\mathcal{T}$ is a fixed expression over program variables $\boldsymbol{x}$ (referred to as *template*) and template parameters $\boldsymbol{\delta}$. This restriction is comparable to choosing an abstract domain in abstract interpretation, i.e. a template captures only those properties of the program state space that are relevant for the analysis. Using this restriction in the form of templates, the second-order search for an invariant is turned into a first-order search for the template parameters [10]:

$$\begin{aligned}\exists \boldsymbol{\delta}\,.\,\forall \boldsymbol{x}, \boldsymbol{x'}\,.\,(Init(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \wedge \\ (\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x'}) \implies \mathcal{T}(\boldsymbol{x'}, \boldsymbol{\delta}))\end{aligned} \tag{2.6}$$

The problem now being expressible in the first-order logic contains quantifier alternation, which is not well handled by the current SMT solvers. Therefore, the formula is solved by iteratively checking the negated Formula 2.6 (to turn $\forall$ into $\exists$) for different choices of constants $\boldsymbol{d}$ as candidates for template parameters $\boldsymbol{\delta}$ [3]. For a value $\boldsymbol{d}$, the template formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ is an invariant if and only if the following formula is unsatisfiable:

$$\begin{aligned}\exists \boldsymbol{x}, \boldsymbol{x'}\,.\,\neg(Init(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \vee \\ \neg(\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x'}) \implies \mathcal{T}(\boldsymbol{x'}, \boldsymbol{d}))\end{aligned} \tag{2.7}$$

Constant $\boldsymbol{d}$ corresponds to an abstract value in abstract interpretation and represents, i.e. *concretises to*, the set of all program states $\boldsymbol{x}$ that satisfy the formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$. An abstract value $\bot$, representing the infimum of the abstract domain denotes the empty set ($\mathcal{T}(\boldsymbol{x}, \bot) \equiv false$), and the supremum $\top$ denotes the whole state space ($\mathcal{T}(\boldsymbol{x}, \top) \equiv true$).

The concretisation function γ is same for each abstract domain:

$$\gamma(\boldsymbol{d}) = \{\boldsymbol{x} \,|\, \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv true\}. \tag{2.8}$$

To get the most precise abstract value representing a given concrete program state $\boldsymbol{x}$, the abstraction function is thus:

$$\alpha(\boldsymbol{x}) = \min(\boldsymbol{d}) \text{ such that } \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv true. \tag{2.9}$$

Since the abstract domain forms a complete lattice, the minimum value $\boldsymbol{d}$ is guaranteed to exist.

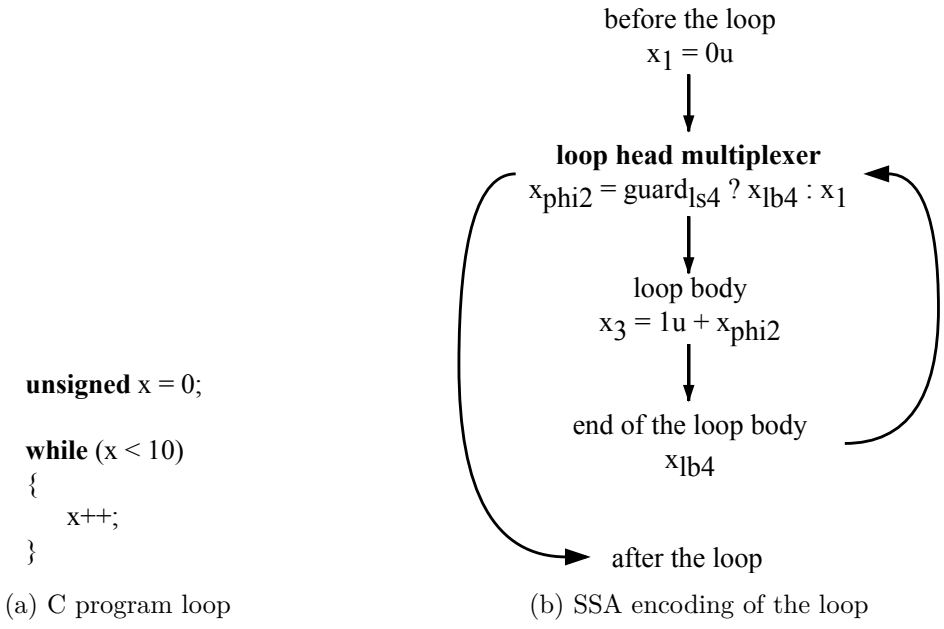
The quantifier-free version of Formula 2.7 is used for the invariant inference, where the value of $\boldsymbol{d}$ is initialized to $\bot$ and the formula is iteratively solved by an SMT solver. If the formula is unsatisfiable, then an invariant has been found. If not, then a model of satisfiability is returned by the solver, representing a counterexample to the current instance of the template being an invariant. The current value of the template parameter $\boldsymbol{d}$ is then refined by joining with the obtained model using a domain-specific join operator $\circ$ [10].

2.4 Program Encoding in Single Static Assignment Form

An essential part of this work is to map the symbolic variables used in an invariant onto actual variables of the original program. Therefore, in this section, we present how programs are represented in the single static assignment form in the 2LS framework.

Instead of representing the analysed program as a transition system (as mentioned in the previous section), it is more efficient to translate it into the *single static assignment form* or SSA. For acyclic programs, the SSA is a formula that exactly represents the strongest post condition of running the program. Generally, the SSA form is an intermediate program representation that satisfies the property that each variable is assigned to only once. For each variable x at each program location i where the value of x is modified, a new copy of x denoted x_i is introduced.

In order to reason about the abstractions of a program with a solver, 2LS extends the concept of SSA by over-approximation of the loops [3]. Figure 2.1 shows the control flow expressed by SSA. In order to track the control flow of the program, special variables called *guards* are used. For each program location i , a Boolean variable $guard_i$ is introduced. Its value represents the reachability of the program location.



(a) C program loop

(b) SSA encoding of the loop

Figure 2.1: Control flow expressed in the SSA encoding of the C loop in 2LS

The over-approximation of program loops can be seen as follows. The program path coming from *before the loop* joins at the *loop head* with the path coming from the *end of the loop body* (assuming that all paths before and within the loop are reachable and thus join). Moreover, in Figure 2.1, the loop has been cut at the end of its body: instead of passing the version of the variable x from the end of the loop body (x_3) to the loop head, a free "loop-back" variable x_{lb4} is introduced and passed, describing the value of the last assignment to the variable x. The *havocing* of the path coming from the end of the loop body represents the acyclicity of the SSA. The value of each variable x at the loop-head is then represented using a so-called *phi variable* — x_{phi2}, whose value is defined by a nondeterministic choice between the value coming from before the loop — x_1, and the loop-back value coming from the end of the loop — x_{lb4}, thus representing a join of the value coming from before the loop and from the end of the loop body. Moreover, at the loop head, a free Boolean "loop select" variable $guard_{ls4}$ is introduced, having an unconstrained value in order to illustrate the non-deterministic choice.

Precision can be improved by constraining the value of the loop-back variable x_{lb4} by means of a *loop invariant*, which is inferred during the analysis. Any property in a given abstract domain that holds at the loop entry (x_1) and at the end of the body (x_3), can be also assumed to hold on the feedback variable x_{lb4} [3].

As described in the previous Section 2.3, 2LS uses templates, which are fixed quantifier-free first-order logic formulas describing the desired property of the program, to efficiently compute loop invariants of the analysed program. As mentioned in Section 2.2 about the concept of invariants, invariants abstract the set of reachable program states (expressed in Formula 2.4). Loop invariant is one of the abstractions based on the concept of invariants. Loop invariant $Inv(\boldsymbol{x}_{loop})$ is obtained by projecting the invariant on to a subset of variables $\boldsymbol{x}_{loop} \subseteq \boldsymbol{x}$ containing the loop-back variables of a loop. Therefore loop invariants are used to constrain values of loop-back variables.

2.4.1 Example of a Program Conversion into SSA Form

Being built upon the CPROVER infrastructure, 2LS uses C parser to parse a C program into an internal representation called *GOTO program*. GOTO programs are control flow graphs [12]. GOTO programs store some metadata about the given source program along with instructions, particularly the original file name and line numbers from which the instructions were generated. This is crucial for the localization of symbolic variables in the original program. Since in the verification process 2LS uses the SSA form, the GOTO program representation is transformed into the SSA form. The transformation is realized in regard to the specific modifications of 2LS outlined previously in this section. The transformation is done in following points:

- Each assignment to a variable x at location i introduces a new version of the variable denoted as x_i which is used at the left-hand side of the assignment. Each used variable on the right-hand side of the assignment is renamed to their last introduced versions.

- For every modified variable inside every conditional statement and loop, a phi node is introduced. The choice between two values in the phi node is made on a basis of the branch condition (in case of a conditional), and on a basis of a free boolean variable (in case of a loop), as described in the introduction of this section.

- A guard variable is introduced for every entry location of each basic block from GOTO program. Guard variables express the reachability of the given basic block through conditions.
- Function calls are replaced by the over-approximating function placeholders.
- Typical operations manipulating with dynamically allocated objects on the heap are specifically represented in the SSA. Each instantiation of a new heap object using `malloc` function is replaced with instantiation of a new *abstract dynamic object*. Then every operation with memory using pointers (read, write or load and store in case of dynamic data structures) is handled over the abstractions of the concrete dynamic objects [10].

```
1  void main()
2  {
3    int x = 1000;
4    int y = x-1;
5
6
7    for (int i = 0; i < 100; i++)
8    {
9
10
11
12
13     x++;
14     y--;
15
16     assert(y <= x);
17
18
19   }
20
21 }
```

(a) The C program

```
1
2  guard_0 = TRUE
3  x_1 = 1000
4  y_3 = 999
5
6  i_5 = 0
7  guard_6 = guard_0
8  x_phi6 = (guard_ls12 ? x_lb12 : x_1)
9  y_phi6 = (guard_ls12 ? y_lb12 : y_3)
10 i_phi6 = (guard_ls12 ? i_lb12 : i_5)
11 guard_7 = !(i_phi6 >= 100) && guard_6
12
13 x_7 = 1 + x_phi6
14 y_8 = -1 + y_phi6
15
16 x_7 >= y_8 || !guard_7
17 i_11 = 1 + i_phi6
18 guard_11 = x_7 >= y_8 && guard_7
19 // *12 loop back to location 6
20 guard_13 = i_phi6 >= 100 && guard_6
21
```

(b) The corresponding SSA

Figure 2.2: Conversion of a C program in to SSA form in 2LS

We illustrate the conversion on the given example in Figure 2.2 [3]. The reachability of the entry location of the program is expressed by the guard variable *guard_0* which is set to *true*, since the entry location is always reachable. Definition of both variables x and y is at lines 3-4 (in the SSA form) along with the definition of the loop control variable i at line 6. The head of the loop contains three *phi nodes* for each of the modified variables inside the loop. The entry of the loop-head is guarded by the variable *guard_6*. Since the loop-head is always reachable, its value is set to the value of *guard_0*. The guard *guard_7* at line 11 expresses that the loop body is only reachable if the negation of the loop condition is false (i.e. the loop condition is true) and the entry of the loop-head (*guard_6*) is reachable. At lines 13-14, with value assignments, new copies of the variables x and y are introduced.

The assertion is expressed by the condition at line 16 and at line 18 the variable *guard_11* requires the condition to be true whenever the body of the loop is reached (*guard_7*). At the end of the loop body, at line 17 the loop control variable i gets updated, introducing a new copy. Finally, *guard_13* at line 20 expresses that the location after the loop is reachable when the loop condition is false and the entry of the loop-head was reached.

2.5 Template-based Abstract Domains

In this section we describe the two most used abstract domains along with brief mention of combination of domains in the 2LS framework. The purpose of this section is to build a theory basis for design of our method used for identification of imprecise variables inside invariants computed in various abstract domains.

In the previous sections, we described the verification approach of 2LS, which consists of computing inductive invariants — a problem expressible in existential second-order logic with quantification, which is then reduced to a problem solvable by an iterative application of a first-order solver. The verification approach is then reduced to computing loop invariants using parametrised templates while utilizing an SMT solver to find suitable values of the template parameters.

In 2LS, all abstract domains, in relation to its unique computational approach, are based on templates. This has many benefits e.g., enables easier combination with other domains to infer various program properties. Each domain has a so-called *template form* which describes the program property that is being analysed. Templates of the domains presented in the following subsections have a form of a conjunction of formulae called *template rows*. Since templates are used to efficiently compute loop invariants which constrain the values of the loop-back variables, each template row corresponds to an SSA loop-back variable x_{lb}. Additionally, the template row has a row parameter, called *abstract row value* or *template row value*. This parameter is computed during invariant inference and represents an abstraction of the concrete value of the variable x_{lb}. A loop invariant is then obtained by projecting the template rows and its corresponding template row values on to a subset of variables containing only the loop-back variables of the given loop [3].

In the following subsections we describe various abstract domains, due to having computation loop based on templates, sometimes labelled as *template abstract domains*. In Subsection 2.5.1 we describe the *abstract polyhedra domain*, in Subsection 2.5.2 we describe the *heap shape domain* and lastly we mention combinations of domains in Subsection 2.5.3.

2.5.1 Abstract Polyhedra Domain

One of the most general form of linear invariant generation is *polyhedra analysis*. The analysis is performed in the abstract polyhedra domain, a domain used for analysis of numerical variables using a system of linear inequalities representing numerical values of variables. Due to its worst-case exponential time and space complexity limitations, a more restricted domains are used instead [11].

In 2LS, there is a class of templates called *template polyhedra domain*, used for analysis of numerical variables. The base template has a form:

$$\mathcal{T} = (\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{\delta}) \tag{2.10}$$

where $\boldsymbol{A}$ is a matrix of fixed coefficients. Each r^{th} *row* of the template is constrained by the r^{th} row of the matrix.

Subclasses of the template polyhedra domain correspond to various known abstract domains, each having either a template working with a single variable x or with a pair of variables x and y over a *pair of template rows*:

- *Intervals* uses the interval template: $\pm x \leq c$,
- *Zones* or *differences* uses the interval and difference template: $x - y \leq c$, $y - x \leq c$,
- *Octagons* uses the interval, difference, and sum templates: $x + y \leq c$, $-x - y \leq c$,

In these template expressions, each variable in the set of program variables $\boldsymbol{x}$ is represented using *bit-vectors* with variable bit-width in order to avoid arithmetic under- and overflows. The $\top$ value corresponds to the respective maximum values in the type of the program variables, whereas the $\bot$ is encoded as a special symbol [3].

Example. We want to express that variable x (of signed integer type) lies in an interval $[2, 10]$ using the intervals template. Then the template, as a conjunction of its template rows and the computed invariant constraining that variable looks like:

$$\mathcal{T} = \mathcal{T}_0(x, 10) \wedge \mathcal{T}_1(-x, -2) \qquad (x \leq 10) \wedge (-x \leq -2)$$

(a) Resulting intervals template (b) Computed invariant

Figure 2.3: Variable x of signed integer type lies in interval $[2, 10]$, expressed using the intervals template

2.5.2 Abstract Shape Domain

Abstract shape domain or abstract heap domain in 2LS is used by *heap manipulation analysis* for modeling the shape of the heap — describes the shape properties of the program heap. The shape analysis in 2LS focuses on describing the *reachable* shape of the heap, mainly analysis of the shape of dynamic data structures like singly and doubly-linked lists. The reachable shape of the heap is described using a set of memory objects a pointer can dereference into.

Shape analysis assumes that the source programs are defined over a finite set of static and dynamic memory objects of various types:

- Var: a finite set of *static memory objects* simply corresponding to program variables.
- $PVar \subseteq Var$: a set of *pointer-typed* variables.
- $SVar \subseteq Var$: a set of *structure-typed* variables, where $PVar \cap SVar = \emptyset$.
- Fld: a finite set of *fields* of structure-typed objects in the given program (corresponds to fields in a C data structure, defined the using keyword `struct`).
- $PFld \subseteq Fld$: a set of all *pointer-typed fields*.

Dynamic memory objects, i.e. memory objects allocated on the heap using `malloc` function, are represented using a set of *abstract dynamic objects* AO, where $Var \cap AO = \emptyset$. Dynamic objects may be allocated in a loop at one program location, therefore abstract

dynamic objects represent a set of all concrete dynamic objects allocated at one *allocation site*. Dynamic objects may also be of structure type composed of fields, therefore elements of the set $AO \times Fld$ represent abstractions of the appropriate fields.

The set of all objects of a program abstraction is therefore $Obj = AO \cup Var$. Then the set of all *pointers* of a given program abstraction is defined as:

$$Ptr = PVar \cup ((SVar \cup Obj) \times PFld) \tag{2.11}$$

Pointers hold *symbolic address* of objects (or the special value `null`) from the set of all addresses $Addr$ defined as:

$$Addr = \{\&o \mid o \in Obj\} \cup \{null\} \tag{2.12}$$

where $\& - operator$ represents the access to the object's address.

As already mentioned, templates are used to compute loop invariants of the analysed program, which describe properties that hold for some program variables at the end of the body of loops after each iteration. Since there is only one loop-back pointer variable for each pointer variable in each loop, the abstract shape domain is limited to the set of all *loop-back pointers* $Ptr^{lb} = Ptr \times L$, where L is the set of all program loops. Elements $(p, l) \in Ptr^{lb}$ are denoted as p_i^{lb}, where i is the program location of the end of the loop l.

The value of element p_i^{lb} is an abstraction of the value of the pointer p coming from the end of the body of the loop l. This property, described by the shape domain is called *may-point-to* relation between the sets Ptr^{lb} and $Addr$. Then the template of the shape domain is a formula of the form:

$$\mathcal{T}^S = \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \tag{2.13}$$

It is a conjunction of the template rows $\mathcal{T}^S_{p_i^{lb}}$, where each row corresponds to a loop-back pointer from the set Ptr^{lb}. The parameter $d_{p_i^{lb}} \subseteq Addr$ of the template row is the *abstract row value* and specifies the set of all addresses from $Addr$ that the pointer p may point to at the location i. We can then express the template row as a quantifier-free formula [13]:

$$\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv \left(\bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a \right). \tag{2.14}$$

Example. Using the template shape domain we express that loop-back pointer p at location 1 may point to objects ao_1, ao_2 or $null$ and another loop-back pointer q at location 2 points to object ao_1. Then the template rows and the resulting template have the form:

$$\mathcal{T}^S_{p_1^{lb}} = (p_1^{lb} = \&ao_1 \vee p_1^{lb} = \&ao_2 \vee p_1^{lb} = null)$$
$$\mathcal{T}^S_{q_2^{lb}} = (q_2^{lb} = \&ao_1 \vee q_2^{lb} = null)$$

(a) Template row for each pointer

$$\mathcal{T}^S = \mathcal{T}^S_{p_1^{lb}}(\{\&ao_1, \&ao_2, null\}) \wedge \mathcal{T}^S_{q_2^{lb}}(\{\&ao_1, null\})$$

(b) The resulting shape domain template

Figure 2.4: Various pointer locations expressed using the shape domain template

2.5.3 Combinations of Template-based Abstract Domains

Apart from efficiency benefits, the template-based approach of domains also allows easier combination of domains to infer various new interesting properties, e.g. inferring properties about numerical data of data structures. In this subsection we briefly introduce two of such domains that we aim to design our method for.

The simplest way to achieve a combination of two abstract domains is to use a *Cartesian product template*. It is a combination of different kinds of templates which are used independently side-by-side. It is accomplished as a conjunction of templates [13]. The representative example of this approach is the *shape interval domain* mentioned in the following list.

A. *Shape interval domain* is a domain which combines the shape domain and the interval domain. Its template is a conjunction of two templates from both domains. The resulting template then consists of two types of templates rows: one for numerical variables, the other for pointer variables. The abstract value for each row is computed in the correspoing domain. This way, it is possible to determine the shape of dynamic data structues from invariants for pointer-typed fields and at the same time the content of nodes of these structures from invariants for numerical fields [10].

B. *Shape domain with symbolic loop paths* is an extension of the shape domain that allows the invariant of a loop to distinguish which loops were or were not executed before reaching a given loop. With a utilization of SSA control-flow variables, specifically, loop-select guards, a *symbolic loop path* is simply a conjunction of these guards, expressing thus a reachability of the given loops. Different invariants are then computed for different symbolic loop paths. This can be used to, e.g. find out which objects were allocated beforehand and can be then processed in a given loop [13].

Chapter 3

Design of Imprecise Variable Identification and Localization Methods

In this chapter, we propose a solution to the predefined goal of this work: (1) a way to analyse the computed invariants in order to determine parts of the invariant that cause undecidability of the verification and (2) a method that uses the obtained information to identify the original program variables that successful verification may depend on. This chapter mainly builds on the information about the 2LS framework presented in the previous chapters.

In 2LS, templates are often composed of multiple parts (e.g., each part corresponds to a single variable) and an abstract value is computed for each of them individually. In order to design a method to identify such parts of the invariant that cause undecidability of the verification, we must first define how to recognise them. In our work, we limit ourselves to find parts of invariants that correspond to the supremum ($\top$) value of the abstract domain, that the invariant is computed in. Therefore, the goal is to determine in each supported abstract domain its representative supremum $\top$ value and to find so-called *template variables* that correspond to parts of invariants having the supremum value.

Since we analyse loop invariants, these variables are the loop-back SSA variables described in Section 2.4. We refer to the discovered variables as to imprecise template variables. In Section 3.1 we propose algorithms for finding imprecise parts of invariants (and subsequently for finding the imprecise template variables) for two supported abstract domains: namely for the *abstract interval domain* (Section 2.5.1) and the *heap domain* (Section 2.5.2).

After identifying imprecise template variables, we can use the existing structures in 2LS, mainly *GOTO programs* representation of the analysed program, which have the form of control flow graphs, to locate the original program variables that the template variables, more specifically SSA loop-back variables, correspond to.

However, template variables do not necessarily correspond to program variables, since they may represent dynamically allocated objects that may pose a problem. In Section 3.2 we describe an algorithm that maps the imprecise template variables, be it static or dynamic, back to the original program using SSA form and GOTO program representation of the analysed program.

3.1 Identification of Imprecise Variables inside Invariants

As described in Section 2.3, 2LS uses templates, which are fixed quantifier-free first-order logic formulas describing the desired property of the program, to efficiently compute loop invariants of the analysed program.

In 2LS each template domain has a defined *template form*, describing the program property that is being analysed. Templates of the domains presented in the following subsections have a form of a conjunction of formulae called *template rows*. Each template row corresponds to a single SSA loop-back variable x_{lb}. The template row has a row parameter, called *abstract row value*. This parameter is computed during invariant inference and represents an abstraction of the concrete value of the variable x_{lb}.

Therefore, in terms of template domains, a loop invariant is obtained by projecting the template rows and its corresponding *template row values* on to a subset of variables, containing only the loop-back variables of the concerning loop.

In the following subsections we present templates of the template polyhedra domain and the heap domain and also the supremum values in each domain that determine the imprecise template variables that we search for.

Finally, we propose algorithms to identify imprecise variables inside computed invariants using templates in the template polyhedra domain, specifically *abstract interval domain* and in the heap domain. The proposed algorithms 1 and 2 are described in the following subsections.

3.1.1 Abstract Interval Domain

As mentioned in the Section 2.5.1, abstract interval domain in 2LS is represented by the template subclass of template polyhedra, a class of templates used for analysis of numerical variables. Generally, the subclasses of template polyhedra have the form $\mathcal{T} = (\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{\delta})$ where $\boldsymbol{A}$ is a matrix with fixed coefficients. *Intervals* subclass template has the following form [3]:

$$\begin{pmatrix} +1 \\ -1 \end{pmatrix} x_i \leq \begin{pmatrix} \delta_{i1} \\ \delta_{i2} \end{pmatrix} \tag{3.1}$$

The i-th *row* of the template is the constraint generated by the i-th row of matrix $\boldsymbol{A}$. Therefore, for each variable x_i there are two template rows making one *template row pair*. Variables of $\boldsymbol{x}$ are represented as signed or unsigned integers.

We analyse invariants generated from these templates and search for variables with unconstrained values. For a variable x_i, this means that we search for template parameters:

$$\begin{pmatrix} +1 \\ -1 \end{pmatrix} x_i \leq \begin{pmatrix} d_{i1}^{\top} \\ d_{i2}^{\top} \end{pmatrix} \tag{3.2}$$

where the computed values of parameters $d_{i1}^{\top}$ and $d_{i2}^{\top}$ correspond to the maximum and the minimum value of the type of variable x_i, respectively.

Algorithm 1 presents the method of identification of imprecise variables inside the computed invariant in the interval domain.

The set of template row values *RowValues* is initialized with values of the computed invariant on line 1. Template rows are searched in a "pair after pair" manner. On lines 5-6 we get the template row value of the first template row using the function GetRowValue

and also we save the template row expression, which corresponds to the name of the variable. On line 8 we get the template row value of the second row of the template row pair. Afterwards we check, on line 9, if both template row values of the pair hold maximum and the minimum values in the given variable type, respectively. The result of the algorithm is the set of all identified imprecise SSA loop-back variable names — *VarNameSet*.

Algorithm 1: Imprecise variable identification in interval domain.

```
1  RowValues ← DomainValues
2  VarNameSet ← ∅
3  foreach Row ∈ TemplateRows do
4      if Row is first row of pair then
5          FirstRowValue ← GetRowValue(Row, RowValues)
6          RowExpr ← row expression of Row
7      else
8          SecndRowValue ← GetRowValue(Row, RowValues)
9          if FirstRowValue is maximum row value and SecndRowValue is minimum
            row value then
10             Name ← GetName(NameSet, RowExpr)
11             VarNameSet ← VarNameSet ∪ {Name}
12 return VarNameSet
```

Example. To better illustrate the identification of imprecise variables in the interval domain using the presented Algorithm 1, we use the illustrative C program from Chapter 2 shown in Figure 2.2.

The verification of the program using intervals domain is inconclusive due to unrecognized dependencies between values of variables x and i, and y and i where variable i determines the number of loop unwindings. Therefore, there is a possible overflow detected for values of variables x and y, and that can be seen in the computed invariant in Figure 3.1b, where values of both variables were set to the maximum and minimum in their promoted type.

$$
\begin{aligned}
&\mathcal{T}_0(+x_{lb12}, \delta_0)\\
&\mathcal{T}_1(-x_{lb12}, \delta_1)\\
&\mathcal{T}_2(+y_{lb12}, \delta_2)\\
&\mathcal{T}_3(-y_{lb12}, \delta_3)\\
&\mathcal{T}_4(+i_{lb12}, \delta_4)\\
&\mathcal{T}_5(-i_{lb12}, \delta_5)
\end{aligned}
\qquad
\begin{aligned}
&\\
&\\
&\\
&(+x_{lb12} \leq \text{INT\_MAX}) \wedge (-x_{lb12} \leq \text{INT\_MIN}) \wedge\\
&(+y_{lb12} \leq \text{INT\_MAX}) \wedge (-y_{lb12} \leq \text{INT\_MIN}) \wedge\\
&(i_{lb12} \leq 100) \wedge (-i_{lb12} \leq -1)
\end{aligned}
$$

(a) Interval domain template rows (b) Computed invariant

Figure 3.1: Generated template and computed invariant of program in Figure 2.2

The generated interval template $\mathcal{T}(\{x_{lb12}, y_{lb12}, i_{lb12}\}, \{\delta_0, \delta_1, \delta_2, \delta_3, \delta_4, \delta_5\})$ has in summary *six template rows*, as shown in Figure 3.1a. Algorithm 1 then compares row values of the first and of the second template row (corresponding to the actual computed values

in the computed invariant 3.1b) in each pair of template rows to the maximum and the minimum value in the promoted type (here signed integer), respectively. The result of the algorithm is then a set of the identified imprecise template row variables:

$$VarNameSet = \{x_{lb12}, y_{lb12}\}. \tag{3.3}$$

3.1.2 Heap Domain

This subsection builds on Section 2.5.2 about *shape domain* also called the heap domain. The heap domain is used by *shape analysis* for modelling the reachable shape of the program heap.

Since, loop invariants constrain values of the loop-back variables, the heap domain limits its template row variables to the set of all loop-back pointers Ptr^{lb} defined in Formula 2.11. An abstract dynamic object also corresponds to an allocation site (e.g. multiple concrete heap objects allocated in a loop). Pointers hold symbolic addresses of objects (or the special value `null`) from the set of all addresses defined in Formula 2.12. The relation between the sets of Ptr^{lb} and $Addr$ described by the shape domain is called *may-point-to* relation. The template of the shape domain has the form defined in Formula 2.13 where the may-point-to relation is expressed between the values of the parameter $d_{p_i^{lb}} \subseteq Addr$ and the template row pointer p of the template row quantifier-free Formula 2.14.

Whenever the shape analysis can not determine the symbolic address of the pointer p at location i, e.g. due to uninitialization or as a result of a dereference of an unknown or invalid (`null`) address, the value of the abstract row (corresponds to parameter $d_{p_i^{lb}}$) is set to a special value "unknown object" denoted as $o_\perp$ [13]. Since this special value corresponds to the supremum value (which is equivalent to *true* [3]) in the heap domain, we are looking imprecise template rows whose abstract value is true.

The method of identifying imprecise template rows in the heap domain is shown in Algorithm 2.

Algorithm 2: Imprecise variable identification in Heap domain.

```
1 RowValues ← DomainValues
2 VarNameSet ← ∅
3 foreach Row ∈ TemplateRows do
4     RowExpr ← row expression of Row
5     RowValue ← GetRowValue(RowExpr, RowValues)
6     if RowValue is true then
7         Name ← ExprName(DomainNameSet, RowExpr)
8         VarNameSet ← VarNameSet ∪ {Name}
9 return VarNameSet
```

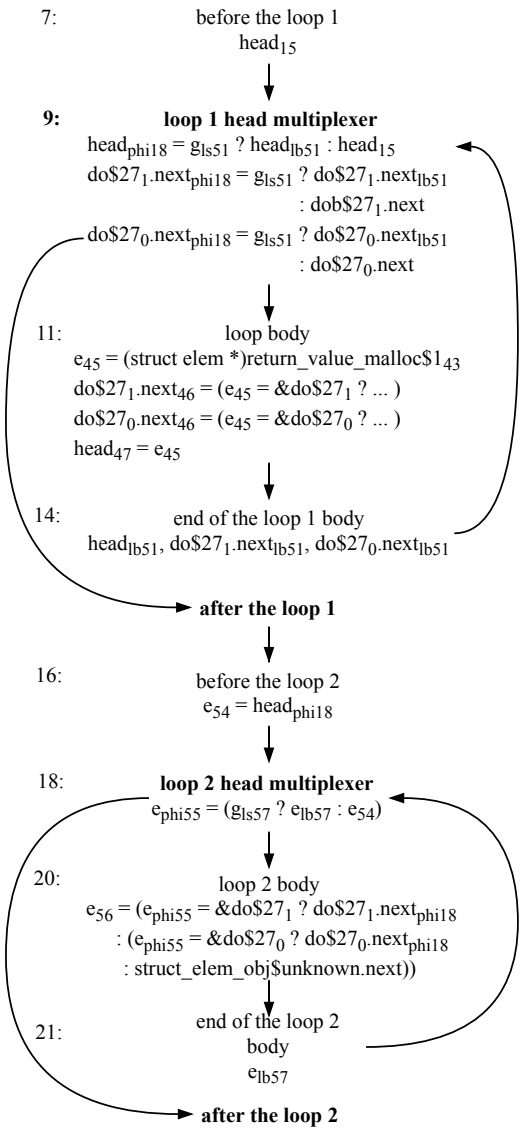
At line 6, each template row value *RowValue* of corresponding template row is checked whether it is *true*, which corresponds to the $\top$ value in the abstract domain. The row value is set to *true* whenever the value of the row object might be non-deterministic. Only if the template row value at line 6 is *true*, the name of the non-deterministic row (SSA variable name) is added to the set of names *VarNameSet* at line 8, which is the result of the algorithm.

Example. On example in Figure 3.2 we illustrate the presented algorithm for identification of imprecise variables in the heap domain. The example 3.2a shows a simple dynamic singly

linked list creation in the first loop between the lines 9-14 and a list traversal in the second loop between the lines 18-21. Due to an uninitialized value of pointer `head` pointing to an `elem_t` type at line 7, the traversing of the list at lines 18-21 results in *memory access violation.*

```
1 typedef struct elem {
2   struct elem *next;
3 } *elem_t;
4
5 void main()
6 {
7   elem_t head;
8
9   for (unsigned i = 0; i < 2; i++)
10  {
11    elem_t e = (elem_t) malloc(
          sizeof(struct elem));
12    e->next = head;
13    head = e;
14  }
15
16  elem_t e = head;
17
18  while (e)
19  {
20    e = e->next;
21  }
22 }
```

(a) Simple list traversal in C



(b) The corresponding SSA form

Figure 3.2: Example of imprecise variable identification in heap domain

The SSA form in Figure 3.2b has had names of dynamic variables and guards shortened to *do* instead of *dynamic_object* and to *g* instead of *guard*. The SSA also shows only the information relevant for imprecise variable identification in the heap domain. The generated heap domain template for the example code 3.2a is shown in Figure 3.3a. The template has been shortened to only four relevant template rows in total. The computed invariant in Figure 3.3b shows the abstract row values (denoted as d_i where i is the r-th template row in Figure 3.3a) for the corresponding template rows of the template.

	$(head_{lb51} = \&dynamic\_object\$27_0 \vee$
$\mathcal{T}_0(head_{lb51}, d_0)$	$head_{lb51} = \&dynamic\_object\$27_1) \quad \wedge$
$\mathcal{T}_1(dynamic\_object\$27_1.next_{lb51}, d_1)$	$(dynamic\_object\$27_1.next_{lb51} = true) \quad \wedge$
$\mathcal{T}_2(dynamic\_object\$27_0.next_{lb51}, d_2)$	$(dynamic\_object\$27_0.next_{lb51} = true) \quad \wedge$
$\mathcal{T}_3(e_{lb57}, d_3)$	$(e_{lb57} = true)$

(a) Heap domain template rows (b) Computed invariant

Figure 3.3: Generated template and computed invariant of program in Figure 3.2

Since the computed invariant only constrains the template row variable $head_{lb51}$, the other template row variables had their template row values set to *true* due to their non-deterministic values. Therefore, the set of identified imprecise template row variables looks like:

$$VarNameSet = \{e_{lb57}, dynamic\_object\$27_1.next_{lb51}, \\ dynamic\_object\$27_0.next_{lb51}\} \tag{3.4}$$

3.2 Localization of Variables in the Original Program

In this section we present a solution to localization of imprecise template row variables identified in various domains, as presented in the previous Section 3.1 in the original program.

Since 2LS is built upon the CPROVER infrastructure, the only information about the source program is through its internal representation — *GOTO programs*. GOTO programs are control flow graphs that maintain various information about the source code lines in its elements — *GOTO instructions*. The translation to the SSA form is made upon the GOTO programs. As outlined in Section 2.4.1 about the SSA conversion, the SSA form consists of SSA variables at various SSA locations also known as *nodes*. Due to the SSA form being an over-approximation of GOTO programs and due to its specific modifications in 2LS, it is not possible to use only the SSA variables when referring to the source program variables and to the lines in the source code at which they are defined or used. Instead, we must utilize the existent GOTO instructions in the given GOTO programs to which the SSA nodes refer.

The localization of the original variables in the source program consists of the following steps outlined in the following subsections: (1) We retrieve the original name of the given variable, this concerns mainly working with the SSA names (Subsection 3.2.1), (2) we localize the corresponding SSA node of the loop-back variable (Subsection 3.2.2), (3) we retrieve the source information using the localized SSA node and its corresponding GOTO instruction (Subsection 3.2.3). Finally, we present the algorithm for localization of the original variables and an illustrative example for both of the abstract domains that are able to find their imprecise template row variables in Subsection 3.2.4.

3.2.1 Original Variable Name Retrieval

Variables we want to localize are identified imprecise SSA loop-back variables of the invariants computed in various abstract domains. Such variables are either static variables or dynamically allocated objects. Such dynamic objects, as mentioned in Section 2.5.2 about the heap domain, are represented by abstract dynamic objects. Since dynamically allocated objects are accessed via pointers only, they do not have any names. Therefore, typical heap-manipulating operations are specifically represented in the SSA. The only operation this solution is concerned with, is the creation of dynamic objects using the `malloc` function. We start by specifying a set of naming rules.

Generally, the original *statically allocated variable name* x is converted to a loop-back SSA variable name in the following way:

$$x \longrightarrow x\#lbN \tag{3.5}$$

where N is the number of the n-th SSA node where the loop-back version of the variable is introduced (the *last SSA node* in the loop).

Since dynamically allocated variables do not have any names, they are specially represented in the SSA. Each `malloc` call is replaced by an instantiation of a member of new abstract dynamic objects and a choice among their symbolic addresses is returned as a result of the call. Uniqueness of the names of all abstract objects is ensured by the replacement of the occurrence of `malloc` in the source program as follows [10]:

$$malloc(\ldots) \longrightarrow g_1^{os}\ ?\ \&do\$i\#1\ :\ (g_2^{os}\ ?\ \&do\$i\#2\ :\ (\ldots)) \tag{3.6}$$

where g^{os} are free Boolean variables, so-called *object-select guards*, $\&do\$i$ is the shortened version of $\&dynamic\_object\$i$, and i is the number of the SSA node of `malloc` call occurence (i.e. SSA node with location of the allocation site).

Additionally, in 2LS, structure-typed heap objects are split into *fields* according to their structure composition, where each field of the structured object is considered a separate variable. Then the loop-back SSA name of such objects is defined as [13]:

$$dynamic\_object\$i\#y.Fld\#lbN \tag{3.7}$$

where $0 < y \leq k$, k is the number of all elements in the set of abstract dynamic objects AO allocated at one allocation site (it is required that $AO_m \cap AO_n = \emptyset$ for $m \neq n$), Fld is the name of the structure-typed object field, and N is the number of the loop-back node, being the last SSA node in the loop.

Therefore, we use as the original *name of the unnamed dynamic objects* the names of the elements of the set AO — $dynamic\_object\$i\#y$. In the case of the structure-typed dynamic objects, we can additionally refer to the name of its *structure field* using the Fld in the corresponding abstract dynamic object name.

3.2.2 SSA Node Localization

Having the SSA loop-back variables, be it static variable (according to the naming Rule 3.5) or dynamic objects (according to the naming Rule 3.7), we locate the SSA node using the N number in the lbN suffix, which corresponds to the last node in the body of the loop where the loop-back variable is created. At the same time due to the loops being cut at the end of their loop bodies, it is the node just before the loop-head node, as illustrated in the SSA form example in Figure 2.1.

In the case of dynamic objects, additionally we can get the node of the allocation site it was allocated at using the i number after the $ sign, according to the naming Rule 3.7.

3.2.3 Source Information Retrieval via GOTO instructions

As stated at the beginning of this section, 2LS uses *GOTO programs* as an internal representation of the analysed program. A GOTO program consists of a set of *GOTO functions*, which are then composed of a set of *GOTO instructions*. Each GOTO instruction corresponds to a step in the given source program function and maintains various information about the source code. Since SSA form is an over-approximation of GOTO program, more than one SSA node may correspond to one GOTO instruction. From each GOTO function an SSA form is built, called *local SSA*.

To get the source code line number of the loop where the variable with the name identified in Subsection 3.2.1 holds the imprecise value, first we need to get the corresponding GOTO instruction. In order to locate the GOTO instruction which corresponds to the SSA node (in the given local SSA) of the imprecise loop-back variable, we need to locate the SSA node itself using the approach from the previous subsection.

Having the loop-back SSA node of the imprecise variable, which is the last node in the loop, we can simply get the successor node — the loop-head node of the loop. Now, we can get the corresponding GOTO instruction. Since a GOTO instruction corresponds to one step in a concrete function we can get the line number of the loop-head in the source program using its kept source code information.

Analogously, we can locate the allocation site of the given dynamic object (i.e., line number where it was allocated using `malloc`). As already specified in the previous subsection, we can get the SSA node, and using the corresponding GOTO instruction, we can ultimately reach the concrete line number in the source code where the allocation occurs.

3.2.4 Localization Method Algorithm

We collect various information about each localized imprecise variable and store it in a summary. The information in the summary corresponds to the concrete variable properties in the analyzed program. In the summary, we also define elements used only by the dynamic objects. The summary $\boldsymbol{S}$ of a variable is a tuple:

$$\boldsymbol{S} = (Var, Loc_{lh}, Field, Loc_{site}) \tag{3.8}$$

where

- Var: is the name of the variable. In case of static variables, it holds the concrete variable name. In case of dynamic objects, it holds the symbolic "SSA-introduced" name.
- Loc_{lh}: is the loop-head location of the loop which the loop-back variable corresponds to.
- $Field$: is the name of the field of structure-typed heap object.
- Loc_{site}: is the allocation site location of the dynamic object.

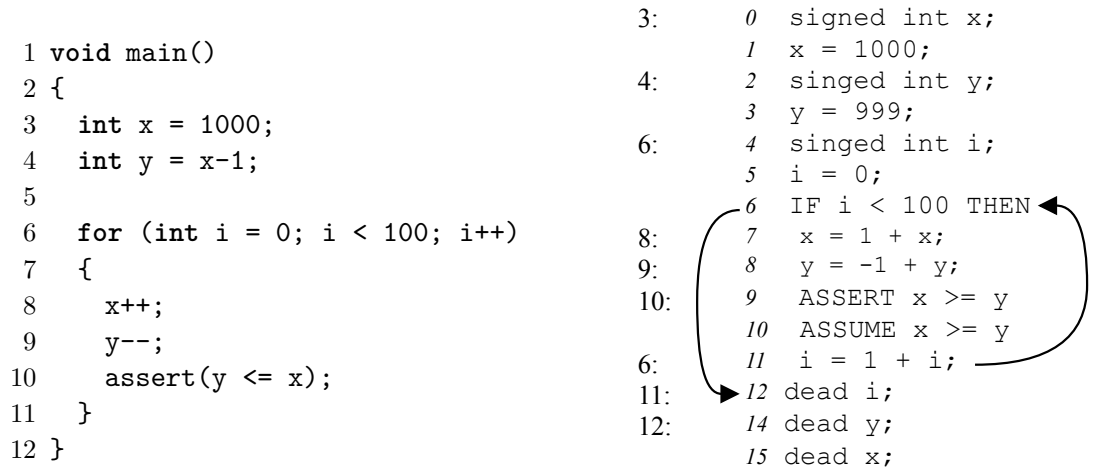
The localization method the of imprecise variables identified in invariants computed in various abstract domains is described in Algorithm 3.

Algorithm 3: Localization of original program variables using their SSA names

```
1  S ← ∅                                                    // a set of summaries
2  Var, Loc_lh, Field, Loc_site ← empty
3  NameSet ← set of all imprecise SSA loop-back variable names
4  IsDynamic ← false          // whether is static variable or dynamic object
5  foreach Name ∈ NameSet do
6      if Name starts with "dynamic_object$" then
7          IsDynamic ← true
8      Var ← GetPrettyName(Name)          // original program variable name
9      Location ← GetNameLocation(Name)                    // n-th SSA node
10     if Location = −1 then                               // is input variable
11         Loc_lh ← input variable
12     else
13         LoopBack_node ← get SSA node at location Location
14         LoopHead_node ← get loop-head node of LoopBack_node
15         Instr_lh ← get corresponding GOTO instruction of LoopHead
16         Loc_lh ← get source location of Instr_lh
17         if IsDynamic = true then                        // is dynamic object
18             Field ← GetDynamicField(Name)
19             Site_node ← get SSA node at location GetSiteLocation(Name)
                // allocation site
20             Instr_site ← get corresponding GOTO instruction of Site_node
21             Loc_site ← get source location of Instr_site
22     S ← S ∪ {Var, Loc_lh, Field, Loc_site}
```

The input of the algorithm is the set of all imprecise SSA loop-back variables $NameSet$, identified in any of the supported domains mentioned in the previous Section 3.1. At line 6, we differentiate between static and dynamic types of variables according to the naming rules of variables specified in the Subsection 3.2.1. At line 8, we get the actual program variable name Var according to the approach presented in Subsection 3.2.1. Then, we get the number of the corresponding SSA node using the method presented in Subsection 3.2.2 at line 9. If the location of the variable's SSA node can not be found then it is considered an input variable and no more information can be determined (lines 10-11). Having the variable's SSA node $LoopBack_{node}$, we can now get the next SSA node — $LoopHead_{node}$ (line 14). Then, at lines 15-16, we get the source location (the line number in the source program) of the GOTO instruction $Instr_{lh}$ that corresponds to the given loop-head node (as outlined in Subsection 3.2.3). Generally, if the given variable is a dynamic object, we get its allocation site location Loc_{site} (using the approach from Subsection 3.2.3) from the metadata about the source code of the given GOTO instruction $Instr_{site}$ which corresponds to the SSA node at allocation site $Site_{node}$ (lines 19-21). If the given dynamic object is structure-typed we can also get the name of its field $Field$, at line 18, according to the approach from Subsection 3.2.1. All gained information about the given variable is then added to the set of summaries $\boldsymbol{S}$ at line 22.

Also to further illustrate the localization described in Algorithm 3, we provide two examples. For the numerical variables in the abstract interval domain (Figure 3.4) and for the pointer-typed variables in the heap domain (Figure 3.5).



(a) Program in C

(b) The corresponding GOTO program

Figure 3.4: Localization of imprecise numerical variables in the original program

Example. The program in Figure 3.4a has been already described as the example in Subsection 3.1.1 about identification of imprecise template row variables in abstract interval domain. Also, the identified imprecise template row variables were shown in Formula 3.3.

Figure 3.4b shows the GOTO program representation of the example program. The numbers in the first column correspond to the line numbers of the original program showing thus which GOTO instructions are on which line. The numbers in the following column show the order of GOTO instructions (they are mainly for reference). The arrows illustrate the control flow between the basic blocks normally expressed using the `GOTO` statements.

The input of the algorithm is the set of imprecise names of variables $NameSet$, we can get the identified names of imprecise variables using Formula 3.3:

$$NameSet \leftarrow \{x_{lb12}, y_{lb12}\} \tag{3.9}$$

The type (static or dynamic) of the variables, their original program name and the location of the corresponding SSA node is determined according to the naming rule of static variables presented in Subsection 3.2.1 and to the approach outlined in Subsection 3.2.2, respectively. Both loop-back variables are static and are at the location 12 (loop-back node) in the SSA form, shown in Figure 2.2. The next node in the SSA is the loop-head node, which is linked to the GOTO instruction number 6. Coincidentally, the instruction corresponds to line 6 in the source program, which is the beginning of the for loop. The resulting set of all summaries $\boldsymbol{S}$ of all imprecise variables looks like:

$$\boldsymbol{S} = \{(x, 6, empty, empty), (y, 6, empty, empty)\} \tag{3.10}$$

Example. Using the already described program in Figure 3.2 we illustrate localization of pointer-typed variables with imprecise values in identified in the invariant computed in the heap domain.

Figure 3.5 shows the GOTO program representation of the program in Figure 3.2. The figure follows the same illustrative rules as already described for the GOTO program in the previous example.

The input of the algorithm is the set of all pointer-typed variables with imprecise values already identified in the example of Subsection 3.1.2. The set of such variable names $NameSet$ is therefore defined according to Formula 3.4:

$$NameSet \leftarrow \{e_{lb57}, dynamic\_object\$27_1.next_{lb51}, \\ dynamic\_object\$27_0.next_{lb51}\} \tag{3.11}$$

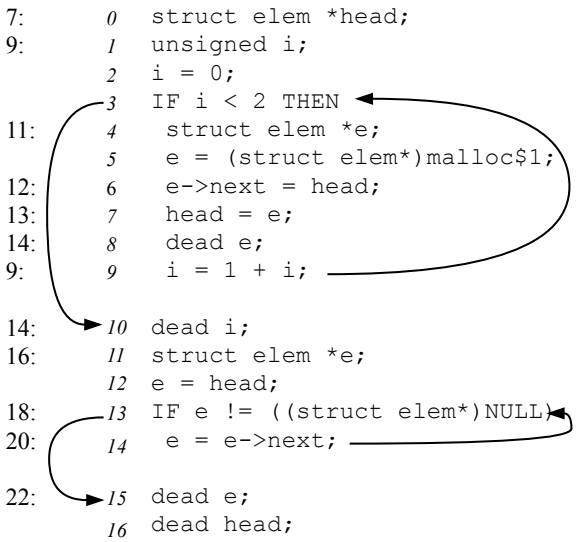


Figure 3.5: The corresponding GOTO program to C program in Figure 3.2a

First, we identify the memory type of variables and determine their original program name using the approach specified in Subsection 3.2.1. Variable e_{lb57} is identified as static and the other remaining elements in the $NameSet$ are identified as dynamic objects with names $dynamic\_object\$27_1$ and $dynamic\_object\$27_0$. The corresponding SSA node of e_{lb57} variable is at location 57 (as shown in the corresponding SSA form in Figure 3.2b). Its loop-head node corresponds to the GOTO instruction 13, which further corresponds to the `while` statement at line 18 in the source program (according to the approach from Subsection 3.2.3). In the same manner, both dynamic objects correspond to the same SSA node at location 51. Using the GOTO instruction 3 of their loop-head node we can identify the line number 9 of the `for` loop in the original program.

Additionally, we identify name of the fields of both structure-typed dynamic objects (approach from Subsection 3.2.1) and the SSA node of their allocation site (see Subsection 3.2.2). SSA node at location 27 (since the SSA form in Figure 3.2b was shortened, node 27 corresponds to the returned value of `malloc` at location 47) maps to GOTO instruction 5 which in the end is the statement at line 11. Finally, the resulting set of all summaries $\boldsymbol{S}$ of all localized variables looks like:

$$\boldsymbol{S} = \{(e_{lb57}, 18, empty, empty), \\ (dynamic\_object\$27_1, 9, next, 11), \\ (dynamic\_object\$27_0, 9, next, 11)\} \tag{3.12}$$

Chapter 4

Implementation

The solution designed in the previous Chapter 3 was implemented in the 2LS framework. The implementation of *Imprecision Invariant Identification* method is generally available for any template-based domain in 2LS. Currently the only supported domains are the *abstract interval domain*, the *heap domain*, and their combination domain *heap polyhedra domain* with the *symbolic paths* extension. In the following sections we describe the Architecture of 2LS, and in Section 4.2 we briefly describe the integration process of our solution in to the 2LS framework.

4.1 Architecture of 2LS

Being built upon the CPROVER infrastructure, 2LS utilizes various modules of the CPROVER framework. The verification process of 2LS is composed of many steps which can be divided into three main categories: *front end*, *middle end*, and *back end*. In the following subsections we describe the categories in steps as shown in Figure 4.1 [12].

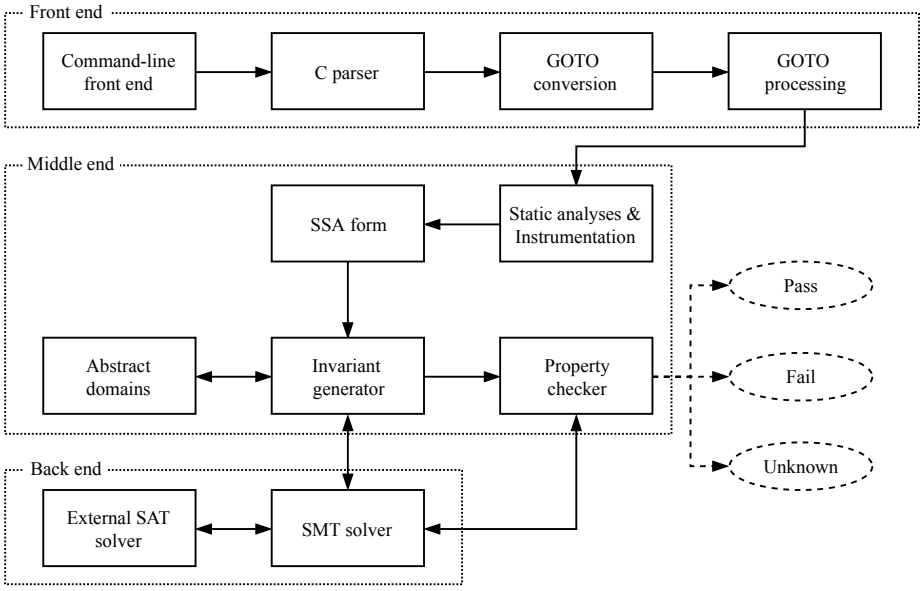


Figure 4.1: 2LS Architecture

4.1.1 Front end

User-supplied parameters are processed by *command-line front end*. Using command-line options, user can configure various properties in 2LS, such as bit-width, type of operating system, architecture, and other instrumentation and back end functions. After successful configuration, user-supplied input source program is parsed using a *C parser* which utilises an off-the-shelf C preprocessor. From the preprocessed source, a parse tree is built. As an internal representation of the source program 2LS uses *GOTO programs* implemented in the CPROVER framework. The translated source program in GOTO program representation has a form of a control flow graph. 2LS then *processes the GOTO program* — performs various transformations upon the intermediate representation such as function pointer resolving, function inlining, or constant propagation.

4.1.2 Middle end

As needed, various *static analyses* are performed upon the GOTO program before SSA conversion. These are mainly *objects analysis* which collects all accessed objects in a function, *points-to* analysis, to determine the set of dereferenceable objects for each pointer, and in the end derives the data flow equations for each GOTO program function. After all needed static analyses, the GOTO program is converted into the *SSA form*.

Due to the over-approximation caused by the SSA form (see Section 2.4), loop invariants and function summaries are inferred and computed in various abstract domains in order to refine this over-approximation. Currently, 2LS supports a number of *abstract domains*, some are more thoroughly described in Section 2.5:

- *Abstract polyhedra domain*: class of domains for analysis of numerical variables. In 2LS numerical variables are signed, unsigned integers and floats represented using *bit-vectors*. Subclasses include: *Interval domain*, *Zones domain*, *Octagon domain*
- *Abstract Shape domain*: domain for modeling the reachable shape of the heap (see Section 2.5.2).
- *Heap polyhedra domain*: a combination domain of the abstract polyhedra and the abstract shape domains (see Section 2.5.3).
- *Heap polyhedra domain with symbolic paths*: an extension domain for the heap polyhedra combination domain. Computes different invariants for various *symbolic loop paths* (see Section 2.5.3).
- *Equalities and disequalities domain*: used for analysis of equalities and disequalities between pairs of variables.
- *Lexicographic linear ranking function domain*: domain used for termination analysis.

Also, to increase efficiency, 2LS performs local constant propagation and expression simplification.

After computing invariants, the property checker is used to check the validity of user-supplied assertions and of other supported properties. The SSA form of the source program is translated into the CNF formula and along with the computed invariantm it is solved using the SMT solver. Then, negations of programs assertions are checked whether they are satisfiable. In case there is a satisfiable negation of an assertion, then the verification has failed due to an error in the analysed program. If negations of all assertions are

unsatisfiable, then the verification is successful and the supplied program is valid. Due to over-approximation, false positives can be found or the verification might be inconclusive [10].

4.1.3 Back end

For computing invariants and property checking, 2LS utilizes an SMT solver. Instead of using poorly supported incremental SMT solver, 2LS uses external incremental SAT solver, namely Glucose 4.0 [2], and SMT theories over bit-vectors provided by the CPROVER framework.

4.2 Integration of the Designed Solution

The designed solution has been implemented in the 2LS framework. The mentioned abstract domains were extended by the method for identification of imprecise variables designed in Section 3.1. Furthermore, the SSA analyzer was extended by the method for localization of the imprecise variables in the original program, as designed in Section 3.2. All algorithms from Chapter 3 were therefore implemented. In this section we describe some details about implementation of these extensions.

4.2.1 Imprecise Invariant Identification Method for Abstract domains

Now, every abstract domain can have the *imprecision invariant identification* method implemented. Currently, this extension is implemented in the polyhedra domain (abstract interval domain only), the heap domain, the combination of both previous domains — the heap polyhedra domain, and the heap polyhedra domain with symblic paths.

In the polyhedra domain, as previously mentioned, numerical variables are represented using *bit-vectors*. Due to the extension of bit-width in order to avoid arithmetic under- and overflows, the minimum and the maximum values in the promoted type to be compared, (as according to the Algorithm 1) must be extended also to the same bit-width.

Heap domain is extended in a straightforward manner according to the Algorithm 2. Both combination domains (heap polyhedra and heap polyhedra sympath domains) call imprecision invariant identification methods implemented in their elementary domains because they are composed of both polyhedra and heap template rows. The result is a unified set of identified imprecise variables from both elementary domains.

4.2.2 Localization Method for SSA Analyzer

A component called *SSA Analyzer* which uses various *strategy solvers* in 2LS' domains to infer invariants, was extended with the localization method of imprecise invariants according to Algorithm 3. After the invariant gets computed, SSA analyzer calls the imprecision invariant identification method in the given domain. The localization method can be activated using the `--show-imprecise-vars` switch. Whenever used, the imprecise variables are localized, except for CPROVER auxiliary variables, and the information is saved in a function summary. At the end of the analysis, the information is printed as a part of the function summary for every individual function.

Chapter 5

Results and Experiments

We have proposed a method for identification of imprecise parts of invariants in the original program and implemented it in the 2LS framework. First, we need to prove that this extension did not break any of the existing analyses in 2LS. That is proved using the regression test suite of 2LS in Section 5.1. After we confirm that the integration in no way affected the existing analyses, we show on our running example the output of this extension and how it can be of help to the user in the verification process in Section 5.2.

In the rest of this chapter, we perform series of large-scale experiments. We run the 2LS regression test suite and benchmarks from the International Competition on Software Verification 2017 (SV-COMP 2017) in which 2LS annually participates. In these experiments, we concentrate on tasks where the verification is currently inconclusive and we run our extension on such tasks. As a result, we give a number of tasks in each of the experiments (and of their categories) where our solution found at least some variables of the analysed program with an imprecise computed invariant. In such tasks, our solution can potentially help developers of 2LS to fix the problem causing the inconclusiveness. Results of the experiments are presented in Section 5.3 (regression tests) and in Section 5.4 (SV-COMP benchmarks).

The listed experiments and tests were performed on the operating system Ubuntu version 16.04 Xenial 64 bit running on a system with following specifications: Intel Xeon X56xx 3.492 GHz quad core. We used 2LS version0̇.7.1.

5.1 Proving the Success of the Integration

We prove that our integration did not break any of the working code in 2LS by running the regression test suite of 2LS on the original implementation of 2LS and then on the implementation with our extension activated. Used 2LS options were presereved according to the predefined options of the suite. The summary of results is shown in Table 5.1.

We can see that our extension did not affect the existing implementation of 2LS. In the last row “elapsed time” we also provide the time taken to run the tests (average value taken after three runs). The point to be proven is that our solution should not inflict any performance hit to the tool. Therefore we can see that the differences between the time intervals are within a margin of error (though it should be taken with consideration in respect to the method used).

	Without extension	With extension
Correct results	183	183
Incorrect results	52	52
Inconclusive results	33	33
Skipped tests	116	116
Elapsed time	333.618 s	328.756 s

Table 5.1: Difference before the extension integration and after the integration on 2LS regression tests

5.2 Experiments

In this section we list the output of our extension when verifying following examples. As outlined in Section 3.2.4 the output of our extension is part of the function summary, which is printed for every analysed function.

Listing 5.1 shows the output of running our extension on example in Figure 3.2a, which is focused on localization of pointer-typed variables of the heap domain. Listing 5.2 shows the output of running our extension on the motivation example in Figure 1.1, which is focused on localization of numerical variables of the abstract interval domain.

```
...
invariant imprecise variables:
 1: Imprecise value of "next" field of "dynamic_object$27#1"
 allocated at line 11; at the end of the loop starting at line 9
 2: Imprecise value of "next" field of "dynamic_object$27#0"
 allocated at line 11; at the end of the loop starting at line 9
 3: Imprecise value of variable "e" at the end of the loop;
 starting at line 18
...
```

Listing 5.1: Identified variables with imprecise values in program 3.2a

We can see that the proposed solution correctly identified the variables with imprecise values, as described in the localization example for the heap domain in Section 3.5. To refer to the concrete objects, we use the symbolic names of dynamic objects introduced in the SSA. We can see that the names of the fields of both structure-typed dynamic objects were correctly identified, we also refer to the allocation location of both of the objects.

```
...
invariant imprecise variables:
 1: Imprecise value of variable "x" at the end of the loop
 starting at line 5
 2: Imprecise value of variable "y" at the end of the loop
 starting at line 5
...
```

Listing 5.2: Identified variables with imprecise values in program 1.1

We can see that our extension correctly identified the variables with unconstrained values inside the program loop, as described in the localization example in Section 3.4. In

response to the motivation example in Section 1.1 user can utilize the output and take its usability a little bit further.

Normally, even such "obvious" program may render the result of the verification of 2LS as inconclusive. In this case, the dependency between the values of variables inside the loop and the value of the control variable i is not recognized by 2LS. The result is the imprecision already shown in the computed invariant in Figure 3.1b in Chapter 3.

In order for 2LS to recognize such dependency, user can use `assume` constructions to constrain the values of such imprecise variables in a fairly obvious way. We can safely assume that the value of variable x will steadily increase over the number of loop iterations starting from its original value. Analogously, the same can be assumed with the value of variable y—it will continuously decrease, but clearly will never reach below 0. Therefore, we insert the appropriate `assume` constructions as seen in Figure 5.1. 2LS then verifies the program, taking in consideration the newly defined constraints and computes a new invariant:

$$\begin{gathered}(+x \leq \text{INT\_MAX}) \wedge (-x \leq -1001) \wedge \\ (+y \leq 998) \wedge (-y \leq 0) \wedge \\ (i \leq 100) \wedge (-i \leq -1)\end{gathered}$$

where we can see that interval of values of both variables x and y do not intersect, in fact the relation corresponds to the condition of the assertion at line 10. Therefore the verification of the program is successful.

```
1  void main() {
2    int x = 1000;
3    int y = x-1;
4
5    for (int i = 0; i < 100; i++) {
6      x++;
7      y--;
8      __CPROVER_assume(x >= 1000);
9      __CPROVER_assume(y >= 0);
10     assert(y <= x);
11   }
12 }
```

Figure 5.1: Successful verification after "assume" construct insertion

5.3 Imprecisions Found in the 2LS Regression Tests

In this section we list the results of running the tests of the regression test suite of 2LS.

In order to assist in the development of new features in 2LS, we want to concentrate on identifying imprecisions in the *Knownbug* tests (column *Kbg*). That is, in tests that describe a faulty or unwanted behavior of 2LS. These tests may either end with the verification result as failed or inconclusive. We mainly look for inconclusive tests (column *Inc*), because as already mentioned, the imprecisions in invariants are one of the causes of such result.

Therefore in Table 5.2, we list the following columns in order: the total number of tasks in every category of this suite, the number of tasks where an imprecision was found, the number of all *knownbug* tests, number of *knownbug* tasks where imprecisions were identified, and lastly number of *knownbug* tests whose verification result is inconclusive where imprecisions were identified.

Category	**Tasks**	**Found**	**Kbg**	**Found & Kbg**	**Found & Kbg & Inc**
Nontermination	43	12	12	6	0
Termination	129	31	24	11	5
kIkI	36	6	6	0	0
Preconditions	8	0	1	0	0
Interprocedural	47	2	10	2	2
Invariants	86	9	18	5	5
Heap	19	9	11	3	3
Heap-data	11	10	1	1	0
Memsafety	4	4	0	0	0

Table 5.2: Regression tests in 2LS. Listed columns: Number of tasks with identified imprecise variables—Found, number of *knownbug* tasks, and *knownbug* tasks with identified imprecisions, and lastly number of *knownbug* tasks with verification inconclusive result where imprecisions were identified.

As we can see, at least in some of the knownbug tasks imprecisions were found. We mainly focus on the *Invariants* and *Heap* categories, where all the identified tasks were both knownbug and inconclusive.

5.4 Benchmarks from SV-COMP 2017

We ran various categories of the SV-COMP benchmarks listed in the table below. All the tests we run with the options `--heap-values-refine`, `--k-induction`, `--unwind 5` and `--competition-mode`, with preferences a memory limit of 15 GB, and 900 s timeout, and on all CPU cores.

In total 1503 tests were run, out of which 364 tests terminated with verification result as inconclusive (column *Inconclusive*). In summary, in 262 tests were identified variables with imprecise values potentially causing such verification result (column *Found*).

Category	**Tasks**	**Inconclusive**	**Found**
ReachSafety-Loops	210	91	59
ReachSafety-Arrays	231	71	45
ReachSafety-BitVectors	50	29	26
ReachSafety-ControlFlow	74	19	19
ReachSafety-Floats	469	11	6
ReachSafety-Heap	265	108	88
MemorySafety-Heap	115	18	13
MemorySafety-Other	89	17	6

Table 5.3: Number of inconclusive tasks were variables with imprecise values in various SV-COMP categories. Number of tasks (tests) in every category, number of tasks with the verification result as inconclusive, and number of

We can see that for each category (except for MemorySafety-Other) in the majority of all tasks with the verification result inconclusive, we identified at least one variable with an imprecise value that may potentially be the cause of such result.

Chapter 6

Conclusion

In this work, we proposed a solution to identify imprecise parts of invariants computed by the 2LS framework that possibly determine whether verification is successful. The solution identifies variables of the original program with imprecise values and the loops in the analysed program where this imprecision occurs. In a case we identify dynamic objects (allocated on the heap), we are able to identify the line of allocation, and for structure-typed objects also the original name of their fields having the imprecise values.

We have proposed two successive methods in order to identify such variables: (1) a method for identification of imprecise parts of invariants computed in various abstract domains and (2) a method for localization of variables corresponding to such parts using the internal representation of C programs utilized by 2LS.

The solution was implemented in the 2LS framework. The method for identification of imprecise parts of invariants can be implemented in any of the existing abstract domains in 2LS. Currently, we have implemented the method in the abstract interval domain, in the heap domain, in their combination — heap interval domain, and lastly in the extension of the heap interval domain with symbolic paths. The method for localization has been integrated into the SSA analyzer module of 2LS.

We have shown on various examples that our solution is able to identify the imprecise variables in the invariants and locate the parts of the analysed program in which these imprecisions occur. Additionally, we have shown an example when such feedback is useful for the user and helps him to correctly verify the given program. We have proved on various benchmarks from the SV-COMP 2017 test suite that in a majority of programs where the verification currently fails, the proposed solution can be used to localize parts of the invariants that potentially cause the undecidability of the verification.

In future, we would like to propose an official integration of this extension to the 2LS framework. This could be mainly of use to the 2LS developers in their feature development process.

Bibliography

[1] *CPROVER - Systems Verification Group*. [Online; seen 21.01.2019]. Retrieved from: http://www.cprover.org/

[2] Audemard, G.; Simon, L.: *The Glucose SAT Solver*. 2014. [Online; seen 13.05.2019]. Retrieved from: http://www.labri.fr/perso/lsimon/glucose/

[3] Brain, M.; Joshi, S.; Kroening, D.; et al.: *Safety Verification and Refutation by k-invariants and k-induction*. In *Static Analysis: 22nd International Symposium, LNCS*, vol. 9291. Springer. 2015. pp. 145–161.

[4] Cousot, P. and Cousot, R.: *Static determination of dynamic properties of programs*. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France. 1976. pp. 106–130.

[5] Cousot, P. and Cousot, R.: *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In *Proceedings of the 4th International Symposium on Programming*. ACM. 1977. pp. 238–252.

[6] Cousot, P. and Cousot, R.: *Basic Concepts of Abstract Interpretation*. In *Building the Information Society*. Springer US. 2004. pp. 359–366.

[7] Kroening, D.: *Diffblue*. 2018. [Online; seen 21.01.2019]. Retrieved from: https://www.diffblue.com/about-us

[8] Kroening, D.; Schrammel, P.: *2LS - Static Analyzer and Verifier*. [Online; seen 21.01.2019]. Retrieved from: https://github.com/diffblue/2ls

[9] Lengál, O.; Vojnar, T.: *Formal Analysis and Verification - Abstract Interpretation*. 2018. [Online; seen 21.01.2019]. Brno University of Technology Faculty of Information Technology. Retrieved from: http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-10.pdf

[10] Malik, V.: *Template-Based Synthesis of Heap Abstractions*. Master's thesis. Brno University of Technology, Faculty of Information Technology. Brno. 2017.

[11] Sankaranarayanan, S.; Sipma, H. B.; Manna, Z.: *Scalable Analysis of Linear Systems Using Mathematical Programming*. In *VMCAI*, vol. 3385. Heidelberg. 2005. pp. 25–41.

[12] Schrammel, P.; Kroening, D.: *2LS for Program Analysis (Competition Contribution)*. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems, *LNCS*, vol. 9636. Springer. April 2016. pp. 905–907.

[13] Viktor Malik and Martin Hruska and Peter Schrammel and Tomas Vojnar: *Template-based verification of heap-manipulating programs*. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD 2018)*. Institute of Electrical and Electronics Engineers. 2018. pp. 103–112.

Appendix A

Contents of the CD

The enclosed CD contains a directory of the 2LS framework with its source codes. The structure of the main directory looks like:

```
/
├── cbmc/ ..... CBMC library (CPROVER framework)
├── doc/ ..... LaTeX source files of this text
├── regression/ ..... 2LS regression tests
├── src/ ..... 2LS source files
├── COMPILING ..... Instructions on how to compile by 2LS
├── install.sh ..... 2LS install script
├── LICENSE ..... 2LS license
└── README.md ..... README file
```

The source files of the 2LS can be found in the directory `src/`. The source directory is divided into multiple subdirectories, we list all the directories and their source files in which we implemented our extension, in the list below:

- `2ls/` Main 2LS directory, consists of source files of the front-end of 2LS.
  - `2ls_parse_options` Consists of parsing options definitions and calls other modules in the verification process of 2LS. We added our own command-line option.
- `domains/` Directory of existing abstract domains in 2LS. The method for identification of invariant imprecision was implemented in ithe following domains/files, and also the localization method was implemented in the `SSA analyzer` module.
  - `domain` We defined the method for invariant imprecision identification for other domains that inherit from this base domain source file. Then we implemented this method in the following source files of the existing domains: `heap_domain`, `heap_tpolyhedra_domain`, `heap_tpolyhedra_sympath_domain`, and in the `tpolyhedra_domain`.
  - `ssa_analyzer` We extended the analyzer module with the localization method.
- `solver/` We added the activation of our command-line option into the `summarizer_fw` source file, in which a function summary is computed and printed, and we defined our summary in `summary` source file, which defines the function summary structure.

Appendix B

How to Compile

The project can be compiled and run using the attached source files. The project can be compiled in two ways:

A. Using the `install.sh` script in the main directory of the CD. It downloads the CBMC library (CPROVER infrastructure), compiles it and then compiles the project itself.

B. Strictly compiling the 2LS source files only using the `src/Makefile`, but the CBMC library must be already compiled in its own `cbmc/` directory using the (`cbmc/src/Makefile`).

2LS with the imprecision identification extension can be run using:

```
$ 2ls SOURCE_FILE --show-imprecise-vars
```

The output of our extension should then be seen at the end of computed function summary (if is a summary is computed). If option `--competition-edition` is used also, then the function summary is not printed by default.

We also enclose 2LS regression tests described in the following appendix, which can be used to test our extension. The flags of the tests of the regression suite were modified so that the option of our extension is used when they are run.

Appendix C

2LS Regression Tests

2LS contains its own regression tests in directory `regression/` divided into number of categories. The tests can be run using the enclosed `Makefile` in the directory. Every category is in its own subdirectory having its own `Makefile` that can be used to run them individually. We present the categories with a one line description:

- `heap` Contains tasks focused on the heap-manipulation of programs.
- `heap-data` Tasks that combine manipulation of unbounded data structures and a need to reason about the data stored in these structures.
- `interprocedural` Contains tasks focused on verifying programs using interprocedural analysis.
- `invariants` Tests the invariant computation in various domains.
- `kiki` Tests features of the *kIkI* algorithm.
- `memsafety` Tasks aimed at verifying memory safety.
- `preconditions` Tasks aimed at computing forward and backward preconditions and postconditions of functions of the analysed programs.
- `nontermination/termination` Tasks focused on analysis of termination of functions.

Also we include custom `bash` scripts, which were used for evaluation of test results. The results shown after running the tests using the enclosed `Makefile` in the `regression` directory, may not correspond to Table 5.2, since a certain number of tests are skipped by default (as shown in Table 5.1), because verification of such tests takes more than 5 minutes. All bash scripts expect the regression tests to be run beforehand.

- `fast_find.sh` Fast search of imprecisions in any of the tasks, prints a simple summary of number of tasks in each category and number of tasks where imprecisions were found.
- `find_imprecision.sh` Prints a text table similar to the enclosed Table 5.2 and summary of results.
- `test_results.sh` Prints a summary of verification results of tasks in each of the categories, similar to Table 5.1.