BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TEMPLATE-BASED SYNTHESIS OF HEAP ABSTRACTIONS

ABSTRAKCIA DYNAMICKÝCH DÁTOVÝCH ŠTRUKTÚR S VYUŽITÍM ŠABLÓN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **VIKTOR MALÍK**
AUTOR PRÁCE

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

BRNO 2017

Abstract

The goal of this work is to propose a shape analysis suitable for the context of the 2LS analyser. 2LS is a program analysis framework for C programs which is based on automatic invariant inference using an SMT solver. The proposed solution includes a way how the shape of a program heap can be described using logical formulae over bit-vectors and how a first-order SMT solver can be used to infer loop invariants and function summaries for each function of the analysed program. Our approach is based on pointer access paths that describe the shape of the heap by expressing the reachability of heap objects from pointer-typed program variables. The information obtained from the analysis can be used to prove various properties of programs manipulating dynamic data structures, mainly linked lists. The solution has been implemented in the 2LS framework and it brought a significant improvement in terms of the capabilities of 2LS in analysing heap-manipulating programs. This is demonstrated on benchmarks taken from the well-known International Competition on Software Verification (SV-COMP) as well as other benchmarks.

Abstrakt

Cieľom tejto práce je návrh analýzy tvaru haldy vhodnej pre potreby analyzátora 2LS. 2LS je nástroj pre analýzu C programov založený na automatickom odvodzovaní invariantov s použitím SMT solvera. Navrhované riešenie obsahuje spôsob reprezentácie tvaru programovej haldy pomocou logických formulí nad teóriou bitových vektorov. Tie sú následne využité v SMT solveri pre predikátovú logiku prvého rádu na odvodenie invariantov cyklov a súhrnov jednotlivých funkcií analyzovaného programu. Náš prístup je založený na ukazateľových prístupových cestách, ktoré vyjadrujú dosiahnuteľnosť objektov na halde z ukazateľových premenných. Informácie získané z analýzy môžu byť využité na dokázanie rôznych vlastností programu súvisiacich s prácou s dynamickýcmi dátovými štruktúrami. Riešenie bolo implementované v rámci nástroja 2LS. S jeho použitím došlo k výraznému zlepšeniu schopnosti 2LS analyzovať programy pracujúce s ukazateľmi a dynamickými dátovými štruktúrami. Toto je demonštrované na sade experimentov prevzatých zo známej medzinárodnej súťaže vo verifikácii programov SV-COMP a iných experimentoch.

Keywords

formal verification, program analysis, 2LS, template-based analysis, shape analysis, linked lists, pointer access paths, abstract interpretation, SSA form, invariant inference, function summaries, dynamic data structures

Kľúčové slová

formálna verifikácia, analýza programov, 2LS, analýza založená na šablónach, analýza tvaru haldy, zreťazené zoznamy, prístupové cesty na halde, abstraktná interpretácia, SSA forma, odvodzovanie invariantov, súhrny funkcií, dynamické dátové štruktúry

Reference

MALÍK, Viktor. *Template-Based Synthesis of Heap Abstractions.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Template-Based Synthesis of Heap Abstractions

Declaration

Hereby I declare that this master thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Dr. Peter Schrammel. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Viktor Malík
September 13, 2017

Acknowledgements

I would like to thank my supervisor prof. Tomáš Vojnar for useful revisions of the project report and remarks and consultations about the theoretical aspect of the work. I would also like to thank Dr. Peter Schrammel of the University of Sussex for continuous support in both theoretical and implementation parts of the project.

Contents

Chapter 1

Introduction

Research in the fields of formal verification and analysis is very wide and getting ahead very fast. Currently, there is a large number of tools available, designed to analyse various properties of programs. However, most of the tools are typically very narrowly focused on a single area of analysis. They usually fail to analyse complex properties of real-life programs (e.g. verifying termination of programs using numerical and pointer variables at the same time) while still being able to scale for realistic programs. On such complex properties and programs, the tools usually give up or produce imprecise results (false positives or even false negatives).

One of the tools trying to combine multiple approaches into a single, scalable framework is 2LS. It integrates different program analysis techniques to work simultaneously and exchange information, which allows it to both prove true properties as well as find errors in programs. Due to using multiple techniques, 2LS offers a possibility to analyse different classes of program properties. Currently, it is well usable to verify termination, data-flow among numerical variables and arrays (using domains of different price and precision), or equality between pairs of variables in the given program. 2LS was developed by Daniel Kroening and Peter Schrammel at the University of Oxford, UK. Currently, it is maintained by Peter Schrammel at the University of Sussex and the spin-off company DiffBlue Ltd. This thesis was solved in cooperation with this company.

One of the important features that 2LS currently lacks is an ability to analyse programs that work with dynamic data structures, such as linked lists, trees, etc., which is a task usually referred to as shape analysis since it is intended to analyse reachable shapes of dynamic data structures. However, many real-world programs use dynamic structures, and therefore it is needed to integrate this kind of analysis into 2LS. Moreover, since 2LS already contains good numerical analysis, its combination with shape analysis could bring new options of analysing interesting program properties that other, single-purpose tools are not able to handle well (e.g. properties based on the length of linked lists).

Hence, the goal of this thesis is to propose a shape analysis suitable for the specific context of 2LS, which very much differs from what is common in other frameworks. Typical shape analyses are mostly based on some form of abstract interpretation that symbolically executes a given program, iteratively going through its loops. In order to avoid generation of infinitely many reachable program configurations, some form of widening/abstraction is used to summarize reachable sets of configurations into a finite number of abstract symbolic configurations. For representing such configurations, rich classes of logics, automata, or other formalisms are used—e.g. separation logic [14], 3-valued predicate logic with transitive

closure [16], forest automata [10], or symbolic memory graphs [9]. 2LS differs from these tools in (at least) 2 important aspects: (1) it is heavily based on the bit-vector logic, ultimately using SAT solving, and (2) it uses a significantly different computation loop. This loop is based on combining k-induction, a notion of invariants based on so-called templates, and a rather specific form of abstract interpretation. Incorporating shape analysis into this framework hence requires a rather specific solution.

In this thesis, we propose a solution to the above problem. In particular, we propose a novel domain for representing sets of reachable heap shapes that can be well integrated into the template-based approach of 2LS. Namely, we represent sets of heap configurations using concept of pointer access paths. This representation of the heap does not concretely describe the shape of the heap, only expresses reachability of heap objects from variables in the analysed program via chains of pointers. Moreover, we propose all algorithms needed for integrating the domain into 2LS, both within intraprocedural as well as interprocedural analysis. This required us to propose specific algorithms for join of abstract values in our abstract domain and for reflecting heap modifications performed by a function within the context of the caller function. In addition, to be able to supply these algorithms with some auxiliary information they need, we also had to modify the points-to analysis and the generation of the static single assignment form that is used as an internal program representation of 2LS.

We have implemented the proposed ideas in 2LS and applied the extended tool to a number of benchmarks. The obtained results indicate that our extension brought a significant improvement in terms of the ability of 2LS in analysing programs manipulating pointers and dynamic data structures on the heap.

The rest of the thesis is organised as follows. First, the basic concepts of program verification and the current state of the art of 2LS are described in Chapter 2. After that, principles of the proposed solution are described in Chapter 3, along with all necessary changes that must be done to the current concepts of 2LS. Chapter 4 outlines the architecture of 2LS and gives details of the implementation of our extension. Results of our experiments are presented in Chapter 5, along with a discussion of what these results show. Finally, a conclusion and future work is in Chapter 6.

This thesis extends the Term project of the same title. Particularly, Chapter 2 and part of Chapter 3 (up to Section 3.6) were taken from the Term project with some minor changes.

Chapter 2

Program Verification in 2LS

The goal of this project is an integration of shape analysis into the 2LS tool [17]. This chapter will briefly introduce the basic concepts of 2LS and then explain in detail those that are needed to understand the methods proposed in this project.

2LS is a program analysis framework built upon the CPROVER verification framework [1]. It is oriented towards analysis of sequential C programs. The core algorithm of 2LS, called *k*I*k*I, efficiently combines bounded model checking (BMC), *k*-induction and abstract interpretation [4]. Although all these three verification approaches can be used simultaneously, we will only use *abstract interpretation* for the shape analysis extension of 2LS proposed within this work. General concepts of this program analysis technique are formally described in Section 2.2.

The abstract interpretation used in 2LS is based on computing so-called *inductive invariants* for all loops and functions of the source program. These are inferred using an SMT solver-based approach and then used to reason about various properties of the analysed program. The approach to the inference of the inductive invariants is formally described in Section 2.3.

Although the below description views source programs for simplicity as transition systems (described in Section 2.1), the implementation of 2LS uses the *static single assignment form* (SSA) as the source program representation since it is better usable with the solver-based approach. The concept of SSA and the conversion of the source program into this form is described in detail in Section 2.4.

In order for 2LS to be usable for larger programs, it is essential to use interprocedural analysis. This includes computing summaries for individual functions from the source program. Formal definition of summaries and their usage for the analysis is explained in Section 2.5.

The concepts described within this chapter are used in various analysis present in the current implementation of 2LS. This mainly includes the analysis of the values of numerical variables (using the polyhedra abstract domain), the termination analysis, and the analysis of equalities among variables.

2.1 Source Programs as Transition Systems

The following description views the source program as a transition system. A *program state* $\boldsymbol{x}$ is the current value of all program variables (including the program counter) and related memory (i.e. the stack and the heap). Let S be a set of program states, and let

the *transition relation* $\tau \subseteq S \times S$ define for each state a set of its possible successors in the program execution.

Assume a sequence of sets of states $S_0 S_1 \dots S_k$ such that $\forall 0 \leq i < k : (S_i, S_{i+1}) \in \tau$. We denote $S_k = \tau^k(S_0)$ the set of states *reachable from* S_0 *after* k *execution steps*. If I is the set of all possible initial states of a program, then the set of *all reachable states* S_r is the least fixed point of τ starting from I defined as:

$$S_r = \bigcup_{i \in \mathbb{N}} \tau^i(I). \tag{2.1}$$

Informally, S_r is the set of all states that the program can get into during its execution.

2.2 Abstract Interpretation

Abstract interpretation is a static analysis technique based on an over-approximation of the set of reachable states of the source program. Generally, the set of all reachable states is not computable. However, since it is usually needed to reason about a certain program property only, to prove this property it is sufficient to approximate program states as elements of a simpler domain, called the *abstract domain*.

Having the *concrete domain* P of program states, we create the abstract domain Q. An element of the abstract domain, called an *abstract value*, corresponds to an element from the concrete domain, which is typically a set of concrete program states. Along with the abstract domain, we define two functions [7]:

- The *concretisation function* defines a mapping from an abstract value to a value of the concrete domain. Formally $\gamma : Q \to P$ and $\gamma(q)$ is a concrete value represented by q.
- The *abstraction function* defines mapping from a concrete value to an abstract value from the abstract domain. Formally $\alpha : P \to Q$ and $\alpha(p)$ is the most precise abstract value in Q whose concretisation contains p.

An abstract interpretation I of a program is then a tuple [8]:

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau^{\#}) \tag{2.2}$$

where

- Q is the abstract domain (along with well-defined abstraction and concretisation functions),
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\sqcup : Q \times Q \to Q$ is the *join operator*, $(Q, \sqcup, \top)$ is a complete semilattice,
- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering on $(Q, \sqcup, \top)$ defined as $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$,
- $\tau^{\#} : \mathit{Instr} \times Q \to Q$ defines the interpretation of *abstract transformers*.

Abstract interpretation approximates the set of reachable states by computing the fix-point of $\tau^{\#}$ in the abstract domain. The result is one abstract value for each execution point of the source program. In case multiple abstract values are obtained (because of multiple execution paths entering the program location), these are accumulated into one using the join operator. The properties of the analysed program are then checked in the computed abstract values. The soundness of the analysis is ensured using a *Galois connection* between the concrete and abstract domains. We say that $(P, \leq, Q, \sqsubseteq)$ is a Galois connection if and only if $(P, \leq)$ and $(Q, \sqsubseteq)$ are partially ordered sets, and there is a following relation between abstraction and concretisation functions [8]:

$$\begin{aligned} &\forall p \in P, q \in Q : \\ &\qquad p \leq \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q. \end{aligned} \tag{2.3}$$

Since the computed abstract value is an over-approximation of the set of all reachable concrete program states, abstract interpretation may generate a *false positive*. It is a situation when a property does not hold for the computed abstract semantics, but it holds for the set of all reachable program states. This incoherence is usually caused by the fact that an abstract value represents multiple concrete program states and may represent also states that are not reachable in the real program. The objective is to minimize the number of the false positives. This may be achieved, for example, by choosing a more precise abstract domain or by a combination with other static analysis approaches (as 2LS does).

2.3 Template-based Verification

This section formally explains the approach to abstract interpretation adopted in the 2LS framework. The key phase of the abstract interpretation part of the *k*I*k*I algorithm is an *inference of inductive invariants*. This problem, which can be expressed in (existential fragment) of second-order logic, is reduced to the problem expressible in quantifier-free first-order logic using so-called *templates*. This reduction enables 2LS to use an SMT solver for automated inference of loop invariants and function summaries. These are then used to check various properties of the analysed program. The whole concept is focused on finite state systems since 2LS uses bit-vectors to analyse software [4].

2.3.1 Program Verification Using Inductive Invariants

2LS uses an SMT solver to reason about programs, thus we adapt the formalisation of source programs from Section 2.1 to use logical formulae for the below presentation. The state of a program is described by a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—the states in the set are defined by models of the formula. Given a vector of variables $\boldsymbol{x}$, a predicate $\mathit{Init}(\boldsymbol{x})$ is the predicate describing the initial states. A transition relation is described as a formula $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$. From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $\mathit{Init}(\boldsymbol{x})$. This is, however, difficult to compute, so instead an *inductive invariant* is used. Inv is an inductive invariant if it has the property:

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

An inductive invariant defined as above is a description of a fixed-point of the transition relation. However, it is not guaranteed to be the least one, nor to include $\mathit{Init}(\boldsymbol{x})$. Moreover,

there are predicates which are inductive invariants, but are not sufficient to be used for proving any properties of the source program (such as predicate *true*, which describes the complete state space) [4]. That is why we will try to compute such invariants that approach the least fixed-point, so that it is enough to use them to check a given property.

A verification task does often require showing that the set of all reachable states does not intersect with the set of error states denoted $Err(\boldsymbol{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

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

2.3.2 Invariant Inference via Templates

In order to exploit the power of the kIkI algorithm, 2LS uses a solver-based approach to computing inductive invariants. To directly handle Formula 2.5 by a solver, 2LS would need to handle second-order logic quantification. Since a suitably general and efficient second order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant Inv to $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. This restriction corresponds to the choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for the template parameters:

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

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\boldsymbol{d}$ as candidates for template parameters $\boldsymbol{\delta}$. For a value $\boldsymbol{d}$, the template formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ is an invariant if and only if Formula 2.7 is unsatisfiable.

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

From the abstract interpretation point of view, $\boldsymbol{d}$ is an abstract value, i.e. it represents (*concretises to*) the set of all program states $\boldsymbol{x}$ that satisfy the formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\boldsymbol{x}, \bot) \equiv false$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv true$ [4].

Formally, the concretisation function γ is same for each abstract domain:

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

As for the abstraction function, it is essential to find the most precise abstract value representing a concrete program state. 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 partially ordered set with ordering $\sqsubseteq$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv$ *true*, existence of such a minimal value $\boldsymbol{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\boldsymbol{d} = \bot$ and iteratively solves the below quantifier-free formula (corresponding to the second disjunct in Formula 2.7) using an SMT solver:

$$\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg(\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})). \tag{2.10}$$

If the formula is unsatisfiable, then an invariant has been found, otherwise the model of satisfiability is returned by the solver. The model represents a counterexample to the current instance of the template being an invariant. The value of the template parameter $\boldsymbol{d}$ is though refined by joining with the obtained model of satisfiability using the domain-specific join operator $\sqcup$ [4].

2.3.3 Incremental Solving

The solver approach used in 2LS is based on a so-called *incremental solving*. This technique aims at checking whether satisfiability of a problem is preserved when the clause set is incremented with new clauses. Instead of re-solving the whole problem, the information from the original problem is used to speed up the solution of the new problem. The original problem (before adding the clauses) is though considered satisfiable, and only the impact of the new clauses is checked [11].

In 2LS, this concept is used as follows. First, the below formula is passed to the solver:

$$\mathit{Init}(\boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}'). \tag{2.11}$$

Providing that a valid source program is passed, Formula 2.11 is satisfiable (each program state has a successor) and is considered to hold in all following iterations. After that, only the current instance of the template formula is passed to the incremental solver in each iteration.

Instead of representing source program as a transition system, it is equivalent and more efficient to convert it into the static single assignment form (SSA), which represents the logical formula describing the whole program. Since the SSA form explicitly expresses control flow, it corresponds to the whole Formula 2.11 and removes the need to (directly) implement the abstract transformers. The SSA form used and the conversion of the source program into it is described in Section 2.4.

2.4 SSA Encoding of Source Programs

As the previous section stated, 2LS translates the program into the *single static assignment form* (SSA). It is a well-known concept of an intermediate program representation. Its general principles are introduced in Section 2.4.1.

For an acyclic code, SSA is a formula that represents exactly the strongest post condition of running the code. 2LS extends the standard SSA form by an over-approximation of the loops so that it allows one to reason about abstractions of the program using a solver [4]. This conversion of the loops, along with other modifications of the standard SSA used in 2LS, are explained in Section 2.4.2.

The mechanism of the transformation of the source program into the SSA form and an example of such conversion are stated in Section 2.4.3.

2.4.1 The General Notion of SSA

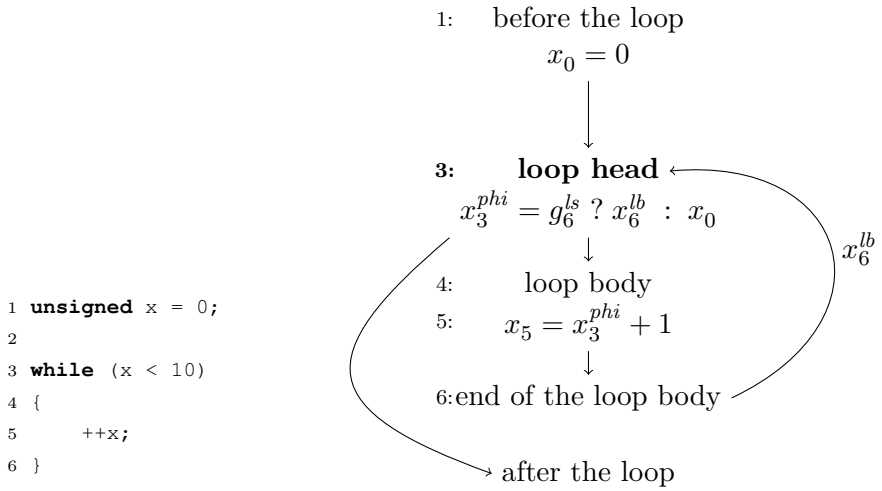
Generally, SSA is an intermediate program representation satisfying the property that each variable is assigned at most once. A translation into the SSA thus involves separating each variable v into several variables v_i. When a node i of the original program contains an assignment to v, it is replaced by an assignment to v_i. Every usage of v is replaced by the appropriate variable v_i where i is the last node where v was assigned before the given use of v.

In order to always have exactly one node of the last assignment of v, additional assignments must be introduced at join points of the original program. These are called Φ (*phi*) nodes and have a form of an assignment $x = \Phi(y, z)$. This expression means that x is assigned the value of y if the control reaches this node via the first entering edge, and x is assigned the value of z if the control reaches the node via the second entering edge [2].

The logical formula corresponding to the original program is then a conjunction of SSA formulae for all program statements.

2.4.2 SSA Used in 2LS

The SSA form used in 2LS extends the general concepts introduced in Section 2.4.1. In order to be usable in the incremental solver, SSA is made acyclic by cutting the loops at the end of the loop body. The example of this conversion is given in Figure 2.1. This figure explains how SSA variables express the control flow in a simple loop [4].



(a) A loop in C (b) Encoding of the loop in SSA

Figure 2.1: Conversion of loops in the SSA form used in 2LS

The loop has been cut at the end of its body: instead of passing the version of x from the end of the loop body (x_5) back to the Φ node in the loop head, a free "loop-back" variable x_6^{lb} is passed. The choice of the value of x in the Φ node is made non-deterministically using the free boolean "loop-select" variable g_6^{ls}. This way, the SSA form is made acyclic, and though it always holds when passed to the solver (which is needed by incremental solving since it represents Formula 2.11, as explained in Section 2.3.3).

Since x_6^{lb} and g_6^{ls} are free variables, this representation is an over-approximation of the actual program traces. The precision can be improved by constraining the value of x_6^{lb} by means of a *loop invariant*, which will be inferred during the analysis. A loop invariant describes a property in the given abstract domain that holds at the loop entry (x_0) and at the end of the loop body (x_5) and though can be assumed to hold on the feedback variable x_6^{lb} [4]. Using the interval numerical domain, the invariant for x_6^{lb} from the example in Figure 2.1 could be:

$$g_6^{ls} \Rightarrow (x_6^{lb} \geq 1 \wedge x_6^{lb} \leq 10). \tag{2.12}$$

The example showed data-flow variables, which correspond to the original program variables. In addition to these, 2LS uses guard variables that capture the branch conditions from conditionals and loops. For example, the SSA form from Figure 2.1 would contain guard variable g_4:

$$g_4 = x_3^{phi} < 10. \tag{2.13}$$

This variable guards the reachability of the assignment of x_5 and is used during the inference of invariant from Equation 2.12.

To facilitate interprocedural analysis, the SSA used in 2LS contains placeholders for function calls in the form $h_i(\boldsymbol{x}_i^{p\_in}, \boldsymbol{x}_i^{p\_out})$ which stands for i-th invocation of the function h with input and output arguments $\boldsymbol{x}_{h_i}^{p\_in}$ and $\boldsymbol{x}_{h_i}^{p\_out}$, respectively. These placeholders assure that the function calls are initially havocked (over-approximated) and can be constrained by computing *function summaries* (see Section 2.5) [5].

Pointer-typed variables have special handling in the SSA used in 2LS. Since this is closely related to the heap analysis performed within this work, it is described in detail later in Section 3.3.

2.4.3 Conversion of the Source Program into SSA

The 2LS framework is built over the CPROVER verification framework. CPROVER provides a compiler for C programs, which parses a C program into its own internal representation called a *GOTO program* [1]. It represents the source program in the form of a control-flow graph containing the locations with statements and the edges between them. Since 2LS uses the SSA form during the analysis, it performs a transformation from a GOTO program into the SSA. This transformation is done in a standard manner as described in Section 2.4.1 with 2LS-specific modifications explained in Section 2.4.2:

- Each variable is split into multiple "versions" for each of its assignments. An assignment of a variable x at location i introduces a fresh symbol x_i which is used at the left-hand side of the assignment. Variables occurring on the right-hand side are renamed to their last assigned versions.

- For each conditional statement and for each loop, a Φ node is introduced for every variable that is altered within the conditional (loop). The choice between two values in a Φ node is controlled using the branch condition in case of a conditional, and a free boolean variable in case of a loop (due to loops cutting as explained in Section 2.4.2).

- A guard variable is introduced for the first location of each basic block from the GOTO program. The guard variable captures the condition of reachability of the given basic block in the source program. This mainly applies to branches of conditional statements and loop bodies.

- Function calls are replaced by the over-approximating placeholders.
- The operations manipulating heap objects are treated specially. Their transformation is newly designed within this project, therefore it is described later, in Section 3.4.

```
1  void main()
2  {
3      unsigned x = 0;
4
5
6      while (x < 10)
7      {
8          ++x;
9      }
10
11     assert(x == 10);
12 }
```

(a) The C program

```
1
2  g_2 = TRUE
3  x_3 = 0
4
5  g_5 = g_2
6  x_6^phi = (g_9^ls ? x_9^lb : x_3)
7  g_7 = (x_6^phi < 10) && g_5
8  x_8 = 1 + x_6^phi
9
10 g_10 = !(x_6^phi < 10) && g_5
11 x_6^phi = 10 || !g_10
12
```

(b) The corresponding SSA

Figure 2.2: Conversion from a C program to SSA

To better understand conversion of a C program, we give an example in Figure 2.2 [4]. Line 2 is the entry location of the program. It is always reachable, therefore g_2 is set to *true*. The definition of x is done at line 3. The head of the loop contains a Φ node (line 6) and since it is directly reachable from the beginning of the `main` function, its guard g_5 is same as the guard of the entry point (g_2). The guard g_7 at line 7 expresses that the loop body is only reachable if the loop head is reachable (g_5) and the loop condition is true ($x_6^{phi} < 10$). Line 8 sets the new value of x. The guard g_{10} at line 10 captures the fact that the location after the loop is reachable when the loop condition is false. Finally, line 11 requires x to be equal to 10 once the assertion is reachable (g_{10} is true).

2.5 Interprocedural Analysis

C programs are typically composed of multiple functions. To correctly analyse such a program, provided it is not recursive, it might be inlined first (by replacing the function calls by the corresponding function bodies). Although this simplifies the analysis (the whole program is in one function), it also brings inefficiency since the inlined program might be much larger than the original one, which can prolong the analysis.

Even though the 2LS framework offers the possibility of full inlining of the program (by using the `--inline` switch), it is designed to perform interprocedural analysis, where each function of the original program is analysed separately.

In this section, we introduce the basic concepts of the interprocedural analysis in 2LS. The original implementation does not handle passing function arguments by reference (pointers). Since this is crucial for analysis of heap manipulation, we implement this mechanism within this work. For that reason, this section describes just the original interprocedural analysis, where function arguments can be passed by value only. The design of passing the arguments by reference is then introduced in Section 3.7.

2.5.1 Input and Output Variables

A function f is specified by its input variables $\boldsymbol{x}_f^{in}$, output variables $\boldsymbol{x}_f^{out}$ (usually referred as formal inputs and outputs), and by its SSA form that has been described in Section 2.4. In the SSA form, a function call of f in a node i is represented by the placeholder $f_i(\boldsymbol{x}_{f_i}^{p\_in}, \boldsymbol{x}_{f_i}^{p\_out})$ where $\boldsymbol{x}_{f_i}^{p\_in}$ and $\boldsymbol{x}_{f_i}^{p\_out}$ represent the actual input and output arguments, respectively. This placeholder havocs the function call and can be constrained by computing a function summary [5].

Input variables $\boldsymbol{x}_f^{in}$ of a function f include parameters $\boldsymbol{param}_f$ and global variables $\boldsymbol{glob}_f^{in}$. For global variables, we only consider those that are actually accessed inside the function. Output variables only contain global variables $\boldsymbol{glob}_f^{out}$, which include same variables as $\boldsymbol{glob}_f^{in}$ (potentially in different SSA versions). The return value of the function is denoted r_f and included in $\boldsymbol{glob}_f^{out}$.

All of the above variables are in the SSA form. The input variables are used without any indices since it is not known where they have been assigned last time. The output variables that are actually written by the function are in the version of their last assignment before the end of the function, while those that are not written (only read) remain in the same form as the input ones.

2.5.2 Function Abstractions

During the function analysis, we use multiple abstractions based on the concept of invariants introduced in Section 2.3.1:

- An *invariant* is a predicate Inv such that:

$$\begin{aligned}\forall \boldsymbol{x}, \boldsymbol{x}' : (&Init(\boldsymbol{x}) \implies Inv(\boldsymbol{x})) \wedge \\ &(Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}')).\end{aligned} \tag{2.14}$$

  Invariants abstract the set of reachable states in the program. If we project the invariant to a subset of variables $\boldsymbol{x}_{loop} \subseteq \boldsymbol{x}$ containing loop-back variables for a loop, we obtain a so-called *loop invariant* $Inv(\boldsymbol{x}_{loop})$. This can be used to constrain the values of loop-back variables (as shown in Section 2.4.2).

- A *summary* abstracts the behaviour of a function. It describes how a function f transforms its formal inputs into outputs. Given an inductive invariant Inv, input and output variables $\boldsymbol{x}^{in}$ and $\boldsymbol{x}^{out}$, and a predicate $Init_f(\boldsymbol{x})$ describing the initial states of the function, a summary of the function f is a predicate Sum such that:

$$\begin{aligned}\forall \boldsymbol{x}, \boldsymbol{x}' : (&\boldsymbol{x}^{in} \subseteq \boldsymbol{x} \wedge Init(\boldsymbol{x}) \wedge \\ &Inv(\boldsymbol{x}') \wedge \boldsymbol{x}^{out} \subseteq \boldsymbol{x}') \implies Sum(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}).\end{aligned} \tag{2.15}$$

  The first line of the implication antecedent expresses that the initial states of the function depend on the input variables. After computing an invariant for the output variables (second line), we obtain a summary $Sum(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ of the function. The summary can be later used to constrain the function call placeholder $f_i(\boldsymbol{x}_{f_i}^{p\_in}, \boldsymbol{x}_{f_i}^{p\_out})$ by replacing formal input and output variables $\boldsymbol{x}^{in}$ and $\boldsymbol{x}^{out}$ in $Sum(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ by actual values of inputs and outputs $\boldsymbol{x}_{f_i}^{p\_in}$ and $\boldsymbol{x}_{f_i}^{p\_out}$. Details can be found in Section 2.5.3.

- A *calling context* abstracts the behaviour of a caller function towards a called function. It specifies the context (actual values of input and output variables) that the function is called in. Given an invariant Inv and values of actual function call inputs and outputs $\boldsymbol{x}_{f_i}^{p\_in}$ and $\boldsymbol{x}_{f_i}^{p\_out}$, a calling context for a call of a function f at a call site i is a predicate $CallCtx_{f_i}$ such that:

$$\forall \boldsymbol{x}_i, \boldsymbol{x}_{i+1} : (\boldsymbol{x}_{f_i}^{p\_in} \subseteq \boldsymbol{x}_i \wedge Inv(\boldsymbol{x}_i) \wedge Trans(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \wedge \boldsymbol{x}_{f_i}^{p\_out} \subseteq \boldsymbol{x}_{i+1}) \implies CallCtx_{f_i}(\boldsymbol{x}_{f_i}^{p\_in}, \boldsymbol{x}_{f_i}^{p\_out}). \tag{2.16}$$

  The function call is included in $Trans(\boldsymbol{x}_i, \boldsymbol{x}_{i+1})$, being the transition relation for the location i. A program state at the location i is denoted $\boldsymbol{x}_i$ and contains the actual input arguments $\boldsymbol{x}_{f_i}^{p\_in}$ of the function call. A program state after the execution of the function call is denoted $\boldsymbol{x}_{i+1}$ and includes $\boldsymbol{x}_{f_i}^{p\_out}$. After computing an invariant for $\boldsymbol{x}_i$ we obtain the calling context $CallCtx_{f_i}(\boldsymbol{x}_{f_i}^{p\_in}, \boldsymbol{x}_{f_i}^{p\_out})$ for a call of f at the location i. The calling context can be used during the analysis of the function f to constrain formal inputs and outputs of the function. More detail of the usage of calling contexts along with an example can be found in Section 2.5.4.

All of these concepts depend on invariants which are computed using the templates from a specific domain. Therefore the obtained constraints are abstractions and describe only those properties that are relevant for the domain used.

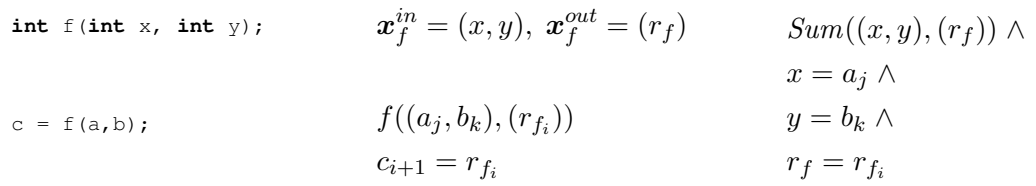
2.5.3 Function Calls Constraining

Function summaries abstract the behaviour of functions and are used to constrain the function call placeholders in the SSA of the caller function. This simulates the effect of the called function in the call site.

Before analysing a function, 2LS first analyses (computes summaries) of all called functions. After that, each function call placeholder f_i is replaced in the solver by the corresponding summary $Sum(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out})$ of the callee function f (having the form of a logical formula). A problem is that the formal input and output variables of the callee function (occurring in the summary) and the actual inputs and outputs at the call site are generally different. In order for the callee summary to simulate the effect of the function on caller's variables, one need to bind formal and actual input and output variables in the solver.

For a better illustration we give a simple example. Let f be a function in C declared and then called as shown in Figure 2.3a. Supposed there are no global variables, the inputs and outputs of f and an example of the SSA corresponding to the call are shown in Figure 2.3b (j and k are the locations of the last assignments of a and b in the caller function, respectively). Note that the call is replaced by a placeholder where a fresh symbol r_{f_i} is introduced for the return value and it is assigned to c in the next location. This is already done by the GOTO program parser in the CPROVER framework. After a summary $Sum((x, y), (r_f))$ of f is computed, we replace the placeholder by an expression in Figure 2.3c. This contains the summary itself and the bindings between formal and actual inputs and outputs of the function.

We propose an algorithm for the binding between formal and actual input and output variables of a function call. The following presentation assumes that a function f is called from a call site f_i. In order to distinguish between formal and actual variables, Table 2.1 shows naming of vectors of variables in the function f and in the call site f_i. Note that



(a) Declaration and call in C (b) Inputs, outputs, and SSA (c) Replacing formula

Figure 2.3: Example of function call constraining

global variables are also different since they occur in different SSA versions in the caller and callee functions.

Table 2.1: The symbols used in binding algorithms

Symbol	Meaning
$\boldsymbol{param}_f$	formal parameters of f
$\boldsymbol{glob}_f^{in}$	formal input global variables of f
$\boldsymbol{glob}_f^{out}$	formal output global variables of f
$\boldsymbol{arg}_{f_i}$	arguments (actual parameters) of the call site f_i
$\boldsymbol{glob}_{f_i}^{p\_in}$	actual global variables at the input of the call site f_i
$\boldsymbol{glob}_{f_i}^{p\_out}$	actual global variables at the output of the call site f_i

We also introduce a function NAME, which takes an SSA variable v and returns the name of the corresponding original program variable without the SSA suffix. The function BIND(v_1, v_2) creates a binding between the variables v_1 and v_2 (i.e., it adds an assumption $v_1 == v_2$ to the solver).

Using these helper functions, the binding between the call site and the callee variables is done using two procedures shown in Algorithm 1.

Algorithm 1 Binding between the call site and the called function

procedure BINDGLOBALS
  for all $x \in \boldsymbol{glob}^{in}, x' \in \boldsymbol{glob}_{f_i}^{p\_in}$. NAME(x) = NAME(x') **do**
    BIND(x, x')
  for all $x \in \boldsymbol{glob}^{out}, x' \in \boldsymbol{glob}_{f_i}^{p\_out}$. NAME(x) = NAME(x') **do**
    BIND(x, x')
procedure BINDPARAMS
  for $j \leftarrow 0..(|\boldsymbol{param}| - 1)$ **do**
    BIND$(\boldsymbol{param}[j], \boldsymbol{arg}_{f_i}[j])$

2.5.4 Context-Sensitive Summaries

For some programs, it is useful (and sometimes necessary for precise analysis) that values of the input variables of a function are constrained with respect to the actual argument values

that the function is called with. In this case, when analysing a function f that is called for the first time in a call site f_i, its *calling context* $CallCtx_{f_i}$ is computed first. The calling context is formally defined in Formula 2.16 and depends on the invariant in the entry of the location i. This is computable before the analysis of f since f is called in location i for the first time. The computed calling context is then used as a precondition for the analysis of f and the inferred summary is stored in the form $CallCtx_{f_i} \Rightarrow Sum_f$.

Using the example from Figure 2.3, if f was called in the call site f_i with a calling context: $CallCtx_{f_i}((a_j, b_k), (r_{f_i})) = (a_j \leq 0 \wedge b_k \geq 0)$, the computed summary would be stored in the form: $(x \leq 0 \wedge y \geq 0) \Rightarrow Sum_f((x, y), (r_f))$.

Since f might be called multiple times within different contexts, the computed summaries are reusable only if the current calling context is subsumed by calling contexts $\bigvee_i CallCtx_{f_i}$. In case it is not, the summary is recomputed and joined conjunctively with previous summaries [5].

Chapter 3

Design of a Heap Analysis for 2LS

In the previous chapter, we described the current state of the art of the 2LS framework. It is well-usable for various analyses, such as the numeric variables analysis or the termination analysis.

The goal of this project is the integration of a new type of analysis—a heap manipulation analysis. We will focus on analysing the shape of dynamic data structures (mainly singly and doubly linked lists). In order to achieve this, there are multiple tasks to be fulfilled.

The most important one is to design a new abstract domain that can describe desired properties of the program heap. In 2LS, this involves a proposal of a suitable form of the template for computing invariants, specifying what an abstract value will be, and creating the corresponding join algorithm. These can be found in Sections 3.5 (the abstract domain) and 3.6 (the abstract value synthesis algorithm). Since shape analysis has already been implemented in various different tools, we can make use of the existing abstract domains. Therefore we first explore the existing work on formal heap analysis and verification in Section 3.1.

Apart from creating the abstract domain, there are more problems to be solved. The first one is related to the SSA form, which 2LS uses for representation of the source programs. The problem is that the SSA uses symbolic names for program variables. Dynamically allocated objects do not have any names (because they are accessed via pointers only), and therefore we have to introduce new symbols to represent them. Moreover, a dereference of the same pointer might result into different heap objects at different execution points. To ensure a correct dereferencing, 2LS runs a simple static *points-to* analysis. Since the analysis is currently not complete (it cannot handle some commands, or function calls) we extend it and formally describe in Section 3.3. Afterwards, the transformation of heap-manipulating operations into the SSA form is based on the results of this analysis. The conversion of typical operations is explained in Section 3.4.

The second problem comes with the interprocedural analysis. Generally, a function can alter objects that are neither global nor passed to the function as parameters but are pointed by those. This is called a side effect of a function and must be included in the function summary. Since 2LS currently passes global objects and actual arguments to the function only, we will introduce passing of heap objects. The problem becomes even more complicated when a function accesses and alters an abstract heap object such as a linked list. The proposed approach to the interprocedural analysis of heap-related properties is described in Section 3.7.

3.1 Related Work on Heap Verification Techniques

Before designing an abstract domain for the description of shape invariants in 2LS, we explore the existing approaches to the heap (shape) verification.

Most of the current tools use intermediate representations of the analysed programs in the form of control-flow graphs (CFG). They compute the abstract state (representing the set of reachable concrete program states) at each program location. However, this approach is not usable in 2LS since the source program representation is in the acyclic SSA form (thus a part of the control-flow is omitted because of the loops cutting). Moreover, a solver for quantifier-free first-order logic is used, and so we have to use an abstract domain that allows us to reason about the shape of a heap using quantifier-free formulae only.

An approach we found to be currently the most suitable for our purposes is based on the so-called *storeless semantics*. Contrary to tools based on more popular store-based program semantics describing the shape of the heap using various logics [14, 13, 16], automata (Forester [10]), or graphs (Predator [9]), which closely correspond to the real state of the heap, storeless approaches represent heap as a set of pointer access paths [15].

An *access path* does not concretely express the state of the heap, it only describes which dynamic objects (i.e. objects allocated by `malloc` function) are reachable from a pointer. Using a set of access paths for each pointer in the program, one can efficiently describe the shape of (the reachable part of) the heap. The approach based on access paths is used e.g. in a tool built over the CPROVER framework [3] (where the heap is described as a conjunction of predicates), or various other tools [6, 15, 12].

The main difference between these and our proposed approach is that they use CFGs and compute the sets of reachable program states iteratively using the abstract interpretation approach. On the other hand, 2LS uses an acyclic SSA form in an incremental SMT solver-based approach. This allows a simpler creation of domains and could also bring a possibility to combine our shape analysis with other analyses already present in 2LS.

We propose an abstract domain for 2LS to describe the shape properties of the program heap. Before that, we have to introduce some changes to the SSA form, so that it is usable along with the proposed domain in the SMT solver.

3.2 Preliminaries and Notation

In this chapter, we assume that the source programs are defined over the following finite sets of objects:

- Var: a set of all statically allocated objects (variables). We assume each variable has its unique name.
- Obj: a set of dynamically allocated objects (on the heap). In 2LS, one dynamic object corresponds to one allocation site, i.e. it might represent an abstraction of multiple concrete heap objects allocated in a loop (such as a segment of a linked list).
- $Ptr \subseteq (Var \cup Obj)$: a set of all pointer-typed objects (both static and dynamic). These either hold an address of an object in the memory or `null`.
- $Str \subseteq Obj$: a set of all structure-typed heap objects.
- Fld: a set of all fields of structured types.

- *Instr*: a set of all program locations.

We also use $\mathbb{B}$ to denote the standard Boolean domain. We use the ordering $\leq$ on $\mathbb{B}$ as follows:

$$\forall x, y \in \mathbb{B}.\ x \leq y \Leftrightarrow (x \Rightarrow y). \tag{3.1}$$

3.3 Points-to Static Analysis

Using the SSA form in a quantifier-free first-order SMT solver in the way 2LS uses it brings some problems. One of them is the fact that each memory object must be identified by its unique name. This would be easy if each object was accessed directly through an associated variable in the original program. However, when pointers are used (which is quite common in low-level code), the situation gets more complicated since a single pointer variable can be dereferenced to different memory objects.

For that reason, we perform a static *points-to* analysis prior to the conversion into the SSA. This analysis determines for each pointer a set of memory objects it can be dereferenced into in each program location where it is used. In case the pointer can be `null`, or its value can be unknown (e.g. because it has not been initialised), this information is also determined. The analysis does not check for errors (such as `null` dereferences), it simply collects all possibilities of pointer dereferencing.

The points-to analysis used is based on a classic abstract interpretation. The abstract domain $\mathcal{PT}$ is defined by a function mapping a pointer to an element of the Cartesian product:

$$\mathcal{PT} = Ptr \rightarrow (2^{Obj \cup Var} \times \mathbb{B} \times \mathbb{B}). \tag{3.2}$$

The abstract value at the program location $i \in Instr$ defines for each pointer $p \in Ptr$ in i a tuple $PointsTo(p_i) \in (2^{Obj \cup Var} \times \mathbb{B} \times \mathbb{B})$ containing the following information:

- A set $ValueSet(p_i) \subseteq (Obj \cup Var)$ holding a set of memory objects that p can be dereferenced into at location i.
- A boolean predicate $isnull(p_i)$ determining if p can be `null` at i.
- A boolean predicate $unknown(p_i)$ denoting that the value of p might be unknown (non-deterministic) at location i.

The points-to analysis is run on a GOTO program, which has the form of a control-flow graph. The algorithm follows the abstract interpretation approach of finding the least fixed point of the abstract domain lattice for every program location. To be able to find a fixed point, we have to define a partial ordering $\sqsubseteq$ on the abstract domain $\mathcal{PT}$. The ordering is defined for two *PointsTo* values for the same pointer only.

$$\begin{aligned} &\forall PointsTo(x), PointsTo(x)' \in \mathcal{PT}. \\ &\quad PointsTo(x) \sqsubseteq PointsTo(x)' \Leftrightarrow ValueSet(x) \subseteq ValueSet(x)' \wedge \\ &\qquad\qquad\qquad\qquad\quad unknown(x) \leq unknown(x)' \wedge \\ &\qquad\qquad\qquad\qquad\quad isnull(x) \leq isnull(x)' \end{aligned} \tag{3.3}$$

The join algorithm is done element-wise. For *ValueSet* sets, the join is the set union ($\cup$), whilst for boolean predicates *isnull* and *unknown*, the join is defined as the logical disjunction ($\vee$).

Next, we define the abstract transformers. The effect of basic commands manipulating pointers on the abstract value is defined in Figure 3.1. We support the statements stated in the figure, with possibility of chaining them and combining them with access to structure fields. We assume that the commands take place at the program location i.

<table>
<tr>
<td>Command: <code>p = null</code>

$ValueSet(p_i) := \emptyset$
$isnull(p_i) := true$
$unknown(p_i) := false$</td>
<td>Command: <code>T *p</code> (declare <code>p</code>)

$ValueSet(p_i) := \emptyset$
$isnull(p_i) := false$
$unknown(p_i) := true$</td>
</tr>
<tr>
<td>Command: <code>p = &o</code>

$ValueSet(p_i) := \{o\}$
$isnull(p_i) := false$
$unknown(p_i) := false$</td>
<td>Command: <code>p = q</code>

$ValueSet(p_i) := ValueSet(q_{i-1})$
$isnull(p_i) := isnull(q_{i-1})$
$unknown(p_i) := unknown(q_{i-1})$</td>
</tr>
<tr>
<td colspan="2">Command: <code>*p = q</code>

$\forall p' \in ValueSet(p_{i-1}) : ValueSet(p'_i) := ValueSet(p'_{i-1}) \cup ValueSet(q_{i-1})$
$isnull(p'_i) := isnull(p'_{i-1}) \vee isnull(q_{i-1})$
$unknown(p'_i) := unknown(p'_{i-1}) \vee unknown(q_{i-1})$</td>
</tr>
<tr>
<td colspan="2">Command: <code>p = *q</code>

$ValueSet(p_i) := \bigcup_{q' \in ValueSet(q_{i-1})} ValueSet(q'_{i-1})$
$isnull(p_i) := \bigvee_{q' \in ValueSet(q_{i-1})} isnull(q'_{i-1})$
$unknown(p_i) := \bigvee_{q' \in ValueSet(q_{i-1})} unknown(q'_{i-1})$</td>
</tr>
</table>

Figure 3.1: Abstract transformers for *points-to* analysis

If 2LS is used to perform interprocedural analysis, the points-to analysis is run on each function separately. In order for it to be correct, we have to define the initial abstract value (since the function may have pointer-typed inputs) and an abstract transformer for a function call. We discuss our approach to interprocedural analysis later, hence we will also present these concepts there (see Section 3.7.1).

3.4 Representation of Heap Operations in SSA

We use the information obtained during the points-to analysis described in the previous section during our transformation of a GOTO program into the SSA form. We now present how typical heap-manipulating operations are represented in the SSA:

malloc Each call of the *malloc* function is replaced by an instantiation of a new abstract dynamic object and returning its symbolic address as a result of the call. This ensures that all heap objects have their unique names. The replacement of the i-th occurrence

of *malloc* in the source program is as follows:

$$\texttt{malloc}(\texttt{sizeof}(t)) \longrightarrow \&\mathit{dynamic\_object}\$i \tag{3.4}$$

where the type of $\mathit{dynamic\_object}\$i$ is t. The created dynamic object is an abstraction since it corresponds to one allocation site and though can represent multiple concrete objects allocated in a loop.

Memory read In C, this operation has a typical form $\texttt{v} = *\texttt{p}$. Assuming that it takes place at the program location i, we use the triple $\mathit{PointsTo}(p_i)$ to construct the corresponding expression in SSA as follows:

$$\texttt{v} = *\texttt{p} \;\longrightarrow\; v_i == E_p(\,\mathit{ValueSet}(p_i), \mathit{isnull}(p_i), \mathit{unknown}(p_i)) \tag{3.5}$$

where the expression E_p is defined recursively:

$$E_p(\emptyset, \mathit{true}, \mathit{false}) = E_p(\emptyset, \mathit{false}, \mathit{true}) = \mathit{unknown\_object} \tag{3.6}$$

$$E_p(\{o\}, \mathit{false}, \mathit{false}) = o \tag{3.7}$$

$$E_p(V, n, u) = (p == \&o \;?\; o \;:\; E_p(V \setminus \{o\}, n, u)\,). \tag{3.8}$$

The expression E_p generates a case split for each object that p may be dereferenced into (3.8). The last case is a placeholder for an unknown heap object if p might be `null` or non-deterministic (3.6). If neither of the *isnull* and *unknown* predicates is true for p, we use the last object from the $\mathit{ValueSet}(p)$ as the last case split (3.7).

Memory write This operation is dual to the memory read, and, in C, it has a typical form $*\texttt{p} = \texttt{v}$. The transformation at location i uses $\mathit{ValueSet}(p_i)$:

$$*\texttt{p} = \texttt{v} \longrightarrow \bigwedge_{o \in \mathit{ValueSet}(p_i)} o_i == (p_i == \&o \;?\; v \;:\; o_{i-1}). \tag{3.9}$$

In this case, the memory location referenced by p is assigned a value. Therefore, we create the SSA equality for each object from the $\mathit{ValueSet}(p_i)$ where it is assigned the new value v in case it was pointed by p, and it keeps its previous value otherwise (o_{i-1}). The solver assigns a concrete address to p_i during solving, though one equality $p_i == \&o$ will be true only, and a single object o from $\mathit{ValueSet}(p_i)$ will be updated.

Load and store These are typical operations manipulating dynamic data structures, such as linked lists. Their form in C is $\texttt{v} = \texttt{p}{\rightarrow}\texttt{f}$ and $\texttt{p}{\rightarrow}\texttt{f} = \texttt{v}$ for *load* and *store*, respectively. Since the arrow operator in C might be rewritten using the dereference and the dot operator ($\texttt{p}{\rightarrow}\texttt{f}$ is equivalent to $(*\texttt{p}).\texttt{f}$), these operations are analogous to memory read and write. The only difference is that a field of a structure is accessed, instead of the whole object.

In 2LS, the structures are split into their fields and each field of a structured object is considered a separate variable. This means that the transformation of *load* will be done similarly to Equation 3.5:

$$\texttt{v} = \texttt{p}{\rightarrow}\texttt{f} \;\longrightarrow\; v_i == E_{p,f}(\,\mathit{ValueSet}(p_i), \mathit{isnull}(p_i), \mathit{unknown}(p_i)). \tag{3.10}$$

The expression $E_{p,f}$ is defined as E_p where object fields are used instead of objects—$\mathit{unknown\_object}.f$ in place of $\mathit{unknown\_object}$ in Equation 3.6 and $o.f$ in place of o in Equation 3.7 and in the second operand of the ternary operator in Equation 3.8.

Analogously, the *store* operation is represented similarly to Equation 3.9:

$$\texttt{p}\rightarrow\texttt{f} = \texttt{v} \longrightarrow \bigwedge_{o \in ValueSet(p_i)} o.f_i == (p == \&o \ ? \ v \ : \ o.f_{i-1}). \tag{3.11}$$

3.5 Template Heap Domain

We now present our abstract domain for modelling the shape of the heap. In 2LS, the abstract domain is specified by a template—a fixed quantifier-free first-order logic formula describing the desired property of a program. In Section 3.1, we proposed to describe the shape of the heap using a set of *access paths*. This section shows how these are used inside a template, and how they are transformed into an appropriate formula usable in the 2LS solver working over the theory of bit-vectors and arrays.

3.5.1 Template Form

We use a simple model of the heap, which does not consider the pointer arithmetic. We restrict the template to use only those memory objects that describe the shape of the heap—the pointer-typed objects (defined by the set *Ptr*) and the structure-typed heap objects (defined by the set *Str*). The formula represented by a template is then a conjunction of expressions, so-called *template rows*, where each row corresponds to one of these memory objects. Since two types of objects are considered (pointer-typed and structure-typed), we split the template into two parts:

- The *pointer* part describes the points-to relation between pointers and pointed objects (which can be pointer-typed, too). The formula of the pointer part is a conjunction of *pointer rows*:

$$\mathcal{T}^P \equiv \bigwedge_{p \in Ptr} \mathcal{T}_i^P(p, d_i^P). \tag{3.12}$$

  Here, $\mathcal{T}_i^P(p, d_i^P)$ is the i-th pointer row that describes the points-to relation of the pointer p. It is a parametrized formula with d_i^P being the abstract row value from the domain δ^P that is defined as:

$$\delta^P = 2^{Obj \cup Var}. \tag{3.13}$$

  The row value specifies a set of (abstract) objects that the row pointer p may point to. A pointer row is thus defined as a formula:

$$\mathcal{T}_i^P(p, d_i^P) \equiv \bigvee_{o \in d_i^P} p = \&o. \tag{3.14}$$

- The *object* part describes the shape of dynamic data structures on the heap using access paths. The formula is a conjunction of *object rows*, where one row is defined for each pair of a heap object and its (pointer-typed) field:

$$\mathcal{T}^O \equiv \bigwedge_{(o,f) \in Str \times Fld} \mathcal{T}_i^O((o, f), d_i^O). \tag{3.15}$$

Here, $\mathcal{T}_i^O((o,f),d_i^O)$ is the i-th object row, which is a formula that characterizes a set of access paths leading from the object o via its pointer field f. An access path for a tuple (o,f) is specified by a destination object and a set of (abstract) objects that it passes through, thus the abstract domain δ^O of the object row value d_i^O is:

$$\delta^O = 2^{Obj \times 2^{Obj}}. \tag{3.16}$$

The object row is then a formula:

$$\mathcal{T}^O((o,f),d_i^O) \equiv \bigwedge_{(d,O)\in d_i^O} path(o,f,d)[O]. \tag{3.17}$$

The semantics of the *path* predicate is defined in Section 3.5.2.

The abstract domains δ^P and δ^O for both pointer and object rows contain two special values $\bot$ and $\top$, which represent the empty set and the whole domain, respectively (as required by 2LS, see Section 2.3.2). The formulae corresponding to rows with these values are defined as:

$$\begin{aligned} \mathcal{T}(x,\bot) &\equiv false \\ \mathcal{T}(x,\top) &\equiv true. \end{aligned} \tag{3.18}$$

Finally, using the formulae for the pointer and the object part, we define the template $\mathcal{T}$ for our domain as:

$$\mathcal{T} \equiv \mathcal{T}^P \wedge \mathcal{T}^O. \tag{3.19}$$

3.5.2 The *path* predicate

It remains to define when the *path* predicate is true. Intuitively, this can be defined recursively:

$$\begin{aligned} path(o,f,d) \Leftrightarrow o.f = \&d \vee \\ path(*(o.f),f,d). \end{aligned} \tag{3.20}$$

However, the recursive expression is not usable in the solver, which means we have to remove the recursion. We do this in an approximative way by introducing the set O which contains all heap objects that the path passes through:

$$\begin{aligned} path(o,f,d)[O] \Leftrightarrow o.f = \&d \vee \\ (\exists o' \in O : o.f = \&o' \\ \wedge\ \exists o'' \in O : o''.f = \&d \\ \wedge\ \forall o_1 \in (O \setminus \{o''\})\ \exists o_2 \in O : o_1.f = \&o_2). \end{aligned} \tag{3.21}$$

The definition in Formula 3.21 can be explained as follows. The first equality on the right side of the equivalence indicates the situation when the destination can be reached from the source in one step by following the field f. In case this is not true, we have to express that the destination might be reachable after multiple steps. This is ensured by the consequent conjunction. Its first part describes the first step on the path, the second part describes the last step on the path, and the last part over-approximates the inner steps of the path (linking between the heap objects that the path passes through). Here, the approximation is done in the sense that we do not store the ordering of objects in O and thus this representation describes more paths than the recursive expression. However, this is not a problem since we also compute access paths for all objects in O during the analysis.

3.6 Abstract Value Synthesis Algorithm

The general core of the template parameter synthesis algorithm was indicated in Section 2.3.2. The domain-specific part of the algorithm is implementation of the join procedure between an abstract value and a model of satisfiability returned by the solver. In order not to over-approximate too much it is useful to design the join algorithm such that a minimal value of the template parameter is found.

For the heap domain, we perform the join row-wise. The update of the row parameter is different for every type of the row. It depends on the value assigned to the pointer (for a *pointer* row) or to the field of the dynamic object (for an *object* row) in the model of satisfiability of Formula 2.10. Since the SMT solver used in 2LS is based on the theory of bit-vectors, it always assigns a value to each variable during solving, which in case of the pointer-typed objects is either an address of an object in the memory or `null`.

The update of a *pointer* row parameter value is simple. The object whose address was assigned to the corresponding pointer in the model of satisfiability is added to the set representing the row value. This way, we collect exactly all objects that the pointer can reference. The ordering on the *pointer* row value is defined by the set inclusion.

The update of an *object* row parameter value is more complicated. We first define the *path relation for the field* f denoted P^f as:

$$\forall x, y \in Obj.\ x P^f y \Leftrightarrow path(x, f, y). \tag{3.22}$$

We use this relation to define a partial ordering on the set of all paths in the heap. Since this relation is transitive, the join algorithm creates the transitive closure over the set of all paths according to the following formula:

$$path(o', f, d)[O] \wedge (o.f = \&o') \Rightarrow path(o, f, d)[O \cup \{o'\}]. \tag{3.23}$$

Algorithm 2 presents two functions for updating *pointer* and *object* rows, respectively. Both functions take for parameters a template row of the appropriate type and the corresponding model of satisfiability (SAT) of Formula 2.10.

In case the model of satisfiability gives that the value of the row object might be non-deterministic, we set the value to $\top$, which represents the whole abstract domain.

The creation of the transitive closure over the set of paths is ensured by the two loops in the function UpdateHeapRow. The loop on lines 18-19 adds all paths from the object pointed by $o.f$ into the current row value, whereas the loop on lines 21-23 adds all paths leading from current row object o into all row values that already contain paths leading to o.

3.7 Interprocedural Analysis

Section 2.5 describes the current approach to the interprocedural analysis in 2LS as follows: for each function, the formal input and output variables of the function are determined, then a summary of the function is computed using invariants, and, finally, a binding between the formal and actual values of inputs and outputs at each call site of the function is created.

In the current implementation of 2LS, only parameters and global variables are considered as the inputs or outputs of a function. Therefore, we must introduce a way to pass the objects that are not included in the parameters (nor global variables) but can be reachable from these via chains of pointers to the called function.

Algorithm 2 Join i-th template row value with the satisfiability model

function UPDATEPOINTERROW($\mathcal{T}^P(p, d_i^P)$, model of SAT: $p = v$)
  if v is non-deterministic **then**
    $d_i^P \leftarrow \top$
  else if $v = \texttt{null}$ **then**
    $d_i^P \leftarrow d_i^S \cup \{\texttt{null}\}$
  else
    assume $v = \&o$
    $d_i^S \leftarrow d_i^P \cup \{o\}$
function UPDATEOBJECTROW($\mathcal{T}^O((o, f), d_i^O)$, model of SAT: $o.f = v$)
  if v is non-deterministic **then**
    $d_i^O \leftarrow \top$
  else
    if $v = \texttt{null}$ **then**
      $d_i^O \leftarrow d_i^O \cup \{(\texttt{null}, \emptyset)\}$
    else
      assume $v = \&o'$
      $d_i^O \leftarrow d_i^O \cup \{(o', \emptyset)\}$
      for all $path(o', f, d)[O']$ **do** // *Adding paths leading from* o'
        $d_i^O \leftarrow d_i^O \cup \{(d, O' \cup \{o'\})\}$
    // *Updating rows having paths leading to* o
    for all $\mathcal{T}^O((\bar{o}, f), d_j^O).(o, \bar{O}) \in d_j^O$ **do**
      for all $(d, O) \in d_i^O$ **do**
        $d_j^O \leftarrow d_j^O \cup \{(d, O \cup \bar{O} \cup \{o\})\}$

A set of objects that a pointer can be dereferenced into can be determined from the points-to analysis (Section 3.3) by querying the *ValueSet* set. In order for the points-to analysis to work with interprocedural analysis, we have to introduce some extensions that are presented in Section 3.7.1.

Next, in Section 3.7.2, we show how binding of objects pointed by parameters and global variables between the caller and the callee functions is done. However, we will also show that passing objects directly pointed by function parameters only is not enough to correctly handle functions which alter the existing heap containing recursive data structures. For that reason, in Section 3.7.3, we will extend the approach to be able to handle functions manipulating linked lists.

3.7.1 Interprocedural Points-to Analysis

In order to provide correct interprocedural analysis, we have to introduce the initial abstract value and the abstract transformer for the function call into the points-to analysis. Since each function is analysed separately, both of these concepts require an abstraction of pointed objects—a pointer-typed parameter of a function may initially point to an unknown object and a pointer passed to a function as an argument may point to a different object after invocation of the function. Because of this, we introduce a function POINTED shown in Algorithm 3 that takes a pointer symbol p and creates a new symbol p^{obj} that represents an

abstraction of the object pointed by p. If p^{obj} is pointer typed (which happens when passing a pointer to a pointer), we add it into the Ptr set holding all pointers in the program.

Algorithm 3 Create abstraction of a pointed object

```
1: function POINTED(p ∈ Ptr)
2:     create p^obj // typeof(p^obj) is the type pointed by typeof(p)
3:     if typeof(p^obj) is pointer then
4:         Ptr ← Ptr ∪ {p^obj}
5:     return p^obj
```

Initial Abstract Value

Using the POINTED function, we define the initial value of the points-to analysis of a function f by Algorithm 4.

Algorithm 4 Initial abstract value for function f

```
1: for all p ∈ (param_f ∪ glob_f^in) ∩ Ptr do
2:     INIT(p)
3: function INIT(p ∈ Ptr)
4:     p^obj ← POINTED(p)
5:     ValueSet(p) ← {p^obj}
6:     unknown(p) ← true
7:     if p^obj ∈ Ptr then
8:         INIT(p^obj)
```

The function INIT is called for every pointer-typed input p of the function (which can be parameter or global). It initializes the value set to contain the abstraction of the object pointed by p and sets the *unknown* predicate for p to *true*. If p^{obj} is again a pointer, INIT is called recursively.

Here, p is used without the location index since it is an input to the function. The symbol p^{obj} will be later bound to the corresponding object from the caller function (see Section 3.7.2).

Example To better illustrate how the initial value is used in the SSA, we show an example of a function `chainNode` which takes a pointer to the head of a linked list as a parameter, allocates a new list node, appends it to the beginning of the list, and sets the head to the new node. The function in C and the corresponding SSA form are shown in Figure 3.2.

A new list node is allocated at line 2. The call to `malloc` is transformed into an instantiation of a new dynamic object. The pointer *node* is then dereferenced into this object at line 3. The parameter *ppnode* is dereferenced into $ppnode^{obj}$ at lines 3 and 4.

Function Call Abstract Transformer

Next, we have to create an abstract transformer for the function call command. A call of a function might invoke a side effect on an object pointed by the return value, by an argument of the call, or by a global variable—after the function invocation, a pointer passed to the function might point to another object than before. Therefore, for each

```
1 void chainNode(struct node **ppnode) {
2     struct node *node = malloc(sizeof *node);
3     node->next = *ppnode;
4     *ppnode = node;
5 }
```

(a) The function in C

```
1
2 node_2 = &dynamic_object$0
3 dynamic_object$0.next_3 = ppnode^obj
4 ppnode_4^obj = node_2
5
```

(b) The corresponding SSA

Figure 3.2: A function accessing a pointed object

pointer-typed argument a of the function, we use the function POINTED to get the symbol a^{obj} which abstracts the object pointed by a after invocation of the function, and add it into $ValueSet(a_i)$, where i denotes the location of the function call. The abstract transformer for a function call $f_i(\boldsymbol{arg}_{f_i})$ is described by Algorithm 5.

Algorithm 5 Abstract transformer for a function call $f_i(\boldsymbol{arg}_{f_i})$

1: **for all** $a \in (\boldsymbol{arg}_{f_i} \cup \boldsymbol{glob}_{f_i}^{p\_out}) \cap Ptr$ **do**
2: $\quad a^{obj} \leftarrow$ POINTED(a)
3: $\quad$ ADD(a, a^{obj})
4: **function** ADD(p, o)
5: $\quad ValueSet(p_i) \leftarrow ValueSet(p_i) \cup \{o\}$
6: $\quad unknown(p_i) \leftarrow true$
7: $\quad$ **for all** $p' \in ValueSet(p_i) \cap Ptr$ **do**
8: $\quad\quad o^{obj} \leftarrow$ POINTED(o)
9: $\quad\quad$ ADD(p', o^{obj})

The function ADD takes a pointer p and an object o being the abstraction of a new object created by f and pointed by p after the invocation of f in location i. It adds o to $ValueSet(p_i)$. In case p is a pointer to a pointer, the function is recursively called for all potentially pointed objects.

Example To illustrate the above, we now provide an example. Let p be of type `int**`, x, y be of type `int*`, a, b be of type `int`, and the *ValueSet* sets at location $i-1$ be as follows:

$$ValueSet(p_{i-1}) = \{x, y\}$$
$$ValueSet(x_{i-1}) = \{a\}$$
$$ValueSet(y_{i-1}) = \{b\}.$$

After the call of a function having the declaration `f(int **q)` with argument p at location i, the sets will be:

$$ValueSet(p_i) = \{x, y, p^{obj}\}$$
$$ValueSet(x_i) = \{a, p^{obj^{obj}}\}$$
$$ValueSet(y_i) = \{b, p^{obj^{obj}}\}$$
$$ValueSet(p_i^{obj}) = \{p^{obj^{obj}}\}.$$

Here, p^{obj} and $p^{obj^{obj}}$ abstract new objects of type `int*` and `int`, respectively, that might have been created in the function f. The object p^{obj} needs not to be there since the value of p itself cannot be changed by the function. However, it is not a problem that we included it (in order to keep the algorithm simple), since we only over-approximate the set of all objects that p might point to and the value of p will not change in the SSA form.

3.7.2 Binding Pointed Objects between Functions

Above, we described changes that have to be made in the points-to analysis in order to handle interprocedural analysis. We are able to compute how a single function affects the shape of the program heap by computing the function summary using the proposed shape domain. The computed summary can be used to constrain function call placeholder in the SSA form in the way explained in Section 2.5.3. What we need to extend here, is the algorithm for binding objects between the caller function and the summary, since Algorithm 1 does not consider objects that are not function parameters, but are pointed by those (and thus can be altered by the function summary).

These objects pose a problem in terms of binding. The reason is that the names of the corresponding heap objects might be different between the caller and the callee. To bind them correctly, we make use of the points-to analysis. We particularly query the *ValueSet* sets of corresponding pairs of pointers (each pair composed of an argument and a parameter) to bind objects pointed by these. E.g., in the above example, at the function input, we would bind objects x and y (objects pointed by the argument p at the function call input) to object q^{obj} (an object representing an abstraction of the object initially pointed by the parameter q).

Since we need to bind multiple objects together, we first extend the function BIND that was presented in Section 2.5.3 to be able to create multiple bindings at once. The new function takes sets of variables as arguments (instead of simple variables) and is shown in Algorithm 6. We introduce a binary operator $\triangleleft$ that adds clauses at its right hand side to the clause set of the SMT solver used by 2LS stated at its left hand side.

Algorithm 6 Extended *Bind* function

1: **function** BIND(V_1, V_2)
2: solver $\triangleleft \bigvee_{(v_1,v_2)\in V_1\times V_2}(v_1 == v_2)$

Algorithm 7 shows the complete binding of the heap objects of a function f at a call site f_i. It works with the sets of objects defined in Table 2.1.

We first define the function DEREF taking a set P of pointer-typed objects and a location i as arguments. It symbolically dereferences all pointers in P and returns the set of all objects that can be pointed by these at the location i. The dereferencing is done by querying the *ValueSet* sets of the pointers at line 2. In order to return the correct SSA symbols of the dereferenced objects at location i, we introduce the function SSA, which for a variable v and a location i returns the corresponding SSA symbol—the variable v in the version of its last assignment before i. This function is applied to all elements of the set D of the dereferenced objects and the resulting set is returned (line 3).

Next, we define the function BINDPOINTED. It takes two sets of objects P_1 and P_2 and the corresponding locations i and j. If the sets contain pointers only, the function binds all objects pointed by elements of P_1 at the location i to all objects pointed by elements of P_2

Algorithm 7 Binding pointed objects

```
1:  function DEREF(P, i)
2:      D ← ⋃_{p∈P} ValueSet(p_i)
3:      return {SSA(d, i) | d ∈ D}
4:  function BINDPOINTED(P_1, i, P_2, j)
5:      if ∀p ∈ (P_1 ∪ P_2) . typeof(p) is a pointer then
6:          D_1 ← DEREF(P_1, i)
7:          D_2 ← DEREF(P_2, j)
8:          BIND(D_1, D_2)
9:          BINDPOINTED(D_1, i, D_2, j)
10: function BINDPOINTEDOBJECTS(f called from a location i)
11:     e ← entry location of f
12:     o ← exit (output) location of f
13:     for all x ∈ glob^in, x' ∈ glob_{f_i}^{p_in} s.t. NAME(x) = NAME(x') do
14:         BINDPOINTED({x}, e, {x'}, i − 1)
15:     for all x ∈ glob^out, x' ∈ glob_{f_i}^{p_out} s.t. NAME(x) = NAME(x') do
16:         BINDPOINTED({x}, o, {x'}, i)
17:     for j ← 0..(|param_f| − 1) do
18:         BINDPOINTED({param_f[j]}, e, {arg_{f_i}[j]}, i − 1)
19:         BINDPOINTED({param_f[j]}, o, {arg_{f_i}[j]}, i)
```

at the location j (lines 6-8). Since the dereferenced objects can be of a pointer type again, the function is called recursively on the dereferenced sets.

The function BINDPOINTED is then used from the main function BINDPOINTEDOBJECTS that finally computes the binding of pointed objects. It binds the objects pointed by formal input global variables of a called function f at the function entry location e to the objects pointed by the actual values of the global variables at the call site i in the caller function (lines 13-14). The same is done for objects pointed by formal output global variables at the exit location o of the function at lines (15-16). Then the binding is done for objects pointed by corresponding pairs of formal parameters and actual arguments at the entry and the exit of the function at lines (17-19).

Example We illustrate the algorithm on the function `chainNode` from Figure 3.2. Let this function be called as shown in Figure 3.3.

```
10 struct node *list = NULL;
11 chainNode(&list);
```

(a) The function call in C

$10\ list_{10} = NULL$

$11\ chainNode_{11}((\&list), ())$

(b) The corresponding SSA

Figure 3.3: A function accessing pointed object

Since there are no global variables, only sets $\boldsymbol{arg}_{chainNode_{11}}$ and $\boldsymbol{param}_{chainNode}$ are non-empty:

$$\boldsymbol{arg}_{chainNode_{11}} = \{\&list\}$$
$$\boldsymbol{param}_{chainNode} = \{ppnode\}.$$

The function BindHeapObjects then executes lines 18 and 19 only:

1. The call at line 18 looks as follows:

$$\textsc{BindPointed}(\{ppnode\}, 0, \{\&list\}, 10).$$

2. In the first invocation of the BindPointed function, the sets D_1 and D_2 are:

$$D_1 = \textsc{Deref}(\{ppnode\}, 0) = \{ppnode^{obj}\}$$
$$D_2 = \textsc{Deref}(\{\&list\}, 10) = \{list_{10}\}$$

   and the created binding is:

$$ppnode^{obj} = list_{10}.$$

3. In the next (recursive) invocation, the sets are:

$$D_1 = \textsc{Deref}(\{ppnode^{obj}\}, 0) = \{ppnode^{obj\,obj}\}$$
$$D_2 = \textsc{Deref}(\{list_{10}\}, 10) = \emptyset \rightarrow \text{ since } list_{10} \text{ is } \texttt{null}$$

   and there is no binding created (since D_2 is empty).

4. The call at line 19 looks as follows:

$$\textsc{BindPointed}(\{ppnode\}, 5, \{\&list\}, 11).$$

5. In the first invocation of the BindPointed function, the sets D_1 and D_2 are:

$$D_1 = \textsc{Deref}(\{ppnode\}, 5) = \{ppnode_4^{obj}\}$$
$$D_2 = \textsc{Deref}(\{\&list\}, 11) = \{list_{11}\} \rightarrow \text{ since } list \text{ can be modified by } \texttt{chainNode}$$

   and the created binding is:

$$ppnode_4^{obj} = list_{11}.$$

6. In the next (recursive) invocation, the sets are:

$$D_1 = \textsc{Deref}(\{ppnode_4^{obj}\}, 5) = \{dynamic\_object\$0_3\}$$
$$D_2 = \textsc{Deref}(\{list_{11}\}, 11) = \{list_{11}^{obj}\} \rightarrow \text{ see Algorithm 5}$$

   and the binding created is:

$$dynamic\_object\$0_3 = list_{11}^{obj}.$$

In total, there are three bindings created:

$$ppnode^{obj} = list_{10} \text{ (input binding)},$$
$$ppnode_4^{obj} = list_{11} \text{ (output binding), and}$$
$$dynamic\_object\$0_3 = list_{11}^{obj} \text{ (output binding).}$$

3.7.3 Functions Manipulating the Existing Dynamic Structures

Even though the proposed approach of passing objects pointed by inputs into a function is able to handle pointers to pointers, it might not be sufficient when analysing functions that take a recursive data structure existing on the heap as their input. We propose a way to analyse this kind of functions with concentration on linked lists which are one of the most often used data structures. A generalisation to other data structures is left for future work.

If a function, for example, traverses and manipulates a singly-linked list, it might alter objects which are not directly pointed by the parameters but are reachable from those via the list linkage. The information about the reachability of objects via pointer chains is available during the analysis of the caller function (by computing access paths for the actual function call arguments) but not during the transformation of the callee function into the SSA form, which is done prior to the actual analysis. Because of this, we are not able to bind the corresponding dynamic objects between the caller and the callee functions using Algorithm 7 (since it uses sets of variables determined during the transformation into the SSA form).

To resolve this problem, we make use of the fact that list traversals are done in a similar way in most programs. Hence, we define so-called *list iterators*, which are special objects representing an abstraction of iterating over a linked list using some pointer. In this section, we formalise the concept of iterators, explain how to create them in the SSA, and, finally, show how iterators can be used to correctly analyse functions manipulating linked lists.

List Iterators

A list iterator is an abstraction of a list node (which is a dynamic object) pointed by a certain pointer in a single iteration of a loop that iterates over the linked list. Formally, we define a *list iterator*, denoted it, as a triple:

$$it \in (Ptr \times Obj \times Fld). \tag{3.24}$$

The elements of an iterator are interpreted as follows:

- $p \in Ptr$: the *induction pointer*. It is a pointer that is used to traverse the linked list—in each iteration, p points to the current list node.
- $o \in obj$: the *initial node*. It is a node that is pointed by p before the first iteration.
- $f \in Fld$: the *iterator field*. It is a field through which a step to the next node is done after each iteration.

Example A typical iterative traversal of a linked list is shown in the function in Figure 3.4.

For the given loop, we create a list iterator $list^{it}$:

$$list^{it} = (list, list^{obj}, next). \tag{3.25}$$

The induction pointer is $list$, the initial node is $list^{obj}$ (it abstracts the object initially pointed by $list$), and the iterator field is $next$ (after each iteration, $list$ is moved to the node pointed by the $next$ field).

```
void traverse(struct node *list)
{
    while (list)
    {
        // do something
        list = list->next;
    }
}
```

Figure 3.4: Traversal of a singly-linked list

Creating List Iterators in the SSA Form

The presence of an iterative access to a linked list may be determined by comparing the values of the points-to relation of the induction pointer on the back edge of a loop and at the loop head. To this end, we integrate a list iterator detection into our points-to analysis. Specifically, the list iteration detection is integrated into the abstract transformer for a loop back edge shown in Algorithm 8.

Algorithm 8 Abstract transformer for loop back edge from location j to location i

1: **for all** $p \in Ptr$ **do**
2: **if** $\exists\, x^{obj} \in ValueSet(p_i) \;\wedge\; \exists\, x^{obj}.f^{obj} \in ValueSet(p_j)$ **then**
3: $ValueSet(p_i) \leftarrow (ValueSet(p_i) \setminus \{x^{obj}\}) \cup \{p^{it}\}$ // $p^{it} = (p, x^{obj}, f)$

The transformer searches for a pattern of iterative access—at lines 1-2 it tries to find a pointer p whose points-to value moved by a field f inside the loop body. In case it finds such an access, it replaces the found value by a new iterator (line 3). In the future, the detection can be improved by a more sophisticated method but so far the proposed simple approach seems sufficient on many case studies.

After finding an iterator on the loop back edge, we need to replace all accesses to the first list node that were obtained in the first traversal of the loop body in the points-to analysis by accesses to the list iterator. We do this by redefining the join operator for the *ValueSet* sets as shown in Algorithm 9. We denote $ValueSet(p_i)_1$ to be the old value set, $ValueSet(p_i)_2$ to be the new value set, and $ValueSet(p_i)$ the set resulting from the join.

We also define two helper operations:

- x contains y if and only if the identifier of y is a substring of the identifier of x.
- $x[y/z]$ is a symbol that is obtained from the symbol x by replacing all occurrences of y in the identifier of x by z.

Algorithm 9 Join of $ValueSet(p_i)$

1: $ValueSet(p_i) \leftarrow ValueSet(p_i)_1 \cup ValueSet(p_i)_2$
2: **for all** $v_2 \in ValueSet(p_i)_2$ s.t. v_2 contains $q^{it} = (q, x, f)$ **do**
3: **for all** $v_1 \in ValueSet(p_i)_1$ s.t. $v_1 = v_2[q^{it}/x]$ **do**
4: $ValueSet(p_i) \leftarrow ValueSet(p_i) \setminus \{v_1\}$

The algorithm applies the standard set union and then filters out those objects that were present in the old value set and can be replaced by an iterator from the new value set—these can be determined by replacing the iterator substring in the new object by the identifier of the initial node of the iterator. Application of this algorithm can be seen in the points-to analysis in the following example (join on lines 8 and 13).

Since a loop might access multiple list nodes in a single iteration and we want to represent these as precisely as possible in the SSA, we extend the initialisation of the abstract value by a loop shown in Algorithm 10 that initialises value sets of pointer-typed fields of data structures. For example, the initial *ValueSet* for a pointer $list^{obj}.next$ (which represents the *next* field of the object initially pointed by the parameter *list*) will contain the object $list^{obj}.next^{obj}$.

Algorithm 10 Initialisation of the abstract value for pointer-typed fields of data structures

1: **for all** $p^{obj}.f$ s.t. $p \in Ptr \wedge \texttt{typeof}(p^{obj}).f$ is *pointer* **do**
2: INIT($p^{obj}.f$)

This algorithm is run along with initialisation by Algorithm 4. In case of recursive data structures, the initialisation might not terminate, which we solve in practice by initialising only those values that are actually used within the SSA form of the analysed function. We also run Algorithm 10 for iterators that are found during the analysis.

Example To illustrate the detection of list iterators, we show an example of conversion of a program into the SSA form for a function transforming a singly-linked list into a doubly-linked list given in Figure 3.5.

```
1  void backLink(struct node *list) {
2
3
4    while (list) {
5
6
7
8      struct node *next = list->next;
9
10
11     next->prev = list;
12
13     list = next;
14   }
15 }
```

(a) The function in C

```
1
2  g_2 = TRUE
3
4  list_4^phi = (g_14^ls ? list_14^lb : list)
5
6  g_6 =!(list_4^phi = NULL)
7
8  next_8 = (list_4^phi = &list^it ? list^it.next
9                                  : unknown_object.next)
10
11 list^it.next^obj.prev_11 = list_4^phi
12
13 list_13 = next_8
14
15
```

(b) The corresponding SSA

Figure 3.5: SSA using list iterators

First, the points-to analysis is run on the function source. Since there is one pointer input, namely, *list*, we create an initial value for it. We also create initial values for fields

of the dynamic objects that will be needed during the computation:

$$\begin{aligned}
&ValueSet(list) = \{list^{obj}\} \wedge unknown(list) = true\\
&ValueSet(list^{obj}.next) = \{list^{obj}.next^{obj}\}\\
&ValueSet(list^{obj}.next^{obj}.prev) = \{list^{obj}.next^{obj}.prev^{obj}\}.
\end{aligned}$$

The computation of the *ValueSet* sets for the particular program lines goes as follows:

$$\begin{aligned}
8: \quad ValueSet(next_8) &= \bigcup_{x \in ValueSet(list_7)} ValueSet(x.next_7)\\
&= ValueSet(list^{obj}.next_7)\\
&= ValueSet(list^{obj}.next)\\
&= \{list^{obj}.next^{obj}\}
\end{aligned}$$

$$\begin{aligned}
11: \quad ValueSet(list^{obj}.next^{obj}.prev_{11}) &= ValueSet(list^{obj}.next^{obj}.prev_{10}) \cup ValueSet(list_{10})\\
&= ValueSet(list^{obj}.next^{obj}.prev) \cup ValueSet(list)\\
&= \{list^{obj}.next^{obj}.prev^{obj}, list^{obj}\}
\end{aligned}$$

$$\begin{aligned}
13: \quad ValueSet(list_{13}) &= ValueSet(next_{12})\\
&= ValueSet(next_8)\\
&= \{list^{obj}.next^{obj}\}
\end{aligned}$$

14 : loop-back edge to 4

$ValueSet(list_4) = ValueSet(list) = \{list^{obj}\}$

$ValueSet(list_{14}) = \{list^{obj}.next^{obj}\}$

$\Rightarrow ValueSet(list_4) = \{list^{it}\}$ where $list^{it} = (list, list^{obj}, next)$

We have found an iterator.

The initial values of its components are the following:

$ValueSet(list^{it}.next) = \{list^{it}.next^{obj}\}$

$ValueSet(list^{it}.next^{obj}.prev) = \{list^{it}.next^{obj}.prev^{obj}\}$

$$\begin{aligned}
8: \quad ValueSet(next_8)' &= \bigcup_{x \in ValueSet(list_7)} ValueSet(x.next_7)\\
&= ValueSet(list^{it}.next_7)\\
&= ValueSet(list^{it}.next)\\
&= \{list^{it}.next^{obj}\}
\end{aligned}$$

8 : join

$$\begin{aligned}
ValueSet(next_8) &= ValueSet(next_8) \circ ValueSet(next_8)'\\
&= \{list^{obj}.next^{obj}\} \circ \{list^{it}.next^{obj}\}\\
&= \{list^{it}.next^{obj}\}
\end{aligned}$$

The iterator replaces the access to the object pointed by the *next* field of the first list node previously detected by the analysis at line 8.

$$\begin{aligned} 11: \quad ValueSet(list^{it}.next^{obj}.prev_{11}) &= ValueSet(list^{it}.next^{obj}.prev_{10}) \cup ValueSet(list_{10}) \\ &= ValueSet(list^{it}.next^{obj}.prev) \cup ValueSet(list_4) \\ &= \{list^{it}.next^{obj}.prev^{obj}, list^{it}\} \end{aligned}$$

$$\begin{aligned} 13: \quad ValueSet(list_{13})' &= ValueSet(next_{12}) \\ &= ValueSet(next_8) \\ &= \{list^{it}.next^{obj}\} \end{aligned}$$

13 : join

$$\begin{aligned} ValueSet(list_{13}) &= ValueSet(list_{13}) \circ ValueSet(list_{13})' \\ &= \{list^{obj}.next^{obj}\} \circ \{list^{it}.next^{obj}\} \\ &= \{list^{it}.next^{obj}\} \end{aligned}$$

The iterator replaces the access to the object pointed by the *next* field of the first list node previously detected by the analysis at line 13.

14 : loop-back edge to 4

$ValueSet(list_4) = \{list^{it}\}$

$ValueSet(list_{14}) = \{list^{it}.next^{obj}\}$

$\Rightarrow ValueSet(list_4) = \{list^{it}\}$

This iterator has already been found.

After the points-to analysis, we run the transformation of the C code into the SSA form. This is done by the following steps:

1. Line 2: The function entry is always reachable, thus the first guard is set to *true*.
2. Line 4: A Φ node is created for the loop head. It joins the value of *list* from before the loop (the parameter *list*) and the value of *list* from the end of the loop body (loop-back variable $list^{lb}_{14}$).
3. Line 6: The loop body is reachable only if the loop condition is satisfied—this is expressed by the guard g_6.
4. Lines 8-9: We query the points-to analysis to correctly dereference *list*:

$$\begin{aligned} ValueSet(list_8) &= ValueSet(list_4) = \{list^{it}\} \\ unknown(list_8) &= true. \end{aligned}$$

Based on the information obtained from the points-to analysis, we create a case split according to Equation 3.10. The expression `list->next` can have values $list^{it}.next$ (line 8 in the SSA form) or $unknown\_object.next$ (line 9 in the SSA form).

5. Line 11: To correctly dereference $next$, we query the points-to analysis:

$$ValueSet(next_{11}) = \{list^{it}.next^{obj}\}$$

   and thus we create the equation as shown at line 11.

6. Line 13: This is a simple assignment without dereferences, which we translate into the SSA form by adding the appropriate variable indices.

The example contains a list iterator $list^{it}$ obtained during the points-to analysis. Since the SSA form uses multiple versions of the same variable, we use the appropriate versions in the iterator, too:

$$list^{it} = (list_4^{phi}, list^{obj}, next).$$

The loop in the function traverses a singly-linked list passed to the function at the input. In each loop iteration, the pointer $list$ points to the current node of the list. In the SSA form, the detected iterator represents an abstraction of a node pointed by $list$, specifically by its version $list_4^{phi}$. The function reads the $next$ field of the iterator (line 8) which abstracts an access to the node that is the successor of the current node, and then writes to the $prev$ field of this node (line 11). We denote these reads and writes as *iterator accesses*.

Iterator Accesses

Formally, an iterator access is associated with an iterator and contains a sequence of fields and a location. Let $\mathcal{I}$ be a domain of all list iterators. We define an iterator access a as:

$$a \in \mathcal{I} \times Fld^n \times Instr \tag{3.26}$$

where Fld^n denotes a n-fold Cartesian product of structure fields with arbitrary n.

An iterator access is interpreted as an access to a node that can be reached from the current node (corresponding to the iterator) by following the given sequence of fields. Since each iterator access is interpreted as a standard variable in the SSA form, the location specifies the SSA suffix of such variable. We showed that the example from Figure 3.5b contains one iterator $list^{it}$. We can detect two iterator accesses:

$$\begin{aligned} a_1 &= (list^{it}, (next), 0) && \text{line 8} \\ a_2 &= (list^{it}, (next, prev), 11) && \text{line 11.} \end{aligned}$$

The location 0 is interpreted as the input version of the variable.

Binding List Iterators

After all list iterators and their accesses in a function are found, we are able to compute the summary of a function for a given calling context, i.e. determine how the function alters the linked list given at its input. In order to do this, we bind the list iterator accesses with the actual heap objects from the function calling context.

Let $CallCtx_{h_l}$ be the calling context of a function h called at location l. It has the form of a heap template with computed values of row parameters for variables of the caller function at the entry of the call site h_l. Next, let $\mathcal{A}$ be a set of all list iterator accesses in h. Using this information, we perform the binding of list iterators of h with actual heap objects from the calling context using the function BindAllIterators shown in Algorithm 11.

Algorithm 11 Binding list iterators to input heap objects

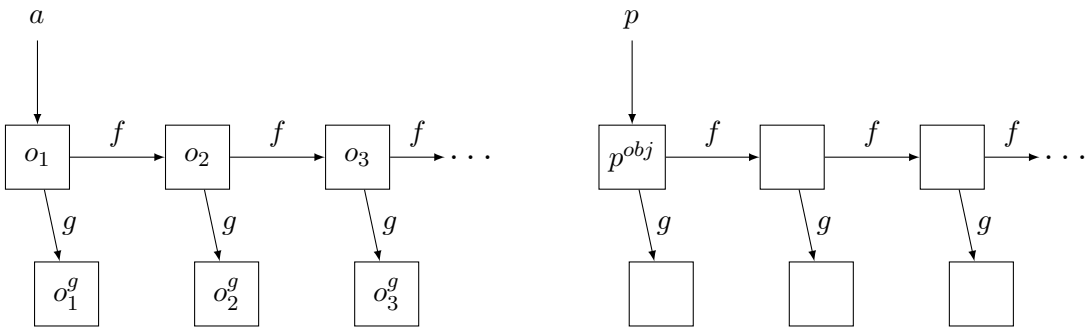
```
1:  function ReachableObjs(o ∈ Obj, f ∈ Fld)
2:      R ← ∅
3:      O ← set of all objects from CallCtx_{h_l} that correspond to o
4:
5:      for all path(ō, f, d)[Ō] ∈ CallCtx_{h_l} such that ō ∈ O do
6:          R ← R ∪ Ō ∪ {d} // add objects on the path and the path destination
7:      return R
8:  function BindIterators(p ∈ Ptr, o, it ∈ Obj, f ∈ Fld, F ∈ Fld^n, loc ∈ Instr, i ∈ ℕ)
9:      result ← true
10:     R ← ReachableObjs(o, f)
11:     for all r ∈ R do
12:         cond ← p = &r
13:         bind ← it = r ∧ it.F[i] = r.F[i]
14:         if i < (|F| − 1) then
15:             bind ← bind ∧ BindIterators(r.F[i], r, it.F[i]^obj, F[i], F, loc, i + 1)
16:         expr ← cond ⇒ bind
17:         result ← result ∧ expr
18:         if i = (|F| − 1) ∧ loc ≠ 0 then
19:             add T_j^O((r, F[i]), d_j^O) to template
20:     return result
21: function BindAllIterators
22:     for all (p^it, F, loc) ∈ A where p^it = (p, o, f) do
23:         solver ◁ BindIterators(p, o, p^it, f, F, loc, 0)
```

First, we define the function ReachableObjs that takes an (abstract) object o and a field f and returns the set of all objects reachable from o via f in the calling context. Since o might be an abstraction, we first get all objects that correspond to o in the calling context (line 3). After that, all objects belonging to paths leading from objects corresponding to o via the field f are collected (lines 5-6) and returned.

The actual binding is created by the BindIterators function. The function returns bindings of a single iterator access with corresponding objects and their fields. It is called from the function BindAllIterators for each iterator access (p^{it}, F, loc) where $p^{it} = (p, o, f)$. It has 7 parameters, 5 of them corresponding to elements of the iterator and the iterator access. The parameter it represents the iterator symbol used in the SSA form and the parameter i is the current level of recursion and is used as the index into the field vector F from the iterator access.

In order to explain how the function works, we illustrate used symbols on simple figures. Let the calling context contain a list linked through a field f, where each node points to a data node by the field g, and let this list be pointed by a pointer a that will be passed to the called function as an argument. This list is illustrated by Figure 3.6a and the corresponding abstraction of the list at the callee site is shown by Figure 3.6b. Here, the list is pointed by the formal parameter p and the first node is abstracted by the object p^{obj}.

Next, let the analysed function contain a loop that traverses the given list, node by node, using p as an induction pointer. We detect a list iterator $p^{it} = (p, p^{obj}, f)$ that, for

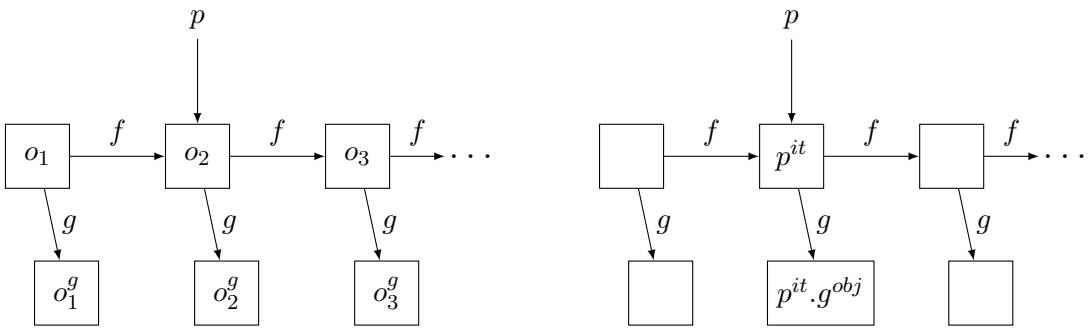


(a) Caller site (calling context).

(b) Callee site.

Figure 3.6: Linked list at the function entry at the caller site and at the callee site.

example, in a second loop iteration, is interpreted as shown in Figure 3.7b. A projection of this situation on the actual list from the calling context is shown in Figure 3.7a.



(a) Actual list from the calling context.

(b) Abstraction of the list using an iterator.

Figure 3.7: Second iteration of a loop traversing a linked list using p as an induction pointer.

When the function BindIterators is called for the iterator $p^{it} = (p, p^{obj}, f)$ and its access $(p^{it}, g, 0)$, it performs the following steps:

1. A set of objects reachable from the initial iterator object p^{obj} (which corresponds to o_1 in the calling context) via the iterator field f is computed using the function ReachableObjs (line 10).

2. Next, a binding for each reachable object r is created. In the situation shown in Figure 3.7, r corresponds to o_2. A binding is composed of a precondition and a binding expression:

   2.1 A precondition is an equality between the iterator induction pointer p and the address of o_2 (line 12).

   2.2 The binding expression is a conjunction of multiple equalities:

   - An equality between the iterator symbol p^{it} and the object o_2. (line 13)
   - An equality between the field of the iterator access $p^{it}.g$ and the corresponding field $o_2.g$ (line 13).

- In case the iterator access is composed of multiple fields (the index i of the current access field is not the index of the last field), the function BindIterators is called recursively. In the presented situation, this could happen if the object pointed by g would point to another object by some field. In this case, the recursive call would have the form:

  $$\text{BindIterators}(o_2.g, o_2, p^{it}.g^{obj}, g, F, loc, 1)$$

  and it would create a binding for objects starting from o_2 following the field g. The returned binding is added to the current one (line 15).

3. A new template row is inserted for each field of the actual object o_2 that corresponds to a write iterator access (this happens in the last recursive call of the function for an iterator access whose location is not 0).

Example We illustrate the algorithm on a concrete example. We use the function `backLink` from Figure 3.5. Let this function be called as `backLink(list)` with the following calling context:

$$\begin{aligned}
&\mathcal{T}^O(list, \{o\})\\
&path(o, next, o')[o]\\
&path(o, next, \texttt{null})[o, o']\\
&path(o', next, \texttt{null})[o']\\
&path(o, prev, \texttt{null})[]\\
&path(o', prev, \texttt{null})[].
\end{aligned}$$

The state of the heap described by this calling context is visualised in Figure 3.8. Dashed arrows denote that o and o' are abstractions of list segments (of arbitrary lengths) linked through *next* field. All *prev* fields are set to `null`.

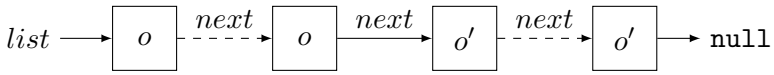


Figure 3.8: Visualisation of the calling context in the example

The function `backLink` contains one iterator $list^{it}$ and two iterator accesses a_1 and a_2:

$$\begin{aligned}
list^{it} &= (list_4^{phi}, list^{obj}, next)\\
a_1 &= (list^{it}, (next), 0)\\
a_2 &= (list^{it}, (next, prev), 8).
\end{aligned}$$

The function BindIterators is then executed for each iterator access. We show its execution for a_2:

1. Line 23: $\text{BindIterators}(list_4^{phi}, list^{obj}, list^{it}, next, (next, prev), 8, 0)$ is called.
2. Line 10: $\text{ReachableObjs}(list^{obj}, next)$ is called, which returns the set $R = \{o, o'\}$.
3. Line 11: The first iteration will create a binding for o.

4. Line 12: $cond = (list_4^{phi} = \&o)$.
5. Line 13: $bind = (list^{it} = o \wedge list^{it}.next = o.next)$.
6. Line 14: $i = 0 < (|F| - 1) = 1$, hence we recursively call $\text{ReachableObjs}(o.next, o, list^{it}.next^{obj}, next, (next, prev), 8, 1)$.
   - 6.1 Line 10: $\text{ReachableObjs}(o, next)$ is called, which returns the set $R = \{o, o'\}$.
   - 6.2 Line 11: The first iteration will create a binding for o.
   - 6.3 Line 12: $cond = (o.next = \&o)$.
   - 6.4 Line 13: $bind = (list^{it}.next^{obj} = o \wedge list^{it}.next^{obj}.prev_8 = o.prev_8)$.
   - 6.5 Line 14: $i = 1 \geq (|F| - 1) = 1$, hence line 15 is skipped.
   - 6.6 Line 16: $expr = (o.next = \&o \implies list^{it}.next^{obj} = o \wedge list^{it}.next^{obj}.prev_8 = o.prev_8)$.
   - 6.7 Line 18: Condition is true, hence we add a new template row: $\mathcal{T}_0^O((o, prev), d_0^O)$.
   - 6.8 Lines 14-22: We repeat the loop for the object o'.
   - 6.9 Line 23: The resulting binding is:

$$(o.next = \&o \Rightarrow list^{it}.next^{obj} = o \wedge list^{it}.next^{obj}.prev_8 = o.prev_8) \wedge$$
$$(o.next = \&o' \Rightarrow list^{it}.next^{obj} = o' \wedge list^{it}.next^{obj}.prev_8 = o'.prev_8).$$

7. Line 19: The iterator access binding for the object o is:

$$list_4^{phi} = \&o \Rightarrow (list^{it} = o \wedge list^{it}.next = o.next \wedge$$
$$(o.next = \&o \Rightarrow list^{it}.next^{obj} = o \wedge list^{it}.next^{obj}.prev_8 = o.prev_8) \wedge$$
$$(o.next = \&o' \Rightarrow list^{it}.next^{obj} = o' \wedge list^{it}.next^{obj}.prev_8 = o'.prev_8)).$$

8. Line 21: The condition is false (correct template rows have been created in the recursive call).
9. Lines 14-22: We repeat the loop for the object o'.
10. Line 23: The whole binding for the iterator access a_2 is:

$$list_4^{phi} = \&o \Rightarrow (list^{it} = o \wedge list^{it}.next = o.next \wedge$$
$$(o.next = \&o \Rightarrow list^{it}.next^{obj} = o \wedge list^{it}.next^{obj}.prev_8 = o.prev_8) \wedge$$
$$(o.next = \&o' \Rightarrow list^{it}.next^{obj} = o' \wedge list^{it}.next^{obj}.prev_8 = o'.prev_8))$$
$$\wedge$$
$$list_4^{phi} = \&o' \Rightarrow (list^{it} = o' \wedge list^{it}.next = o'.next \wedge$$
$$(o'.next = \&o' \Rightarrow list^{it}.next^{obj} = o' \wedge list^{it}.next^{obj}.prev_8 = o'.prev_8)).$$

Moreover, two new template rows are created:

$$\mathcal{T}_0^O((o, prev), d_0^O)$$
$$\mathcal{T}_1^O((o', prev), d_1^O).$$

Similarly, bindings for the iterator access a_1 are created. In this case, the location of a_1 is 0, and so no new template rows are generated.

Using the given bindings and the new template rows, the summary for the function `backLink` will contain access paths for o and o' corresponding to the created back link via the *prev* field in the linked list.

Last, we have to ensure that the computed summary can be used in the caller function to constrain the function call. This is normally done by binding corresponding objects between the caller and the callee functions using Algorithm 1 and Algorithm 7. These, however, cannot be used for functions that contain iterators. The reason is that Algorithm 7 binds objects pointed by corresponding pointers using the results of the points-to analysis. For a function manipulating existing linked lists, a *ValueSet* of a formal output contains an iterator object but the summary contains the actual objects from the calling context. We resolve this by introducing Algorithm 12 that binds the heap objects that were changed within the called function (hence they were added to the shape domain template by Algorithm 11) to new versions of the same objects in the caller function site. This algorithm is run along with other binding algorithms which handle those objects that are not represented by iterators.

Algorithm 12 Binding objects represented by iterators between f and a call site f_i

1: $e \leftarrow$ exit location of f
2: **for all** $\mathcal{T}_k^O((o, f), d_k^O)$ added by Algorithm 11 **do**
3: $\quad solver \triangleleft \text{SSA}(o.f, i) = \text{SSA}(o.f, e)$

In the above example, two new template rows for o and o' were created. If the function `backLink` was called from location 20, then running Algorithm 12 would create the following bindings:

$$o.prev_{20} = o.prev_8$$
$$o'.prev_{20} = o'.prev_8$$

Chapter 4

Implementation

We have implemented the solution proposed in Chapter 3 into the 2LS tool. In Section 4.1, we first describe the architecture of 2LS and the sequence of the main steps that 2LS performs during program analysis. Section 4.1 also contains short description of all analyses currently available in 2LS. In Section 4.2, we show how we integrated shape analysis into this architecture, giving the most important implementation details. In the end, we outline how our shape analysis can be used simultaneously with other analyses by implementing a simple combination of abstract domains.

4.1 The Architecture of 2LS

2LS is built over CPROVER infrastructure and uses multiple components of this framework. The overall architecture of 2LS can be divided into three main parts: front-end, middle-end, and back-end. For many operations in the front-end and in the back-end, the mechanisms from the CPROVER framework or other external tools are used. The main steps performed by 2LS are outlined in Figure 4.1 [17].

We now describe the particular parts and steps in more detail.

4.1.1 Front-End

The *command-line front-end* first configures 2LS according to user-supplied parameters. There are many options that can be set, the complete list is available via the `--help` switch. After that, the source program is parsed and translated into a GOTO program. This is ensured by *GOTO program parser* from the CPROVER framework, which uses an external C preprocessor. A GOTO program is an internal program representation having the form of a control flow graph. In the end, 2LS performs various transformation of the GOTO program, such as function inlining or constants propagation. From our point of view, the most important transformation is splitting chains of dereferences occurring in one command into multiple commands. This ensures that each statement contains one dereference only and hence the points-to analysis and representation of heap operations in the SSA form does not have to handle situations when multiple dereferences occur in one command.

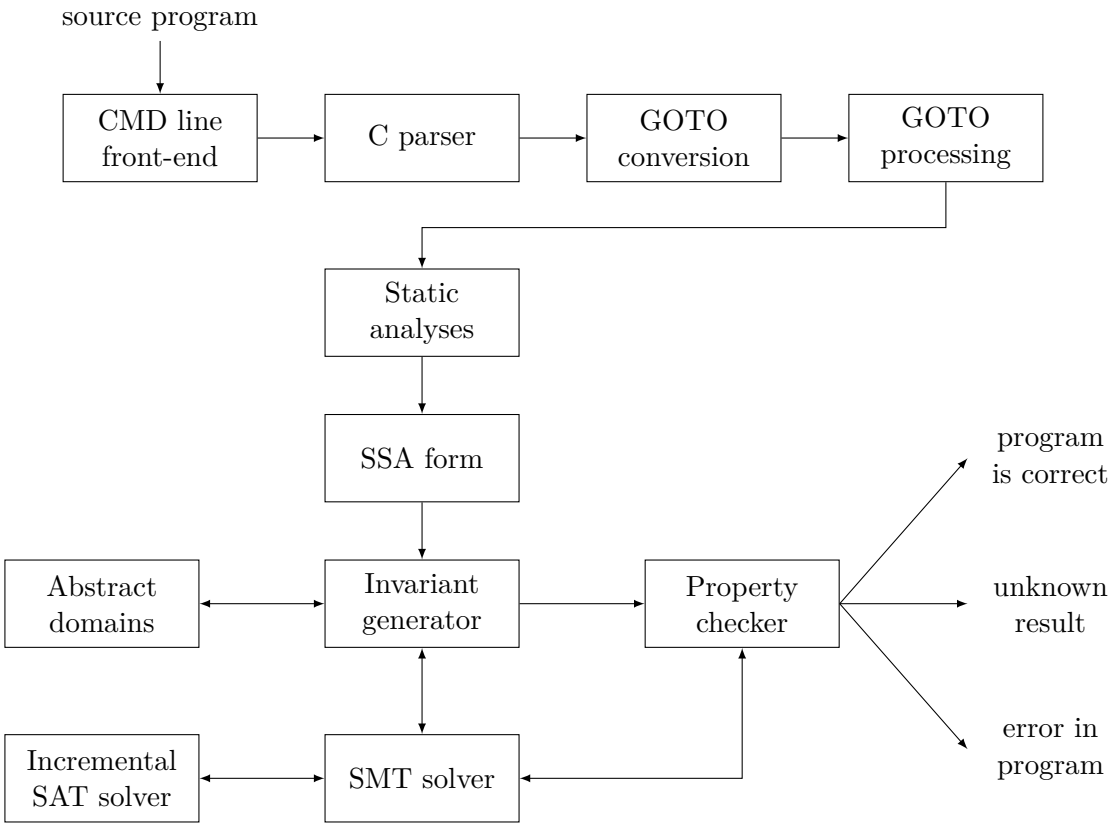


Figure 4.1: The architecture of 2LS

4.1.2 Middle-End

The middle-end is the part of 2LS where most of the program analysis is done. For that reason, we describe the steps included into this phase in more detail.

Static Analyses and Conversion into the SSA Form

First, several static analyses are performed on the GOTO program in order to obtain information that will be needed for the subsequent conversion into the SSA form. These include *objects analysis* that collects all objects accessed in a function, and *assignments analysis* that, for each object, determines program locations where the object is assigned. This analysis is crucial for the SSA generation since the SSA versions of variables will be created based on computed locations.

Another important analysis performed in this phase is the *points-to* analysis that we extended and described in Section 3.3.

After performing all needed static analyses, the GOTO program is converted into the SSA form. The approach of the conversion was described already in Section 2.4.3.

Invariant Generator

The SSA form is an over-approximation of the GOTO program (due to the way loops are cut with the values returned over the back-edges made random and function calls havocked).

2LS refines this over-approximation by inferring loop invariants and function summaries. These can be computed in various abstract domains that are chosen by command line options. Currently, 2LS supports the following domains:

- *Polyhedra domains*: domains for analysis of numerical variables—signed and unsigned bit-vector integers and floats. These include:
  - *Interval domain*: The value of each numerical variable is constrained by an interval. Thus, for each variable x, constraints in the form $\pm x \leq C$, where C is a constant, are discovered.
  - *Zones domain*: Uses constraints of the form $x - y \leq C$ for each pair of program variables x and y.
  - *Octagon domain*: Uses invariants of the form $x + y \leq C$ for each pair of program variables x and y.
- *Equalities domain*: A domain for analysis of equalities and disequalities between pairs of variables.
- *Lexicographic ranking domain*: A domain used for termination analysis.

Property Checker

After all invariants are computed, the property checker checks validity of all user-supplied assertions in the source program. This is done by solving the formula representing the source program obtained from the SSA form along with the computed invariants in the SMT solver and checking the satisfiability of negations of program assertions. In case the negations of all assertions are unsatisfiable, the program is valid, on the other hand if there is an assertion whose negation is satisfiable, the program contains an error. If an error is found, it may be spurious due to over-approximation (when the inferred invariants are too weak) and thus the analysis result might be indecisive.

4.1.3 Back-End

Both invariant inference and property checking are done using an SMT solver. Since 2LS requires an incremental solver and its support is much better in SAT solvers, it uses an external SAT solver Glucose 4.0 or MiniSAT 2.2.0. The needed support for the SMT theories of bit-vectors and arrays are provided by the CPROVER infrastructure.

4.2 Shape Analysis Integration

We have implemented our solution of shape analysis into the above described architecture of 2LS. All algorithms from Chapter 3 are implemented in a straightforward manner. In this section, we state the parts of 2LS where the particular algorithms are implemented and present some implementation details.

4.2.1 Points-to Analysis and Heap Operations in the SSA Form

The points-to analysis we use is implemented as one of the static analyses that are performed prior to the conversion into the SSA form. It uses the classical abstract interpretation approach. In order to implement it, we make use of the C++ template for abstract interpretation provided by the CPROVER infrastructure.

The replacement of the calls of the `malloc` function is done in the phase of processing the GOTO program. For each call, a new object is created and inserted into the symbol table, and the address of this object is used in place of the call.

The representation of heap operations is included into the transformation of the GOTO program into the SSA form.

4.2.2 Shape Domain

We created a new abstract domain for shape analysis. It is composed of two main parts: the *domain* itself, and the *strategy solver*. The domain defines the form of the template and the abstract value, while the strategy solver implements the algorithm for join of the current abstract value with the model of satisfiability of the formula returned by the solver. The inference of invariants is ensured by the invariant generator of 2LS. The usage of our shape domain can be activated using the `--heap` switch.

The domain also contains an implementation of the initial binding of list iterators (Algorithm 11) that is performed at the beginning of analysis of each function.

The binding between the caller and the callee functions is ensured by a component called *SSA inliner*, which we extended by our binding algorithms (Algorithm 7 and Algorithm 12).

4.3 Combination of Abstract Domains

One of the main directions of this work is to provide a possibility to combine multiple abstract domains together, which could bring the possibility to analyse complex properties of programs, such as those that depend both on pointer and non-pointer variables.

As a proof of concept, we have implemented a simple domain combination that combines our shape domain with the interval domain already present in 2LS. We make use of the fact that interval domain uses template rows for numerical variables, similarly to our shape domain using template rows for pointer variables. The implementation is based on creating a new abstract domain, whose template is composed of two parts: one part being for numerical variables and the other for pointer variables. For each row, the abstract value is computed in the corresponding domain using the corresponding join algorithm.

If we split structure-typed variables of the original program into particular fields, we can determine the shape of dynamic data structures from invariants for pointer-typed fields, and the content of nodes of these structures from invariants for numerical fields. This way, we may be able to prove properties that could not be proven when each analysis was done separately.

Chapter 5

Results and Experiments

We have proposed and implemented a shape analysis designed particularly for the 2LS framework. In order to prove that our extension brought an improvement of the capabilities of 2LS, we performed a series of experiments. These were run on a benchmark from the International Competition on Software Verification 2017 (SV-COMP 2017) and on a benchmark from the Predator shape analyser. The execution and results of these experiments are described in Sections 5.1 and 5.2.

Since our solution involves major updates to parts of 2LS that are run for each kind of analysis (modifications of the SSA form generation, points-to analysis, etc.), it is essential to prove that our changes did not affect other analyses. To this end, we use the regression tests that are present in 2LS. The obtained results are described in Section 5.3.

5.1 Benchmark from SV-COMP 2017

One of the most relevant collections of benchmarks in the community of software verification and analysis is the collection of benchmarks from the International Competition on Software Verification (SV-COMP). The goal of this competition is to provide a possibility to compare different verification tools in terms of their precision and performance. This is done by establishing a set of benchmarks that are composed of a large number of verification tasks. Each task consists of a C program and a property (reachability, memory safety, termination) to be verified.

The tasks are divided into several categories and their subcategories, based on the verified properties. Since we aim to analyse properties related to the shape of the heap, the most important category is the Heap Reachability category. The tasks of this category aim to verify user-supplied assertions that check reachability of objects in dynamic data structures such as lists, trees, etc.

The result of running a tool on the benchmark of an SV-COMP category is a score expressing the performance of the tool on the tasks included into the given benchmark. For the Heap Reachability category, the expected result of a verification task is either *true*, which expresses that the program is error-free, or *false*, which expresses that the program contains an error meaning that some broken assertion is reachable. The scoring system is the following:

- +2 points for each program that is correctly proven to be error-free (*correct true*).
- +1 point for each program where an existing error is found (*correct false*).

- -16 points for each correct program in which an error is reported (*incorrect false*, a so-called false positive).
- -32 points for each program where an error is present, but was not discovered (*incorrect true*, a so-called false negative).
- 0 points for an inconclusive result, which also includes a tool crash, or out of resources error.

We executed 2LS on the Heap Category benchmark from SV-COMP 2017 without and with our extension. The obtained results are shown in Table 5.1. The experiments were run on an Intel Xeon 5000 processor at 3.5 GHz running Ubuntu 16.04. Each run was limited to 15 GB of memory and 60 s of CPU time.

Table 5.1: A comparison of 2LS without and with our extension on the SV-COMP'17 Heap Reachability category

	2LS	
	Without extension	**With extension**
Number of tasks	173	173
Correct results	77	82
Correct true	54	62
Correct false	23	20
Incorrect results	17	4
Incorrect true	6	3
Incorrect false	11	1
Inconclusive	79	87
Score	**-237**	**32**
CPU time per finished task (s)	0.31	0.37

We can see that our analysis increased the number of correctly analysed tasks and decreased the number of incorrect results, which led to a significant increase of the score. However, the results bring some interesting observations.

Even though the total number of correct results increased, the number of errors correctly found decreased. A likely explanation is that the previous "correct false" results were just coincidences, which is justified by a large number of "incorrect false" results. Since 2LS had but minimal support for shape analysis, manipulation of the heap often caused errors.

Generally, we can observe that the most significant improvement was in proving correct programs and avoiding false positives. This can be explained by the fact that our shape analysis uses the abstract interpretation approach of 2LS, which over-approximates the program and thus is sound in proving program correctness. Moreover, when an error is found, it is not guaranteed to be reachable in the real program (due to over-approximation) and thus 2LS often ends with an "unknown" result in case the error might be spurious. This possibility is supported by the increase of inconclusive results obtained when using our extension of 2LS.

The next observation is that there is a significant number of tasks that are successfully verified even without our extension. This is mainly caused by the fact that these tasks do not contain any loops, thus no invariant is to be computed, and the SSA form with the SMT solver is enough to prove the program correctness or to find an error.

Last, the table also shows the average CPU time spent to verify a task. This calculation includes only those tasks whose analysis finished without error (hence we remove tasks that ended by the tool crash or out of resources error). We can observe that our shape analysis increased the verification time by few percent only and hence preserved a high performance of 2LS.

5.2 Experiments from the Predator Tool

Apart from SV-COMP benchmarks, other sets of relevant examples can be found in the distributions of existing tools for shape analysis. Currently, one of the best tools in this area is the Predator tool [9]. It has won several gold medals in the Heap Manipulation category in previous editions of SV-COMP (this category was replaced by the Heap Reachability category this year).

We extracted the regression tests from this tool that work with singly (SLL) and doubly linked lists (DLL). We added program assertions into these programs so that they are usable for analysis with 2LS. In the tests, we are not interested in lists destruction, since our extension does not support checks for memory leaks, yet. Similarly to the previous experiment, we ran 2LS on the benchmark without and with our shape analysis, and compared the numbers of successfully verified examples. The results are shown in Table 5.2.

Table 5.2: A comparison between the number of successfully verified tasks from the Predator benchmark with and without our extension of 2LS

	Tasks	**Correct results**	
		before our extension	**after our extension**
SLL	17	6	**14**
DLL	8	2	**7**

We can observe that our analysis notably increased the number of successfully verified programs in both categories. We can see that there is a number of tasks that 2LS handles without our extension, which is caused by the fact that these programs either do not contain any loops, or check for properties that can be proven without an invariant for the shape of the heap.

5.3 2LS Regression Tests

2LS contains a large set of regression tests checking various properties of programs. A majority of the tests is aimed towards existing analyses—the analysis of numerical variables and the termination analysis. We re-ran these tests after the integration of our solution and compared the results with the previous ones. This way, we show that our changes did not corrupt the current analyses of 2LS.

The tests are divided into 5 main categories, each containing a number of verification tasks. Every task contains a C program to be verified and a test specification that defines

the expected verification result. A short description of each category can be found in Appendix C. Since 2LS is still in development, there are tasks that 2LS is currently not able to verify correctly. The numbers of successfully verified tasks before and after the integration of our solution are shown in Table 5.3.

Table 5.3: A comparison between the number of successfully verified 2LS regression tests with and without our extension

Category	**Tasks**	**Correct results**	
		before our extension	**after our extension**
Termination	125	89	**90**
kIkI	36	31	**31**
Preconditions	8	8	**8**
Interprocedural	46	31	**31**
Invariants	86	64	**64**

The results show that our changes did not negatively affect the existing analyses in 2LS. On the contrary, there is one additional successful test in the Termination category. Even though the test does not use the shape domain, our changes to passing pointers between functions helped to perform correct verification of the benchmark.

Chapter 6

Conclusion

In this work, we proposed a way of integrating shape analysis into the 2LS framework. This included creating an abstract domain capable of describing the shape of dynamic data structures in the heap. To this end, we use the concept of pointer access paths that describe the shape of the heap by expressing a reachability of heap objects from pointer-typed variables in the analysed program. Moreover, we introduced changes to other parts of 2LS needed to successfully perform the shape analysis. Specifically, we improved the generation of the SSA form, extended the points-to analysis, and proposed methods needed to perform interprocedural analysis of functions working with pointers and recursive data structures.

The proposed mechanisms show how pointer operations and the shape of the heap can be described using quantifier-free formulae in the first-order logic. Solving these in an SMT solver working with the theory of bit-vectors allows one to automatically prove properties of a C program regarding dynamic data structures, especially linked lists.

We have implemented the proposed concepts into the 2LS framework and performed a series of experiments to demonstrate usefulness of our extension. The experiments were run on benchmarks from the Heap Reachability category of SV-COMP 2017 and from the Predator tool. The results show that our shape analysis in 2LS brought a significant improvement of the capabilities of 2LS to analyse programs working with pointers and dynamic data structures. We also showed that the implementation did not negatively affect other analyses that were already present in 2LS.

Current analyses in 2LS include a good-quality analysis of values of numerical variables. Its combination with the proposed shape analysis could bring the possibility of analysing interesting properties of the heap, such as those that depend on lengths of the lists, or offsets of the addresses. In our implementation, we showed how a simple combination of domains can be done in 2LS. In the future, extending this concept could allow one to efficiently analyse properties that other tools cannot cope with and thus to handle more complex programs.

Bibliography

[1] CPROVER. http://www.cprover.org/.

[2] Alpern, B.; Wegman, M. N.; Zadeck, F. K.: Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 1988. pp. 1–11.

[3] Brain, M.; David, C.; Kroening, D.; et al.: Model and Proof Generation for Heap-Manipulating Programs. In *Proceedings of the 23rd European Symposium on Programming*. Springer. 2014. pp. 432–452.

[4] 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.

[5] Chen, H.; David, C.; Kroening, D.; et al.: Synthesising Interprocedural Bit-Precise Termination Proofs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2015. pp. 53–64.

[6] Chong, S.; ; Rugina, R.: Static Analysis of Accessed Regions in Recursive Data Structures. In *Proceedings of the 10th International Static Analysis Symposium*. Springer. 2003. pp. 463–482.

[7] Cousot, P.; 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.

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

[9] Dudka, K.; Peringer, P.; Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In *Proceedings of the 20th International Static Analysis Symposium*. Springer. 2013. pp. 215–237.

[10] Habermehl, P.; Holík, L.; Rogalewicz, A.; et al.: Forest Automata for Verification of Heap Manipulation. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer. 2011. pp. 424–440.

[11] Hooker, J. N.: Solving the incremental satisfiability problem. *JLP*. vol. 15, no. 1&2. 1993: pp. 177–186.

[12] Matosevic, I.; Abdelrahman, T. S.: Efficient Bottom-up Heap Analysis for Symbolic Path-based Data Access Summaries. In *Proceedings of the International Symposium on Code Generation and Optimisation*. ACM. 2012. pp. 252–263.

[13] Møller, A.; Schwartzbach, M. I.: The Pointer Assertion Logic Engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2001. pp. 221–231.

[14] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society. 2002. pp. 55–74.

[15] Rinetzky, N.; Bauer, J.; Reps, T.; et al.: A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN Symposium on Principles of Programming Languages*. 2005. pp. 296–309.

[16] Sagiv, M.; Reps, T.; Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 1999. pp. 105–118.

[17] 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.

Appendix A

Contents of the CD

The attached CD contains source codes of the project. The main directory structure of the CD is the following:

```
/
├── 2ls/ ..... 2LS directory
│   ├── regression/ ..... 2LS regression tests
│   └── src/ ..... 2LS source files
├── cbmc/ ..... CBMC (CPROVER framework) directory
├── doc/ ..... Source files of this text
└── README ..... README file
```

Our extension is implemented as a part of the 2LS framework and thus the source files can be found in `2ls/src`. This directory is divided into multiple subdirectories, some of which contain components of our extension:

`domains` Contains abstract domains used in 2LS. Our shape domain is implemented by the following components:

- `heap_domain` Defines the template form and the abstract value.
- `list_iterator` The representation of list iterators.
- `strategy_solver_heap` Implements the abstract value synthesis algorithm.

`ssa` Contains components related to the creation and manipulation of the SSA form. Important components are:

- `local_ssa` The definition of the SSA form. It also contains algorithm for transformation of the GOTO program into the SSA form, which includes representation of heap-manipulating operations.
- `malloc_ssa` The replacement of `malloc` calls by dynamic objects.
- `ssa_inliner` Responsible for binding between the caller and the callee functions in interprocedural analysis.
- `ssa_pointed_objects` A library for working with pointed objects abstractions.
- `ssa_value_set` The points-to analysis.

The directory `doc` contains the LaTeX source files and the PDF version of this text.

Appendix B

Compilation and Running

The project can be compiled and run using the source files that are attached on the CD. Compilation can be done by the following steps:

1. Compile CBMC—this is a library for the whole CPROVER infrastructure that 2LS is built on. CMBC in correct version can be found on the CD and compiled using `cbmc/src/Makefile`.

2. Compile 2LS—requires CBMC to be compiled in the `cbmc/` folder. Compilation of 2LS can be done using `2ls/src/Makefile`.

2LS with our shape domain can be run by the following command:

```
2ls/summarizer/2ls --heap --no-propagation SOURCE_FILE
```

It is recommended to use the `--no-propagation` switch that turns off a propagation of constants in the GOTO program, which can sometimes cause problems for our points-to analysis. The file `SOURCE_FILE` must be a correct compilable sequential C program. Process and results of the analysis are printed to `stdout` and `stderr`.

We also recommend to use the 2LS regression tests that define a simple way of analysing programs with 2LS. Existing tests can be found in subdirectories of `2ls/regression`. Each test contains a C program to be verified and a test specification `test.desc` that defines the parameters and the expected results of the analysis.

Appendix C

2LS regression tests

2LS contains a number of regression tests divided into multiple categories. These can be found in `2ls/regression` directory where each category is contained in one subdirectory. Each category contains various verification tasks situated in separate folders. A task folder contains a C program to be analysed and a task specification. The result of the analysis is also stored in the task directory. All tests from a category can be run using `<category>/Makefile`. Apart from running each category separately, it is also possible to run all categories together using `2ls/regression/Makefile`.

There are 5 categories that contain the original 2LS regression tests that were used in the experiment in Section 5.3:

interprocedural Tasks in this category are aimed at verifying programs using interprocedural analysis. Here, a summary is computed for each function of the analysed program. The tests include both context sensitive and context insensitive analyses.

invariants Tasks in this category are aimed at computing invariants using various (mainly numeric and equalities) domains. The verified programs typically contain the `main` function only or are analysed using the `--inline` switch.

kiki Contains tasks aimed at checking features of the *k*I*k*I algorithm, mainly k-induction.

preconditions Tasks in this category are aimed at computing forward and backward preconditions and postconditions of functions of the analysed program, which is one of the features of 2LS. All tests are run with the `--preconditions` switch.

termination Tasks in this category are aimed at analysis of termination of functions in the analysed program. All tests are run with the `--termination` switch. The analysis uses lexicographic domain.

Within this work, we added 3 more categories of regression tests into 2LS that aim at checking properties related to the shape of the heap:

heap Contains tasks using interprocedural analysis of heap-manipulating programs. Here, it is possible to find functions that were used as examples in this thesis.

predator-dls Contains tasks from the Predator tool aimed at checking properties of doubly-linked lists. These tasks were used in the experiment in Section 5.2.

predator-sls Contains tasks from the Predator tool aimed at checking properties of singly-linked lists. These tasks were used in the experiment in Section 5.2.