Informatics

Enhancing Abstraction and Symbolic Execution for Shape Analysis of C-Programs operating on Linked Lists

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

David Kaindlstorfer, BSc
Matrikelnummer 01624252

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Mitwirkung: Prof. Ing. Tomáš Vojnar, Ph.D.
Doc. Mgr. Adam Rogalewicz, Ph.D.
Ing. Veronika Šoková

Wien, 25. Jänner 2022

David Kaindlstorfer   Florian Zuleger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Enhancing Abstraction and Symbolic Execution for Shape Analysis of C-Programs operating on Linked Lists

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

David Kaindlstorfer, BSc
Registration Number 01624252

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Assistance: Prof. Ing. Tomáš Vojnar, Ph.D.
Doc. Mgr. Adam Rogalewicz, Ph.D.
Ing. Veronika Šoková

Vienna, 25th January, 2022

David Kaindlstorfer Florian Zuleger

Erklärung zur Verfassung der Arbeit

David Kaindlstorfer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Jänner 2022

David Kaindlstorfer

Acknowledgements

First I want to thank my supervisor Florian Zuleger. The regular meetings were very helpful and valuable especially at the beginning of our collaboration where it was necessary to understand a quite complex analysis tool. Also I would like to thank for the fast and uncomplicated help at the time when it was not clear how to proceed with the problems I had.

Secondly I want to thank the team I collaborated with at VUT University in Brno for their effort in the meetings, their time during my visit in Brno and the support to understand Broom's source code. Tomáš Vojnar helped in getting an idea of the Predator analyser and gave high-level ideas how to resolve problems of Broom. Adam Rogalewicz helped me in debugging, understanding the abstraction procedure and resolving bugs. Veronika Šoková helped me in understanding the symbolic execution and working with the Infer analyser.

Kurzfassung

Diese Arbeit beschreibt Verbesserungen für das statische Analysetool Broom. Broom analysiert Programmteile eines C-Programmes und beweist, dass es frei von Speicherfehlern ist. Im Besonderen ist Broom in der Lage, Programme, die mit verschiedenen Varianten von verketteten Listen arbeiten, zu analysieren. Falls Broom das Eingabeprogramm als sicher klassifiziert, erzeugt es für jede Funktion eine Menge an Zusicherungen, die in einer bestimmten Form von Separation Logic definiert werden.

Broom ist derzeit noch ein Prototyp und die Komplexität der Listen, die analysiert werden können, ist begrenzt. Deshalb wurde der Abstraktionsalgorithmus von Broom im Rahmen dieser Arbeit so erweitert, dass er auch Listen unterstützt, bei denen jeder Knoten einen Zeiger auf ein gemeinsames Objekt hat. Diese Verbesserung ist praktisch relevant, da diese Listen gerne in systemnahen Code verwendet werden.

Darüber hinaus beschäftigt sich diese Arbeit mit dem Problem der Explosion des Zustandsraumes bei der Analyse komplexerer Funktionen. Es wurde in weiterer Folge eine Prozedur implementiert, die diesen Zustandsraum so beschneidet, dass die Zahl der Zusicherungen für eine Funktion signifikant reduziert wird. Besipielsweise führt diese Verbesserung dazu, dass die Zahl der Zusicherungen, die für eine Funktion, die über eine verschachtelte Liste iteriert, von 400 auf zwei reduziert wird. Als Konsequenz konnte die Dauer der Analyse des entsprechenden Codes auf vier Minuten reduziert werden, während die Analyse vor der Implementierung der Verbesserungen praktisch unmöglich war.

Abstract

This thesis describes improvements of the static analyser Broom. Broom can analyse fragments of programs written in the programming language C proving the absence of memory bugs. Specifically, the tool supports programs operating on different kinds of linked lists. If Broom considers the input program as safe, it generates for each function a set of contracts, which are defined in a special flavour of Separation Logic.

Broom is still a prototype and the complexity of the lists that can be analysed is limited. To this end we extended Broom's abstraction procedure such that also lists with pointers to shared non-global heap objects, which are common in system code, can be analysed.

Additionally we tackled the state space explosion problem, which may occur when Broom analyses more complex functions. We implemented a pruning strategy that significantly reduces the number of contracts for these functions. This improvement allowed us to reduce the number of contracts that would be generated for a function traversing a nested list from over 400 to two. This results in an analysis of the respective code within approximately four minutes while analysing this code was practically infeasible before the pruning was implemented.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Using dynamic data structures such as (doubly) linked lists, which can be circular and nested in the C programming language is error-prone. Pointer arithmetic may be used to achieve a more efficient implementation, which can cause nasty programming errors as well. These complex lists are used for instance in the Linux kernel. Therefore static analysis techniques for these lists are of practical relevance.

1.2 State of the Art

In the past different static analysis methods for C-code manipulating linked lists have been developed. They detect invalid memory access, double-free errors and memory leaks (which occur when allocated memory is not freed at program exit). These techniques analysing dynamic pointer-linked data structures are called *Shape Analysis* and are known as specifically hard to design.

Most of the existing approaches have different drawbacks such as a lack of support for low-level pointer operations or a low-level of automation and are therefore not applicable in practise. [HPR+22]

The tool Broom is able to overcome many of these issues. The source code is written in OCaml and publicly available [1] under GNU GPLv3. Broom is a sound static analyser that either finds memory-related bugs or determines for each function a contract, which consists of a pre- and post-condition. This contract reflects the absence of invalid memory access, double-free errors and memory leaks. The pre- and post-conditions are specified in a special flavour of Separation Logic with inductive list predicates and describe the

[1] `https://pajda.fit.vutbr.cz/rogalew/broom`

configuration of the heap and stack on a byte-precise level. As an example look at figure 1.1. Read the *-operator - for simplicity - as a logical "and". The predicate $x \mapsto y$ is the points-to predicate stating that at address x value y is stored. With this semantics the contract specifies exactly the behaviour of the store-operator in C.

```
void store(int *x, int y){
    *x = y;
}
```

Figure 1.1: Simple store-function with $P \equiv x = X * y = Y * X \mapsto z$ and $Q \equiv x = X * y = Y * X \mapsto Y$

The analysis process of Broom can be described as follows. For each basic statement in a function there is a contract specified. To obtain a contract for a whole function, Broom symbolically executes each statement following the function's control flow graph and iteratively tries to extend the contract. To avoid that this symbolic execution diverges for loops, Broom needs to automatically generate inductive loop invariants. Typically these loop invariants contain inductive list predicates. For instance the list predicate $\mathsf{ls}(\varepsilon_1, \varepsilon_2)$ states that there is a singly-linked list of length l (with $l \geq 0$) from a block located at address ε_1 to a block located at address ε_2. The example in figure 3.1 shows a simple loop traversal where the loop invariant is $\mathsf{ls}(x, \text{null})$.

In fact Broom got inspired by the functionalities of the Predator analyzer [DPV13] but aims at removing one of its major limitations: The need that the input program is closed, which means that it contains a main function as unique entry point. To that end Broom first analyses the functions at the bottom of the call tree and uses the computed contracts to analyse functions higher up in the call tree (which currently prevents Broom from analysing recursive functions). The ability to analyze open programs is crucial because the necessity to provide a main function as entry point requires programmers to write harnesses initializing all data structures.

One well-known analyser which is capable of dealing with open programs and was also an inspiration for Broom is the Infer analyser [CD11]. Similarly to Broom, Infer is based on Separation Logic and Bi-abduction. However the variant of Separation Logic used in this thesis allows for more precise reasoning as also seen in section 2.3. Additionally Infer has the drawback that it is not very complete (as also seen in the experiments of [HPR+22]).

1.3 Problem Statement

Unfortunately Broom is still a prototype. One of its issues is its low performance: The analysis is very time-consuming such that analysing even quite simple functions such as a function traversing nested lists is intractable.

Another issue of Broom is that the set of lists which can be modeled and analysed is relatively limited. This is related to the fact that the abstraction procedure, which generates the loop invariants, is quite basic. For instance it does not support lists with shared nodes/data elements, which are common in C system code (see e.g. [BCC⁺07]). For these lists we have for each node a pointer to a unique allocated block.

The aim of this thesis is to improve Broom's analysis capabilities by tackling the above mentioned problems. To this end we have to identify the reasons why the performance of the analysis process is low and have to implement techniques resolving the underlying problems. Secondly we introduce support for lists with shared nodes/data elements. This is non-trivial because these types of lists cannot be represented with our current definition of Separation Logic.

1.4 Methods

The methodological approach contains extensive testing of Broom. First we need to do systematic experiments to delimit the reasons for Broom's low performance. After designing improvements for the underlying problems on an abstract level, we implement them in Broom's source code. Finally we again perform experiments with Broom on a set of benchmark instances which mostly come from the Predator benchmark suite to evaluate the efficacy of the improvements. We will designate a certain set of benchmark instances which can be handled due to the improvements implemented.

1.5 Contributions

In our initial experiments we found out that for programs whose functions have multiple contracts and for programs with a deep function call tree such as the program traversing a nested list, the main reason for Broom's low performance is related to an explosion of the state space. We describe this problem in section 3.1. In sections 3.2, 3.3 and 3.4 we see how we can tackle this state space explosion by implementing pruning strategies.

In section 4.1 we describe how the Separation Logic used to specify contracts was extended in order to allow Broom modelling of lists with shared nodes. In the following sections 4.2 and 4.3 we give the first detailed description of the abstraction procedure of Broom, which also includes the changes implemented in order to support the altered syntax of the Separation Logic. In [HPR⁺22] the abstraction procedure was not described in detail, which is why people not knowing the source code of Broom had difficulties in understanding some problems related to abstraction. One of these problems is described in section 4.4: Some shapes of nested lists could not be abstracted which resulted in a diverging analysis process for these lists.

In section 5 we will finally give some sample C-files, which are mostly taken from the benchmark suite for the Predator analyser. These benchmark instances should illustrate the increased variety of lists which can be analysed thanks to the changes implemented.

CHAPTER 2

Preliminaries

In this section we will define everything described in the previous section more formally mostly summing up [HPR+22].

2.1 Memory Model

In order to define the operational semantics of the C programming language as our low-level language we need to define a memory model. With that memory model we will define how the stack and heap evolves by executing a basic statement. Later we will also use this memory model to define the contracts (P, Q) of whole functions in a special flavour of Separation Logic.

Our analysis should reason about programs on a byte-precise level. Therefore we define the set of values Val as sequences of bytes. For a machine with an N-byte architecture we can then designate the set of memory locations $Loc \subseteq Val$ as the set of all N-byte words.

With these definitions we can define configurations in our memory model as stack-block-memory triplets (S, B, M). Let Var be a set of variables. Additionally each variable $x \in Var$ has a constant positive size denoted $\mathsf{size}(x)$, which determines the number of bytes used by x.

Stack. S represents the stack and is a total function $Var \to Val$. So $S(x)$ is a sequence of bytes of length $size(x)$.

Blocks. B represents the set of allocated blocks in the heap. That is a set of intervals $\{\ [l, u) \mid 0 < l < u$ where $l, u \in Loc\}$ which is non-overlapping. The constraint $0 < l$ ensures that there cannot be a block allocated at address null.

Memory. $M : Loc \rightharpoonup Byte$ is a partial function defining the contents of allocated memory locations. We have that for every $\ell \in Loc$ s.t. $M(\ell)$ is defined, there is a block $[l, u) \in B$ s.t. $\ell \in [l, u)$

2.2 Operational Semantics of the Low-level language

Using the memory model defined in the previous section we can define the operational semantics of our low-level language. Our low-level language is close to the intermediate languages produced by common C compilers such as `gcc` or `clang`. We assume a type checker which checks that the LHS and RHS for assignments are compatible.

To illustrate the operational semantics, we will just give the semantics of the store-operator (used in figure 1.1). We use $M[\ell, \ell')$ to denote the byte sequence $M(\ell)M(\ell+1)\cdots M(\ell'-1)$. For a (partial) function f, $f[a \hookrightarrow b]$ denotes the (partial) function identical to f up to $f[a \hookrightarrow b](a) = b$. The full specification of the low-level language with its semantics is given in [HPR+22].

$$(S, B, M) \xrightarrow{*x:=y} \text{ if } \mathfrak{b}_B(S(x)) = 0 \text{ or } S(x) + \mathsf{size}(y) > \mathfrak{e}_B(S(x)), \\ \text{then } \mathit{error} \text{ else } (S, B, M[[S(x), S(x) + \mathsf{size}(y)) \hookrightarrow S(y)])$$

The functions $\mathfrak{b}_B, \mathfrak{e}_B : Loc \to Loc$ return the base or end address, respectively, of a block in B to which a given location belongs. More formally: Given some $\ell \in Loc$, $\mathfrak{b}_B(\ell) = l$ in case there is some $[l, u) \in B$ with $\ell \in [l, u)$. Otherwise $\mathfrak{b}_B(\ell) = 0$. Likewise for $\mathfrak{e}_B(\ell)$

2.3 Separation Logic

In the following we will introduce a Separation Logic which will allow us to define contracts for functions of our low-level language such as the contract defined in figure 1.1.

The semantics of the Separation Logic uses the memory model defined in section 2.1.

The **syntax** for a formula φ is given as:

$$\begin{aligned} \varphi \ ::= \ & \varepsilon_1 \mapsto \varepsilon_2 \mid \varepsilon_1 \mapsto k[\varepsilon_2] \mid \varepsilon_1 \mapsto \top[\varepsilon_2] \mid \varphi_1 * \varphi_2 \mid \phi_1 \vee \phi_2 \mid \mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2) \mid \\ & \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2') \mid \mathsf{emp} \mid \mathit{true} \mid \varepsilon_1 \bowtie \varepsilon_2 \mid \exists x.\varphi \\ \bowtie \ ::= \ & = \mid \neq \mid \leq \mid < \mid \geq \mid > \qquad \varepsilon \ ::= \ k \mid x \mid \mathfrak{b}(\varepsilon) \mid \mathfrak{e}(\varepsilon) \mid \mathsf{uop}\ \varepsilon \mid \varepsilon_1 \ \mathsf{bop}\ \varepsilon_2 \end{aligned}$$

The unary operator uop and binary operator bop can be the following set of operators of common low-level languages: $+$, $-$, $*$, &, |. Additionally there is a concatenation operator $\odot$ on byte sequences. We call x, y and z variables, k a constant, ε, ε_1, ε_1', ε_2, ε_2' expressions and φ_1 and φ_2 formulas. We call predicates of form $\varepsilon_1 \bowtie \varepsilon_2$ *pure* and points-to predicates as well as list predicates *spatial* predicates.

We will first define the formal **semantics** w.r.t. SBM triplets and then explain it informally. However we will only give the semantics of the separating conjunction operator $*$ as well as the ls and dls predicate. We chose the separating conjunction operator as an illustrative example. The list predicates are presented because their semantics will be extended in section 4.1. The exact semantics of the other operators can be found in [HPR+22].

$$(S, B, M) \models \varphi_1 * \varphi_2 \text{ iff there are } M_1, M_2 \text{ with } M = M_1 \uplus M_2, (S, B, M_i) \models \phi_i$$

In fact the separating conjunction operator $\varphi_1 * \varphi_2$ has more properties than a simple logic "and". It requires that the heap M can be split into two *disjoint* sub-heaps M_1, M_2 s.t. $(S, B, M_1) \models \varphi_1$ and $(S, B, M_2) \models \varphi_2$. For points-to predicates $\varepsilon_1 \mapsto \varepsilon_1'$ in φ_1 and $\varepsilon_2 \mapsto \varepsilon_2'$ in φ_2 this means that the byte sequences ε_1' and ε_2' are non-overlapping in M. Notice that in our setting of Separation Logic we use a *per-field separating conjunction*. Therefore we define fields of object structures to be non-overlapping which allows us to reason about programs on a precise level. This is a big improvement compared to a *per-object separating conjunction* that has been used in e.g. in [CDOY11] and in the Infer analyser.

$$\begin{aligned}(S, B, M) \models \mathsf{ls}_{\Lambda(a,b)}(\varepsilon_1, \varepsilon_2) \text{ iff } & (S, B, M) \models \varepsilon_1 = \varepsilon_2 \text{ or} \\ & (S, B, M) \models \varepsilon_1 \neq \varepsilon_2 * \mathsf{true} \text{ and there is some } \ell \in \mathit{Loc} \\ & \text{and a fresh variable } u \in \mathit{Var} \text{ s.t.} \\ & (S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u) * \mathsf{ls}_{\Lambda(a,b)}(u, \varepsilon_2)\end{aligned}$$

As one can see in the definition above the singly-linked list predicate ls is parameterized by a formula Λ. We call Λ the list segment predicate because it expresses local information for each list segment/node. The list segment predicate has form $\Lambda(a, b) \equiv \exists x_1, ...x_k.\varphi$ where φ is quantifier-free and does not contain the disjunction operator $\vee$. With these restrictions we can describe data fields in blocks or even nested lists as we will see in section 3. For instance the list segment predicate for the structure defined in figure 3.1 would be simply $\Lambda(a, b) \equiv a \mapsto b$. If we add to the structure a data field, we would have $\Lambda(a, b) \equiv \exists d.a \mapsto b * a + 8 \mapsto d * \mathfrak{b}(a) = \mathfrak{b}(a + 8)$.

Coming back to definition of $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2)$ we see that a list from node ε_1 to ε_2 is either empty or there is a node fulfilling Λ at ε_1 which is linked to a tail list $ls_\Lambda(u, \varepsilon_2)$. As the definition of ls uses a the ls predicate itself, it is an inductive definition. The meaning of the list predicates is also depicted in figure 2.1.

$$\begin{aligned}(S, B, M) \models \mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2') \text{ iff } & (S, B, M) \models \varepsilon_1 = \varepsilon_2' * \varepsilon_2 = \varepsilon_1' \text{ or} \\ & (S, B, M) \models \varepsilon_1 \neq \varepsilon_2' * \varepsilon_2 \neq \varepsilon_1' * \mathsf{true} \text{ and there is some } \ell \in \mathit{Loc} \\ & \text{and a fresh variable } u \in \mathit{Var} \text{ such that} \\ & (S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u, \varepsilon_2) * \mathsf{dls}_{\Lambda(x,y,z)}(u, \varepsilon_1, \varepsilon_1', \varepsilon_2')\end{aligned}$$

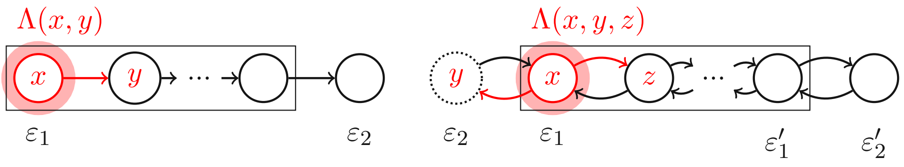

Figure 2.1: An illustration of $\mathsf{ls}_{\Lambda(x,y)}(\varepsilon_1, \varepsilon_2)$ and $\mathsf{dls}_{\Lambda(x,y,z)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$

The definition of the dls predicate is very similar to the definition of the ls predicate. We notice that the list segment predicate Λ has three parameters, because the current node is linked to the predecessor and the successor node. The parameters of the dls predicate allow to describe different types of doubly-linked lists: For instance a circular list has $\varepsilon_1 = \varepsilon_1'$ and $\varepsilon_2 = \varepsilon_2'$.

2.4 Bi-Abduction

So far we only defined the semantics of our low-level language and the Separation Logic which we will use to define contracts of functions. However we have only informally discussed in the introductory section how Broom generates these contracts.

We first need to determine contracts defining the semantics of all basic statements of our programming language (which we will also model as functions) in order to infer a contract for a whole function. We will not give the complete list of contracts for the basic statements but refer again to [HPR+22]. As an example the contract of the store function is shown in figure 1.1. All contracts created by Broom have form $(C, D_1 \vee ... \vee D_l)$ where $C, D_1, ..., D_l$ are formulas in our Separation Logic not containing disjunction.

Let $f(x_1, ..., x_n)$ be the function for a basic statement in our low-level language and let (P, Q) be a contract for f. (P, Q) is **sound**, which means that for all triplets (S, B, M) s.t. $(S, B, M) \models P$ and all executions of f that start from (S, B, M) and end in some configuration (S', B', M') we have that $(S', B', M') \models Q$.

This follows directly from the semantics of our low-level language as stated in section 2.2.

In general the set of variables Var can be partitioned into two sets: The set of logical variables $LVar$ and the set of program variables $PVar$. For a function $f(x_1, ..., x_n)$ we always have $\{x_1, ..., x_n\} \subseteq PVar$. In C $LVar$ covers all local variables while $PVar$ contains the function parameters and all global variables.

Let us first focus on functions f whose function calls have contracts with conjunctive post-conditions. Furthermore assume that f does not use branching and looping. Therefore we can see f as a sequence $s_1, ..., s_n$ of function calls. In section 2.5 we will lift all these constraints.

As already stated in the introductory section, the symbolic execution iteratively extends an initial state (P_{init}, Q_{init}) for all function calls $s_1, ..., s_n$. In each step this state reflects a sound contract for the function up to a specific function call s_i. To extend a state and symbolically execute one function call $s_i = g(y_1, ..., y_m)$ we have to perform bi-abduction: Let (C, D) be the contract for g and $Q \equiv \exists U_Q.Q_{free} * Q_{free} * Q_{eq}$, so we split Q into its pure and spatial part. We also split D: $D \equiv \exists U_D.D_{free} * D_{eq}$. The bi-abduction problem is given by

$Q_{free} * [?] \models C' * [?]$

where $C' \equiv (C[a_i/y_i])[\varepsilon_j/x_j]$ and $Q_{eq} \equiv x_1 = \varepsilon_1 * ... * x_n = \varepsilon_n$

So we construct C' by instantiating all parameters $y_1, ..., y_m$ by their actual values $a_1, ..., a_n$ in the function call and "apply" Q_{eq} by renaming which gives us a formula without any program variable $x_1, ..., x_n, y_1, ..., y_m$.

We can see that in the bi-abduction problem there are two formulae computed: The missing formula on the LHS of the entailment is called *antiframe* and is denoted by M. Intuitively it represents the additional constraint needed to safely execute s_i. The missing formula on the RHS is called *frame*, denoted by F, and represents the part of the heap that stays untouched by s_i. Using the computed antiframe and frame we can update our state:

$P_{after} := M * P$

$Q_{after} := \exists U_{Q_{after}}.Q'_{free} * Q'_{eq}$

where $U_{Q_{after}} = (var(Q'_{free} * Q'_{eq}) \cap LVar) \setminus var(P_{after})$, $Q'_{free} = F * D_{free}$ and Q'_{eq} is constructed from D_{eq} for variables which are passed to g and from Q_{eq} for variables which are not passed to g. For later use we write $biabduct(P, Q, g(a_1, \ldots, a_m), C, D)$ if we apply one step of the bi-abduction procedure as explained.

The computation of the frame and antiframe in the abduction process is done using a set of *abduction rules*. As a deeper understanding of this computation is not required in the remaining sections, we refer the interested reader to [HPR+22]. There you can also find a soundness proof for the bi-abduction procedure.

Before we explain the general setting of symbolic execution for functions with branching and functions with disjunctive post-conditions we need to define the following:

Let $C_1 = (P_1, Q_1)$ and $C_2 = (P_2, Q_2)$ be contracts or (intermediate) symbolic states. We have

$$(P_1, Q_1) \sqsubseteq (P_2, Q_2)$$

$$\Leftrightarrow$$

$$\exists LVar(P_1).P_1 \models \exists LVar(P_2).P_2 \text{ and } \exists LVar(Q_1).Q_1 \models \exists LVar(Q_2).Q_2$$

where $LVar(\varphi) = var(\varphi) \cap LVar$.

If $C_1 \sqsubseteq C_2$, we say that C_2 covers C_1.

2.5 Symbolic Execution

In this section we will remove the constraints from the previous section that contracts are disjunctive and that functions are a sequence of function calls. Instead of a list of function calls we are now given a control flow graph (CFG) with $(V, E, entry, exit)$ where edges are labeled by a function call. The entry/exit points of the CFG are denoted by $entry, exit \in V$ respectively. There are also special cut-points $V_{cut} \subseteq V$ that represent loop headers. Additionally there is a mapping $symb(v)$ which stores for each node v a list of states. This mapping $symb(v)$ is maintained in the symbolic execution by the following worklist algorithm:

```
1  procedure symbolic_execution($(V, E, entry, exit), P_{init}, Q_{init}$):
2      $symb(entry)$ := $\{(P_{init}, Q_{init})\}$
3      let $v \in V$ be some node which needs to be processed
4      foreach $(v, v') \in E$
5          let $g(a_1, ..., a_k)$ be the function label for $(v, v')$
6          let $\mathcal{C}(g)$ be the set of contracts for function $g$
7          foreach $(C, D_1 \vee ... \vee D_l) \in \mathcal{C}(g)$
8              for $i$ = 1 to $l$
9                  $(P_{after}, Q_{after})$ := biabduct($P, Q, g(a_1, ..., a_k), C, D_i$)
10                 if $v' \in V_{cut}$
11                     foreach $(P, Q) \in symb(v')$
12                         if $((P_{after}, Q_{after}) \sqsubseteq (P, Q))$
13                             break
14                     $symb(v)$ := $symb(v)$ $\cup$ $\{(\alpha(P_{after}), \alpha(Q_{after}))\}$
15                 else
16                     $symb(v)$ := $symb(v)$ $\cup$ $\{(P_{after}, Q_{after})\}$
17     goto 3
```

Listing 2.1: Pseudocode for the symbolic execution algorithm

In the algorithm α is an abstraction procedure which replaces spacial predicates by ls or dls predicates. The abstraction procedure is described in detail in chapter 4. On one hand abstracting states means losing information like the length of the list but on the other hand this is crucial for termination of the analysis of loops. This is connected to the second requirement needed for termination of loop analysis: The need to do entailment checks in line 12. We call these entailment checks *invariant checks* because they check whether a newly created state is already covered by a state in $symb(v')$. If this is true, we can safely discard (P_{after}, Q_{after}). Notice that the code presented is Broom's standard symbolic execution algorithm. However you can adjust Broom's behaviour by passing an option which will make Broom perform abstraction before entailment checks. An example of how the analysis of a loop proceeds is given in figure 3.2.

The implementation of entailment checks will not be described in this thesis as it is not required for the understanding of the further sections. However in [HPR+22] it is described how entailment checking was reduced to bi-abduction.

Notice that executing one round of the worklist algorithm in Listing 2.1 does not give sound contracts for a function f but only a set of contracts whose pre-conditions are *candidate* preconditions. Following [CDOY11], Broom implements a two-round analysis for functions with branching:

```
procedure symbolic_execution_branching($G = (V, E, entry, exit)$):
    // round 1
    let $P_{init} \equiv Q_{init} \equiv x_1 = X_1 * ... * x_n = X_n$
    let $(P_1, Q_1), ..., (P_k, Q_k)$ = symbolic_execution($G, P_{init}, Q_{init}$)
    let $\mathcal{C}(f) = \emptyset$
    // round 2
    for i = 1 to k
        let $P_{init} \equiv Q_{init} \equiv P_i$
        let $(P_i, Q_1), ..., (P_i, Q_l)$ be the result of
            symbolic_execution($G, P_{init}, Q_{init}$) with
            requirement $M = \mathsf{emp}$ in $biabduct$
        $\mathcal{C}(f)$ := $\mathcal{C}(f) \cup (P_i, Q_1 \vee ... \vee Q_l)$
    return $\mathcal{C}(f)$
```

Listing 2.2: Pseudocode for the two-round analysis of functions with branching

In Listing 2.2 we can see that in the first round we only generate candidate preconditions for a function f. In the second round we use these pre-conditions as initial states and do not allow the strengthening of the pre-conditions in $biabduct(\cdot)$. If therefore *biabduct* fails, we discard this pre-condition. On success we return the sound contract $(P_i, Q_1 \vee ... \vee Q_l)$.

Notice that in the second phase for a given pre-condition P_i, Broom internally stores for each possible post-condition Q_j (with $j = 1, ..., l$) computed for P_i, a new contract (P_i, Q_j) instead of a single contract $(P_i, Q_1 \vee ... \vee Q_l)$. This facilitates the abduction application for a given contract without changing the semantics. To be consistent with Broom's behaviour, we will also replace disjunctive contracts by multiple contracts not containing the disjunction in the following section.

CHAPTER 3

Enhancing Symbolic Execution

In section 3.1 we will outline Broom's current problems when analyzing functions dealing with (nested) lists. These problems are mostly related to the state space explosion. In section 3.2 there will be a pruning strategy for contracts and states defined. Implementing these optimizations will drastically improve the running time and allow Broom to analyze functions traversing nested lists.

In the following we use $\mathsf{ls}(\varepsilon_1, \varepsilon_2)$ and $\mathsf{dls}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ instead of $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2)$ and $\mathsf{dls}_\Lambda(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ if the exact definition of Λ is irrelevant or implicit.

3.1 Motivation

As one can see in [HPR+22], where the analysis of the loop traversal function in figure 3.1 is described, there will be typically several contracts created for functions with loops.

In [HPR+22] we see that the set of contracts computed after the first analysis phase is given by:

- $C_1 = (P_1 \equiv X = \text{null} * x = X, Q_1 \equiv X = \text{null})$
- $C_2 = (P_2 \equiv X \mapsto \text{null} * x = X, Q_2 \equiv X \mapsto \text{null})$
- $C_3 = (P_3 \equiv \mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X, Q_3 \equiv \mathsf{ls}(X, \text{null}) * X \neq \text{null})$

Contracts C_2, C_3 require the run of the second analysis phase, because they are retrieved from symbolic executions which traversed the loop body. In the following we will show how the second analysis round evolves. We will only show here the post-conditions as the pre-conditions do not change in the second round.

```
struct sll {struct sll *next;};

void loop(struct sll *x){
    while(x!=NULL){
        x=x->next;
    }
}
```

Figure 3.1: Simple list traversal

The second run for C_2 will initially evaluate `x!=NULL` to *true* because this can be derived from P_2. The loop body will be symbolically executed which will give us symbolic state $Q \equiv X \mapsto \text{null} * x = \text{null}$. When now the loop header `x!=NULL` is evaluated according to P_2, the result will be *false* and the second run will terminate with contract $C_2' = C_2$.

The second run of contract C_3 is a bit more complex. It is schematically depicted in figure 3.2. Again `x!=NULL` will initially be evaluated to *true*, so the loop body will be executed ending up in symbolic state $Q \equiv X \mapsto L_1 * \mathsf{ls}(L_1, \text{null}) * x = L_1$. When reaching the loop header again, we do the invariant checks and execute abstraction function α which both fail. As we now cannot derive `x!=NULL`, we do a case distinction: For the branch of $x = L_1 = \text{null}$, we stop symbolic execution and obtain final contract $C_3' = (P_3, X \mapsto L_1 * \mathsf{ls}(L_1, \text{null}) * L_1 = \text{null})$ which can be simplified to $C_3' = (P_3, X \mapsto \text{null})$ because of $L_1 = \text{null}$ and the semantics of the ls predicate for empty lists. If $L_1 \neq \text{null}$, we again execute the loop body obtaining symbolic state $Q \equiv X \mapsto L_1 * L_1 \mapsto L_2 * \mathsf{ls}(L_2, \text{null}) * x = L_2$. Again we do invariant checks and perform abstraction. The invariant checks fail, but abstraction succeeds and we get $Q \equiv \mathsf{ls}(X, L_2) * \mathsf{ls}(L_2, \text{null}) * x = L_2$. We cannot derive `x!=NULL`, so similarly to the former iteration we have to do case distinction: If $x = L_2 = \text{null}$, we stop symbolic execution and obtain contract $C_3'' = (P_3, \mathsf{ls}(X, L_2) * \mathsf{ls}(L_2, \text{null}) * L_2 = \text{null})$ which can be simplified to $C_3'' = (P_3, \mathsf{ls}(X, \text{null}))$. If $L_2 \neq \text{null}$ we again execute the loop body obtaining symbolic state $Q \equiv \mathsf{ls}(X, L_2) * \mathsf{ls}(L_3, \text{null}) * L_2 \mapsto L_3 * x = L_3$. When finally reaching again the loop header, we do invariant checks, which succeed because we have $\exists L_2 \exists L_3. \mathsf{ls}(X, L_2) * \mathsf{ls}(L_3, \text{null}) * L_2 \mapsto L_3 * x = L_3 \models \exists L_2. \mathsf{ls}(X, L_2) * \mathsf{ls}(L_2, \text{null}) * x = L_2$. Therefore the analysis is aborted and we have the following final contracts for the function `loop`:

1. $C_1 = (X = \text{null} * x = X, X = \text{null})$
2. $C_2' = (X \mapsto \text{null} * x = X, X \mapsto \text{null})$
3. $C_3' = (\mathsf{ls}(X, \text{null}) * x = X * X \neq \text{null}, X \mapsto \text{null})$
4. $C_3'' = (\mathsf{ls}(X, \text{null}) * x = X * X \neq \text{null}, \mathsf{ls}(X, \text{null}))$

$$\begin{tikzcd}
 & \mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X \arrow[d, dash, "\text{x != NULL}"] & & \\
 & \mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X \arrow[d, dash, "\text{x = x} \to \text{next}"] & & \\
 & X \mapsto L_1 * \mathsf{ls}(L_1, \text{null}) * x = L_1 \arrow[dl, dash, "\text{x = NULL}"'] \arrow[dr, dash, "\text{x != NULL}"] & & \\
\underline{X \mapsto \text{null}} & & X \mapsto L_1 * \mathsf{ls}(L_1, \text{null}) * L_1 \neq \text{null} * x = L_1 \arrow[d, dash, "\text{x = x} \to \text{next}"] & \\
 & & X \mapsto L_1 * L_1 \mapsto L_2 * \mathsf{ls}(L_2, \text{null}) * x = L_2 \arrow[d, dash, "\alpha"] & \\
 & & \mathsf{ls}(X, L_2) * \mathsf{ls}(L_2, \text{null}) * x = L_2 \arrow[dl, dash, "\text{x = NULL}"'] \arrow[dr, dash, "\text{x != NULL}"] & \\
 & \underline{\mathsf{ls}(X, \text{null})} & & \mathsf{ls}(X, L_2) * \mathsf{ls}(L_2, \text{null}) * L_2 \neq \text{null} * x = L_2 \arrow[d, dash, "\text{x = x} \to \text{next}"] \\
 & & & \mathsf{ls}(X, L_2) * \mathsf{ls}(L_3, \text{null}) * L_2 \mapsto L_3 * x = L_3 \\
 & & & \text{STOP - entailment!}
\end{tikzcd}$$

Figure 3.2: Execution of the second analysis round of simple list traversal code (see figure 3.1) for contract C_3 with $P \equiv \mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X$

All in all we see that the second analysis phase is very similar to the first analysis phase. The main difference is that in the first analysis phase the pre-condition is strengthened whereas in the second phase the post-condition is strengthened.

Now consider the code for traversing a nested loop in figure 3.3.

```
struct outer_sll {
    struct outer_sll *next;
    struct sll *nested;
};

void outer_loop(struct outer_sll *x){
    while(x!=NULL){
        loop(x->nested);
        x=x->next;
    }
}
```

Figure 3.3: Simple traversal of a list and its nested lists

How would Broom analyze function `outer_loop` which traverses each nested list?

In the first analysis phase the initial state S_0 will be branched for `x!=NULL`. So the contract $(X = \text{null} * x = X, X = \text{null})$ will be created. This is the only contract which does not require a second run. The other branch will evolve to a symbolic state $S'_0 = (X \neq \text{null} * x = X, X \neq \text{null} * x = X)$. Whenever `loop(x->nested)` is symbolically executed, each state is split into four new successor states because there are four final contracts calculated for the function `loop`. Therefore after executing the loop body, we obtain states $S_{1,1}, S_{1,2}, S_{1,3}, S_{1,4}$ describing a single node with a possibly empty, singleton or non-empty nested list. When the symbolic execution reaches the loop header again, entailment checks and abstraction fail. So we start from the beginning and branch again for `x!=NULL`. This will create candidate contracts $C_{1,1}, C_{1,2}, C_{1,3}, C_{1,4}$ for the case $x = \text{null}$. For the case $x \neq \text{null}$ we continue with states $S'_{1,1}, S'_{1,2}, S'_{1,3}, S'_{1,4}$. The symbolic execution will branch them in the loop body to new states $S_{2,1}, ..., S_{2,16}$ and abstraction will succeed on these states because they describe two consecutive nodes. Branching for `x!=NULL` at the loop header node will finally give us candidate contracts $C_{2,1}, ..., C_{2,16}$. After one more loop iteration the entailment check for all $16 \cdot 4 = 64$ states will succeed. So all in all the first analysis round will give us $16 + 4 = 20$ candidate contracts which need a second run. The process is schematically depicted in figure 3.4.

When trying to run the second phase for each candidate contract, we will encounter the same exponential blowup. So we would have to expect $20 \cdot 20 = 400$ final contracts and $20 \cdot 16 \cdot 4 = 1280$ succeeding entailment checks. Knowing that abstraction and entailment checks are quite costly, we can easily see that executing this is infeasible. In fact the real computation effort is even higher because in some symbolic states abstraction

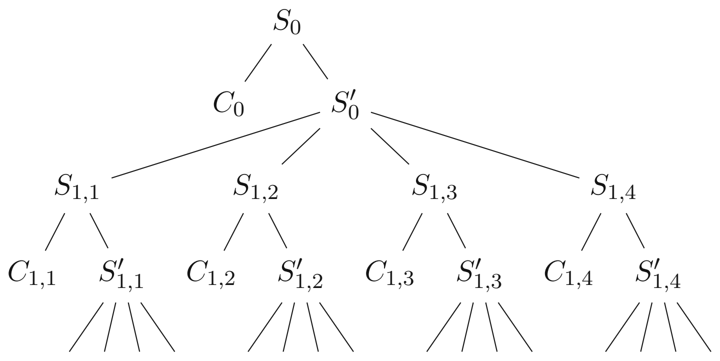

Figure 3.4: Schematic computation tree for first analysis round of nested list traversal code (see figure 3.3) with all four contracts for inner loop (see page 14)

cannot succeed after two iterations as `loop(x->nested)` creates a dummy variable for `x->nested` which will become a program variable pointing to the nested list of the second node.

3.2 Pruning of Contracts

It is clear that many of the contracts that Broom computes are redundant. Some of them are even syntactically equivalent. The plan is to remove as many contracts as possible without compromising soundness and by just using program functionalities already implemented - entailment checks and abstraction.

The idea is the following: If $C_1 \sqsubseteq C_2$, it is safe to remove C_1, because all program executions covered by C_1 are already covered by C_2. By doing this we can prune the set of contracts for a function without compromising soundness nor increasing the false positive rate.

In our two-round analysis this pruning works as follows. At the end of the first analysis round, we check for each pair of contracts (C_1, C_2) with $C_1 = (P_1, Q_1)$ and $C_2 = (P_2, Q_2)$ if $C_1 \sqsubseteq C_2$ and remove C_1 if this holds. If additionally $C_2 \sqsubseteq C_1$ holds, C_1 and C_2 are semantically or even syntactically equivalent. In this case we remove only one of C_1 and C_2. However after the first analysis phase we do not perform both entailment checks as given in the definition of $C_1 \sqsubseteq C_2$, but we only check $\exists LVar(P_1).P_1 \models \exists LVar(P_2).P_2$ which is justified by the fact that we drop the post-conditions Q_1, Q_2 anyway when we start the second analysis round.

We do something similar at the end of the second analysis round: We check for each pair of contracts (C_1, C_2) if $C_1 \sqsubseteq C_2$ and remove C_1 if this holds. Again we can do an optimization reducing the number of entailment checks needed: To determine $C_1 \sqsubseteq C_2$ it is sufficient to check $P_1 = P_2$ and $\exists LVar(Q_1).Q_1 \models \exists LVar(Q_2).Q_2$. This is justified by the fact that $\exists LVar(P_1).P_1 \models \exists LVar(P_2).P_2$ holds in the second phase only if $P_1 = P_2$

as otherwise C_1 or C_2 would have been pruned after the first analysis round.

To illustrate that, consider the new execution for the simple loop example in figure 3.1:

After the first run of the analysis we see that $C_2 = (X \mapsto \text{null} * x = X, X \mapsto \text{null}) \sqsubseteq C_3 = (\mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X, \mathsf{ls}(X, \text{null}) * X \neq \text{null})$ holds. In fact as we are in the first analysis round, we only see the entailment for the pre-conditions. Therefore we discard contract C_2 and only proceed with contracts C_1 and C_3.

At the end of the second run we will detect $C'_3 = (\mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X, X \mapsto \text{null}) \sqsubseteq C''_3 = (\mathsf{ls}(X, \text{null}) * X \neq \text{null} * x = X, \mathsf{ls}(X, \text{null}))$. Therefore we discharge contract C'_3 and only get contracts C_1 and C''_3 as final result for the `loop` function.

This reduction from four to two contracts for the `loop` function greatly reduces the computation effort needed for the nested loop function in the second run because in each iteration we do not branch into four but only into two sub-states. This allows Broom to analyze the `outer_loop` function as described in the following.

The first analysis phase of the `outer_loop` function now proceeds as depicted in figure 3.5. In each loop iteration we again branch at the loop header `x!=NULL`. The symbolic execution along the "else"-branch (where $x = \text{null}$ holds) again immediately stops giving a candidate contract, which does not require a second run. The other branch will execute the loop body splitting symbolic execution into two sub-states because we are given two contracts for the `loop` function. One sub-state will define an empty, the other a non-empty nested list. This continues until the symbolic states contain two successive nodes because this will allow abstraction to succeed.

After abstraction succeeds, one more iteration has to be performed (not depicted in figure 3.5): The abstracted symbolic states are branched at the loop header and four candidate contracts defining null-terminated lists with nested lists are returned. The states following the "if"-branch continue their symbolic execution and are again branched in the loop body reaching finally states with $P \equiv \mathsf{ls}(X, L_1) * L_1 \mapsto L_2 * L_1 + 8 \mapsto \text{null} * \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * x = L_2$ and $P \equiv \mathsf{ls}(X, L_1) * L_1 \mapsto L_2 * L_1 + 8 \mapsto L_3 * \mathsf{ls}(L_3, \text{null}) * \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * L_3 \neq \text{null} * x = L_2$. So we retrieve $2 \cdot 4 = 8$ states of that forms. When reaching the loop header we do invariant checks which succeed for all symbolic states. At this point the first phase for the analysis of `outer_loop` stops.

In figure 3.5 one can see that in the first round there are six candidate contracts created which require a second run (two after the first iteration and four after the second iteration). However contract $C_{nested} = (\mathsf{ls}_\Lambda(X, \text{null}) * X \neq \text{null} * x = X, \mathsf{ls}_\Lambda(X, \text{null}) * X \neq \text{null})$ with $\Lambda(a, b) \equiv a \mapsto b * a + 8 \mapsto c * \mathsf{ls}_{\Lambda'}(c, \text{null}) * \mathfrak{b}(a) = \mathfrak{b}(a + 8)$ and $\Lambda'(a, b) \equiv a \mapsto b$ covers each of the other five candidate contracts which require a second run. Therefore those contracts will be pruned and only contract C_{nested} will require the run of the second analysis phase.

The second run for contract C_{nested} is very similar to the first run. The only difference in terms of the enumeration tree will be that at the initial state we will not branch for `x!=NULL` because $x \neq \text{null}$ is given in C_{nested}'s pre-condition. So the second phase will

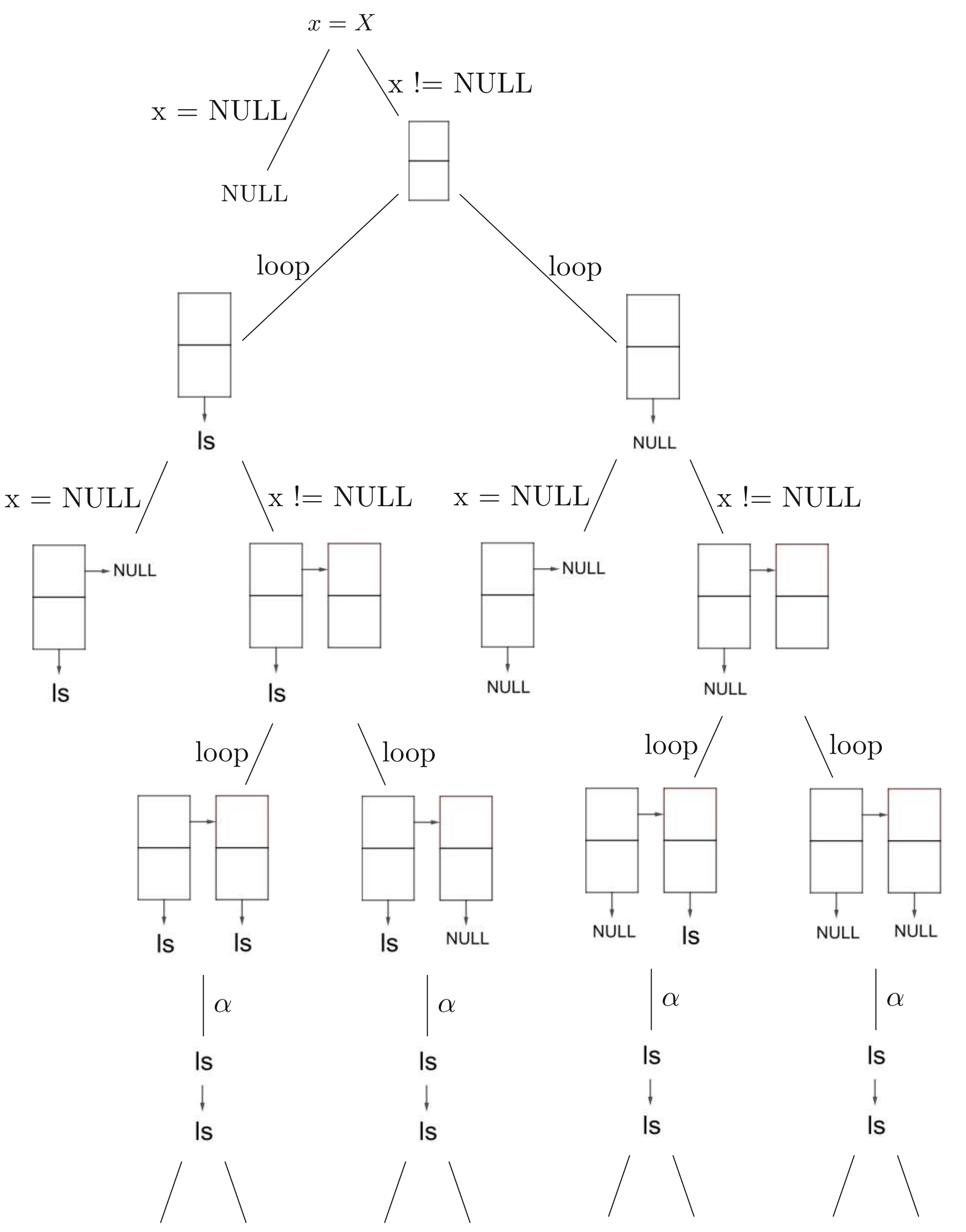

Figure 3.5: Computation tree for first analysis round of nested list traversal code (see figure 3.3) with contracts C_1, C_3'' for inner loop

yield the same six contracts as the first phase with the only difference that for all of them we have pre-condition $\mathsf{ls}_\Lambda(X, \text{null}) * X \neq \text{null} * x = X$. Therefore the pruning after the second phase will again give exactly one contract which is equivalent to C_{nested}.

Finally we obtain contracts $(X = \text{null} * x = X, X = \text{null})$ and C_{nested} for the `outer_loop` function. This is in fact the same number of contracts as given for the `loop` function. So the defined pruning strategy avoids an (exponential) blowup also w.r.t. the nesting depth of a function. While before our improvements even the analysis of the nested list example (of depth 1) was infeasible, Broom can now analyze a function traversing a list with nesting depth two within approximately four minutes. Details about running times are given in chapter 5.

3.3 Pruning of Intermediate States

In the previous section we described a strategy for pruning of (candidate) contracts (P, Q) of a whole *function*. Notice that this set of contracts is different from the mapping *symb* as defined in section 2.4. Most importantly for cut nodes $v \in V_{cut}$ in our CFG, we do the invariant checks w.r.t. $symb(v)$. More precisely, if symbolic execution executes v on (P, Q), we check if there is a state (P', Q') s.t. $(P, Q) \sqsubseteq (P', Q')$. If this succeeds, we discard (P, Q), otherwise we apply abstraction and execute v on the abstraction result.

For states $(P_1, Q_1), (P_2, Q_2) \in symb(v)$ we might have $(P_1, Q_1) \sqsubseteq (P_2, Q_2)$ just like for the final/candidate contracts of a function. Therefore it makes sense to apply a similar pruning strategy than the one for contracts: Whenever symbolic state (P, Q) reaches a cut point v we do not only check if there is a $(P', Q') \in symb(v)$ s.t. $(P, Q) \sqsubseteq (P', Q')$ (invariant check) but we also check if $(P', Q') \sqsubseteq (P, Q)$ and remove (P', Q') from $symb(v)$ if this holds. This might happen for instance if P or Q is the result of a successful abstraction and therefore describes lists of arbitrary length while P' or Q' describes a list of specific length or a list of special shape. In further iterations we can then avoid entailment checks w.r.t. (P', Q'). For the pruning of the symbolic states, we can also do a similar optimization as described in section 3.2: During the first analysis round we check entailment only w.r.t. to the pre-condition and during the second analysis round we check for equality of pre-conditions and entailment of post-conditions. Doing this optimization is sound because if we cannot detect a loop invariant in $symb(v)$ in a loop iteration i, we also cannot detect an invariant in a subset of $symb(v)$ for that iteration i, so symbolic execution will not terminate in an earlier iteration due to that optimization.

3.4 Optimizing Traversal of States for Invariant Checks

In the original version of Broom when doing the invariant checks for state (P, Q) Broom traversed $symb(v)$ in a FIFO order. This means we first check entailment for states (P', Q') which were added earlier in the respective analysis round. However the states which were added later are potentially more abstract than the states added before as we apply abstraction within every iteration. Therefore the likelihoods that $(P, Q) \sqsubseteq (P', Q')$

succeeds for those states is higher and we implemented a LIFO traversal of $symb(v)$ in order to reduce the number of failing entailment checks.

CHAPTER 4

Enhancing list abstraction

In this chapter we will describe how we extended the list (segment) predicates to handle lists with shared nodes (section 4.1). After that we describe the new abstraction procedure of Broom in detail. It searches for pairs of points-to predicates, pairs of a points-to and a list predicate and pairs of list predicates. In either case the respective spatial predicates are replaced by a list predicate. Since the procedure for the second and third case is built on top of the procedure for the first case, we first describe the case of abstracting two points-to predicates in section 4.2. The other cases are described in section 4.3.

Generally the abstraction procedure α is a function that takes as input a formula φ and returns a formula φ' s.t. $\varphi \models \varphi'$. However in our case we do not want to lose information about the current program variables $PVar = \{p_1, ..., p_k\}$ if they point to intermediate nodes. This is why we need to know $PVar$ in the abstraction procedure. So in fact $\alpha(\varphi, PVar)$ is a binary function.

4.1 New List (Segment) Predicate

Originally our list predicates were of form $\mathsf{ls}_{\Lambda(a,b)}(\varepsilon_1, \varepsilon_2)$ or $\mathsf{dls}_{\Lambda(a,b,c)}(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2')$ where $\Lambda(a,b)$ or $\Lambda(a,b,c)$ is of shape $\exists x_1, ..., \exists x_k.\varphi$ and φ is a quantifier-free symbolic heap with free variables a, b resp. a, b, c. This restriction prevents us from specifying non-global objects referenced by various list nodes. Following the approach of [BCC+07] we extend our logic to describe also this kind of objects by adding a parameter $\bar{\epsilon}$ to the parameters of the list predicates. $\bar{\epsilon} = [\bar{\varepsilon_1}, ..., \bar{\varepsilon_n}]$ is a list of expressions which define addresses referenced by all nodes of the respective list. To this end we call $\bar{\epsilon}$ the list of *shared expressions.*

With our new list predicates $ls_\Lambda(\varepsilon_1, \varepsilon_2, \bar{\epsilon})$ and $dls_\Lambda(\varepsilon_1, \varepsilon_2, \varepsilon_1', \varepsilon_2', \bar{\epsilon})$ there arises a need of extending the list segment predicates Λ as well. For each shared expression we add a new parameter s_i to Λ (with $i = 1, ..., n$).

The ordering of the shared expressions in the list predicate gives us the correspondence of the additional parameters of the lambda to the shared expressions. Therefore for ls predicates the list segment predicate is of form $\Lambda(a, b, s_1, ..., s_n)$ and for dls predicates the list segment predicate is of form $\Lambda(a, b, c, s_1, ..., s_n)$.

The semantics of the list predicates ls and dls are re-defined in the following way:

$$\begin{aligned}(S, B, M) \models \mathsf{ls}_{\Lambda(x,y,s_1,...,s_n)}(\varepsilon_1, \varepsilon_2, \bar{\epsilon}) \text{ iff } &(S, B, M) \models \varepsilon_1 = \varepsilon_2 \text{ or}\\ &(S, B, M) \models \varepsilon_1 \neq \varepsilon_2 * \mathit{true} \text{ and there is some } \ell \in \mathit{Loc}\\ &\text{and a fresh variable } u \in \mathit{Var} \text{ s.t.}\\ &(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u, \bar{\epsilon}) * \mathsf{ls}_{\Lambda(x,y,s_1,...,s_n)}(u, \varepsilon_2, \bar{\epsilon})\end{aligned}$$

$$\begin{aligned}&(S, B, M) \models \mathsf{dls}_{\Lambda(x,y,z,s_1,...,s_n)}(\varepsilon_1, \varepsilon_2, \varepsilon'_1, \varepsilon'_2, \bar{\epsilon}) \text{ iff}\\ &\quad(S, B, M) \models \varepsilon_1 = \varepsilon'_2 * \varepsilon_2 = \varepsilon'_1 * \mathit{true} \text{ or } (S, B, M) \models \varepsilon_1 \neq \varepsilon'_2 * \varepsilon_2 \neq \varepsilon'_1 * \mathit{true}\\ &\quad\text{and there is some } \ell \in \mathit{Loc} \text{ and a fresh variable } u \in \mathit{Var} \text{ s.t.}\\ &\quad(S[u \hookrightarrow \ell], B, M) \models \Lambda(\varepsilon_1, u, \varepsilon_2, \bar{\epsilon}) * \mathsf{dls}_{\Lambda(x,y,z,s_1,...,s_n)}(u, \varepsilon_1, \varepsilon'_1, \varepsilon'_2, \bar{\epsilon})\end{aligned}$$

Broom relies on a procedure for materialising nodes out of a list segment on several places. Materializing means handling the case where $(S, B, M) \models \varepsilon_1 \neq \varepsilon_2 * \mathit{true}$ or $(S, B, M) \models \varepsilon_1 \neq \varepsilon'_2 * \varepsilon_2 \neq \varepsilon'_1 * \mathit{true}$ respectively. The materialization is used for instance for the rules `slseg-pt-ls-left` and `slseg-pt-ls-right` but also in the abstraction process when trying to abstract a node and a list segment into a single list segment. With the new list (segement) predicates the materialization procedure was updated following the new semantics.

4.2 Abstraction of Points-to Predicates

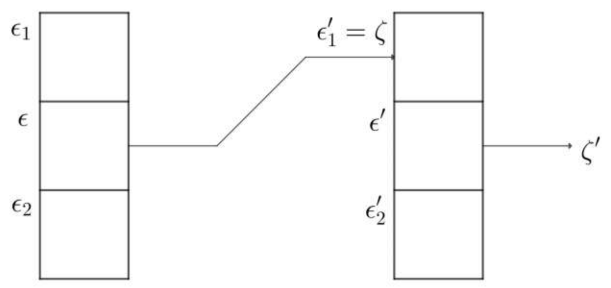

Figure 4.1: General setting for abstracting two points-to predicates

We will now give the procedure abstracting pairs of points-to predicates to list predicates. For a better understanding, the description is only semi-formal and contains figures for the different cases. Overall the abstraction procedure is divided into a pairing procedure, which pairs fields of blocks s.t. their offsets are equal and a checking procedure which checks if two fields "behave in a compatible way".

Pairing procedure

Given an input formula φ and a list of program variables $PVar$, the pairing procedure checks for *each* pair of points-to predicates $(\varepsilon \mapsto \zeta, \varepsilon' \mapsto \zeta')$ in φ if $\varepsilon, \varepsilon'$ can be linking fields for two nodes:

1. Check VALIDITY of $\mathsf{size}(\zeta) = \mathsf{size}(\zeta')$
2. Check VALIDITY of $\mathfrak{b}(\zeta) = \mathfrak{b}(\varepsilon')$
3. Check SATISFIABILITY of $\mathfrak{b}(\varepsilon) \neq \mathfrak{b}(\varepsilon')$
4. Check SATISFIABILITY of $\varepsilon - \mathfrak{b}(\varepsilon) = \varepsilon' - \mathfrak{b}(\varepsilon')$
5. Assume VALIDITY of 3. and 4.
6. Check SATISFIABILITY of $\mathfrak{b}(\varepsilon') \neq \mathfrak{b}(x)$ with $x \in PVar$
7. Try to pair all n points-to predicates defining the blocks located at $\mathfrak{b}(\varepsilon)$ and $\mathfrak{b}(\varepsilon')$ obtaining pairs of form $(\varepsilon_i \mapsto \zeta_i, \varepsilon'_i \mapsto \zeta'_i)$ with $\varepsilon_i = \mathfrak{b}(\varepsilon) + offset_i$ and $\varepsilon'_i = \mathfrak{b}(\varepsilon') + offset_i$ and $0 \leq offset_i \leq \mathfrak{e}(\varepsilon) - \mathfrak{b}(\varepsilon) = \mathfrak{e}(\varepsilon') - \mathfrak{b}(\varepsilon')$ for $i = 1, ..., n$. Exclude the linking fields ε and ε' because their compatibility is already checked.

   So in the underlying queries of the paring procedure we

   a) check SATISFIABILITY of $\varepsilon_i - \mathfrak{b}(\varepsilon) = \varepsilon'_i - \mathfrak{b}(\varepsilon')(= offset_i)$
   b) check SATISFIABILITY of $\mathsf{size}(\zeta_i) = \mathsf{size}(\zeta'_i)$

8. For $i = 1, ..., n$ call the checking procedure with $(\varepsilon_i \mapsto \zeta_i, \varepsilon'_i \mapsto \zeta'_i)$ which is defined below

The fact that in some queries we only check for SATISFIABILITY even tough it seems that VALIDITY is required comes from the necessity to handle intrusive lists and Linux lists: For these lists we have to deal with incomplete information because we only know $\mathfrak{b}(\varepsilon) \leq \varepsilon$ for a list node field ε.

Checking procedure

To describe the checking procedure we have to define the following set: Let $F(\varepsilon)$ be the set of forward links for expression ε. This is the set of points-to predicates of form $\varepsilon' \mapsto \_$ and list segment predicates of form $\mathsf{ls}(\varepsilon', \_, \_), \mathsf{dls}(\varepsilon', \_, \_, \_, \_)$ such that $\mathfrak{b}(\varepsilon') = \mathfrak{b}(\varepsilon)$.

We compute $F(\zeta_i)$ and $F(\zeta_i')$. To this end we check for each spatial predicate in φ if it is contained in one of these sets. Therefore we perform the VALIDITY queries as given in the definition of $F(\cdot)$.

For the sets $F(\zeta_i)$ and $F(\zeta_i')$ the checking procedure continues in the following cases:

- $F(\zeta_i) = F(\zeta_i') = \emptyset$

  Case: Data field

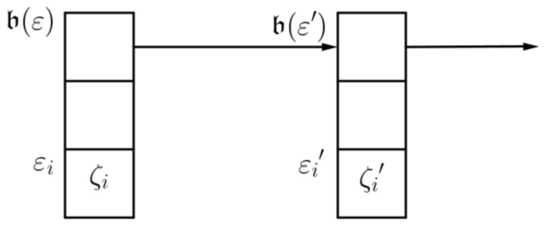

Figure 4.2: Case $F(\zeta_i) = F(\zeta_i') = \emptyset$: Data field

The constraint $F(\zeta_i) = F(\zeta_i') = \emptyset$ mans that ε_i and ε_i' are not fields pointing to an allocated block but are both data fields storing numeric constants ζ_i, ζ_i' (which can have value $\top$). Let a be the first parameter of the lambda. If checking VALIDITY of $\zeta_i = \zeta_i'$ succeeds, we add to the respective (partial) lambda $a + offset_i \mapsto \zeta_i$. Otherwise $a + offset_i \mapsto \top$ is added.

- $F(\zeta_i) \neq \emptyset$, $F(\zeta_i') \neq \emptyset$

  This constraint itself does not imply compatible behaviour, so sub-cases need to be distinguished.

  Sub-case: Field pointing to node itself

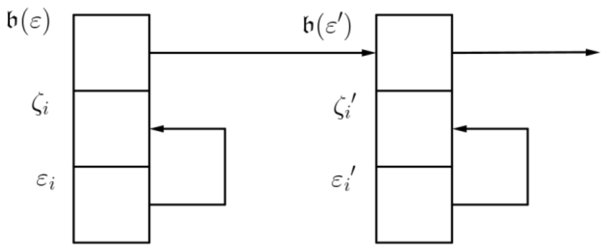

Figure 4.3: Sub-case of $F(\zeta_i) \neq \emptyset, F(\zeta_i') \neq \emptyset$: Pointers point "back" to same node

If the following checks succeed, we know that fields ε_i and ε_i' point "back" to the blocks located at $\mathfrak{b}(\varepsilon)$ resp. $\mathfrak{b}(\varepsilon')$.

1. Check VALIDITY of $\mathfrak{b}(\zeta_i) = \mathfrak{b}(\varepsilon)$
2. Check VALIDITY of $\mathfrak{b}(\zeta_i') = \mathfrak{b}(\varepsilon')$
3. Check SATISFIABILITY of $\zeta_i - \mathfrak{b}(\zeta_i) = \zeta_i' - \mathfrak{b}(\zeta_i')$

No we can add to the respective (partial) lambda $a + offset_i \mapsto a + (\zeta_i - \mathfrak{b}(\zeta_i))$.

Sub-case: Field points to shared object

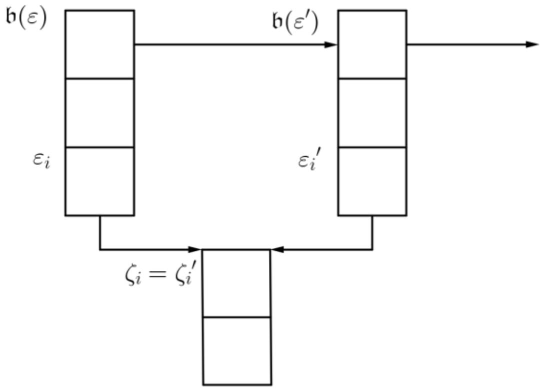

Figure 4.4: Sub-case of $F(\zeta_i) \neq \emptyset, F(\zeta_i') \neq \emptyset$: Pointers pointing to a shared node

To detect the case that ε_i and ε_i' point to the same shared allocated block ζ_i, our enhanced abstraction checks

1. VALIDITY of $\zeta_i = \zeta_i'$

We can now add a fresh variable u to the respective (partial) lambda's parameters together with the constraint $a + offset_i \mapsto u$. Additionally we need to add ζ_i to $\bar{\epsilon}$, which is the list of shared expressions for the list segment predicate which will be finally obtained after the folding.

Sub-case: Backward field of doubly-linked list segment

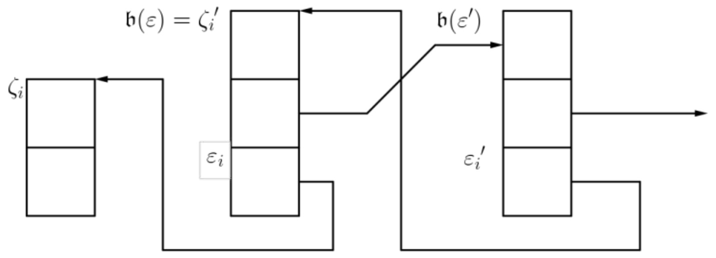
$\mathfrak{b}(\varepsilon) = \zeta_i'$
$\mathfrak{b}(\varepsilon')$
ζ_i
ε_i
ε_i'

Figure 4.5: Sub-case of $F(\zeta_i) \neq \emptyset, F(\zeta_i') \neq \emptyset$: Doubly-linked list segment

If the following checks succeed, we can conclude that ε_i' points to an address ζ_i' with base $\mathfrak{b}(\varepsilon)$, which means that ε_i' is the back link field of a doubly-linked list node.

1. Check SATISFIABILITY of $\mathfrak{b}(\zeta_i) \neq \mathfrak{b}(\zeta_i')$
2. Check SATISFIABILITY of $\zeta_i - \mathfrak{b}(\zeta_i) = \zeta_i' - \mathfrak{b}(\zeta_i')$
3. Check VALIDITY of $\mathfrak{b}(\zeta_i') = \mathfrak{b}(\varepsilon)$
4. Check SATISFIABILITY of $\mathfrak{b}(\zeta_i) \neq \mathfrak{b}(\varepsilon)$

Therefore we have to fold at the end of the abstraction procedure into a dls predicate instead of a ls predicate. Additionally we have to add a parameter c to our (partial) lambda together with the constraint $a + offset_i \mapsto c + (\zeta_i - \mathfrak{b}(\zeta_i))$. Notice that we do not check compatibility with the block located at $\mathfrak{b}(\zeta_i)$. This is fine because at the end of the abstraction procedure ζ_i will be passed to the second parameter of dls and when materializing nodes out of a doubly-linked list we do not instantiate the second parameter following the semantics of dls as described in section 4.1.

Sub-case: Field pointing to a nested object

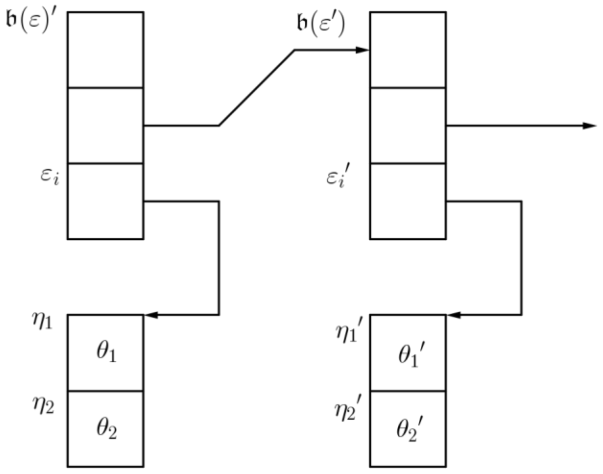

Figure 4.6: Sub-case of $F(\zeta_i) \neq \emptyset, F(\zeta_i') \neq \emptyset$: Nested node

To check if we ε_i and ε_i' point to compatible nested structures, we first need to do the following base checks:

1. Check SATISFIABILITY of $\mathfrak{b}(\zeta_i) \neq \mathfrak{b}(\zeta_i')$
2. Check SATISFIABILITY of $\zeta_i - \mathfrak{b}(\zeta_i) = \zeta_i' - \mathfrak{b}(\zeta_i')$

Unfortunately there are different shapes of nested objects which need different treatment. Let $P_i \in F(\zeta_i)$ and $P_i' \in F(\zeta_i')$ (here it does not matter which element from the sets you choose).

Depending on the types of P_i and P_i', different checks need to be performed:

- $P_i = \eta \mapsto \theta$ and $P'_i = \eta' \mapsto \theta'$
  1. Check SATISFIABILITY of $\mathfrak{b}(\eta) \neq \mathfrak{b}(x)$ with $x \in PVar$
  2. Check SATISFIABILITY of $\mathfrak{b}(\eta') \neq \mathfrak{b}(x)$ with $x \in PVar$

  Now we do the same pairing as in the pairing procedure's point 7. However instead of ε and ε', we use η and η' and we also include the points-to predicates $\eta \mapsto \theta$ and $\eta' \mapsto \theta'$. So if the pairing process succeeds for all m fields, we will be given pairs $(\eta_l \mapsto \theta_l, \eta'_l \mapsto \theta'_l)$ for all $l = 1, ..., m$. Then we call the checking procedure recursively.

- $P_i = ls_\Lambda(\eta_1, \eta_2, \bar{\epsilon})$ and $P'_i = ls_{\Lambda'}(\eta'_1, \eta'_2, \bar{\epsilon}')$

  To check if the two nested lists $ls_\Lambda(\eta_1, \eta_2, \bar{\epsilon})$ and $ls_{\Lambda'}(\eta'_1, \eta'_2, \bar{\epsilon}')$ are compatible we first check if Λ and Λ' are compatible. If $\Lambda \models \Lambda'$, we proceed with $\Lambda_{new} = \Lambda'$ and if $\Lambda' \models \Lambda$, we proceed with $\Lambda_{new} = \Lambda$. Otherwise we fail.

  Then we have to
  1. Check SATISFIABILITY of $\mathfrak{b}(\eta_1) \neq \mathfrak{b}(x)$ with $x \in PVar$
  2. Check SATISFIABILITY of $\mathfrak{b}(\eta'_1) \neq \mathfrak{b}(x)$ with $x \in PVar$
  3. Check that there is a one-to-one correspondence between elements $\bar{\varepsilon} \in \bar{\epsilon}$ and $\bar{\varepsilon}' \in \bar{\epsilon}'$ s.t. checking VALIDITY of $\bar{\varepsilon} = \bar{\varepsilon}'$ succeeds.

  Now we consider two cases:

  * $F(\eta_2) = F(\eta'_2) = \emptyset$
    This means the last elements of the nested list are not allocated blocks but constants (for e.g. null-terminated lists). Therefore we check VALIDITY of $\eta_2 = \eta'_2$.
    If this succeeds, we return (partial) lambda $a{+}offset_i \mapsto u{*}ls_{\Lambda_{new}}(u, \eta_2, \bar{\epsilon})$ for a fresh variable u. Otherwise we return $a{+}offset_i \mapsto u{*}ls_{\Lambda_{new}}(u, \top, \bar{\epsilon})$
  * $F(\eta_2) \neq \emptyset, F(\eta'_2) \neq \emptyset$
    This means η_2 and η'_2 are allocated blocks. Therefore we check if the lists are circular by
    1. Checking VALIDITY of $\mathfrak{b}(\eta_1) = \mathfrak{b}(\eta_2)$
    2. Checking VALIDITY of $\mathfrak{b}(\eta'_1) = \mathfrak{b}(\eta'_2)$

    Finally the (partial) lambda $a + offset_i \mapsto u * ls_{\Lambda_{new}}(u, u, \bar{\epsilon})$ is returned

- $P_i = dls_\Lambda(\eta_1, \eta_2, \eta_3, \eta_4, \bar{\epsilon})$ and $P'_i = dls_{\Lambda'}(\eta'_1, \eta'_2\eta'_3, \eta'_4, \bar{\epsilon}')$

  This case is very similar to the previous case with $P_i = ls_\Lambda(\eta_1, \eta_2, \bar{\epsilon})$ and $P'_i = ls_{\Lambda'}(\eta'_1, \eta'_2, \bar{\epsilon}')$. There are just some additional checks ensuring compatibility of both endpoints. From a conceptual point of view, describing this in detail is not very interesting. So we do not explain this here in further detail.

• $F(\zeta_i) = \emptyset, F(\zeta'_i) = \{ls_\Lambda(\eta_1, \eta_2, \_)\}$ or
  $F(\zeta_i) = \{ls_\Lambda(\eta_1, \eta_2, \_)\}, F(\zeta'_i) = \emptyset$

Case: Field pointing to possibly empty nested list

This case corresponds to the two states in the middle at the bottom of the tree in figure 3.5, where one nested list is empty and the other list is non-empty. Assume w.l.o.g. $F(\zeta_i) = \emptyset, F(\zeta_i') = \{ls_\Lambda(\eta_1, \eta_2, \_)\}$.

We have to

1. Check VALIDITY of $\zeta_i = \text{null}$

In this case our abstraction assumes ζ_i to be an empty list with the same shape as the nested list of ζ_i'. This does not compromise soundness because $ls_\Lambda(\text{null}, \text{null}, \_)$ is valid and we only make the state more abstract. By our assumption we return the (partial) lambda $a + offset_i \mapsto u * \mathsf{ls}_{\Lambda_{0+}}(u, \eta_2)$. The definition of Λ_{0+} is given in section 4.4.

When looking at figure 3.5 one can see why this case needs to be considered within abstraction for the nested list example. If we only merge pairs of nested points-to predicates and pairs of nested list predicates for $F(\zeta_i)$ and $F(\zeta_i')$, we cannot abstract the two states in the middle. So these states would be expanded further giving infinitely long lists where nodes with empty and non-empty nested lists alternate. This would prevent Broom from terminating for the nested list example.

After the procedure finished, we need to remove from φ all predicates associated to nodes $\mathfrak{b}(\varepsilon_1)$ and $\mathfrak{b}(\varepsilon')$, which are the nodes traversed by the checking procedure. We replace these predicates by $\mathsf{ls}_\Lambda(\mathfrak{b}(\varepsilon), \zeta', \bar{\epsilon})$ or $\mathsf{dls}_\Lambda(\mathfrak{b}(\varepsilon), \varepsilon_b, \mathfrak{b}(\varepsilon'), \zeta', \bar{\epsilon})$ where Λ is the list segment predicate returned by the checking procedure and ε_b is the back link of node $\mathfrak{b}(\varepsilon)$ which was discovered in the checking procedure.

4.3 Abstraction of Other Spatial Predicates

Within symbolic execution it might be necessary to abstract states whose pre- or post-conditions define heaps of form $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2, \bar{\epsilon}) * \varepsilon_2 \mapsto \varepsilon_3$ due to e.g. some global variable preventing abstraction in a former iteration. There might even arise cases where $\mathsf{ls}_{\Lambda_1}(\varepsilon_1, \varepsilon_2, \bar{\epsilon_1}) * \mathsf{ls}_{\Lambda_2}(\varepsilon_2, \varepsilon_3, \bar{\epsilon_2})$ needs to be abstracted. The same holds of course for dls predicates but we focus here for simplicity on ls predicates because the described procedure can be easily extended for dls predicates.

- Abstraction of $\varepsilon_1 \mapsto \varepsilon_2$ and $\mathsf{ls}_\Lambda(\varepsilon_1', \varepsilon_2', \bar{\epsilon})$ in formula φ

  1. Check VALIDITY of $\mathfrak{b}(\epsilon_2) = \mathfrak{b}(\varepsilon_1')$
  2. Check SATISFIABILITY of $\mathfrak{b}(\varepsilon_2) \neq \mathfrak{b}(x)$ with $x \in PVar$

  If the checks succeed, we materialize one node out of $\mathsf{ls}_\Lambda(\varepsilon_1', \varepsilon_2', \bar{\epsilon})$ obtaining a formula φ' where $\mathsf{ls}_\Lambda(\varepsilon_1', \varepsilon_2', \bar{\epsilon})$ is replaced by $\Lambda(\varepsilon_1', u, \bar{\varepsilon_1}, ...\bar{\varepsilon_n})$ for a fresh variable u. Then we pair points-to predicates like in our pairing procedure in section 4.2.

However we use as linking fields ε_1 and the linking field in $\Lambda(\varepsilon_1', u, \bar{\varepsilon_1}, ...\bar{\varepsilon_n})$. On success, this will give us pairs $(\varepsilon_i \mapsto \zeta_i, \varepsilon_i' \mapsto \zeta_i')$ as defined on page 25. So we can call the checking procedure which will succeed only if the node located at $\mathfrak{b}(\varepsilon_1)$ is compatible with Λ. In the end we erase from φ' all spatial predicates which were traversed by the checking procedure and are therefore associated to $\mathfrak{b}(\varepsilon_1)$. Additionally we replace $\Lambda(\varepsilon_1', u, \bar{\varepsilon_1}, ...\bar{\varepsilon_n})$ by $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2', \bar{\epsilon})$ and return the resulting formula.

- Abstraction of $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2, \bar{\epsilon})$ and $\varepsilon_1' \mapsto \varepsilon_2'$ in formula φ

  As this case can be treated analogously to the former case (just the names of the variables differ), we do not explain it here.

- Abstraction of $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2, \bar{\epsilon})$ and $\mathsf{ls}_{\Lambda'}(\varepsilon_1', \varepsilon_2', \bar{\epsilon}')$ in formula φ

  This case is different from the other cases but can be handled very easily:

  1. Check VALIDITY of $\varepsilon_2 = \varepsilon_1'$
  2. Check SATISFIABILITY of $\mathfrak{b}(\varepsilon_2) \neq \mathfrak{b}(x)$ with $x \in PVar$
  3. Check that there is a one-to-one correspondence between elements $\bar{\varepsilon} \in \bar{\epsilon}$ and $\bar{\varepsilon}' \in \bar{\epsilon}'$ s.t. checking VALIDITY of $\bar{\varepsilon} = \bar{\varepsilon}'$ succeeds.

  If $\Lambda \models \Lambda'$, we return $\mathsf{ls}_{\Lambda'}(\varepsilon_1, \varepsilon_2', \bar{\epsilon})$ and if $\Lambda' \models \Lambda$ we return $\mathsf{ls}_\Lambda(\varepsilon_1, \varepsilon_2', \bar{\epsilon})$. Otherwise abstraction fails.

4.4 Universal List Segment Predicate for Nested Lists

In figure 3.5 we did not make the parameters of the ls predicates explicit. In this section we will do that because this shows another interesting issue which needed to be solved in order to allow Broom analyze the nested list example. In the first analysis phase when abstraction succeeds, we would get in fact three different list segment predicates.

According to our abstraction procedure the first symbolic state (from the left) with $P \equiv Q \equiv X \mapsto L_1 * X + 8 \mapsto L_2 * \mathfrak{b}(X) = \mathfrak{b}(X+8) * \mathsf{ls}_{\Lambda_n}(L_2, \text{null}) * L_2 \neq \text{null} * L_1 \mapsto L_3 * L_1 + 8 \mapsto L_4 * \mathsf{ls}_{\Lambda_n}(L_4, \text{null}) * L_4 \neq \text{null} * \mathfrak{b}(L_1) = \mathfrak{b}(L_1 + 8) * x = L_3$ will be abstracted to $\mathsf{ls}_{\Lambda_{1+}}(X, L_3) * x = L_3$ where $\Lambda_n(a, b) \equiv a \mapsto b$ and $\Lambda_{1+}(a, b) \equiv a \mapsto b * a + 8 \mapsto c * \mathsf{ls}_{\Lambda_n}(c, \text{null}) * c \neq \text{null} * \mathfrak{b}(a) = \mathfrak{b}(a+8)$. The abstraction result obtained from the last state in that iteration (the rightmost state) with $P \equiv Q \equiv X \mapsto L_1 * X + 8 \mapsto \text{null} * \mathfrak{b}(X) = \mathfrak{b}(X+8) * L_1 \mapsto L_3 * L_1 + 8 \mapsto \text{null} * \mathfrak{b}(L_1) = \mathfrak{b}(L_1+8) * x = L_3$ is given by $\mathsf{ls}_{\Lambda_0}(X, L_3) * x = L_3$ where $\Lambda_0(a, b) \equiv a \mapsto b * a + 8 \mapsto \text{null} * \mathfrak{b}(a) = \mathfrak{b}(a+8)$. To obtain a sound abstraction procedure when abstracting the other two states, we can neither use Λ_0 nor Λ_{1+} because the obtained list is neither always empty nor always non-empty. So we need to find some list segment predicate which is more abstract than both Λ_0 and Λ_{1+}. To this end we define $\Lambda_{0+}(a, b) \equiv a \mapsto b * a + 8 \mapsto c * \mathsf{ls}(c, \text{null}) * \mathfrak{b}(a) = \mathfrak{b}(a+8)$ which will be the list segment predicate used for these two states as depicted in figure

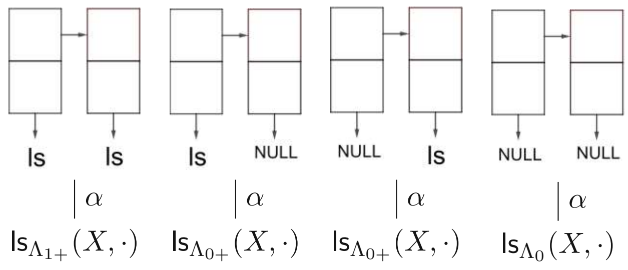
ls
ls
ls
NULL
NULL
ls
NULL
NULL
α
α
α
α
$\mathsf{ls}_{\Lambda_{1+}}(X, \cdot)$
$\mathsf{ls}_{\Lambda_{0+}}(X, \cdot)$
$\mathsf{ls}_{\Lambda_{0+}}(X, \cdot)$
$\mathsf{ls}_{\Lambda_{0}}(X, \cdot)$

Figure 4.7: Abstraction of the states in figure 3.5 with list segment predicates

4.7. In fact Λ_{0+} is obtained by removing constraint $c \neq \text{null}$ from Λ_{1+} and describes a lists whose nested lists *can* be empty.

The existence of various list segment predicates itself is not problematic because Broom has a procedure to determine whether $\Lambda \models \Lambda'$ for some list segment predicates Λ, Λ' used within the entailment checks and abstraction procedure. In fact the introduction of Λ_{0+} is necessary to get a succeeding invariant check for all computation branches for the nested list example. In the loop iteration after abstraction success we would have originally had symbolic states with $P \equiv Q \equiv \mathsf{ls}_{\Lambda}(X, L_3) * L_3 \mapsto L_5 * L_3 + 8 \mapsto \text{null} * \mathfrak{b}(L_3) = \mathfrak{b}(L_3 + 8) * x = L_5$ and symbolic states with $P \equiv Q \equiv \mathsf{ls}_{\Lambda}(X, L_3) * L_3 \mapsto L_5 * L_3 + 8 \mapsto L_6 * \mathsf{ls}(L_6, \text{null}) * L_6 \neq \text{null} * \mathfrak{b}(L_3) = \mathfrak{b}(L_3 + 8) * x = L_5$ where $\Lambda \in \{\Lambda_0, \Lambda_{1+}\}$. Consider for instance state (P, Q) with $P \equiv Q \equiv \mathsf{ls}_{\Lambda_{1+}}(X, L_3) * L_3 \mapsto L_5 * L_3 + 8 \mapsto \text{null} * \mathfrak{b}(L_3) = \mathfrak{b}(L_3 + 8) * x = L_5$: We neither have $(\mathsf{ls}_{\Lambda_0}(X, L_3) * x = L_3, \mathsf{ls}_{\Lambda_0}(X, L_3) * x = L_3) \sqsubseteq (P, Q)$ nor $(\mathsf{ls}_{\Lambda_{1+}}(X, L_3) * x = L_3, \mathsf{ls}_{\Lambda_{1+}}(X, L_3) * x = L_3) \sqsubseteq (P, Q)$.

4.5 Changes for Abduction

By adding an additional parameter to the list predicates there arises the need to change some abduction rules as well. However this only requires small changes for abduction rules which create new list predicates. For the `slseg-ls-ls`-rule and the analogous version for the dls predicate we need to add to the LHS the constraint that the list of shared expressions for the matched lists are equal. Therefore the `slseg-ls-ls`-rule is re-defined as

$$\texttt{slseg-ls-ls}\ \frac{\varphi * \bar{\varepsilon_1} = \bar{\varepsilon_1}' * \ldots * \bar{\varepsilon_n} = \bar{\varepsilon_n}' * [M] \ \triangleright\ \psi * ls_{\Lambda'}(\zeta, \zeta', [\bar{\varepsilon_1}, \ldots, \bar{\varepsilon_n}])}{\varphi * ls_{\Lambda}(\varepsilon, \zeta, [\bar{\varepsilon_1}, \ldots, \bar{\varepsilon_n}]) * [M] \ \triangleright\ \psi * ls_{\Lambda'}(\varepsilon', \zeta', [\bar{\varepsilon_1}', \ldots, \bar{\varepsilon_n}'])}\ C$$

where C is the constraint $\varphi \models \varepsilon = \varepsilon' * \mathit{true}$ and $\Lambda \models \Lambda'$.

The abduction rules `slseg-pt-ls-right` and `slseg-pt-ls-left` (as well as their analogous version for dls predicates) rely on materialization. So they change implicitly as the materialization procedure changes as described in section 4.1.

CHAPTER 5

Experiments

In this section we will present some C-code examples which can be analysed by Broom due to the changes described in this thesis. You can find a summary about the experiments in table 5.1. All of these benchmark instances except for the files nested.c and nested-2.c come from the Predator benchmark suite and can be found in [1] . The tests were run on a machine with an Intel i7-4770 processor with 32 GiB of memory. We invoked Broom with option –entailment-limit=7 which means that symbolic execution for loops aborts after seven loop unfoldings. In the following we will provide some details about the test files.

Name	#Fncs	Time [s]	Infer
test-0227.c	2	31	✓
test-0056.c	4	146	✓
test-0233.c	2	34	✗
sll-headptr.c	1	79	✓
test-0192.c	4	277	✓
nested.c	2	254	✓
nested-2.c	3	570	✓

Table 5.1: Test files which can be verified by Broom thanks to the changes implemented, their number of functions, the running time in seconds which Broom needed for the analysis and an indication whether the Infer analyser was able to generate correct contracts

In the test file test-0227.c a singly-linked list of length at least two is allocated. Every node points to a shared data element.

The file test-0056.c is similar but allocates a doubly-linked list of arbitrary length where each node points to a shared data element.

[1] `https://pajda.fit.vutbr.cz/rogalew/broom/-/tree/master/tests/new`

The test file test-0233.c is also similar but it allocates a non-empty singly-linked list where each node points to a special shared list node which is allocated when the first node is allocated.

In the test file sll-headptr.c we allocate a singly-linked list in which every node has a pointer to the list head. Once allocated, we just iterate over the list. Broom constructs formulas of shape $\mathsf{ls}_\Lambda(h, \text{null}, [h])$ where h represents the shared head node and $\Lambda(a, b, c) \equiv a \mapsto b * a + 8 \mapsto c$. The allocated list is depicted in figure 5.1. Here one can see that shared nodes may also be part of the list itself.

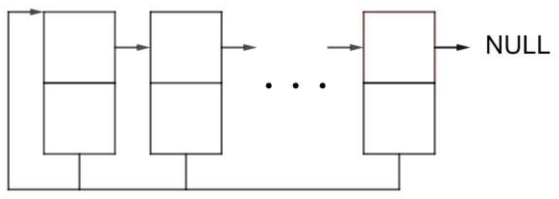

Figure 5.1: Shape of the list allocated in test file sll-headptr.c which is abstracted to a list with the list head as shared node

In test-0192.c we can see an example of a list which can be analyzed thanks to the pruning of contracts. The main function creates a list of length three which has nested lists of arbitrary length. In the original version of Broom for the function creating the nested lists (`create_hlist`) we would have nine contracts after the second analysis round. The function `insert_vlist_node` which calls `create_hlist` would then have 18 contracts because it contains one if-statement. As `create_vlist` calls `insert_vlist_node` three times, we would have $18^3 = 5832$ contracts computed for the `create_vlist` function which is infeasible. If we apply pruning, `create_vlist` has a total of nine contracts describing all $2^3 = 8$ combinations of empty or non-empty nested lists plus one extra contract where the boolean variable `real_nested_list` is set to false.

The example shows that the state space explosion problem does not only appear for programs with nested loops. In general the more complex a program and the deeper its function call tree, the more contracts will be created. Therefore we expect that the implemented optimizations will become relevant for more benchmark instances once Broom supports more complex lists.

Before implementing the optimizations Broom was unable to verify these test files. In the case of nested.c and nested-2.c, Broom could not terminate because of its inability to abstract nested lists with a specific shape (see section 4.4). Here the states which were generated up until the seventh loop iteration got very complex which is why Broom needed 2637 s (approx. 44 min) to abort for nested.c. In the case of the test files related to shared nodes, abstraction could not succeed either in any loop iteration. However as here the states did not get as complex as for the experiments with nested.c, analysis aborted quite fast. For instance the analysis for test-0227.c aborted after 79 s.

Concerning the running time it is clear that an increase in the nesting depth of a list increases the running time. However as one can see in table 5.1, the increase is not exponential. For a simple loop traversal as depicted in figure 3.1 the analysis only takes seconds. For the analysis of nested.c we need 254 s while for the analysis of the outermost loop of nested-2.c (itself) we need (570 - 254) = 316 s which is not much longer than the analysis of the nested loop (of depth 1). This comes from the fact that for all of these functions we only obtain two contracts after the pruning. Considering that in the initial version of Broom even the analysis of nested.c was infeasible, we see the efficacy of the pruning strategies defined.

Concerning the pruning of intermediate states of the mapping *symb* as described in section 3.3, the experiments for nested.c show that the optimization reduces the maximum cardinality of $symb(v)$ from 11 to 7 where v is the cut point in function `outer_loop` as defined in figure 3.3. As this is only a small decrease and the pruning of $symb(\cdot)$ comes at the cost of performing additional entailment checks, the experiments could not show an improvement in running time. The pruning mostly pays off if $symb(\cdot)$ contains a lot of elements which is not yet the case for our still quite simple examples. However as we can reuse a lot of code already implemented for the pruning of contracts and we have to expect that the mapping *symb* grows once Broom can analyse more complex code, we decided to keep that optimization.

Finally we also invoked the Infer analyser (version 1.1.0) on our benchmark instances. The list predicates used by Infer also contain a list with references to shared allocated blocks. Therefore it is not surprising that Infer is able to generate contracts which are similar to the contracts generated by Broom for most of our examples. The only example for which Infer could not generate contracts is test-0233.c. The output of Infer shows that the analysis fails in the loop which destroys the allocated list when applying Infer's join operator.

CHAPTER 6

Conclusion

The goal of the thesis was to improve the capabilities of the static analyser Broom. This tool creates for each C-function a set of contracts in a Separation Logic with inductive list predicates if the function is memory safe.

We identified the state space explosion problem as the main reason why Broom's analysis is very time-consuming for programs whose functions have many contracts and programs with a deep function call tree. Therefore we implemented a pruning strategy reducing the number of contracts after the first and second analysis phase and showed theoretically and experimentally that this can drastically decrease the running time. In fact the state space explosion originally prevented Broom from analysing nested lists, because Broom would have needed to compute over 400 contracts. After applying the pruning we only obtain two contracts for functions traversing nested lists of any depth. This results in an almost linear increase of running time w.r.t. nesting depth of the list. Additionally we could reduce the number of entailment checks performed within pruning of contracts and invariant checking: It turned out that it suffices to only check entailment between the pre-conditions in the first analysis phase resp. between the post-conditions in the second analysis phase.

Furthermore we extended the list predicates used in our Separation Logic such that Broom can now also analyse lists with pointers to shared non-global heap objects. To this end we also needed to add a new sub-case into the abstraction procedure. As our experiments show, these lists also include lists where each node has a pointer to dedicated (head) nodes.

During our analysis of the nested list example we also encountered an interesting issue which prevented Broom from successfully analysing the nested list examples and is related to abstraction: If the first node has an empty and the second node has a non-empty nested list, we need to abstract this to a list whose nested lists are *possibly* empty. In this case in fact we applied an early version of a join operator. This is a well-known operator from static analysis tools (e.g. [DPV13]) which returns for each pair of states S_1, S_2 a state S s.t. $S \sqsubseteq S_1$ and $S \sqsubseteq S_2$. Implementing this operator in future work may not only be required for termination of some examples but would also further reduce the number of contracts created by Broom. For example the function traversing a list as depicted in figure 3.1 could be described with the single contract $(\mathsf{ls}(X, \mathrm{null}) * x = X, \mathsf{ls}(X, \mathrm{null}))$ but Broom currently returns two contracts (one for the empty and one for the non-empty list).

Another problem which shall be resolved in future work is the traversal of the Linux lists: As currently the abstraction procedure requires a one-to-one correspondence between all fields of the first and the second node, abstraction fails for Linux lists because the block boundaries are unknown. We plan to resolve this by defining separate "fields" for the padding at the beginning and end of a block ε. For these padding fields the size parameter can be parametric to $\mathfrak{b}(\varepsilon)$ and $\mathfrak{e}(\varepsilon)$. Finally adding support for numerical abstractions in future work would make Broom applicable for more real-world benchmark instances.

Bibliography

[BCC$^+$07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

[CD11] C. Calcagno and D. Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proc. of NFM'11*, volume 6617 of *LNCS*. Springer, 2011.

[CDOY11] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6), 2011.

[DPV13] K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

[HPR$^+$22] L. Holík, P. Peringer, A. Rogalewicz, V. Šoková, T. Vojnar, and F. Zuleger. Low-Level Bi-Abduction. Technical report, VUT Brno, https://arxiv.org/abs/2205.02590, 2022.