Informatics

Automated Reasoning over Arrays in the Superposition Calculus

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Christoph Hochrainer, BSc.
Matrikelnummer 01429786

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.techn. Laura Kovács, MSc
Mitwirkung: DI Pamina Georgiou, BSc

Wien, 1. Mai 2020

Christoph Hochrainer   Laura Kovács

Informatics

Automated Reasoning over Arrays in the Superposition Calculus

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Christoph Hochrainer, BSc.
Registration Number 01429786

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.techn. Laura Kovács, MSc
Assistance: DI Pamina Georgiou, BSc

Vienna, 1st May, 2020

Christoph Hochrainer	Laura Kovács

Erklärung zur Verfassung der Arbeit

Christoph Hochrainer, 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, 1. Mai 2020

Christoph Hochrainer

Danksagung

In erster Linie möchte ich Univ.Prof. Dr.techn. Laura Kovács meinen aufrichtigsten Dank aussprechen für das Ermöglichen und Betreuen dieser Masterarbeit.

An dieser Stelle möchte ich mich auch bei Dipl.-Ing. Pamina Georgiou und Dipl.-Ing. Bernhard Gleiss bedanken für die anfängliche Einführung in das Rapid-Framework und die nötige Unterstützung mich in die Thematik einzuarbeiten.

Mein Dank gilt außerdem allen, die mich im Prozess der Erstellung dieser Arbeit unterstützt, motiviert und begleitet haben.

Zuletzt möchte ich besonderen Dank an meine Familie aussprechen, die mir im Laufe meines Studiums immer beiseitegestanden ist und ohne die ich es nicht soweit geschafft hätte.

Kurzfassung

In den letzen Jahren gab es enorme Fortschritte im Bereich automatisierter Softwareverifikation und Programanalyse. Dennoch wurden einige Herausforderungen noch nicht vollständig bezwungen, sowie das automatisierte Verifizieren von unbeschrenkten Datenstrukturen, insbesondere Arrays, und Programme, die diese manipulieren. In dieser Arbeiter konzentrieren wir uns auf die Programmanalyse und -verifizierung in vollständiger Logik erster Ordnung.

Diese Arbeit soll theoretische Grundlagen für die Automatisierung von Beweisen zu partieller Korrektheit für Eigenschaften über ganzzahlige Arrays in dem Superposition Calculus liefern. Besonders interessieren uns hierbei Eigenschaften mit verschiedenen Quantifizierungen: $\exists$, $\forall$ und $\forall\exists$. Jede der ausgewählten Eigenschaften beschreibt bestimmte Programmverhaltensweisen, welche häufig in der Praxis anzutreffen sind und es ist daher wichtig diese zu beweisen.

Als Grundlage für diese Arbeit verwenden wir die erst kürzlich vorgestellte *Trace-Logik*, eine Instanz der Logik erster Ordnung, und das Framework Rapid, welches die logische Codierung der Programme liefert. Durch Verwendung einer generischen Eigenschaft $\mathtt{P}$, die eine Eigenschaft über Arrays darstellt, wurden die notwendigen Lemma und die Axiomatisierung der *Trace-Logik* verallgemeinert und es konnten allgemeinere Beweise erstellt werden. Diese können nun verwendet werden, um Beweis für eine konkretisierte Eigenschaft zu instanziieren. Mit den durch die Beweise gewonnenen Erkenntnissen sind wir nun in der Lage automatisierte Beweise der Eigenschaften in allgemeiner oder konkretisierter Form, mit Hilfe des Superposition-basiertem Theorembeweiser Vampire, durchzuführen.

Abstract

Automated reasoning for program verification has made a lot of advancements in the past years. Nevertheless, one of the remaining challenges is automated reasoning over unbound data structures, in particular arrays, and programs manipulating them. In this thesis we will focus on program analysis and verification in full first-order logic.

This work aims to provide theoretical basis to automate partial correctness proofs for software properties over integer arrays in the superposition calculus. Particularly, we are interested in properties of different quantifier settings: $\exists$, $\forall$ and $\forall\exists$. Each of the chosen properties describe specific program behaviors that occur often in practice and are therefore important to prove.

As foundation for this work we use the recently introduced *trace logic*, an instance of first-order logic and the framework Rapid, which provides the logical encoding of the programs. By using a generic property P, representing some property over arrays, we were able to generalize necessary lemmas and axiomatization of *trace logic* to produce more general proofs. These can be used to quickly instantiate a proof for a concrete property. With the insights gained throughout this work, we are now able to automate the proofs of the specific properties in general and concrete form, using the superposition-based theorem prover Vampire.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Introduction 1
1.2 Structure of the Thesis 3

2 Preliminaries **5**
2.1 Many-sorted First-order Logic with Theories 5
2.2 Trace Logic 7
2.3 Input Language $\mathcal{W}$ 9
2.4 Superposition Calculus 9
2.5 Experiments and Tooling 11

3 Safety Properties in Trace Logic **13**
3.1 Safety Properties 13
3.2 Concrete Safety Properties 34

4 State of the Art **37**
4.1 Software Verification 37
4.2 Verification of Array Manipulating Programs 43

5 Conclusion **47**
5.1 Conclusion 47
5.2 Challenges and Future Work 47

List of Figures **49**

Bibliography **51**

CHAPTER 1 Introduction

1.1 Introduction

The digitalization of our surrounding is inevitable. With the introduction of new technologies in our lives, e.g. self driving cars, we also face new problems. One of these problems is the handling of critical software. Numerous applications affecting our everyday life are often extensively tested on functionality but rarely proven to really meet the specific requirements or safety specifications. This can lead to data loss, invasion of privacy or, in extreme cases, to life-threatening situations. One might think that because of our progress in computer science, software bugs hardly occur anymore, but that is wishful thinking. The following fairly recent events of software failures should underline the importance of better handling of software verification:

- On the 11th of April 2019 an Israel spacecraft crashed into the moon, due to a software bug within its engine system.
- On the 7th of August 2019 over hundred flights of British Airways were cancelled due to a stated software problem in their system.
- On the 18th of March 2018 a self-driving Uber car on the roads of Arizona hit and killed a pedestrian.

Now the question arises, why simple software testing is not enough? The main problem with software testing, as Dijkstra famously stated [BR70], is that it shows only the presence of logical errors but not their absence.

A solution to this problem is viewing a program as a mathematical object and mathematically prove the desired program properties [Dij72, Hoa69]. Commonly, verification frameworks tackle this by encoding the source code in some sort of logical representation and perform proofs by using a sophisticated solver. Mathematically proving correctness of program properties is the key factor for improving quality of all kinds of digital products. Considering that proving correctness of program properties by hand is time consuming and needs expert knowledge, it becomes more and more infeasible in practice, due to the increasing demand for software products and growing code bases. Subsequently, it is necessary to keep researching and developing novel and better formal verification techniques and tools.

To better understand the ulterior motive of the present thesis, let us look at the following example. Figure 1.1 shows a simple program that assigns every element in an array the value 1. For the sake of this example let us assume that this is part of a highly critical software and that all elements in the array must never have the value 0 at the end of the program execution.

```
func main() {
    Int alength;
    Int[] a;
    Int i = 0;

    while(i < alength) {
        a[i] = 1;
        i++;
    }
}
```

Figure 1.1: Example Program

The property we are interested in would therefore be that assuming `alength` has the value of the length of the array `a`, all elements of `a` are not equal 0. Of course one can easily verify this for the program in figure 1.1, but once programs become bigger and once they consist of additional control structures, verifying these properties automatically becomes very hard.

This thesis uses first-order theorem proving to validate software properties expressed in first-order logic with theorems. The main focus will lay on the framework Rapid [BEG$^+$19]. Rapid takes as input the program source code and encodes it into first-order logic for *superposition-based* solvers such as Vampire [KV13]. We call the axioms constituting the program *trace axioms* and some of its inherent properties *trace lemmas* to form *trace logic* $\mathcal{L}$, our instance of first-order logic with time-point reasoning. A detailed formal description concerning Rapid and *trace logic* is provided in Chapter 2. In contrast to *SMT-based* approaches [Lei10, KGC16, BFMP11], handling full first-order logic with theories allows for arbitrarily quantified properties over integer and time-points. In particular, the focus of this thesis will lay on three specific program properties with a different quantifier prefix and the needed *trace lemmas* to successfully prove them.

Contribution The main contributions are summarized below:

1. We explain and define trace lemmas needed for the desired proofs of the generalized array properties. Specifically, we provided a full explanation of the proofs with the main focus laying on applying the trace lemmas and showing why they are needed and how they are used for a successful inductive reasoning (Section 3.1).
2. We substituted the generic properties with concrete ones to demonstrate the reusability of the proof and the introduced lemmas for a whole class of related programs and properties (Section 3.2).
3. The knowledge gained by the proofs concerning the lemmas can now be used to prove various subclasses of these array properties and provides thereby a good foundation to improve existing tools based upon our discoveries. In the case of Rapid, the result can be used to get a better inside on the needed trace lemmas and on the generated proof structure to enable automated proofs for the whole subclass of related array properties. Especially exciting is the fact that with the described method we were able to automatically prove a safety property with quantifier alternation (Chapter 3).

1.2 Structure of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we give a short review of first-order logic and state the additional theories we will be using for our proofs in Chapter 3. Furthermore, we will report on some preliminaries concerning the syntax and semantic of our programming language and the resulting encoding, which will accompany us throughout this thesis. Finally, we will conclude Chapter 2 by an introduction to the superposition calculus and the SatViz [GKS19] tool that was used to visualize, conduct and verify all proofs within this thesis. In Chapter 3, we introduce and discuss *trace lemmas* in detail. Furthermore, we show their usefulness by using them to prove safety properties from two generalized while-programs. In the course of it, we take a closer look on the generated clauses provided by the Rapid [BEG+19] framework and discuss the semantic meaning of the proof parts. At the end of Chapter 3, we illustrate the reusability of the generalized proofs on a concrete example program and discuss the observed results. In Chapter 4, we present an overview of historical and state-of-the-art research of proof automation related to this thesis. Finally we conclude and summarize the thesis in Chapter 5.

CHAPTER 2

Preliminaries

2.1 Many-sorted First-order Logic with Theories

As we will discuss later, our universe is not a homogeneous collection of objects, but consists of objects with different sorts. To reflect this, we consider many-sorted first-order logic. Furthermore, we consider our first-order logic to have built in equality. In our proofs all our formulae are well-formed and well-sorted.

2.1.1 Syntax

We define the signature $\Sigma_\Gamma = \{\Omega, \Pi, \Gamma, v\}$ for our many sorted first-order logic as follows:

- Ω, Π are the sets for our functions and predicates;
- Γ is a non empty set of sort symbols, e.g. $\mathbb{N}$. (Note that we do not add the boolean sort defined with $\mathbb{B}$, used by our logical operations);
- v is a function assigning sorts to our function, predicate and variable symbols.

We inductively define the set of well-sorted terms $T^S_{\Sigma_\Gamma}$ of a sort $S \in \Gamma$ over Σ_Γ as follows:

- a variable $x \in T^S_{\Sigma_\Gamma}$ if $v(x) = S$;
- if $t_1, \ldots, t_n$ are well-sorted terms in $T^{S_1}_{\Sigma_\Gamma}, \ldots, T^{S_n}_{\Sigma_\Gamma}$, f is a function with arity n and $v(f) = S_1, \ldots, S_n, S$, then $f(t_1, \ldots, t_n)$ is a well-sorted term.

Note that a constant is a function of arity 0. Further, $\bigcup_{s\in\Gamma} T^S_{\Sigma_\Gamma}$ forms the set of well-formed terms over Σ_Γ.

We inductively define the set of well-sorted / well-formed formulae F_{Σ_Γ} over Σ_Γ as follows:

- $\top$, $\bot$ are formulae. (Note that $v(\top) = v(\bot) = \mathbb{B}$);
- if $t_1, \dots, t_n$ are well-sorted terms in $T^{S_1}_{\Sigma_\Gamma}, \dots, T^{S_n}_{\Sigma_\Gamma}$, P is a predicate with arity n and $v(P) = S_1, \dots, S_n, \mathbb{B}$, then $P(t_1, \dots, t_n)$ is a well-formed formula;
- if $\alpha, \beta \in F_{\Sigma_\Gamma}$ then also $\alpha \circ \beta$ is a well-formed formula, where $\circ \in \{\wedge, \vee \rightarrow, \leftarrow, \leftrightarrow\}$. (Note that $v(\circ) = \mathbb{B}, \mathbb{B}, \mathbb{B}$);
- if $\alpha \in F_{\Sigma_\Gamma}$ then also $\neg\alpha$ is a well-formed formula. (Note that $v(\neg) = \mathbb{B}, \mathbb{B}$);
- if $\alpha \in F_{\Sigma_\Gamma}$ and α contains an unbound variable x of sort S, then also $Qx^S.\alpha$ is a well-formed formula, where $Q \in \{\exists, \forall\}$. Note that in this formula we quantify only over the universe corresponding to the sort S.

Our built-in equality is denoted by $\simeq$ and we define $\neg(t \simeq s)$ to be $t \not\simeq s$ for arbitrary terms s, t.

2.1.2 Semantic

We only consider "strict" models, which means our sort universes are disjoint. We define a model $\mathcal{A} = \{\mathcal{U}_\mathcal{A}, \mathcal{F}_\mathcal{A}, \mathcal{P}_\mathcal{A}, \mathcal{X}_\mathcal{A}\}$ for a signature Σ_Γ as follows:

- $\mathcal{U}_\mathcal{A} = \{M_{S_1}, \dots, M_{S_n}\}$, for sort universe $M_{S_i} \neq \emptyset$ and it corresponds to a specific sort $S_i \in \Gamma$;
- $\mathcal{F}_\mathcal{A}$ is a set consisting of interpretations for all functions in Σ_Γ. That is $\{f_i^\mathcal{A} : M_{S_1} \times \dots \times M_{S_n} \rightarrow M_S | f_i \in \Omega \ \wedge \ v(f_i) = S_1, \dots, S_n, S \ \wedge \ arity(f_i) = n\}$;
- $\mathcal{P}_\mathcal{A}$ is a set of relations for each predicate in Σ_Γ.
  That is $\{P_i^\mathcal{A} \subseteq M_{S_1} \times \dots \times M_{S_n} | P_i \in \Pi \ \wedge \ v(P_i) = S_1, \dots, S_n, \mathbb{B} \ \wedge \ arity(P_i) = n\}$;
- The interpretations of variables $\mathcal{X}_\mathcal{A}$ can be treated like functions with arity 0, that is $\mathcal{X}_\mathcal{A} = \{x_S^\mathcal{A} : M_S | x_S \in X\}$, where X is the set of variables.

In our proofs we will use refutations to show that no such model $\mathcal{A}$ exists for a specific formula F.

2.1.3 Theories

A first-order theory $T_\mathcal{S}$ consists of:

1. a signature $\Sigma_\mathcal{S}$ from which we can derive well-formed formulae;
2. axioms, which are a set of well-formed formulae over that signature $\Sigma_\mathcal{S}$.

For our proofs, we make use of the theory of linear integer arithmetic $T_{\mathbb{I}}$. Specifically, the theory is used to reason about integer variables or integer-valued arrays. The integer sort is denoted as $\mathbb{I}$ and we consider the signature of

$$\Sigma_{\mathbb{I}} = \{\{\dots, -1\backslash 0,\ 0\backslash 0,\ 1\backslash 0, \dots,\ +\backslash 2,\ *\backslash 2\}, \{< \backslash 2\}, \{\mathbb{I}\}, v\},$$

where v is defined as expected.

Furthermore, we make use of the theory of term algebras $T_{\mathcal{A}}$ over the natural numbers $\mathbb{N}$. We define the signature of the natural numbers as follows,

$$\Sigma_{\mathbb{N}} = \{\{zero\backslash 0,\ succ\backslash 1,\ pred\backslash 1\}, \{< \backslash 2\}, \{\mathbb{N}\}, v\},$$

where v is defined as expected. Note that we extend the theory of term algebra $T_{\mathcal{A}}$ with an incompletely axiomatized "<" to describe an ordering on the natural numbers $\mathbb{N}$. Natural numbers will come to use in describing loop iterations discussed in Section 2.2.

The axioms for $\Sigma_{\mathbb{I}}$ and $\Sigma_{\mathbb{N}}$ are generated by the RAPID [BEG+19] framework or by the VAMPIRE [KV13] theorem prover, respectively.

Finally, we introduce an uninterpreted sort $\mathbb{L}$, which will describe our timepoints defined in Section 2.2.

2.2 Trace Logic

The concept of *trace logic* [BEG+19] is the core notion of this thesis. *Trace logic* is an instance of many-sorted first-order logic with equality. We will use it to express the semantics and properties for while-programs. In contrast to [BEG+19], we are only interested in program properties and not relational properties.

For now we focus on expressing locations, timepoints and program variables in *trace logic*. The axiomatization of while-programs and the *trace lemma* needed for inductive reasoning will be discussed as we go through the proofs and are therefore left out for now.

2.2.1 Locations and Timepoints

Throughout this thesis, we consider a program as a set of locations. A program location corresponds to a point in the program on which an interpreter could stop. We consider the set of the individual visits of the locations as timepoints of sort $\mathbb{L}$. Furthermore, we introduce uninterpreted constant and function symbols for each program statement s. For this, we distinguish between following cases:

i) s is only visited once, i.e. it is not enclosed by a loop;

ii) s is enclosed by exactly one loop;

iii) s is enclosed by multiple loops.

For every s falling into the case i), we introduce an uninterpreted constant symbol l_s of the sort $\mathbb{L}$.

For each s falling under ii), we add an uninterpreted function symbol l_s, with signature $l_s : \mathbb{N} \rightarrow \mathbb{L}$. The function l_s is defined over the natural numbers, which are used to describe the current iteration. Additionally to l_s, we introduce an uninterpreted constant symbol n_s of sort $\mathbb{N}$, which represents the first iteration start of our enclosing loop, where our loop condition does not hold.

To cover all statements of case iii), we extend the idea of ii). This time, we introduce two uninterpreted function symbols l_s and n_s, with $l_s : \mathbb{N}^m \rightarrow \mathbb{L}$ and $n_s : \mathbb{N}^{m-1} \rightarrow \mathbb{N}$, where m represents the number of enclosing loops of s and each argument represents the current iteration of the related enclosing loop. Moreover, an extra uninterpreted constant symbol l_{end} of sort $\mathbb{L}$ is introduced to represent the last timepoint of a program execution.

Throughout this work we define the following macros for the most commonly used timepoints. We define it^s to be a function, which returns for each while-statement s a unique variable of sort $\mathbb{N}$. In the following definitions we consider an arbitrary statement s, its enclosing loops $w_1, \dots w_k$ and it an arbitrary term of sort $\mathbb{N}$.

$$\begin{array}{ll} tp_s := l_s(it^{w_1}, \dots, it^{w_k}) & \text{if } s \text{ is not a while-statement} \\ tp_s(it) := l_s(it^{w_1}, \dots, it^{w_k}, it) & \text{if } s \text{ is a while-statement} \\ lastIt_s := n_s(it^{w_1}, \dots, it^{w_k}) & \text{if } s \text{ is a while-statement} \end{array}$$

We denote the first timepoint for an arbitrary statement s as $start_s$:

$$start_s = \begin{cases} tp_s(0) & \text{if } s \text{ is a while statement} \\ tp_s & \text{otherwise} \end{cases}$$

We define end_s as follows:

$$end_s = \begin{cases} start_{s'} & \text{if } s' \text{ occurs after } s \text{ in a context} \\ end_{s'}, & \text{if } s \text{ is last statement in if-branch of } s' \\ end_{s'}, & \text{if } s \text{ is last statement in else-branch of } s' \\ tp_w(succ(it^w)), & \text{if } s \text{ is last statement in body of } w \end{cases}$$

We refer to end_s as the first timepoint after the execution of s.

2.2.2 Program Variables

To reason over program behavior we express properties over program variables v. In the definition of [BEG+19], the authors do so by capturing the value of program variables v at timepoints (from $\mathbb{L}$) in arbitary program execution traces. In our setting we are not interested in non-relational properties, such as safety properties, about programs and will omit the trace argument for simplicity. Hence we model program variables v as function $v : \mathbb{L} \to \mathbb{I}$, where $v(tp)$ gives the value of v at timepoint tp. If the variable v is an array, we add an additional argument of sort $\mathbb{I}$, which corresponds to the position at which the array is accessed. For non-mutable variables we can simply omit the the timepoint argument.

2.3 Input Language $\mathcal{W}$

We denote our programming language by $\mathcal{W}$, and consider the input language of the Rapid framework from [BEG+19]. It is a simple while-like programming language, supporting the standard control flow statements *while*, *if-then-else* and *skip*. The entry point of our programming language is a top level *main* function. $\mathcal{W}$ allows the declaration of mutable and immutable, that is constant, integer and array, variables. It includes standard side-effect free expressions over booleans and integers, such as multiplication, addition or logical connectives. Finally, control flow statements and expressions may also be arbitrary nested.

Figure 1.1 shows an example of a program written in the programming language $\mathcal{W}$.

2.4 Superposition Calculus

Vampire [KV13] uses the *superposition inference system* to derive proofs. It uses *inference rules* together with a *simplification ordering* ($\succ$) on terms and a *selection function*. Since we want to select the literals by hand without restrictions we assume a selection function which allows the selection of any literal.

Assuming such a fixed *simplification ordering* and *selection function*, the inference system consists of following rules. We will underline the selected literals and denote the used rules on the derivations.

Resolution

$$\frac{\underline{A} \lor C_1 \qquad \underline{\neg A'} \lor C_2}{(C_1 \lor C_2)\sigma} \text{ Res}$$

where σ is the most general unifier (mgu) of A and A'

Equality Resolution

$$\frac{\underline{s \not\simeq s'} \lor C}{C\sigma} \text{ Res}$$

where σ is the mgu of s and s'

Superposition

$$\frac{\underline{l \simeq r} \vee C_1 \qquad \underline{L[l']} \vee C_2}{(L[r] \vee C_1 \vee C_2)\sigma}\ \text{Sup}$$

where σ is the mgu of l and l', l' is not a variable, $r\sigma \not\succeq l\sigma$ and $L[l']$ is not an equality literal.

$$\frac{\underline{l \simeq r} \vee C_1 \qquad \underline{t[l'] \simeq t'} \vee C_2}{(t[r] \simeq t' \vee C_1 \vee C_2)\sigma}\ \text{Sup}$$

where σ is the mgu of l and l', l' is not a variable, $r\sigma \not\succeq l\sigma$ and $t'\sigma \not\succeq t[l']\sigma$

$$\frac{\underline{l \simeq r} \vee C_1 \qquad \underline{t[l'] \not\simeq t'} \vee C_2}{(t[r] \not\simeq t' \vee C_1 \vee C_2)\sigma}\ \text{Sup}$$

where σ is the mgu of l and l', l' is not a variable, $r\sigma \not\succeq l\sigma$ and $t'\sigma \not\succeq t[l']\sigma$

Factoring

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\sigma}\ \text{Fact}$$

where σ is the mgu of A and A'

Equality Factoring

$$\frac{s \simeq t \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee \underline{t \not\simeq t'} \vee C)\sigma}\ \text{Fact}$$

where σ is the mgu of s and s', $t'\sigma \not\succeq s\sigma$ and $t'\sigma \not\succeq t\sigma$

A special inference rule is *demodulation*. Demodulation is a simplification and a special form of the superposition rules. It does not have to be applied to selected literals and it deletes one of its parents.

$$\frac{\underline{l \simeq r} \qquad \cancel{\underline{L[l']} \vee C}}{L[r\sigma] \vee C}\ \text{Dem}$$

where $l\sigma \simeq l', l\sigma \succ r\sigma$ and $(L[l'] \vee C)\sigma \succ (l\sigma \succ r\sigma)$

2.5 Experiments and Tooling

To generate program semantics we rely on the Rapid framework [BEG+19]. The proofs of the problems in this work rely on the first-order theorem prover Vampire [KV13]. It is a refutational full first-order prover with built-in equality reasoning and theory support. To better present and engage with the proofs we use SatViz [GKS19]. SatViz is a tool for interactively visualizing proofs and proof attempts of the first-order theorem prover Vampire. It also provides an interactive graphical interface to Vampire, which we use to make and verify the proofs for the program properties. We will provide some of the proofs and proof parts as screenshot from the SatViz output. Figure 2.1 shows how such an output will look like. One can see a sub-tree of a proof with some of its used clauses visualized as nodes. Arrows indicate the derivation direction and mark the child and parent clauses. The clause is displayed inside of the node. Note that the clauses displayed do not have to be in conjunctive normal form. For our setting the blue colored nodes represent the theorem axioms provided by Vampire, the dark green node represents the negated conjecture and the light green ones the extra added lemmas. The light and dark grey nodes are derived clauses and the empty clause is annotated by `$false`.

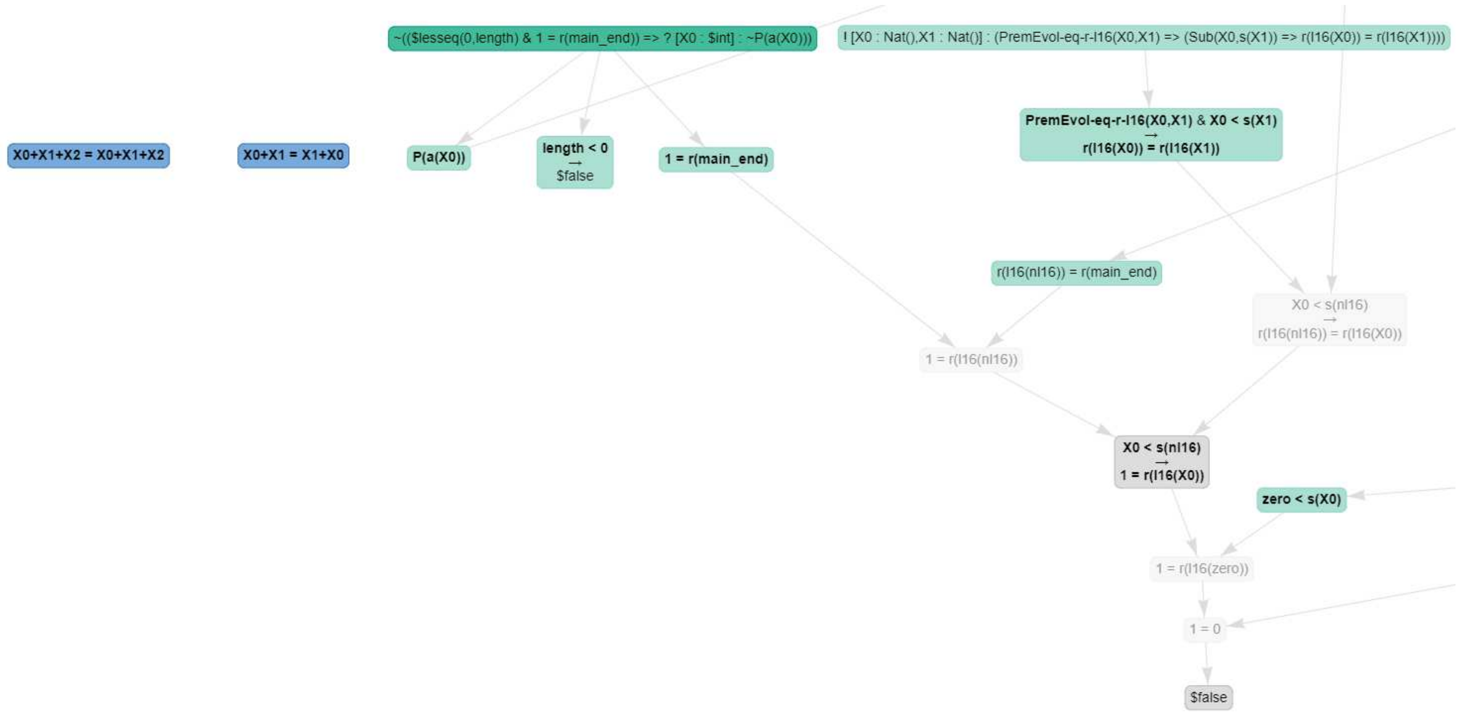
~(($lesseq(0,length) & 1 = r(main_end)) => ? [X0 : $int] : ~P(a(X0)))
! [X0 : Nat(),X1 : Nat()] : (PremEvol-eq-r-l16(X0,X1) => (Sub(X0,s(X1)) => r(l16(X0)) = r(l16(X1))))
X0+X1+X2 = X0+X1+X2
X0+X1 = X1+X0
P(a(X0))
length < 0 → $false
1 = r(main_end)
PremEvol-eq-r-l16(X0,X1) & X0 < s(X1) → r(l16(X0)) = r(l16(X1))
r(l16(nl16)) = r(main_end)
X0 < s(nl16) → r(l16(nl16)) = r(l16(X0))
1 = r(l16(nl16))
X0 < s(nl16) → 1 = r(l16(X0))
zero < s(X0)
1 = r(l16(zero))
1 = 0
$false

Figure 2.1: SatViz example for parts of the proof for Figure 3.1

CHAPTER 3

Safety Properties in Trace Logic

In this chapter we will look at proofs of generic safety properties over a predicate P, representing some property over arrays. Specifically, we look into three proofs showcasing properties of different quantifier settings: $\exists$, $\forall$ and $\forall\exists$. We will walk through the proofs in detail and explain the introduced lemmas needed for the proofs. Replacing P with different properties over array elements produces similar properties, which can be summarized in a program class related to the original program. Furthermore, using a generic property P instead of a concrete property gives the ability to produce more general proofs. The resulting proofs are more general in the sense, that they can be used to quickly instantiate proofs with $\exists$, $\forall$ and even $\forall\exists$ quantified program safety properties of the related program classes. Examples of these instantiations are provided in Section 3.2 of this chapter.

The logical encoding of the programs is provided by Rapid [BEG+19]. Going into detail of the preprocessing would exceed this thesis so it is left to the reader. All proofs in this thesis are carried out using the interactive SatViz [GKS19] tool and the superposition based theorem prover Vampire [KV13].

3.1 Safety Properties

3.1.1 $\exists$ Quantifier Setting

For our first safety property we look at the program in figure 3.1. It shows a program that iterates over the indices of a constant array `a`. Variables that are declared constant (`const`) are immutable and can not change throughout the whole program execution. Figure 3.1 initializes an integer variable `r` with the value 0 and assigns the value 1 to it if the predicate `P` is satisfied for the current iteration element `a[i]`. Accordingly, we want to prove (assuming termination) that if `r` is 1 at the end of the execution, there exists an array index `k` such that the element `a[k]` satisfies the predicate `P`. The program variable `length` denotes the length of our array `a` and should therefore not be negative. We formalize our desired safety property as follows:

$$0 \leq length \wedge r(end) \simeq 1 \rightarrow \exists k^{\mathbb{I}}.P(a(k))$$

As discussed in the Chapter 2, we introduce for the program variable `r` and the iterator variable `i` uninterpreted functions $r : \mathbb{L} \rightarrow \mathbb{I}$ and $i : \mathbb{L} \rightarrow \mathbb{I}$ over timepoints ($\mathbb{L}$) of sort integer and for the constant `length` an uninterpreted symbol of sort integer. Note that although we recognize variables for example as loop iterator or array lengths, we make no semantic difference between them and other program variables. Since our array `a` is a constant array we introduce an uninterpreted function over integers and leave out the timepoints. This leaves us with the uninterpreted constant function $a : \mathbb{I} \rightarrow \mathbb{I}$ of sort integer. For the last timepoint in our program the constant $end : \mathbb{L}$ of the sort timepoint is used.

```
    func main() {
        const Int[] a;
        const Int length;

        Int r = 0;
        Int i = 0;

        while(i < length) {
            if(P(a[i])) {
                r = 1;
            } else {
                skip;
            }
            i = i + 1;
        }
    }

```

Figure 3.1: Example program with generic functional behavior involving $\exists$.

General Proof Idea

Using the refutation prover we are interested in deriving an empty clause at some point in our proof search. So instead of proving the formula

$$0 \leq length \wedge r(end) \simeq 1 \rightarrow \exists k^{\mathbb{I}}.P(a(k))$$

we show that the negation is unsatisfiable. Applying the negation to the property we arrive at the negated conjecture represented as following set of clauses:

$$\{\ 0 \leq length,\ \ r(end) \simeq 1,\ \ \forall k^{\mathbb{I}}.\neg P(a(k)))\ \}$$

Specifically, using backwards reasoning we start our proof with the negated conjecture and use the program axiomatization and the theories of integer and natural numbers to derive the empty clause. The program axiomatization consists of trace logic axioms for program semantics and trace lemmas for inductive reasoning over loops.

The variable `r` is initialized with 0 at location 5, which is encoded as $r(tp_{l_5}) \simeq 0$. Using $\forall k.\neg P(a(k))$ from the negated conjecture and the semantics we will show that `r` never changes throughout the program execution, contradicting the clause from the conjecture $r(end) \simeq 1$. To show that `r` never changes inside the loop execution we use the *value evolution lemma*, given below.

Value Evolution Lemma

To represent the fact of a mutable program variable not being changed between certain bounds of a loop execution we use a trace lemma called *value evolution*. Let `w` be an arbitrary while-statement, let `v` a mutable program variable (`r` in our example) and let $\unlhd$ be a reflexive and transitive predicate (e.g $<$, $>$, $\simeq$). Then the *value evolution* lemma is formalized as

$$\forall it_L^{\mathbb{N}} \forall it_R^{\mathbb{N}}.($$

$$\forall it^{\mathbb{N}}.(it_L \leq it < it_R \land v(tp_w(it_L)) \unlhd v(tp_w(it)) \rightarrow v(tp_w(it)) \unlhd v(tp_w(succ(it))))$$

$$\rightarrow$$

$$it_L \leq it_R \rightarrow v(tp_w(it_L)) \unlhd v(tp_w(it_R))$$

$$)$$

In this thesis we will use this lemma only for the equality $\simeq$ predicate. To get a clear understanding of how it is used in the Rapid setting, let us take a look at the instantiation for the predicate $\simeq$ and the mutable program variable `r` in our setting. We use the uninterpreted function l_8 to denote the timepoint of the loop starting at the line number 8 in Figure 3.1. The uninterpreted function l_8 is parameterized by a natural number denoting the iterations.

$\forall it_L^{\mathbb{N}} \forall it_R^{\mathbb{N}}.($

$$\forall it^{\mathbb{N}}.(it_L \leq it < it_R \wedge r(l_8(it_L)) \trianglelefteq r(l_8(it)) \rightarrow r(l_8(it)) \trianglelefteq r(l_8(succ(it))))$$

$$\rightarrow$$

$$it_L \leq it_R \rightarrow r(l_8(it_L)) \trianglelefteq r(l_8(it_R))$$

$)$

For readability we split the lemma into two parts, namely the premise and the conclusion. For this we introduce a new predicate $PE_{r,l_8} : \mathbb{N}, \mathbb{N} \rightarrow \mathbb{B}$, representing the premise, over two natural numbers, representing the bounds.

Premise:

$\forall it_L^{\mathbb{N}} \forall it_R^{\mathbb{N}}.($

$$PE_{r,l_8}(it_L, it_R)$$

$$\leftrightarrow$$

$$\forall it^{\mathbb{N}}.(it_L \leq it < it_R \wedge r(l_8(it_L)) \simeq r(l_8(it)) \rightarrow r(l_8(it)) \simeq r(l_8(succ(it))))$$

$)$

Conclusion:

$\forall it_L^{\mathbb{N}} \forall it_R^{\mathbb{N}}.(\ PE_{r,l_8}(it_L, it_R) \rightarrow (it_L \leq it_R \rightarrow r(l_8(it_L)) \simeq r(l_8(it_R)))\ )$

We now show how to derive this lemma for our current example. To do so, we will need to show that the premise holds in order to use the conclusion. First of all, to use this lemma effectively we have to think about the bounds it_L and it_R. Most of the time these bounds become clear when we have a certain goal in mind. Considering our current interest in showing that `r` never changes its value throughout the execution of the loop, a good bound for it_L would be the first and for it_R the last execution of the loop. To show that the premise holds we have to show that the induction step holds for an arbitrary iteration within these bounds, that is for some iteration it we show that $r(l_8(it))$ is equal to $r(l_8(s(it)))$. If this is successful, we can use the derived premise to derive our conclusion and gain a loop invariant related to `r`. Considering our instantiated lemma, this would mean that `r` after the loop execution has the same value as `r` before the loop execution.

Proof

For our first example we make a partial proof to derive the *value evolution* lemma. In our assisting proof tool SatViz, we use the superposition calculus with a selection function that let us select any clause and literal in no particular order, since we select our clauses manually and do not need to restrict our search space. Besides from some clauses of the program semantics, for our first proof we mainly use the conjecture and the *value evolution* lemma. From now on we denote sK_i as skolem constants or functions and X_i as implicit universal quantified variables. Also we shorten *succ* to simply s and end_{l_i} to nl_i. Recall the clause set of the negated conjecture:

Conjecture:

$$\{\ 0 \leq length,\ r(end) \simeq 1,\ \neg P(a(X_0)))\ \}$$

Moreover, our proof assisting tool SatViz transforms the two parts of the *value evolution* lemma (premise and conclusion) for equality and the variable `r` into conjunctive normal form. Note that $sK_0 : \mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}$ is a skolem function representing the inner universal quantified iteration (it), parameterized by the outer universal quantified bounds (it_L, it_R).

Value evolution premise:

$$\begin{gathered}\{\ r(l_8(X_1)) \not\simeq r(l_8(s(sK_0(X_1, X_2))))) \vee PE_{r,l_8}(X_1, X_2),\\ X_0 \leq sK_0(X_1, X_2) \vee PE_{r,l_8}(X_1, X_2),\\ sK_0(X_1, X_2) < X_2 \vee PE_{r,l_8}(X_1, X_2),\\ r(l_8(X_1)) \simeq r(l_8(sK_0(X_1, X_2))) \vee PE_{r,l_8}(X_1, X_2)\ \}\end{gathered}$$

Value evolution conclusion:

$$\{\ \neg PE_{r,l_8}(X_3, X_4) \vee X_3 \geq s(X_4) \vee r(l_8(X_3)) \simeq r(l_8(X_4))\ \}$$

Furthermore we will take a look at some of the needed clauses produced by Rapid to model the desired program semantics. We use nl_8 of sort $\mathbb{N}$ to denote the first iteration where the loop condition does not hold.

To express the else part of our if statement in Figure 3.1 we use following clause:

$$\neg P(a(i(l_8(X_5)))) \vee X_5 \geq nl_8 \vee r(l_8(X_5)) \simeq r(l_{15}(X_5))$$

This clause represents parts of the semantics of the `else` branch inside the loop. It asserts that for any iteration it of the loop, where $P(a(i(l_8(it))))$ and $it < nl_8$, the program variable `r` has the same value at the end ($l_{15}(it)$) as at the start ($l_8(it)$) of the loop.

Next we define parts of the semantics of the `while` statement. Given an arbitrary iteration it, where the loop condition is true, the variable `r` at the start of the next iteration ($l_8(s(it))$) is defined as `r` at the end of the current iteration ($l_{15}(it)$). This is formalized as follows:

$$r(l_{15}(X_5)) \simeq r(l_8(s(X_5))) \vee X_5 \geq nl_8$$

Finally formalize the initialization of r as

$$0 \simeq r(l_8(zero))$$

and the final result of r as

$$r(l_8(nl_8)) \simeq r(end)$$

First we want to show that `r` is not changed during the loop execution. To achieve this, we derive the premise (PE_{r,l_8}) for our *value evolution* lemma for the bounds *zero* and nl_8, representing the first and the last visit of the loop head. To achieve this we show for an arbitrary iteration inside those bounds $(sK_0(X_1, nl_8))$ that we can execute one loop iteration without `r` being changed. We start by combining the arbitrary iteration from our lemma with the loop semantics that states that the variable `r` at the start of an arbitrary iteration, which of course is not the first iteration, has the same value as at the end of the previous loop iteration. That is,

$$\frac{\underline{X_5 \geq nl_8} \vee r(l_{15}(X_5)) \simeq r(l_8(s(X_5))) \qquad \underline{sK_0(X_1, X_2) < X_2} \vee PE_{r,l_8}(X_1, X_2)}{r(l_{15}(sK_0(X_1, nl_8))) \simeq r(l_8(s(sK_0(X_1, nl_8)))) \vee PE_{r,l_8}(X_1, nl_8)}\ \text{Res}$$

with mgu $= \{\ X_2 \rightarrow nl_8, X_5 \rightarrow sK_0(X_1, nl_8)\ \}$.

Now we have already fixed one of our bounds. In the next step we use the inductive step from our lemma to narrow our proof down to show that if `r` is unchanged for an arbitrary iteration, respecting the bounds of the premise, we proved the premise. Hence,

$$\frac{\dfrac{\underline{\begin{array}{c}PE_{r,l_8}(X_1, nl_8) \vee \\ r(l_{15}(sK_0(X_1, nl_8))) \simeq r(l_8(s(sK_0(X_1, nl_8))))\end{array}} \qquad \underline{\begin{array}{c}PE_{r,l_8}(X_1, X_2) \vee \\ r(l_8(X_1)) \not\simeq r(l_8(s(sK_0(X_1, X_2))))\end{array}}}{\underline{PE_{r,l_8}(X_1, nl_8)} \vee \underline{PE_{r,l_8}(X_1, nl_8)} \vee r(l_8(X_1)) \not\simeq r(l_{15}(sK_0(X_1, nl_8)))}\ \text{Sup}}{PE_{r,l_8}(X_1, nl_8) \vee r(l_8(X_1)) \not\simeq r(l_{15}(sK_0(X_1, nl_8)))}\ \text{Fact}$$

with mgu $= \{X_2 \rightarrow nl_8\}$,

To show that the variable `r` is not altered during a loop iteration, we used parts of the conjecture and the semantics of the enclosed `if` and `skip` statement:

$$\frac{\underline{P(a(X_0))} \qquad \underline{\neg P(a(i(l_8(X_5))))} \vee X_5 \geq nl_8 \vee r(l_8(X_5)) \simeq r(l_{15}(X_5))}{X_5 \geq nl_8 \vee r(l_8(X_5)) \simeq r(l_{15}(X_5))}\ \text{Res}$$

with mgu $= \{X_0 \rightarrow i(l_8(X_5)))\}$

The result of this resolution step states that for an arbitrary iteration smaller than nl_8, the value of `r` is the same at the start as it is at the end of the iteration. This is exactly what we need except not with any arbitrary iteration, but with the arbitrary iteration used for the inductive step from our lemma:

$$\dfrac{\underline{X_5 \geq nl_8} \lor r(l_8(X_5)) \simeq r(l_{15}(X_5)) \qquad \underline{sK_0(X_1, X_2) < X_2} \lor PE_{r,l_8}(X_1, X_2)}{r(l_8(sK_0(X_1, nl_8))) \simeq r(l_{15}(sK_0(X_1, nl_8))) \lor PE_{r,l_8}(X_1, nl_8)}\ \text{Res}$$

with mgu $= \{X_2 \rightarrow nl_8, X_5 \rightarrow sK_0(X_1, nl_8)\}$

Now we can combine our derived clauses together with parts of the premise from the *value evolution* lemma and complete the inductive reasoning to derive the premise predicate for our desired bounds:

$$\dfrac{\dfrac{\begin{array}{c} PE_{r,l_8}(X_1, nl_8) \lor \\ \underline{r(l_8(sK_0(X_1, nl_8))) \simeq r(l_{15}(sK_0(X_1, nl_8)))} \end{array} \qquad \begin{array}{c} PE_{r,l_8}(X_1, nl_8) \lor \\ \underline{r(l_8(X_1)) \not\simeq r(l_{15}(sK_0(X_1, nl_8)))} \end{array}}{\underline{PE_{r,l_8}(X_1, nl_8)} \lor \underline{PE_{r,l_8}(X_1, nl_8)} \lor r(l_8(X_1)) \not\simeq r(l_8(sK_0(X_1, nl_8)))}\ \text{Sup}}{r(l_8(X_1)) \not\simeq r(l_8(sK_0(X_1, nl_8))) \lor PE_{r,l_8}(X_1, nl_8)}\ \text{Fact}$$

with mgu $= \{X_2 \rightarrow nl_8\}$

Thus,

$$\dfrac{\dfrac{\begin{array}{c} PE_{r,l_8}(X_1, X_2) \lor \\ \underline{r(l_8(X_1)) \simeq r(l_8(sK_0(X_1, X_2)))} \end{array} \qquad \begin{array}{c} PE_{r,l_8}(X_1, nl_8) \lor \\ \underline{r(l_8(X_1)) \not\simeq r(l_8(sK_0(X_1, nl_8)))} \end{array}}{PE_{r,l_8}(X_1, nl_8) \lor PE_{r,l_8}(X_1, nl_8)}\ \text{Res}}{PE_{r,l_8}(X_1, nl_8)}\ \text{Fact}$$

with mgu $= \{X_2 \rightarrow nl_8\}$.

Given the premise predicate from the *value evolution* lemma, we are now able to derive the conclusion of the *value evolution* lemma. Additionally, we conclude with the help of the semantics and parts of the conjecture that `r` has the value 1 after the loop execution:

$$\dfrac{\underline{PE_{r,l_8}(X_1, nl_8)} \qquad \underline{\neg PE_{r,l_8}(X_3, X_4)} \lor X_3 \geq s(X_4) \lor r(l_8(X_3)) \simeq r(l_8(X_4))}{r(l_8(nl_8)) \simeq r(l_8(X_1)) \lor X_1 \geq s(nl_8)}\ \text{Res}$$

with mgu $= \{X_3 \rightarrow X_1,\ X_4 \rightarrow nl_8\}$

$$\frac{\underline{r(l_8(nl_8)) \simeq r(end)} \qquad \underline{1 \simeq r(end)}}{1 \simeq r(l_8(nl_8))}\ \text{Sup}$$

$$\frac{\underline{r(l_8(nl_8)) \simeq r(l_8(X_1))} \vee X_1 \geq s(nl_8) \qquad \underline{1 \simeq r(l_8(nl_8))}}{X_1 \geq s(nl_8) \vee 1 \simeq r(l_8(X_1))}\ \text{Sup}$$

Finally, we can derive the empty clause by showing that `r` has different values at the start and at the end of the program execution. Note that although we no longer have the premise predicate $P(X_1, nl_8)$, the implicit universal variable X_1, which represented our start bound, is now fixed to *zero*. This gives us following proof subtree to conclude the proof. Note that $X_1 \geq s(l_8)$ can also be written as $X_1 > l_8$ which we will use for the next resolution step:

$$\frac{\dfrac{\underline{zero \leq X_6} \qquad \underline{X_1 > l_8} \vee 1 \simeq r(l_8(X_1))}{\underline{1 \simeq r(l_8(zero))}}\ \text{Res} \qquad \underline{0 \simeq r(l_8(zero))}}{\dfrac{1 \simeq 0}{\square}\ \text{Eval}}\ \text{Sup}$$

with mgu $= \{X_1 \rightarrow zero,\ X_6 \rightarrow l_8\}$

In Figure 3.2 one can see our SatViz proof output concerning the before discussed ∃-quantified property. Note that the numbers of the line annotation and the implicit universal variable annotation may differ from the presented proof parts. The visual output is the same as discussed in the preliminaries.

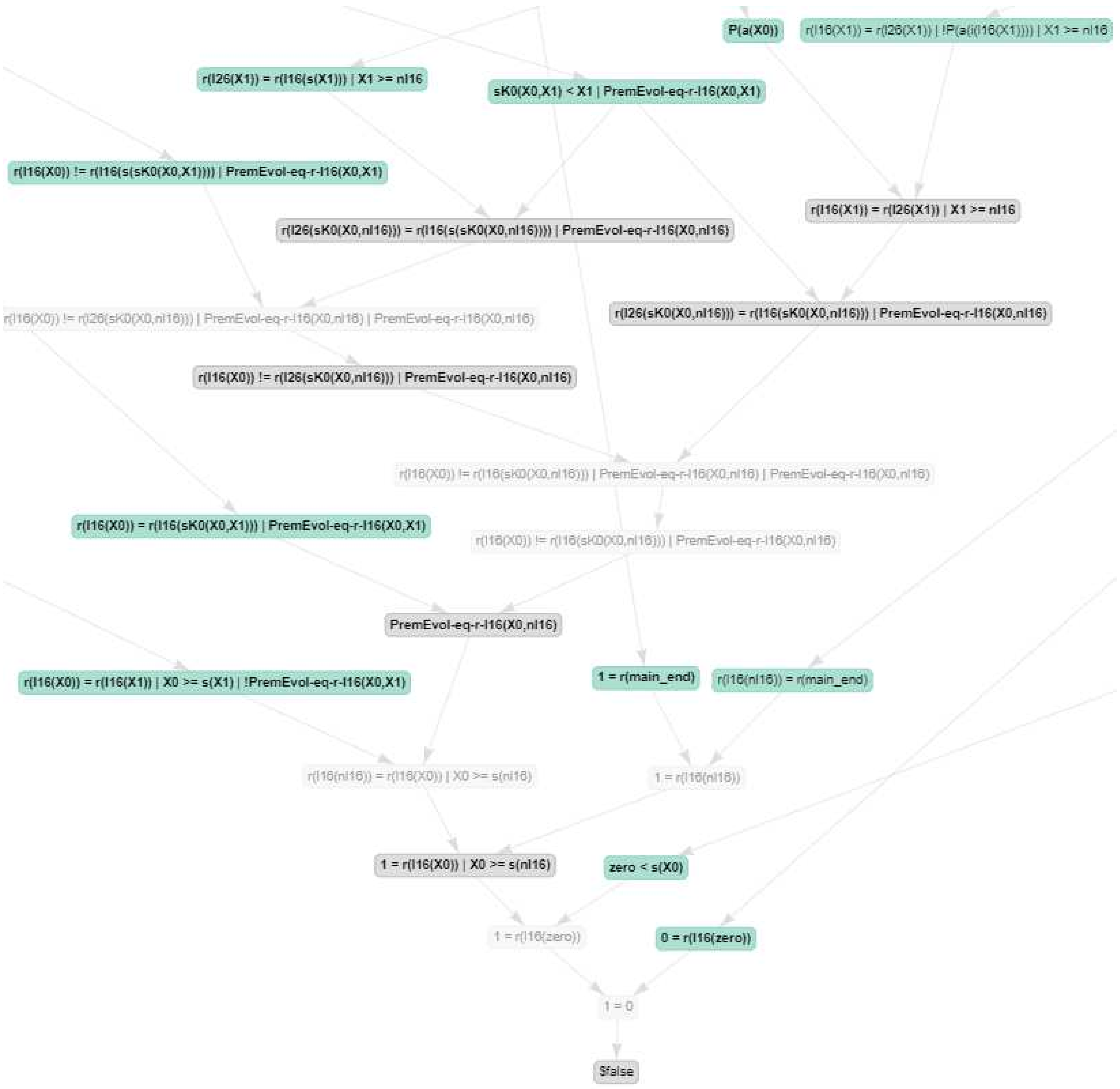
P(a(X0))
r(l16(X1)) = r(l26(X1)) | !P(a(i(l16(X1)))) | X1 >= nl16
r(l26(X1)) = r(l16(s(X1))) | X1 >= nl16
sK0(X0,X1) < X1 | PremEvol-eq-r-l16(X0,X1)
r(l16(X0)) != r(l16(s(sK0(X0,X1)))) | PremEvol-eq-r-l16(X0,X1)
r(l16(X1)) = r(l26(X1)) | X1 >= nl16
r(l26(sK0(X0,nl16))) = r(l16(s(sK0(X0,nl16)))) | PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) != r(l26(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16) | PremEvol-eq-r-l16(X0,nl16)
r(l26(sK0(X0,nl16))) = r(l16(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) != r(l26(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) != r(l16(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16) | PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) = r(l16(sK0(X0,X1))) | PremEvol-eq-r-l16(X0,X1)
r(l16(X0)) != r(l16(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16)
PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) = r(l16(X1)) | X0 >= s(X1) | !PremEvol-eq-r-l16(X0,X1)
1 = r(main_end)
r(l16(nl16)) = r(main_end)
r(l16(nl16)) = r(l16(X0)) | X0 >= s(nl16)
1 = r(l16(nl16))
1 = r(l16(X0)) | X0 >= s(nl16)
zero < s(X0)
1 = r(l16(zero))
0 = r(l16(zero))
1 = 0
$false

Figure 3.2: SatViz proof for the general ∃ property setting

3.1.2 ∀ Quantifier Setting

For our second safety property we look at the program in Figure 3.3. It shows a program that iterates over the indices of a constant array `b`. If the element of the current iteration `b[i]` satisfies some predicate `P`, we add it to the last position of an integer array `a` indicated by `alength` and increase `alength`. We want to prove that at the end of the program execution all copied elements in `a` satisfy `P`. This can be formalized as follows.

$$\forall k^{\mathbb{I}}.(0 \leq blength \land 0 \leq k \land k < alength(end) \rightarrow P(a(end, k)))$$

Similar to our first setting we introduce a constant $blength : \mathbb{I}$ of sort integer, an uninterpreted function $alength : \mathbb{L} \rightarrow \mathbb{I}$ over timepoints of sort integer represent the indices and uninterpreted functions for the arrays `a,b` as $a : \mathbb{L}, \mathbb{I} \rightarrow \mathbb{I}$ and $b : \mathbb{I} \rightarrow \mathbb{I}$ over timepoints and integers of sort integer. The variable $end : \mathbb{L}$ has the same meaning as before and is again a constant timepoint.

```
func main() {
    const Int[] b;
    const Int blength;

    Int[] a;
    Int i = 0;
    Int alength = 0;

    while(i < blength) {
        if (P(b[i])) {
            a[alength] = b[i];
            alength = alength + 1;
        } else {
            skip;
        }
        i = i + 1;
    }
}

```

Figure 3.3: Example program with generic functional behavior involving ∀ and ∀∃.

General Proof Idea

To prove the ∀-quantified property we use the same approach as for the ∃-quantified property in Section 3.1.1. This gives us additionally to our lemmas and theories the following negated conjecture to refute:

$$\exists k^{\mathbb{I}}.(0 \leq blength \land 0 \leq k \land k < alength(end) \land \neg P(a(end, k)))$$

First, let us separate the proof in the following steps:

1. Show that `alength` is *dense*.
2. Show that after assigning a value to a specific index `i` in `a` ,`a[i]` is unchanged until the end of the execution.
3. Show that only values satisfying `P` are copied to `a`.

These steps are enough to conclude our proof and derive an empty clause. For our first step we will use the semantics of `alength` to show that it satisfies the property of being *dense* so we can use this fact in step two. In the second part of the proof we will fix the iteration *it* corresponding to the value of `i` in `a[i]` for the loop split. For this we introduce the *intermediate value* lemma. Furthermore, we introduce the *iteration injectivity* lemma to show that during the loop execution we never go back to the iteration *it*. Finally, we will use the already introduced *value evolution* lemma to show that `a[i]` remains unchanged between our fixed iteration *it* and the last loop iteration *lastIt* and thereby conclude step two. The last step is to combine this reasoning with the fact that all values copied from `b` to `a` satisfy the predicate `P`.

Density Property

The density property helps to deal with iterator variables inside of loops. In this property and in the next two lemmas we assume an integer variable v and a while loop w. In the following formula we say that our integer variable v is *dense* in the execution of the while loop w:

$$Dense_{w,v}$$

$$\leftrightarrow$$

$$\forall it^{\mathbb{N}}.(it < lastIt_w \rightarrow (v(tp_w(succ(it))) \simeq v(tp_w(it))+1) \vee (v(tp_w(succ(it))) \simeq v(tp_w(it))))$$

For an integer variable to be *dense* in our setting, it means that every iteration the variable is either increased by exactly 1 or it stays equal to its value from the previous iteration. This property is most of the time used to determine iterator-like variables to be further used for the *intermediate value* lemma and the *iteration injectivity* lemma.

Intermediate Value Lemma

The *intermediate value* lemma gives us a possibility to reason with loop iterator variables. It is formalized as follows:

$$\forall x^{\mathbb{I}}.($$
$$(Dense_{w,v} \wedge v(tp_w(0)) \leq x \wedge x < v(tp_w(lastIt_w)))$$
$$\rightarrow$$
$$\exists it_2^{\mathbb{N}}.(it_2 < lastIt_w \wedge v(tp_w(it_2)) \simeq x \wedge v(tp_w(succ(it_2))) \simeq v(tp_w(it_2))) + 1$$
$$)$$

stating that if a iterator variable v is *dense* inside a certain while loop w and its starting value $(tp_w(0))$ is different from its end value $(tp_w(lastIt_w))$, then, for each value inbetween the first and the last value of v, there exist an iteration it_2 during the loop execution, where v has exactly that value.

To verify this property, we use this lemma for `alength` at location l_9. Again we can split the lemma into two parts for readability. For this we introduce a premise predicate $PInt_{alength,l_9} : \mathbb{I} \rightarrow \mathbb{B}$ annotated by the variable of interest and the location of the loop beginning over integers. The first part, representing the premise, is formalized as:

Premise:

$$\forall x^{\mathbb{I}}.($$
$$PInt_{alength,l_9}(x)$$
$$\leftrightarrow$$
$$(Dense_{l_9,alength} \wedge alength(l_9(0)) \leq x \wedge x < alength(l_9(nl_9)))$$
$$)$$

and the second part, representing the conclusion, is formalized as:

Conclusion:

$$\forall x^{\mathbb{I}}.($$
$$PInt_{alength,l_9}(x)$$
$$\rightarrow$$
$$\exists it_2^{\mathbb{N}}.(it_2 < nl_9 \wedge alength(l_9(it_2)) \simeq x \wedge alength(l_9(succ(it_2))) \simeq alength(l_9(it_2))) + 1$$
$$)$$

We use this lemma to get the specific iteration where the iterator variable `alength` has the value k from our property. Given that `alength` is dense, together with the program semantics, we can prove the premise predicate. This gives us the iteration where we would like to split the loop and which we will use as initial bound for the *value evolution* lemma. This will be discussed in more detail later.

Iteration Injectivity Lemma

We define the *iteration injectivity* lemma as follows:

$\forall it_1^{\mathbb{N}} it_2^{\mathbb{N}}.($

$$Dense_{w,v} \wedge it_1 < it_2 \leq lastIt_w \wedge v(tp_w(succ(it_1))) \simeq v(tp_w(it_1) + 1)$$

$$\rightarrow$$

$$v(tp_w(it_1)) \not\simeq v(tp_w(it_2))$$

$)$

expressing that given a *dense* variable v, an arbitrary timepoint tp_w and loop iteration it_1 where v is increased by 1, then for any following iteration it_2 of the loop execution the value of v is distinct from the current iteration it_1. This lemma is especially useful when iterating over arrays. In this context it is used to express that an iterator variable `alength` used as index can only visit each array element at most once.

Proof

Showing a full formal proof in the superposition calculus of the $\forall-$property would exceed the space of this thesis. Instead we will provide a high level proof and formal proof parts. To save space and for better readability, duplicate literal elimination is done implicitly.

We use the same notation for skolemization and implicit quantified variables as in Section 3.1.1. For our program in Figure 3.3 we get following negated conjecture in conjunctive normal form:

$$\{\neg P(a(end, sK_0)),\ blength \geq 0,\ sK_0 \geq 0,\ sK_0 < alength(end)\}$$

Our first goal is to derive $Dense_{alength,l_9}$ from our density property for `alength` at line 9. We start with the clausification and skolemization of the property instantiated for $Dense_{alength,l_9}$:

$\{$

$$alength(l_9(sK_2)) \not\simeq alength(l_9(s(sK_2))) \vee Dense_{alength,l_9},$$
$$alength(l_9(sK_2)) + 1 \not\simeq alength(l_9(s(sK_2))) \vee Dense_{alength,l_9}$$
$$sK_2 < nl_9 \vee Dense_{alength,l_9},$$

$\}$

Furthermore, we need the program semantics for the `if` and `while` statement concerning `alength`. The following clauses express the `if` part, where `alength` is increased and the if-condition $(P(b(i(l_9(X_1)))))$ was satisfied:

$$X_1 \geq nl_9 \vee \neg P(b(i(l_9(X_1)))) \vee alength(l_9(X_1)) + 1 = alength(l_{16}(X_1))$$

and the `else` part, where `alength` is left unchanged and the if-condition was unsatisfied:

$$X_1 \geq nl_9 \lor P(b(i(l_9(X_1)))) \lor alength(l_9(X_1)) = alength(l_{16}(X_1))$$

At the start of a new loop iteration ($l_9(s(X_1))$), `alength` has the same value as `alength` at the end of the previous loop iteration ($l_{16}(X_1)$) expressed as the clause:

$$X_1 \geq nl_9 \lor alength(l_{16}(X_1)) = alength(l_9(s(X_1)))$$

Since we established all needed clauses we can start by proving the *density* of `alength` for both execution paths of the `if`. Since both paths satisfy the property of `alength` being *dense*, we will combine them and get rid of the if-condition with a resolution on `P`. First, we combine both paths with the density condition of `alength` staying the same or increase.

$$\dfrac{\underline{\begin{array}{c} X_1 \geq nl_9 \lor \neg P(b(i(l_9(X_1)))) \lor \\ alength(l_9(X_1)) + 1 = alength(l_{16}(X_1)) \end{array}} \qquad \underline{\begin{array}{c} Dense_{alength,l_9} \lor \\ \underline{alength(l_9(sK_2)) + 1} \\ \not\leq alength(l_9(s(sK_2))) \end{array}}}{\begin{array}{c} alength(l_9(s(sK_2))) \not\leq alength(l_{16}(sK_2)) \lor \\ sK_2 \geq nl_9 \lor Dense_{alength,l_9} \lor \neg P(b(i(l_9(sK_2)))) \end{array}}\ \text{Sup}$$

with mgu $= \{X_1 \rightarrow sK_2\}$

Then,

$$\dfrac{\underline{\begin{array}{c} X_1 \geq nl_9 \lor P(b(i(l_9(X_1)))) \lor \\ alength(l_9(X_1)) = alength(l_{16}(X_1)) \end{array}} \qquad \underline{\begin{array}{c} Dense_{alength,l_9} \lor \\ alength(l_9(sK_2)) \not\leq alength(l_9(s(sK_2))) \end{array}}}{\begin{array}{c} alength(l_9(s(sK_2))) \not\leq alength(l_{16}(sK_2)) \lor \\ sK_2 \geq nl_9 \lor Dense_{alength,l_9} \lor P(b(i(l_9(sK_2)))) \end{array}}\ \text{Sup}$$

with mgu $= \{X_1 \rightarrow sK_2\}$.

Then we combine the two paths to get rid of the if-condition:

$$\dfrac{\begin{array}{c} alength(l_9(s(sK_2))) \not\leq alength(l_{16}(sK_2)) \lor \\ Dense_{alength,l_9} \lor sK_2 \geq nl_9 \lor \\ \underline{P(b(i(l_9(sK_2))))} \end{array} \qquad \begin{array}{c} alength(l_9(s(sK_2))) \\ \not\leq alength(l_{16}(sK_2)) \lor \\ Dense_{alength,l_9} \lor sK_2 \geq nl_9 \lor \\ \underline{\neg P(b(i(l_9(sK_2))))} \end{array}}{alength(l_9(s(sK_2))) \not\leq alength(l_{16}(sK_2)) \lor sK_2 \geq nl_9 \lor Dense_{alength,l_9}}\ \text{Res}$$

The following proof parts verify that sK_2 is bounded:

$$\cfrac{\begin{array}{cc} \multicolumn{2}{l}{alength(l_9(s(sK_2))) \not\simeq alength(l_{16}(sK_2)) \lor} \\ \underline{sK_2 \geq nl_9} \lor Dense_{alength,l_9} & \underline{sK_2 < nl_9} \lor Dense_{alength,l_9} \end{array}}{alength(l_9(s(sK_2))) \not\simeq alength(l_{16}(sK_2)) \lor Dense_{alength,l_9}}\ \text{Res}$$

Now we complete proving that sK_2 ia a loop iteration:

$$\cfrac{\underline{X_1 \geq nl_9} \lor alength(l_{16}(X_1)) = alength(l_9(s(X_1))) \qquad \underline{sK_2 < nl_9} \lor Dense_{alength,l_9}}{alength(l_9(s(sK_2))) = alength(l_{16}(sK_2)) \lor Dense_{alength,l_9}}\ \text{Res}$$

with mgu $= \{X_1 \rightarrow sK_2\}$.

$$\cfrac{\begin{array}{c} Dense_{alength,l_9} \lor \\ \underline{alength(l_9(s(sK_2))) = alength(l_{16}(sK_2))} \end{array} \qquad \begin{array}{c} Dense_{alength,l_9} \lor \\ \underline{alength(l_9(s(sK_2)))} \\ \underline{\not\simeq alength(l_{16}(sK_2))} \end{array}}{Dense_{alength,l_9}}\ \text{Res}$$

We derived $Dense_{alength,l_9}$ and thereby showed that `alength` is indeed *dense* inside the loop starting at location l_9.

Secondly we are interested in the timepoint $sK_1 : \mathbb{I} \rightarrow \mathbb{N}$ where `alength` has the value of sK_0, since this will be used to split the loop and mark the start for the *value evolution* lemma. For this we prove the premise of the instantiated *intermediate value* lemma for `alength` at line 9 with sK_0 ($PInt_{alength,l_9}(sK_0)$). This is a quite simple task since we already have $Dense_{alength,l_9}$. We start with the clausification and skolemization of the instantiated *intermediate value* lemma premise and conclusion:

Premise:

$$X_0 < alength(l_9(zero)) \lor \neg Dense_{alength,l_9} \lor PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9))$$

Conclusion:

{

$$\neg PInt_{alength,l_9}(X_0) \lor sK_1(X_0) < nl_9,$$
$$\neg PInt_{alength,l_9}(X_0) \lor alength(l_9(sK_1(X_0))) = X_0,$$
$$\neg PInt_{alength,l_9}(X_0) \lor alength(l_9(s(sK_1(X_0)))) = alength(l_9(sK_1(X_0))) + 1$$

}

To prove $PInt_{alength,l_9}(sK_0)$, we need to define following two clauses of the program semantics. The following clause models the initialization, that is the value at the start ($l_9(zero)$) of the loop execution, of `alength` with 0:

$$0 = alength(l_9(zero))$$

and the last value of `alength`, that is the value at the end of the loop execution ($l_9(nl_9)$):

$$alength(l_9(nl_9)) = alength(end)$$

First we make use of our *density* property:

$$\frac{\begin{array}{c}PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9)) \lor \\ \underline{\neg Dense_{alength,l_9}} \lor X_0 < alength(l_9(zero))\end{array} \qquad \underline{Dense_{alength,l_9}}}{PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9)) \lor X_0 < alength(l_9(zero))}\ \text{Res}$$

Now we show that `alength` was smaller than sK_0 at the start ($l_9(zero)$) and bigger than sK_0 at the end ($l_9(nl_9)$) of the loop execution. We start by using the semantics of the initialization of `alength`:

$$\frac{\begin{array}{c}PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9)) \lor \\ \underline{\underline{X_0 < alength(l_9(zero))}}\end{array} \qquad \underline{\underline{0 = alength(l_9(zero))}}}{X_0 < 0 \lor PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9))}\ \text{Dem}$$

and use $sK_0 \geq 0$ from the negated conjecture to show that it actually is smaller:

$$\frac{\underline{X_0 < 0} \lor PInt_{alength,l_9}(X_0) \lor X_0 \geq alength(l_9(nl_9)) \qquad \underline{sK_0 \geq 0}}{PInt_{alength,l_9}(sK_0) \lor sK_0 \geq alength(l_9(nl_9))}\ \text{Res}$$

with mgu $= \{X_0 \rightarrow sK_0\}$.

For showing that sK_0 is smaller than `alength` at the end of the loop execution we use the semantics of `alength` asserting that `alength` is not changing after the loop execution ($alength(l_9(nl_9)) = alength(end)$):

$$\frac{\underline{\underline{alength(l_9(nl_9)) = alength(end)}} \qquad \underline{\underline{sK_0 \geq alength(l_9(nl_9))}} \lor PInt_{alength,l_9}(sK_0)}{sK_0 \geq alength(end) \lor PInt_{alength,l_9}(sK_0)}\ \text{Dem}$$

and use $K_0 < alength(end)$ from the negated conjecture to derive the premise predicate $PInt_{alength,l_9}(sK_0)$:

$$\frac{\underline{sK_0 \geq alength(end)} \lor PInt_{alength,l_9}(sK_0) \qquad \underline{sK_0 < alength(end)}}{PInt_{alength,l_9}(sK_0)}\ \text{Res}$$

The premise predicate enables us now, using resolution, to derive the following facts from the conclusion part of the *intermediate value* lemma:

- $sK_1(sK_0) < nl_9$, a constraint on the iteration $sK_1(sK_0)$ of not being the last one;
- $sK_0 = alength(l_9(sK_1(sK_0)))$, an equation with the information about the iteration $sK_1(sK_0)$, when `alength` has the value sK_0;
- $alength(l_9(s(sK_1(sK_0)))) = alength(l_9(sK_1(sK_0))) + 1$, an equation stating, that `alength` is increased by one in the next iteration $(s(sK_1(sK_0)))$ after the iteration $sK_1(sK_0)$.

To finish step two, we want to show that after the iteration $sK_1(sK_0)$ the array element at position sK_0 of the array `a` is not going to change until the end of the program execution. To achieve this, we will derive the premise of the *value evolution* lemma with the help of the *iteration injectivity* lemma. Since we already showed a detailed proof using the *value evolution* lemma for the last property, we continue with a high level proof.

We instantiate the *iteration injectivity* lemma for `alength` at line 9, which we already have proven to be *dense*. We fix the first iteration to be our $sK_1(sK_0)$, which we retrieved from the conclusion of the previous derived *intermediate value* lemma and from which we know that `alength` is increased in the current iteration. Recall that the *iteration injectivity* lemma expresses that given an iteration $(sk_1(sK_0))$ in which a *dense* variable $(alength)$ is increased by one, the variable will never have the same value until the end of the loop execution (nl_9). This can be expressed as the following formula:

$$\forall it^{\mathbb{N}}.sK_1(sK_0) < it \leq nl_9 \rightarrow alength(l_9(sK_1(sK_0))) \not\simeq alength(l_9(it)))$$

Now we can use the equation from the conclusion of the *intermediate value* lemma $sK_0 = alength(l_9(sK_1(sK_0)))$ to derive the formula:

$$\forall it^{\mathbb{N}}.sK_1(sK_0) < it \leq nl_9 \rightarrow sK_0 \not\simeq alength(l_9(it)))$$

This formula expresses that in every loop iteration it, following the iteration $sK_1(sK_0)$, `alength` will never be equal sK_0. Since `alength` is used as index for the array `a`, this means implicitly for our program, that the value of `a` at index position sK_0 can not change after the loop iteration $sK_1(sK_0)$. Now to complete step two, we need to derive the premise predicate for the *value evolution* lemma instantiated for the array element at position sK_0, starting from the iteration after the assignment $(s(sK_1(sK_0)))$:

$$PE_{a[sK_0],l_9}(s(sK_1(sK_0)), nl_9)$$

Recall that for proving the premise predicate of the *value evolution* lemma we have to make an inductive step. Given an arbitrary iteration it, which respects the bounds $s(sK_1(sK_0)) \leq it < nl_9$, we have to show that the value of the array element stays the same until the next iteration $s(it)$. This inductive reasoning is handled by the program semantics and the *iteration injectivity* lemma conclusion. To prove it, we proceed with a case split for the iteration it:

Case 1) The predicate `P` is satisfied and the `if` branch is executed;

Case 2) The predicate `P` is unsatisfied and the `else` branch is executed.

Considering the first case, we know from our *iteration injectivity* lemma that the array element at position sK_0 of the array `a` is not visited again inside of the loop after the iteration $sK_1(sK_0)$. Hence the array element at position sK_0 at iteration it is still the same in the iteration $s(it)$.

For the second case, we use the semantics of the `skip` statement. This is quite trivial since `skip` has no impact on program variables, so again the array element at position sK_0 at iteration it is the same as in the iteration $s(it)$.

Since both cases satisfy the condition to use the conclusion of the *value evolution* lemma we can derive the formula:

$$a(l_9(s(sK_1(sK_0))), sK_0) \simeq a(nl_9, sK_0)$$

which expresses that the array element of the array `a` at index sK_0 has the same value at the end of the loop execution (nl_9) as it had at iteration $s(sK_1(sK_0))$.

Finally since `a` is not altered outside of the loop we use our program semantics together with the previous derived conclusion of the *value evolution* lemma to show:

$$a(nl_9, sK_0) \simeq a(end, sK_0)$$

Since `alength` is increased by one in iteration $sK_1(sK_0)$ and this can only happen in the `if` branch, we can backwards reason to prove `P` to be satisfied for the element at position $i(sK_1(sK_0))$ in the constant array `b`. We can link the element sK_0 from `a` to the element $i(sK_1(sK_0))$ from `b`. Using the program semantics with equality we can show:

$$P(a(l_9(sK_1(sK_0)), sK_0))$$

Combining the last two formulae, we can conclude that the element at position sK_0 of the array `a` not only satisfies `P` at the timepoint $l_9(sK_1(sK_0))$ but also at the end of the program execution end giving us:

$$P(a(end, sK_0))$$

This statement contradicts a part of the conjecture ($\neg P(a(end, sK_0))$) and we can derive the empty clause by resolution, which concludes our proof.

3.1.3 ∀∃ Quantifier Setting

Another interesting property for the program in Figure 3.3 is the relation between the copied elements in the integer array `a` and its original counterparts in the integer array `b`. As property we are expressing that every element in `a` has an equivalent element in `b` at the end of the execution. We formalize this as following formula:

$$\forall i^{\mathbb{I}} \exists j^{\mathbb{I}}.(0 \leq i < alength(end) \land 0 \leq blength) \rightarrow a(i, end) \simeq b(j)$$

The meaning and definition of *alength*, *blength*, *a*, *b* and *end* are unchanged to the previous section.

General Proof Idea

The proof for the ∀∃-property is actually quite similar to the proof for the ∀-property from Section 3.1.2. The negated conjecture we get in addition to the lemmas and program axiomatization is as follows:

$$\exists i^{\mathbb{I}} \forall j^{\mathbb{I}}.(0 \leq i < alength(end) \land 0 \leq blength) \land a(i, end) \not\simeq b(j)$$

To disprove this formula we show that for a fixed index i, respecting the given bounds, we can always find an index j, such that $a(i, end) \simeq b(j)$. As before, let us separate the proof in the following steps:

1. Show that `alength` is *dense*;
2. Show that after assigning a value to a specific index `i` in `a`, then `a[i]` is unchanged until the end of the execution;
3. Show that the value assigned to `a[i]` is equal to an element of the constant array `b`.

Since this proof is almost identical to the proof from the previous property of Section 3.1.2, we use the same methodology to produce our proof.

Proof

Due to the similarity of the proofs, we will reuse almost all parts of the previous proof sketch from Section 3.1.2. Since we only provided a high level overview of the *iteration injectivity* lemma we will provide a proof tree for this part now. First, we construct our skolemized form of the negated conjecture and compare it to the previous conjecture.

Clause set of the conjecture of the ∀∃ setting:

$$\{0 \leq sK_0,\ sK_0 < alength(end),\ 0 \leq blength,\ a(end, sK_0) \not\simeq b(X_0)\}$$

Clause set of the conjecture of the ∀ setting:

$$\{0 \leq sK_0,\ sK_0 < alength(end),,\ 0 \leq blength,\ \neg P(a(end, sK_0))\}$$

The rest of the program axiomatization stays the same. Comparing the conjectures the only different is the last clause, which in both cases is only effecting the last step of the proof.

We can thus reuse the proof of the previous property and assume $Dense_{alength,l_9}$ and the premise predicate $PInt_{alength,l_9}(sK_0)$ to be already proven. For providing a detailed proof tree for the *iteration injectivity* lemma we start with the skolemization clause representing the lemma in conjunctive normal form:

$$\neg Dense_{alength,l_9} \vee X_2 \geq X_3 \vee X_3 > ln_9 \vee$$

$$alength(l_9(s(X_2))) \simeq alength(l_9(X_2)) + 1 \vee alength(l_9(X_2)) \not\simeq alength(l_9(X_3))$$

Recall that the *iteration injectivity* lemma expresses that given a *dense* variable `alength` and a arbitrary loop iteration X_2, where `alength` is increased by 1, for all iteration X_3, between the bounds $X_2 < X_3 \leq ln_9$, `alength` is never equal to its value at the start of the iteration X_2 again.

First we combine the lemma with the fact that `alength` is *dense*:

$$\dfrac{\underline{\begin{array}{c} alength(l_9(X_2)) \not\simeq alength(l_9(X_3)) \vee \\ alength(l_9(s(X_2))) \simeq alength(l_9(X_2)) + 1 \vee \\ \neg Dense_{alength,l_9} \vee X_2 \geq X_3 \vee X_3 > ln_9 \end{array}} \qquad \underline{Dense_{alength,l_9}}}{\begin{array}{c} alength(l_9(X_2)) \not\simeq alength(l_9(X_3)) \vee \\ alength(l_9(s(X_2))) \simeq alength(l_9(X_2)) + 1 \vee \\ X_2 \geq X_3 \vee X_3 > ln_9 \end{array}}\ \text{Res}$$

Recall that we are interested in the iteration where `alength` has the value sK_0. We use the second clause of the derived conclusion of the *intermediate value* lemma to introduce sK_0 and the iteration $sK_1(sK_0)$ into our proof:

$$\dfrac{\underline{\begin{array}{c} alength(l_9(X_2)) \not\simeq alength(l_9(X_3)) \vee \\ X_2 \geq X_3 \vee X_3 > ln_9 \vee \\ alength(l_9(s(X_2))) \simeq alength(l_9(X_2)) + 1 \end{array}} \qquad \underline{sK_0 \simeq alength(l_9(sK_1(sK_0)))}}{\begin{array}{c} alength(l_9(s(sK_1(sK_0)))) \simeq sK_0 + 1 \vee \\ alength(l_9(sK_1(sK_0))) \not\simeq alength(l_9(X_3)) \vee \\ sK_1(sK_0) \geq X_3 \vee X_3 > ln_9 \end{array}}\ \text{Sup}$$

with mgu $= \{X_2 \rightarrow sK_1(sK_0)\}$.

We also use the third clause of the conclusion of the *intermediate value* lemma clause to show that `alength` is increasing in the iteration $sK_1(sK_0)$:

$$\frac{\begin{array}{l} alength(l_9(sK_1(sK_0))) \not\simeq alength(l_9(X_3)) \vee \\ \qquad sK_1(sK_0) \geq X_3 \vee X_3 > ln_9 \vee \\ \underline{alength(l_9(s(sK_1(sK_0))) \simeq sK_0 + 1} \end{array} \qquad \underline{sK_0 + 1 \simeq alength(l_9(s(sK_1(sK_0))))}}{\begin{array}{c} alength(l_9(sK_1(sK_0))) \not\simeq alength(l_9(X_3)) \vee \\ sK_1(sK_0) \geq X_3 \vee X_3 > ln_9 \end{array}} \text{ Res}$$

We derived a clause that expresses that for any iteration X_3 in the bounds $sK_1(sK_0) < X_3 \geq nl_9$, `alength` will never be equal to `alength` at the start of the iteration $sK_1(sK_0)$. The last step now is to show that `alenght` is never equal to sK_0 after the iteration $sK_1(sK_0)$ to use it for the *value evolution* lemma:

$$\frac{\begin{array}{l} \qquad sK_1(sK_0) \geq X_3 \vee X_3 > ln_9 \vee \\ \underline{alength(l_9(sK_1(sK_0))) \not\simeq alength(l_9(X_3))} \end{array} \qquad \underline{sK_0 \simeq alength(l_9(sK_1(sK_0)))}}{\begin{array}{c} sK_0 \not\simeq alength(l_9(X_3)) \vee \\ sK_1(sK_0) \geq X_3 \vee X_3 > ln_9 \end{array}} \text{ Sup}$$

Transforming the clause to use a implication and showing it with explicit quantifier, one can see much better what we derived:

$$\forall it^{\mathbb{N}} sK_1(sK_0) < it \leq nl_9 \rightarrow sK_0 \not\simeq alength(l_9(it))$$

The formula now enables us to say that for any iteration *it* after the iteration $sK_1(sK_0)$ until the end of the loop execution `alength` will never have the value sK_0. This is exactly what we discussed before and we can now use it in the same manner as before to derive the *value evolution* premise $PE_{a[sK_0],l_9}(s(sK_1(sK_0)), nl_9)$

Finally, we have to make a connection between the element of `a[`sK_0`]` to its original element of `b`. We needed the connection between $a(sk_1(sK_0), sK_0)$ and $b(i(sK_1(sK_0)))$ in the previous proof as intermediate step to show that $P(a(s(sk_1(sK_0)), sK_0))$ and $P(a(end, sK_0))$. All that is left to prove the current property, is to disprove the statement

$$a(end, sK_0) \not\simeq b(X_0)$$

which states that no element in `b` is equal to the element in `a` at position sK_0 at the end of the execution. We can simply use the previous made intermediate step and the semantics to fix the arbitrary iteration X_0 from the conjecture to be $i(sk_1(sK_0))$ and derive:

$$a(end, sK_0) \simeq b(i(sk_1(sK_0)))$$

which contradicts the conjecture and we derive the empty clause.

3.2 Concrete Safety Properties

Establishing the property proofs for the programs in Figure 3.1 and Figure 3.3 enables us to not only reason about those programs but a variety of programs related to the program classes. We can replace the generic property `P` with arbitrary properties. Examples are provided in Figure 3.4 and Figure 3.5.

```
func main() {
    const Int[] a;
    const Int[] b;
    const Int length;

    Int r = 0;
    Int i = 0;

    while(i < length) {
        if(a[i] != b[i]) {
            r = 1;
        } else {
            skip;
        }
        i = i + 1;
    }
}

```

Figure 3.4: Example program as an instance of Figure 3.1

3.2.1 Concrete ∃ quantifier setting

Figure 3.4 shows now a program that iterates over the indices of constant arrays `a` and `b`. We instantiate `P` with the equality of elements from `a` and `b` at the position of `i` at the current iteration. The program initializes an integer variable `r` with the value 0 and assigns the value 1 to it if the elements at position `i` of `a` and `b` are not equal. Accordingly, we want to prove (assuming termination) that if `r` has not the value 0 at the end of the execution, there exists an array index k such that the elements at the position k of `a` and `b` are equal. The program variable `length` represents the length of our arrays `a` and `b` and should therefore not be negative. We formalize our desired safety property as follows:

$$0 \leq \mathit{length} \land r(\mathit{end}) \simeq 1 \rightarrow \exists k^{\mathbb{I}}.a(k) \not\simeq b(k)$$

```
func main() {
    const Int[] b;
    const Int blength;

    Int[] a;
    Int i = 0;
    Int alength = 0;

    while(i < blength) {
        if (b[i] != 0) {
            a[alength] = b[i];
            alength = alength + 1;
        } else {
            skip;
        }
        i = i + 1;
    }
}

```

Figure 3.5: Example program as an instance of Figure 3.3

Proof

Since the proof is very similar to Section 3.1, it is a good opportunity to show the full proof output using our assisting proof tool SatViz. In Figure 3.6 one can see almost the same proof as in the previous $\exists$ setting. Since we used almost the same notation as the tool it should be no problem to follow the proof. The grayed formulae are derived clauses and the green ones are the produced axiomatization, the negated conjecture and the used *value evolution* lemma.

As one can see, the proof for the generic property can be very quickly extended to the proof for a concrete property with almost no additional complexity. The same holds for the other safety properties.

3.2.2 Experimental Results

Finally, we can use the knowledge of our previous results concerning the used lemmas to construct the axiomatization of our concrete examples, add the needed lemmas. We add only the needed theory axioms and let the safety property be verified automatically using Vampire [KV13]. For the experiments we use a special branch of Vampire, which prefers to pick our lemma clauses over other clauses. The SMT-files are created using the axiomatization of the program done by the Rapid [BEG$^+$19] framework and are then manually modified to only contain the needed lemmas and theory axioms. Also we turn off all built in axioms of Vampire so we have full control over the clauses needed for

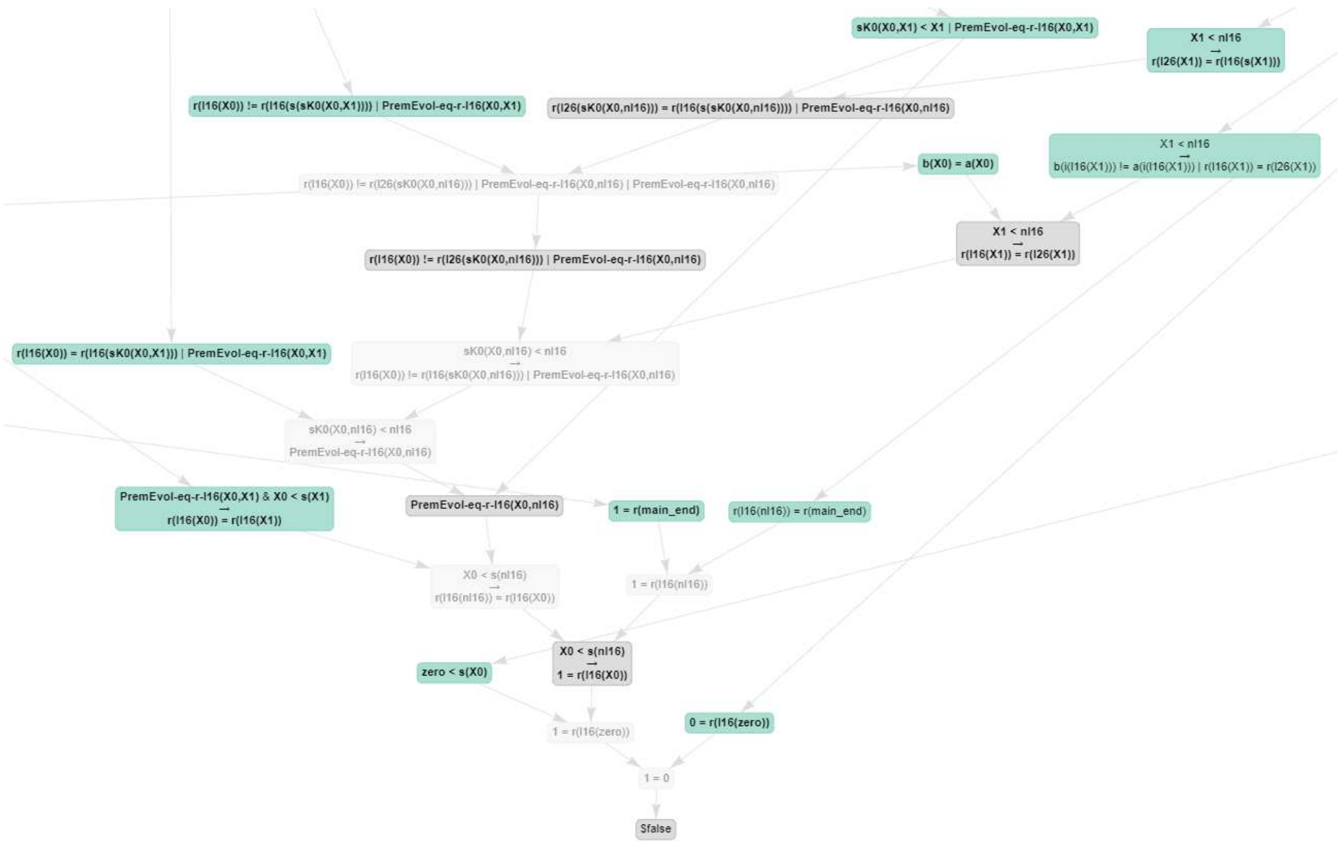
sK0(X0,X1) < X1 | PremEvol-eq-r-l16(X0,X1)
X1 < nl16 → r(l26(X1)) = r(l16(s(X1)))
r(l16(X0)) != r(l16(s(sK0(X0,X1)))) | PremEvol-eq-r-l16(X0,X1)
r(l26(sK0(X0,nl16))) = r(l16(s(sK0(X0,nl16)))) | PremEvol-eq-r-l16(X0,nl16)
X1 < nl16 → b(i(l16(X1))) != a(i(l16(X1))) | r(l16(X1)) = r(l26(X1))
b(X0) = a(X0)
r(l16(X0)) != r(l26(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16) | PremEvol-eq-r-l16(X0,nl16)
X1 < nl16 → r(l16(X1)) = r(l26(X1))
r(l16(X0)) != r(l26(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16)
r(l16(X0)) = r(l16(sK0(X0,X1))) | PremEvol-eq-r-l16(X0,X1)
sK0(X0,nl16) < nl16 → r(l16(X0)) != r(l16(sK0(X0,nl16))) | PremEvol-eq-r-l16(X0,nl16)
sK0(X0,nl16) < nl16 → PremEvol-eq-r-l16(X0,nl16)
PremEvol-eq-r-l16(X0,X1) & X0 < s(X1) → r(l16(X0)) = r(l16(X1))
PremEvol-eq-r-l16(X0,nl16)
1 = r(main_end)
r(l16(nl16)) = r(main_end)
X0 < s(nl16) → r(l16(nl16)) = r(l16(X0))
1 = r(l16(nl16))
zero < s(X0)
X0 < s(nl16) → 1 = r(l16(X0))
1 = r(l16(zero))
0 = r(l16(zero))
1 = 0
$false

Figure 3.6: Proof for the concrete ∃ quantifier setting in the SatViz tool

the proof. For the experiments Vampire version 4.4.0 with the following additional flags were used. That is, `-tha off`, to get rid of all unnecessary theory axioms and `-lls on`, to enable the mentioned preference towards selecting our lemmas.

The insights gained on the lemmas enables now to produce an automated proof for the mentioned properties. An interesting insight is that the quantifier alternation does not affect the overall proof performance compared to the ∀ quantified setting. Furthermore we observed that Vampire can handle the concrete ∀ and ∀∃ quantified properties way better than the general ones, due to the ability to process the inequality in the proof better than the uninterpreted predicate. These results underlines the usefulness of the proofs and the resulted knowledge for the generic cases as base for proofs of the related program classes.

CHAPTER 4

State of the Art

In this chapter we overview the related approaches to our work in software verification. This chapter is separated in two parts: the first one focuses on different methods for formal verification apart from theorem proving in general (Section 4.1). The second part gives detailed examples of novel and state-of-the-art methods for the verification of programs dealing with arrays (Section 4.2).

4.1 Software Verification

4.1.1 Deductive Software Verification and Proof Assistants

Deductive verification consists of generating a mathematical representation of the system and a mathematical description of its specifications. Proving the correctness of these implies conformance of the system with the given specification.

One of the most established and extended techniques doing so is *Hoare Logic* [Hoa69]. Hoare logic is a formal system with a set of logical rules, which can be used to reason about the correctness of programs. The idea is to reduce the program verification to proving so called *Hoare Triples*. A *Hoare Triple* is of the form $\{A\}P\{B\}$, where A and B are first-order formulae respectively expressing pre- and postcondition and P is a program. A *Hoare Triple* of this form is valid if and only if, if A is *true* before the execution of P, B is *true* after the execution of P.

Automating program verification is hard and resorting to proving the correctness of complex programs by hand is thereby sometimes inevitable. A lot of research has been carried out in this area to make this task easier and one answer to this are proof-assisting tools [dt04, Pau94, ORS92]. Isabelle [Pau94] is a generic proof assistant, which allows the expression of mathematical formulas in a formal language and provides tools for proving those formulas in a logical calculus. Isabelle has an extension for higher-order logic, Isabelle/HOL [NPW02], which for example can be used to model program semantics in Hoare logic and proving assertions with it [Nip02]. Furthermore, there exists the formal proof management system Coq [dt04]. It consists of a small kernel, based on a language with few primitives and on top of this a rich extendable environment for designing theories and proofs. Many verification tools show the versatility of the Coq proof assistant as a platform for verification. Two examples of this are CertiCrypt [BGZB09], an environment of formal proofs for computational cryptograph and Ynot [CMM$^+$09], a library to turn Coq into an environment for writing and verifying higher-order imperative programs. Another interactive tool used for software verification is PSV [ORS92], which stands for a *prototype verification software*. It uses higher-order logic as its specification language to specify libraries of theories. The theorem prover of PSV has a collection of basic inference rules, which are combined with higher level proof strategies and applied interactively within a sequent calculus framework. Proofs in PSV yield a proof script which can be manipulated and even replayed.

One of the biggest advantages of using proof assistants over automated tools is their ability to not only deal with more expressiveness like higher-order logic but also their very easily extensible environments in terms of theories, axiomatization or proof strategies used. The biggest downside compared to automated approaches is the time and expert knowledge needed to specify the needed theories and program semantics to successfully use them. Nevertheless, interactive systems can be very useful for prototyping automated solutions, as seen in this thesis with SatViz [GKS19] in combination with the interactive mode of Vampire [KV13].

4.1.2 SMT-Solving and Intermediate Verification Languages

Since software verification is a complex topic and no optimal approach has been found yet due to its undecidability, many solutions combine different techniques and tools into one framework. These frameworks are often designed to be very modular and extendable to experiment with different settings and tools. Most of the time a framework would take a programming language and transforms it into an intermediate verification language. This is done to separate the verification process from the programming language and makes verification of multiple languages with the same tool possible. Due to the great advances in the area of SMT-solvers, most verification frameworks utilize SMT-solvers in their proving process to verify these generated intermediate verification languages.

A successful approach to verify safety properties and an example of utilizing multiple tools ([KGC16, BNSV14]) is the SeaHorn [GKN15] framework. SeaHorn compiles a given C program with manually inserted assertions to optimized *LLVM* [LA04] byte code. The resulting byte code is transformed to Horn clauses, which serves as intermediate representation. Additionally to the Horn clauses, SeaHorn uses different abstract domains to generate program invariants. The Horn clauses and the generated invariants can then be solved by any off the shelf Horn constraints solver. The authors of [GKN15] claim that the flexibility, of being able to parameterize SeaHorn with different encodings and solvers, is one of the biggest strengths of SeaHorn. Although SeaHorn is a very sophisticated framework for the C programming language, it is hard to encode quantified properties. SeaHorn only uses C-like assertions inside the code and has no native support for ∀ and ∃ quantifiers, which limits its expressiveness compared to using full first-order logic.

A framework which offers a better opportunity for comparison is the programming language and verifier Dafny [Lei10]. Dafny has rich programming language features, translated to the intermediate verification language Boogie [Lei08]. The Dafny eco-system automatically infers some invariants of the provided program and generates verification conditions which can be passed to a SMT-solver. The eco-system provides an interface to various SMT-solvers and is thereby bound by the used solver, by default the SMT-solver Z3 [DMB08] is used. Dafny supports a variety of language features like classes, functions, sequences, sets, algebraic datatypes, ect. But it also comes with many other verification features, which are built into the language. Dafny is far more superior concerning its language compared to the other mentioned frameworks. Dafny does not only support in code assertions like SeaHorn, but takes this even further by allowing quantification inside of them, which is similar to our defined safety properties. Furthermore, it has native language support for quantified pre- and postconditions for functions and supports annotations for loop invariants and annotations to help verify termination of a program. Dafny provides the possibility of defining predicates inside the language and use them anywhere like a function which returns a boolean. This expressiveness within the language increases usability and readability. On the other hand, the automated invariant generation is not powerful enough to successfully reason about our example programs. This means that our example programs can not be automatically solved without additional expert knowledge, that is program specific invariants.

Another tool that makes use of an intermediate verification language is Why3 [BFMP11]. It comes with the programming and specification language WhyML and logic language Why3. Why3 is a first-order logic with polymorphic types and several extensions, for example recursive definitions or algebraic datatypes. Why3 has been successfully used as intermediate verification language for Java programs [MPMU04], C programs [MM18, BBCD16] and more. Similar to Boogie, the program can be annotated with pre-, postconditions and invariants, for which the the Why3 logic is used. The resulting representation is put through a chain of proof task transformations to produce a suitable input for a variety of solvers, e.g. Z3, Vampire or Coq. In contrast to the previous frameworks, Why3s plugin-like architecture makes it useable as a *meta* solver allowing for more flexibility in use, like switching between manually guided or fully automated solvers or writing custom transformations. Why3s ability to provide extra lemmas and using a refutation solver could be used to achieve a similar setup as we have with Rapid and Vampire but would require a lot of additional work and engineering.

4.1.3 First-Order Logic Approaches

In the past years not only advancements in SMT-solvers but also in theorem provers like Vampire [KV13] were made. Full first-order logic with its expressive power provides a good foundation for specifying program and especially array properties.

One state-of-the-art approach to encode and verify first-order properties is the Rapid framework [BEG+19]. This work and the formal characterization of *trace logic* is the foundation of my thesis. *Trace logic* is an instance of many-sorted first-order logic with equality. That allows to express properties over program locations and loop iterations. Rapid extends the idea of *Hoare Logic* by encoding a program P, with explicit timepoint reasoning and a relational property F into a set of first-order logic formulae. The validity of this set of formulae ensures that the program P satisfies such a property F. The program semantics are encoded as a set of program locations such that every program variable is encoded as an uninterpreted function over program locations. As opposed to standard *Hoare-Logic*-like semantics, program locations are extended with a notion of iterations allowing to refer to (arbitrarily nested) loop iterations. This expressivity allows for explicit timepoint reasoning. That is, we can formalize quantified inductive properties over loops in the language. Such invariants are referred to as *trace lemmas*.

Using first-order logic to reason about programs with loops has also been studied before by [GKR18]. This work describes a method to automatically reason about programs consisting of a simple loop. In contrast to the Rapid approach, the authors do not only consider partial correctness properties, but also temporal properties, in particular program termination. In [GKR18] the *first-order language of extended expressions* is used to describe desired properties and thereby gaining more expressiveness than by just using simple assertions as in programming languages like C, common in tools like SeaHorn [GKN15]. While C-like assertions can only express program properties for concrete values and array-bounds, thanks to the expressiveness of full first-order logic the *extended expressions* calculus allows reasoning over unbound arrays and arbitrary integer values. The method of [GKR18], implemented in Quit, is restricted to programs with simple loops, while Rapid can already handle programs with arbitrary nested while statements.

4.1.4 Abstract Interpretation

Static program analysis is the analysis of programs, which is performed without the actual execution of the programs. A well studied and extended technique to tackle static program analysis, is the concept of *Abstract Interpretation* [CC77, CC92]. The main idea is to create a sound approximation of the program semantics and partially executing it. The information gained for the abstract program behavior can then be linked to the behavior of the actual program. It gives the possibility to perform type, control-flow or data-flow analysis without the full knowledge of runtime values or the need to perform all calculations.

An abstract interpretation consists of a concrete and abstract domain, which are usually lattices, a concretization and abstraction function that form a Galois connection between the two domains and a sound abstract semantic function. The main challenge in abstract interpretation is to find the right abstraction domain without loosing too much precision but also keeping the computation cost low. These domains can be categorized into non-relational domains, for example interval domain [CC77] and congruence domain [Gra89], and relational-domains, for example polyhedral domain [CH78] and weakly relational domains [Min02]. Non-relational domains have the benefit of being very fast while losing a lot of precision, where in relational-domains we face the opposite problem. An example of extending this idea of abstract domains to the setting of arrays is *FunArray* [CCL10]. It uses a non-relational domain to categorize arrays in segments determined by their access. The work leaves space to extend the approach to deal with relations based upon these segments to gain even more information.

Summarizing abstract interpretation, choosing a too conservative approximating can lead to false negatives, that is the rejection of correct programs based on the violation of some constraints. Hence, balancing cost and precision for problem-specific domains is a widespread and still ongoing research and many problems require problem specific abstract domains to be efficiently solved. Compared to the approach presented in this thesis, we face similar problems by fighting with undecidability instead of false positives, due to over-approximation. The effort of finding and instantiating program specific trace lemmas can be compared to experimenting with specific abstract domains.

4.1.5 Model Checking

Furthermore, there exists the technique of *Model Checking* [CJGK+18] for automated software verification. In general, model checking is the process of determining wether a given formula f is true in a given model M. The main idea to use it in software verification is to view programs as finite state transition systems and specify the desired requirements in temporal logic. To verify certain safety or liveness properties all reachable states of our system are traversed. While the former declares what should not happen or what should always happen, the latter declares what should eventually happen. If a property is false, a counter example is generated. In case of a safety property we get a trace of states that led to the falsification. In case of liveness properties we get a path to an infinite loop which never reaches a specific desired state.

In the past, model checking was using Binary Decision Diagrams (BDDs) as decision procedure, which led to a huge state explosion [BCM+92]. Due to this problem, *Bounded Model Checking* [BCCZ99] emerged. Instead of using the exponentially growing BDDs as decision procedure, bounded model checking relies on propositional satisfiability (SAT). Looking at the iteration over arrays, we see a huge difference in the approach presented in this thesis and bounded model checking. Firstly, most of the tools, e.g. NuSMV2[CCG+02], are unable to deal with infinite arrays or missing information on bounds. Secondly, bounded model checking deals with loops by unrolling them to find counterexamples, which limits the approach to bounded loops. Looking at recent trends of (bounded) model checking, we can see a move towards SMT-solvers [AMP06, GG06, CFMS12] and away from SAT-solvers. One of the main advantages of using a SMT-based approach instead of a propositional one is the preserved structure and the ability to combined various decision procedures for the theories of uninterpreted functions, arrays, linear arithmetic or very specific operations on data structures like bit-vectors. The ability of bounded model checking to quickly explore a large portion of the state space and successfully revealing program bugs on short paths makes it a base component in many verification tools. Current state of the art tools implementing bounded model checking are CBMC[CKL04], ESBMC [CMNF12] or NuSMV2[CCG$^+$02].

Through the years bounded model checking was improved by extending it to induction [SSS00] (more precisely k-induction, e.g. PKind[KT11]) and various forms of abstractions [CGJ$^+$03, McM03, HJMS02, KGCC13]. Bounded model checking transforms the constraints of its transition system into conjunctive normal form. The SAT-solver either finds an assignment (i.e. a counterexample of at most k length) or a refutation proof that the clause set is unsatisfiable. In [McM03] the author used the refutation proof of the SAT-solver and *Craig-Interpolation* to create an over-approximation of reachable states and gained thereby an *Unbounded Model Checking* procedure that guarantees to find a proof (i.e. a counterexample) if one exists. Another successful extension of bounded model checking using over-approximation is counterexample guided abstraction refinement [CGJ$^+$03]. Model checking is applied on an abstraction of the program, which gets refined throughout the search for a valid counterexample. State of the art tools that utilize model checking together with abstraction technique is SLAM [BR01, BBKL10] and Spacer [KGCC13]. SLAM is used to verify safety properties of C programs. It abstracts C programs to boolean programs using C2BP [BMMR01] and uses the model checker Bebop [BR00] to find counterexamples. Since Bebop preserves the control-flow graph of the program, it is left to check if the path is feasible in the C program. If this is the case an actual error path is found, if not the boolean program is refined and another iteration of the process is started. An especially interesting abstraction technique is the one of Spacer. Spacer uses not only an over-approximation based on counterexample guided abstraction refinement, but also an under-approximation based on assumptions.

In general model checking has a lot of advantages. Nearly no human interaction and expert knowledge is required and a variety of great automatic tools are available. In case of a failure, model checking produce a counter example by design, which is not the case considering our approach of using a first-order theorem prover. In practice model checking offers good opportunities for program debugging and is very fast (especially on small counterexample paths). A big drawback considering model checking is the state explosion which affects even modern SMT-approaches due to unrolling. Another disadvantage is the fact that most model checking tools are limited to finite domains and finite systems. Comparing to our approach model checking is great for debugging finite systems but not suitable for dealing with proofs of quantified functional correctness.

4.2 Verification of Array Manipulating Programs

The problems and approaches of array verification do usually not differ from general software verification discussed before. Most of the tools mentioned are capable of dealing with arrays and verifying safety properties of programs using arrays. The challenges most of the tools face are handling interesting array properties, which are usually ∀-quantified properties, and dealing with unbound arrays, unbound loops and loop invariants.

An interesting method of dealing with array manipulating programs is by using induction. By induction over array related *ranks* [ISIRS20] or over the entire program [CGU20] dealing with loop invariants and unboundness is no longer an issue.
A novel approach of using induction to deal with unbound arrays is *inductive rank reduction* [ISIRS20]. The technique verifies the safety properties based on induction on a user-defined *rank* of the program states. A *rank* is the size of a program state, for example represented as the length of an array. The aim is to quickly establish a base case on a bound version of the program using bounded model checking or symbolic execution. Then a so called *squeezing function*, a function representing the induction step, has to be provided or generated. A *squeezing function* maps one program state to another program state with a smaller or equal *rank*, decreases the size of the state and thereby lifting the proof from a bounded case to an unbounded case. For the process to be sound a *squeezing function* has to satisfy some conditions (*initial anchor*, *simulation inducing* and *fault preservation*), which are verified using a SMT-solver. Since these functions are relatively simple for a simple program class, candidates can be created by using grammatically-correct functions and testing them against arbitrary generated examples. The technique was applied to a bidirectional summation program, which is very hard to proof using our approach, since we would need some understanding of the sum function. In contrast to our approach, the inductive reasoning in [ISIRS20] is not done over timepoints or iterations but on the size of the input state. Furthermore, *squeezing functions* are much simpler and more natural to come up with than invariants or inductive lemmas. On the downside, this approach is prone to small changes in the starting *rank*, which can lead to false positives.
The second method mentioned is the framework Vajra [CGU20]. It uses the parametric array size N to perform induction over the whole program, which is represented as parameterize Hoare triple $\{\phi N\}P_N\{\psi N\}$. Similar to the first approach a very simplistic base case is established by unrolling the loops for a fixed number of times and proven the correctness by using a bounded model checker. The induction step was handled by a novel algorithm utilizing an SMT-solver and is based on deriving the Hoare triple $\{\phi N\}P_N\{\psi N\}$ from the hypothesis $\{\phi N-1\}P_{N-1}\{\psi N-1\}$.
On one hand, using induction on specific program classes or combining them with other tools could lead to great results. On the other hand, these approaches scale and perform poorly with increasing complexity of the control flow, like nested loops, which yields no problem to the approach presented in this work.

One state of the art and highly refined framework for verifying programs manipulating unbound arrays is Booster [AGS14a]. It encapsulates a lot of techniques and combines

them to a powerful tool, which makes it a good candidate to represent the previous discussed techniques on model checking and abstraction. Booster takes as input a program with assertions which is parsed into a control flow graph and then transformed to a *cutpoint graph*. A vertex of a *cutpoint graph* represents the entry/exit block of the program or a loop-head. The edges are labeled with sequences of assumptions or assignments. The verification process of Booster can now be split into two parts.
First, the framework starts two rather lightweight analysis with the cutpoint graph as input. One is a rough analysis done by a bounded model checker with a low number of unwindings, which purpose is to quickly find safety violations of the given input. The second analysis tries to *accelerate* all the loops with the help of *Flat Array Properties* [AGS14b]. *Flat Array Properties* combines restrictions on quantifier prefixes and limitations on dereferencing to become a decidable fragment of the quantified theory of arrays. The authors introduced a class of programs called $\mathbf{simple}^0_A$-*programs*, which can be *accelerate*. These programs must have a flat control-flow structure, which means each location belongs to at most one loop, and comprise only loops that can be accelerated as a *Flat Array Properties*. This can be checked by a template-based pattern matching task given the patterns in [AGS14b]. If the input is indeed a $\mathbf{simple}^0_A$-*program*, all loops are substituted with their *accelerated* counterparts and the SMT-solver Z3 [DMB08] is used to solve the resulting queries. This concludes the first part of the verification.
The second part is started under the condition that neither the SMT-solver nor the bounded model checker were successful and it translates the cutpoint graph into a transition system. This transition system is then fed to an extended version of the MCMT[GR10] model checker, which contains an implementation of *Lazy Abstraction with Interpolants* for arrays [ABG+12] and *Acceleration Procedures* for array relations [AGS13]. Acceleration for array relations means in this context being able to express some classes of relation, involving arrays and counters, in first order language and thereby avoid reachability analysis divergence. Together with the abstraction and, in contrast to the precise process before, an over-approximation is created and refined to deal with the unbounded arrays.
One of the biggest strength of this framework compared to our approach is its complexity in the architecture. This helps the framework to take advantage of diverse syntactic particularities in a quick manner using *acceleration*. Especially the first verification part is a huge advantage when it comes to verifying the safety of $\mathbf{simple}^0_A$-*programs* or disproving safety properties with a counterexample, which is in generally hard for a theorem prover. On the other hand, our approach should perform equally good when it comes to certain programs with nested loops or a complex control-flow, since we do not rely on certain syntactic patterns.

Another approach focused on verification of programs with arrays is [DAFPP15]. The authors use constraint logic programs [JL87] as a metalanguage for representing imperative programs, their executions and the properties of interest. They presented a technique to generate a transition system as constraint logic program, which interprets the execution of the program. Additionally, they introduce a predicate *incorrect*, which represents all bad program states. If *incorrect* is now a consequence of an encoded program and its

initial configuration, then the property of interest does not hold for the program. The novel approach of this paper is the use of constraint logic programming transformations [EG96], especially *constraint replacement* for the array read write constraints, and their novel *generalization strategy for array constraints* for the introduction of new predicate definitions required for the verification of the properties of interest. The constraint replacement applies the laws of arrays and simplifies the constraints with array access to integer constraints. The new introduced predicate definitions corresponds to invariants and hold throughout the program execution. The generalization is achieved by using the widening and convex hull operators. The work is successfully implemented in the tool VeriMAP [DAFPP14].
Encoding the program into a logic representation and then using a solver to prove desired properties is closely related to the first-order logic and SMT-based approaches. The aspect of using widening and convex hull operators to gain some form of abstraction can be compared to abstract interpretation approaches. This is also the big difference to the approach we took in this thesis. We enable inductive reasoning by introducing *trace lemmas*, in contrast to the here applied method of generating invariants using abstraction.

Concluding on array verification one can say that, many different techniques are explored and experimented with, but no solution has been found that performs well on all problems. The picked state of the art approaches in this and the previous section cover a wide variety of methods with their strengths and weaknesses when it comes to (array) verification. The big difference to all of them compared to our method using Rapid are the *trace logic* lemmas and the way we handle the program semantics with timepoints. This gives us a good foundation for inductive reasoning and makes it very easy to coupe with complex control flow structures like nested loops and if statements.

CHAPTER 5

Conclusion

5.1 Conclusion

This thesis provides the theoretical basis to automate partial correctness proofs for a handful of properties over integer arrays on the basis of the RAPID framework [BEG$^+$19] using *trace logic* in combination with the superposition-based solver VAMPIRE [KV13].

By using a generic property $\mathtt{P}$, representing some property over arrays, we were able to generalize necessary lemmas and axiomatization needed to prove three safety properties of different quantifier settings: $\exists$, $\forall$ and $\forall\exists$. These proofs gave the theoretical basis to instantiate proofs for a whole class of programs by replacing $\mathtt{P}$ with an arbitrary property over arrays. All the proofs were conducted using SATVIS [GKS19] and the superposition based theorem proofer VAMPIRE. With the gained insides, we were able to achieve fully automated proofs of the generalize and concretized properties by small adaptations on the RAPID encodings and passing it to VAMPIRE.

This work and the resulting proofs illustrate how the inductive loop properties, expressed by the instantiate *trace lemmas*, are a perfect fit when it comes to reasoning about array manipulating programs in the superposition calculus. *Trace logic* and especially its timepoints, makes it possible to conveniently express the safety properties and the program semantics in first-order logic without any restrictions on quantifiers and loop nesting.

5.2 Challenges and Future Work

One of the main challenges of this work was the selection of the right *trace lemmas* for specific proof parts. Especially in the more complex $\forall$-quantified setting and quantifier alternation settings, a lot of the generated clauses related to the *trace lemmas* were not used in the proof. An improvement on this part could be some sort of pre-analysis of the program structure together with the safety property of interest to exclude some of the generated clauses from the clause set. Hereby the biggest challenge would be to not exclude needed clauses and making a proof impossible, but excluding enough to improve efficiency of an automated prover.

Additionally, one could always extend and experiment with the generated *trace lemmas*. One direction worth exploring could be identifying recurring patterns in (array) programs and generating more syntax specific lemmas to improve inductive reasoning on certain program classes.

Another line of work concerning Rapid is the extension of the input language $\mathcal{W}$ to more complicated constructs like function calls, pointers, or dynamic structures. This would require additional program semantics, *trace lemmas* and maybe even new theories. At the moment, Rapid can only handle one function with arbitrary loop nesting, which is in most real world applications not enough.

List of Figures

1.1 Example Program 2

2.1 SatViz example for parts of the proof for Figure 3.1 12

3.1 Example program with generic functional behavior involving ∃. 14
3.2 SatViz proof for the general ∃ property setting 21
3.3 Example program with generic functional behavior involving ∀ and ∀∃. . . 22
3.4 Example program as an instance of Figure 3.1 34
3.5 Example program as an instance of Figure 3.3 35
3.6 Proof for the concrete ∃ quantifier setting in the SatViz tool 36

Bibliography

[ABG+12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 46–61, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[AGS13] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, pages 23–39, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[AGS14a] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Booster: An acceleration-based verification framework for array programs. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 18–23, Cham, 2014. Springer International Publishing.

[AGS14b] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Decision procedures for flat array properties. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 15–30, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. In Antti Valmari, editor, *Model Checking Software*, pages 146–162, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BBCD16] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. Wp plug-in manual. *CEA LIST, Software Safety Laboratory*, 2016. https://frama-c.com/download/frama-c-wp-manual.pdf.

[BBKL10] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2: Static driver verification with under 4In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, page 35–42, Austin, Texas, 2010. FMCAD Inc.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, page 193–207, Berlin, Heidelberg, 1999. Springer-Verlag.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[BEG$^+$19] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei. Verifying relational properties using trace logic. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 170–178, 2019.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, May 2001.

[BNSV14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. Ikos: A framework for static analysis based on abstract interpretation. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 271–277, Cham, 2014. Springer International Publishing.

[BR70] J. N. Buxton and B. Randell. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, pages 113–130, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, page 103–122, Berlin, Heidelberg, 2001. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 08 1992.

[CCG$^+$02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, page 359–364, Berlin, Heidelberg, 2002. Springer-Verlag.

[CCL10] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL'11 - Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, December 2010.

[CFMS12] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Trans. Softw. Eng.*, 38(4):957–974, July 2012.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[CGU20] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 22–39, Cham, 2020. Springer International Publishing.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 84–96, New York, NY, USA, 1978. Association for Computing Machinery.

[CJGK$^+$18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and*

Algorithms for the Construction and Analysis of Systems, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[CMM+09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. *SIGPLAN Not.*, 44(9):79–90, August 2009.

[CMNF12] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. Context-bounded model checking with esbmc 1.17. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, page 534–537, Berlin, Heidelberg, 2012. Springer-Verlag.

[DAFPP14] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verimap: A tool for verifying programs through transformations. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–574, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[DAFPP15] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 140:329–355, 09 2015.

[Dij72] Edsger W. Dijkstra. *Chapter I: Notes on Structured Programming*, page 1–82. Academic Press Ltd., GBR, 1972.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[dt04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0, http://coq.inria.fr.

[EG96] Sandro Etalle and Maurizio Gabbrielli. Transformations of clp modules. *Theoretical computer science*, 166(1):101–146, 1996.

[GG06] Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, page 794–801, New York, NY, USA, 2006. Association for Computing Machinery.

[GKN15] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. Seahorn: A framework for verifying c programs (competition contribution). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–450, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop analysis by quantification over iterations. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 381–399. EasyChair, 2018.

[GKS19] Bernhard Gleiss, Laura Kovács, and Lena Schnedlitz. Interactive visualization of saturation attempts in vampire. *Integrated Formal Methods*, page 504–513, 2019.

[GR10] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 22–29, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, January 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[ISIRS20] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the squeeze on array programs: Loop verification via inductive rank reduction. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 112–135, Cham, 2020. Springer International Publishing.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 111–119, New York, NY, USA, 1987. Association for Computing Machinery.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, Jun 2016.

[KGCC13] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic abstraction in smt-based unbounded software model checking. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 846–862, Berlin, Heidelberg, 2013. Springer-Verlag.

[KT11] Temesghen Kahsai and Cesare Tinelli. Pkind: A parallel k-induction based model checker. *Electronic Proceedings in Theoretical Computer Science*, 72:55–62, Oct 2011.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–86, USA, 2004. IEEE Computer Society.

[Lei08] K. Rustan M. Leino. This is boogie 2. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/, June 2008.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[McM03] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Min02] Antoine Miné. A few graph-based relational numerical abstract domains. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, page 117–132, Berlin, Heidelberg, 2002. Springer-Verlag.

[MM18] C Marché and Y Moy. *The Jessie plug-in for deduction verification in Frama-C*. INRIA, 2018. Version 2.40, http://krakatoa.lri.fr/jessie.pdf.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[Nip02] Tobias Nipkow. *Hoare Logics in Isabelle/HOL*, pages 341–367. Springer Netherlands, Dordrecht, 2002.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, page 748–752, Berlin, Heidelberg, 1992. Springer-Verlag.

[Pau94] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.