Reasoning About Loops Over Arrays using Vampire

Loop Invariant Generation
using a First-Order Theorem Prover

Master's thesis in Computer Science

CHEN YUTING

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2016

MASTER'S THESIS IN COMPUTER SCIENCE

Reasoning About Loops Over Arrays using Vampire

Loop Invariant Generation
using a First-Order Theorem Prover

CHEN YUTING

Department of Computer Science and Engineering
Computer Science-Algorithms, Languages and Logic (MPALG)
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2016

Reasoning About Loops Over Arrays using Vampire
Loop Invariant Generation
using a First-Order Theorem Prover

CHEN YUTING

Master's thesis 2016:05
ISSN 1652-8557
Department of Computer Science and Engineering
Computer Science-Algorithms, Languages and Logic (MPALG)
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2016

Reasoning About Loops Over Arrays using Vampire
Loop Invariant Generation
using a First-Order Theorem Prover

Master's thesis in Computer Science
CHEN YUTING
Department of Computer Science and Engineering
Computer Science-Algorithms, Languages and Logic (MPALG)
Chalmers University of Technology
University of Gothenburg

Abstract

The search for automated loop invariants generation has been popularly pursued due to the fact that invariants play a critical role in the verification process. Invariants with quantifiers are particularly interesting for these quantified invariants can be used to express relationships among the elements of array variables and other scalar variables.

Automated invariant generation using a first-order theorem prover was first introduced by the work of Kovács and Voronkov [KV09a] in 2009. This approach employs a theorem prover, in our case the Vampire, as consequences inferencing engine and further to perform the symbol elimination for invariant generation. The entire approach can be separated into two phases: first, by performing static program analysis, one collects a set of static properties as static knowledge about the program and its variables. Second, these properties are sent to Vampire to infer invariants. This novel idea was further developed into a robust implementation introduced by the work of Ahrendt et al. [AKR15]. Our research originated from the idea of enhancing the existing implementation by adding in domain-specific theory. Particularly, we extended the work of Ahrendt et al. [AKR15] with first-order array theory reasoning.

Using first-order array theory, we present an extension on the existing automated invariant generation approach by this research. The extension aims at enhancing the reasoning process of automated invariant generation for loop programs over unbounded arrays. In addition to the extension on domain specific theory reasoning, this study also explored the enhancement of the static program analysis phase by proposing new static properties over the indexing variables. Experiment results compared with the previous implementation showed the improvement in reasoning over previously unprovable examples. The improvement came from both domain specific theory reasoning and the newly proposed static property.

Our study suggests the inclusion of domain specific theory can enrich the reasoning process of a theorem prover, in our case the first-order theorem prover Vampire. This enhancement can be further applied in program verification purposes such as automated invariant generation and direct proof of correctness. Also, with the theory-specific reasoning, the first-order theorem provers can deliver complex reasoning results containing quantifier alternation. We illustrate our approach on a number of examples coming from program verification.

Keywords: Loop invariant generation, First-order theorem prover, Array theory, Static program analysis, Vampire

Acknowledgements

I would like to first thank the supervisions provided by both professor Laura Kovács and Simon Robillard, their guidance kept me from getting lost in the vest field of formal methods. Special thanks to professor Wolfgang Ahrendt, for he inspired me with the lectures of formal methods. Finally, I'd like to thank Hedy for her gentle company, thank you for withstanding my temper and the continous encouragement throughout.

Contents

Abstract **i**

Acknowledgements **i**

Contents **iii**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Research Question 2
1.3 Project Objectives 2
1.4 Research Scope and Aims 3
1.5 Delimitations 3
1.6 Contributions 3
1.7 Research Impact in Society 4

2 Preliminaries **5**
2.1 Formal Definition of Loop Invariants 5
2.2 Correctness by Contract (Pre- and Post-conditions) 6
2.3 First-order Logic 6
2.3.1 First-order Logic Language 6
2.3.2 First-order Logic Theories 7
2.3.3 Array Theory 8
2.4 Automated Theorem Prover 9
2.5 The First-Order Theorem Prover Vampire 9
2.5.1 Alternating Quantifier and Skolemization 9

3 Literature Review **11**
3.1 Verification of the Loops 11
3.2 Literatures of Invariant Generation using Vampire 12
3.3 Other Approaches for Invariant Generation 12

4 Program Analysis **13**
4.1 Invariant Generation in Vampire 13
4.2 Preprocessing of Invariant Generation 13
4.3 Syntax: Simple Guarded Command Language 14
4.4 Extended Expressions 15
4.4.1 Assertion Language 15
4.4.2 Extended Expression Language 16
4.5 Extracting Loop Properties 17
4.5.1 Static Properties of Scalar Variable 17
4.5.2 Update Properties of Array Variable 18
4.5.3 Array Non-update 18
4.5.4 Monotonic Indexing 19

5 Reasoning with Vampire **20**
5.1 Logical Inferences using Vampire 20
5.2 Vampire and the FOOL: Polymorphic Arrays 20
5.3 First-Order Reasoning about Array Properties 21
5.4 Logical Inferences 21
5.5 Symbol Elimination 22
5.6 Reaching Correctness in Different Ways 22
5.7 Overview of Invariant Generation Workflow 23

6 Results **24**
6.1 Results of all test cases 24
6.2 Case study: swap 26
6.3 Other Invariants 27

7 Conclusion and Future Work **29**
7.1 Conclusion 29
7.2 Future Work 29

References **31**

1 Introduction

1.1 Background and Motivation

The correctness of a piece of code could be invaluable in the software-controlled modern days. To be able to ensure the correctness and formally verify the desired properties has long been the goal of the research communities of computer science. Yet the formal verification and its derivation can be overwhelming and far too verbose for manual efforts. Ideally, one would like to employ a program for automated verification process, as the step-wise concrete yet highly repetitive process is what computers are particularly good at.

Let us further motivate our research project with one concrete example. Here is a small imperative program with loop over arrays, written in Java-like syntax:

```
int [] A, B, C;
int a, b, c;
a = 0; b = 0; c = 0;
while (a < A.length) {
  if (A[a] >= 0) {
    B[b] = A[a];
    b = b + 1; a = a + 1;
  }
  else {
    C[c] = A[a];
    c = c + 1; a = a + 1;
  }
}
```

Figure 1.1: *An imperative loop over three arrays. The program, which we referred as the "`partition`" example , separates the non-negative values from the negative ones by element-wise copying into two result arrays.*

The program copies the non-negative integer values of array `A` into array `B`, and the negative ones into array `C`. While the semantics of the program is rather trivial, the control over array indices introduces the extra complexity. Furthermore, there exists another difficulty with the loop: exactly how many iterations will be carried out? Given the unbounded nature of loops and arrays, the loop iteration can be any arbitrarily big number. Although the termination problem is not our focus here (the loop iteration can be infinite, in that case it does not terminate if given infinite resources), one still wishes to establish some properties of the loop for the derivation of correctness. Here are some properties of the loop above:

1. Each element of the array `B`, starting from `B[0]` to `B[b-1]`, is a non-negative integer. The value equals to one element in the array `A`.
2. Each element of the array `C`, starting from `C[0]` to `C[c-1]`, is a negative integer. The value equals to one element in the array `A`.
3. Each non-negative element of the array `A` equals to one of element in the array `B`.
4. Each negative element of the array `A` equals to one of element in the array `C`.
5. Any element with index beyond the final value of `b` and `c` is not updated at the end of loop.

These properties capture the algorithmic nature of the loop while expressing our semantics of the program. Also, these properties are obeyed regardless the total iteration of loop, therefore they all evaluate to `TRUE` without the knowledge of iteration count. One also refers these properties as "loop invariants" given these properties do not change (always true) throughout the loop. However, to draw the connection between the program with these invariants often requires manual effort, demanding some insights of invariants extracting process itself and the semantics embedded in these invariants. Given another program with the same computational intention as the program above but with a different implementation, the invariants may no longer be true.

Therefore research efforts had been made aiming to provide an automated process of reasoning these invariants without any pre-defined guidance of the user.

One of the automated invariants generation approaches has been developed by the research team within our department. Using a first-order theorem prover, this approach is capable to derive complex loop invariants with alternating quantifiers. The line of development of our existing approach dated all the way back to 2009, when the work of Kovács [KV09a] first applied the symbol elimination method on automated invariant generation. This method relies on an efficient logical inferencing engine. In our case, it is one of the champions [SS04] in first-order theorem proving, Vampire [KV13]. In the continuation of the initial work, Ahrendt et al. [AKR15] further extended the idea to a more robust system. Not only can the new implementation generate invariants for a given program, it is now also capable to prove the correctness given the pre- and post-condition of the loop. Additionally, the new approach is interfaced toward real world programming languages via a intermediate language called *simple guarded command language*. Their work also included the translation tool from Java to the simple guarded command language, together with the connection with the JML verification tool, KeY [Ahr+05].

1.2 Research Question

Automatic loop invariant generation using symbol elimination has been initially demonstrated in the work of Kovács [KV09a]. This method was further improved in the work of Ahrendt et al. [AKR15]. The approach utilizes first-order theorem proving for loop invariant generation. With over half of the test cases listed in [AKR15] solved (11 solved out of total 20), this approach for automated invariant generation shows promising potential. By building on the existing implementation of [AKR15], **this project aims at enhancing the automated reasoning power of loops over arrays, hence improving the invariant generation**. From the previously failed test cases, we observed the theorem prover failing to provide the necessary logical consequences for invariant generations. Instead of treating the array variables as uninterpreted functions which take the indices as function arguments, we propose a novel approach, which directly encode array operations as their original semantics. This new approach provides clear distinction between reading and writing an array element. Additionally, the domain knowledge of array operations are supplied via the axioms of first-order array theory. We wish to find out if one can further extend the reasoning power of this existing system, particularly by extending the theorem prover Vampire, with domain-specific theory reasoning. In our project, used first-order theorem proving in combination with the polymorphic theory of arrays and improved this combination for improving invariant generation.

1.3 Project Objectives

With the aim to incorporate array theory into the existing system and conduct experiments to benchmark the resulting system, we listed the following research objectives:

- Study the first-order theories, especially the theory of array and understand how to reason with the axioms of array theory.
- Study existing approach, and understand how the loop invariant generation works in the current implementation, along with necessary details of Vampire.
- Utilize and incorporate the new Vampire branch, namely Vampire with FOOL [Kot+16].
- Incorporate the axiomatization of the array theory into the invariant generation process, including the necessary implementation work.
- Conduct benchmark experiments with the same set of test cases as in Ahrendt et al. [AKR15].
- Enhance the static program analysis and propose new static properties for improving program analysis in Vampire.

1.4 Research Scope and Aims

Our research aims at extending the existing method implemented by Ahrendt et al. [AKR15] with array theory reasoning. Apart from implementation works on the extension itself, our work is compared with the previous approaches by conducting the experiment over the same set of test cases. The invariant generation aims primarily on imperative loops over unbounded arrays, written in our simple guarded command language. Our simple guarded command language includes two primitive types, the integer and the Boolean type. We do not incorporate sophisticated paradigms such as object classes. This simple guarded language was developed with the existing system along with a translation tool providing the translation from Java programs with loop into this guarded command language. In our master project, we only focus on the invariant generation process hence the project does not aim to improve the translation of Java programs. However, since we do not alter the input syntax of the existing system, the translation between simple guarded language and Java is expected to be reusable without any modification. The only implication on the existing tool chain should be limited to the additional signatures accompanied with the theory of array (namely `select` and `store`). These additional signatures can be included in the invariants generated hence require additional care in the piping process.

1.5 Delimitations

Loop invariants can be coupled with the properties of sophisticated data structures as the program traverses through the data structure, resulting in sophisticated and domain specific invariants. In this master project, we focus on the primitive data structure, namely the arrays, and explore associated properties regarding the invariant generations. This project is conducted with the following delimitations in mind:

- We focus on the partial correctness on imperative programs with simple loop; only terminating programs are considered and we do not explicitly check for termination of the input programs.
- Simple loops are imperative loop without nesting. In addition, in our input syntax, only `while` loops are accepted.
- The program syntax is based on the simple guarded command language, same as the syntax introduced in [AKR15]. This syntax contains limited syntactical language constructs; program constructs such as `if b then t else s` are not supported yet. However, the conditionals can be captured using the guarded statements.
- The arrays studied in our project are limited to one dimensional arrays, meaning nested arrays (multi-dimensional arrays) are not considered. However, the unbounded nature of arrays is within the scope of this study.

1.6 Contributions

Our contributions in this master thesis project are listed as follows:

- As the foundation of our implementation, we extended the work of Kovács and Robillard [KR16] by including the array theory reasoning. For the explanation on implementation details, one can refer to Chapter 5 of this report.
- Comparing to the work of Kovács and Robillard [KR16], we also extended the static property in the program analysis phase. The detail explanation can be found in Chapter 4 program analysis. Specifically, we propose a new static property, the monotonic indexing, which can be found in Section 4.5.
- From the implementation perspective, we utilized the infrastructure of the latest Vampire branch by Kotelnikov et al. [Kot+16] for the polymorphic array implementation. Details regarding this infrastructure can be found in Chapter 5. Our contribution to the implementation can be found in Section 5.3.

- Our implementation is experimented with the same set of test cases as in Ahrendt et al. [AKR15]. The result shows the improvement in reasoning power with array theory. All experiment results can be found in Chapter 6, while detail examination over the generated invariants can be found in Section 6.2 and Section 6.3.

1.7 Research Impact in Society

In modern days, more and more equipments are controlled by software code. Software is reputable for its consistency over countless iterations of operation, it does not get tired or bored over the repetitions; nor does it automatically sense the subtle issues during and between these iterations. Occasionally, these subtle issues get aggregated and lead to catastrophic events. The chain of trust collapses like a domino toppling its neighbor, unexpected behaviour for a small piece of code with unforeseen corner cases results in total failure of a system. Apart from the direct cost of damage, it is extremely difficult to hold anyone responsible for the loss.

Unlike other linear imperative program constructs, imperative loops are difficult for human to reason syntactically. Yet the regular and iterative behaviour is what we seek in a software controlled system. The practicality of iterative programs are not replaceable by other means, hence one must be able to provide an assurance that these programs perform as intended. Luckily, the iterative nature bears large similarity with inductive mathematic reasoning. The bridging between formal deductions and computer programs requires careful joints.

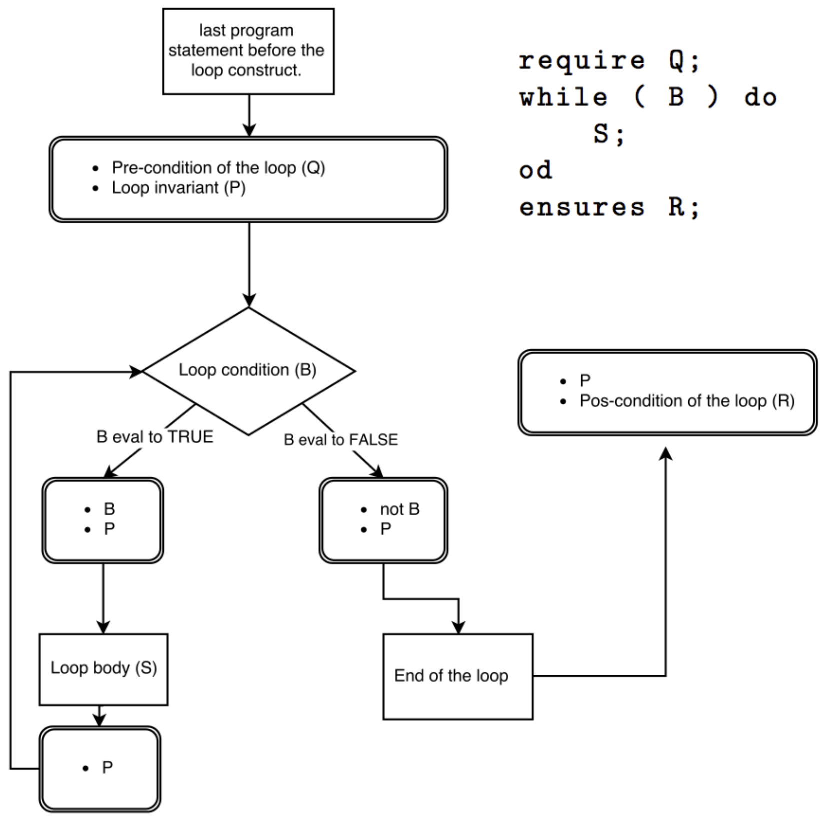
Our research aims at easing the need of manual annotations by providing automated process of invariant finding and further derivation of correctness. Yet the automation does not completely waive the responsibility of programmer for the assurance of correctness. An example would be the pre- and post-condition of the program, a set of suitable pre- and post-conditions, hence the assumptions and the intentions, must be provided by the programmer. Apart from providing the intentions of the program, it is also crucial for the programmer to pay extra care to the communication with others. The vague intention of a piece of code and misunderstanding can be yet another source of bugs.

We hope that the research result of our project can help the work of programmers as well as raise the awareness of responsible coding. The correctness may be automatically verified but to sustain the correctness over time requires more than just automated verification.

2 Preliminaries

2.1 Formal Definition of Loop Invariants

In order to further explore the idea of automated loop invariant generation, one must first formally define the notion of loop invariants.


last program statement before the loop construct.
Pre-condition of the loop (Q)
Loop invariant (P)
Loop condition (B)
B eval to TRUE
B eval to FALSE
B
P
not B
P
Loop body (S)
End of the loop
P
P
Pos-condition of the loop (R)

```
require Q;
while ( B ) do
    S;
od
ensures R;
```

Figure 2.1: *Flowchart of a simple **while-loop** program. The rectangles represent program statements while the rounded rectangles represent the properties hold in particular program state.*

Figure 2.1 shows a simple imperative loop program (found at the upper right corner) with the flow of program execution. The loop invariant `P` can be defined as follows:

1. `P` holds before the very first iteration of the loop, even before the first evaluation of loop condition.

2. After the loop condition gets evaluated, the loop invariant `P` still holds; in case of loop condition gets evaluated to true, the loop body `S` is executed, otherwise the loop terminated with `P & not B` holds.

3. After the execution of loop body `S`, loop invariant `P` still holds.

Summarizing the definitions above would lead to one conclusion: the loop invariant holds prior to the first iteration of the loop and holds before and after each loop iteration. Finally, the loop invariant still holds after the last iteration of the loop. This formulation makes the loop invariants constantly true regardless of the actual number of iterations carried out. It is this particular consistency of the loop invariants which one can leverage during the formal verification of a program with loops. More precisely, this definition of loop invariant is also called the *inductive invariant* in the literature, for it connects the post-condition of the loop and it is used in correctness reasoning.

2.2 Correctness by Contract (Pre- and Post-conditions)

To reason about the correctness of a given program one must provide the notion of correctness. A functionally correct program should achieve certain desirable properties while avoiding other undesirable ones. These properties need to be formally specified as the formal specifications, written in a formal language such as the first-order logic. The desirable outcomes are specified as the post-condition of the program, which must always hold for a program to of the program. Post-conditions can be used for describing the intended behaviour of a program so it performs the intended computation. On the other hand, pre-conditions can lift the restriction of the program by stating a condition assumed to be true prior to the program execution. In case of violation of pre-conditions, the program specification becomes undefined and no longer required to satisfy the post-conditions after its execution. Together, the pre- and post-conditions form a verification contract, which is used in formal verification of programs.

2.3 First-order Logic

Extending the notion of correctness by contract, one can observe that the connection between the contracts (the assumptions and the intentions) and the program must bridged by a form of formalization, preferably a language in which the desired assertions can be easily expressed. This formalization must be formal for the reasoning procedure to be sound, while the expressiveness of such formalization must be adequate for expressing the properties of interest. In the field of theorem proving, one often resolve this formalization problem using different forms of logic. In our case, Vampire reasons in the classical many-sorted first-order logic. In this following section, we briefly introduce the first-order logic and relevant theories to our research.

2.3.1 First-order Logic Language

Also known as the predicate logic, the language of first-order logic consists of:

- *Variables*: an unbounded set of variables, often denoted with lowercase letters $x, y, z, ...$
- *Logical symbols*: a set of predefined symbols, capturing logical operations such as conjunctions, disjunctions, implication, and quantifications. These symbols are part of the basic syntax of first-order logic and does not change their semantics under different interpretations.
  - *Logical connectives*: logical connectives similiar to propositional logic, including $=$ (equality), $\vee$(logical or), $\wedge$(logical and),$\neg$(logical negation), $\Rightarrow$(implication), etc.
  - *Quantifiers*: quantifier ($\forall$ or $\exists$) takes a variable name and a first-order formula to form a quantified first-order formula. The inclusion of quantifiers is the critical difference from the propositional logic, making first-order logic expressive enough for stating 1) a property holds for all possible values; or 2) the existence of particular element with specific properties.
- *Other symbols*: Apart from quantifications with quantifiers, first-order logic also extends propositional logic with the nonlogical symbols, which capture the abstraction of functions.
  - *Predicates*: also known as the relation symbols, predicate symbols come with their arity (≥ 0) and return either true or false based on the arguments applied. Predicate symbols are often denoted with uppercase letters $P(x), Q(x, y), ...$
  - *Functions*: function symbols come with their arity (≥ 0) and often denoted with lowercase letters $f(x), g(x, y), ...$

- *Constants*: a special case of function symbols, specifically describing the functions with arity 0.

A first-order logic *term* can be formulated from: 1) variables, 2) constants, or 3) expressions such as $f(t_1, t_2, ...t_n)$ (well-typed function application) where f is a function symbol with arity n. Examples of first-order logic term are listed as follows:

$$f(t_1, t_2, ...t_n)$$
$$3 + 4 - x = 0$$
$$f(3 * x) = g(x)$$

First-order logic *formulae* can be built from atomic formulae (predicate symbols with well-typed arguments applied) with logical connectives. We show few examples of first-order logic formulae here:

$$\exists x \in \mathbb{Z}.\, \forall y \in \mathbb{Z}.\, x > y$$
$$\forall x \in human.\, isMortal(x) \land bornEqual(x)$$

The *interpretation* maps every term without free variables to an value element in the domain. The interpretation gives the *semantics* to both the interpreted function symbols and the interpreted predicate symbols. On top of the interpretation is the structure. A first-order logic structure consists of the following:

- Set of sorts. (i.e. booleans, integers, arrays of integers); In our particular case, Vampire is a many-sorted first-order logic theorem prover.
- Set of function symbols with their corresponding function types and arities. Constants can be regarded as the function with 0 arity and has a fixed sort.
- Set of predicate symbols with their corresponding types and arities.
- An interpretation maps each sort in the signature to the domain of the sort. As the convention of the common rules of first-order inferences, all domains are assumed to be nonempty sets.

A formula A with free variables (variables not quantified by any quantifier) is satisfiable if and only if there exists structure I such that A holds in I.

2.3.2 First-order Logic Theories

With the interpretation defined, we move on to the axioms: a collection of formulae defined to be valid. The axioms effectively restrict the interpretations, only those interpretations which make the axioms valid are considered. This approach of classifying sets of interested interpretations is called *axiomatic approach.* Another way to describe interested interpretations classes is via the theories. First-order logic theories are interpretations in which some function symbols are interpreted with a fixed semantics. One common example is the theory of equality, works on the interpretations which the = symbol is interpreted as equality relation between two terms. On the theory level, one considers only the semantics of nonlogical symbols. A set of nonlogical symbols can therefore be called as the signature of the theory. Given a signature S, a first-order logic theory T consists of a set of axioms (without free variables) based on the nonlogical symbols in S. The formula A is said to be valid if and only if for all structures which satisfy the axioms of T also satisfy A.

Some theories such as theory of equality can be axiomatized by a set of axioms (reflexivity, symmetry, and transitivity). This process of defining the theory via a set of axioms is known as axiomatization. While some theories can be axiomatized into a concise form, not all theories can be finitely axiomatized (there exists no describable set of axioms), such as the theory of natural number.

Being more expressive than the propositional logic, classical first-order logic can describe many programming related properties while still perserve feasible computation complexity during automated logical inferences. Extending the propositional logic by the capacity of expressing objects in domain of interest, first-order logic contains both the abstraction over functions and the variable quantification. However, this extension on expressiveness comes with a tradeoff in the general decidability problem. While propositional logic is a decidable logical system, the first-order logic is semi-decidable. A logical system $\mathbb{L}$ is *decidable* if there exists an effective method $\mathbb{M}$ which can:

- If given an arbitrary formula A, one can use the method $\mathbb{M}$ to decide whether A is a theorem under $\mathbb{L}$.

By effective method, we meant a method which always terminates and produces only correct results. However, in a semi-decidable logical system $\mathbb{L}'$:

- There exists a method which can generate theorems under $\mathbb{L}'$.
- However, given an arbitrary formula A, one cannot find an effective method to check whether A is a theorem under $\mathbb{L}'$.

First-order logic belongs to the later group of logical system, there exists no effective way to check for the validity of arbitrary formula. Still, there are some well founded first-order theories which are decidable under the fragments of first-order logic.

2.3.3 Array Theory

Our research aims to experiment existing approach with the extension based on reasoning in the theory of array. The theory of array is an first-order theory with only two axioms (more accurately two axiom schemas for the underlying element type of the array). The signature of array theory consists of two function abstractions over array operations:

- `read` (a.k.a `select`): The binary read function takes an array `A` of sort τ and an index `p`. The read function then returns a value of sort τ which is the value of array `A` at position `p`.
- `write` (a.k.a `store`): The write function is a ternary function which takes an array `A`, an element with matching sort, and the index position where new element is stored into. The return value of write function is the modified array `A'`.

The axiom schemas are as follows:

- *read-over-write*: which states in case of reading the same position as writing, one can directly return the written value.

$$\begin{aligned}&\forall a : array_\tau \ . \ \forall i, j : Index \ . \ \forall v : \tau. \\ &\quad (i = j \ \rightarrow select(\, store(\, a, i, v\,), j\,) = v \wedge \\ &\quad (i \neq j \ \rightarrow select(\, store(\, a, i, v\,), j\,) = select(a, j)\end{aligned}$$

- *extensionality*: which states the two arrays are equal if they are element-wise equal.

$$\begin{aligned}&\forall a : array_\tau \ . \ \forall b : array_\tau \\ &\quad (\forall i : Index \ . \ select(\, a, i\,) = select(\, b, i\,)) \rightarrow a = b\end{aligned}$$

2.4 Automated Theorem Prover

Theorem provers are programs which reason about the validity of the given conjecture, based on the logical consequences derived from a given set of statements. The given statements are often referred as the "axioms" which are regarded as the "truth" by the theorem prover. The logical consequences are generated using the inference rules and hence they are inevitably true under the axioms. Broadly speaking, the theorem provers can be categorized into two major camps: the automated theorem provers (ATP) and the interactive theorem prover (ITP or sometimes referred as a proof assistant). From the user perspective, the value of automated theorem prover is the proof steps built by the theorem provers for the steps are the evidences of correctness. The proof steps form a tree of inferences, which also referred as the derivation of the goal.

The notion of soundness makes sure the inference rules used produce only the correct logical consequences. In other words, one can only trust the proof result of a theorem prover with sound inferences. On top of soundness, another useful notion of theorem prover is called completeness. If one theorem prover is complete, it is guaranteed to provide the proof if the proof exists, provided with unbounded resources (including unbounded computation time).

Typically, one would use theorem prover to show a specific statement (called the conjecture) is *logically implied by* of the set of axioms. One may also think the conjecture is the conclusion drawn from the user and the theorem prover is there to judge whether the conclusion is reasonable or not. The application area of conjecture proving can be directly associated with different fields such as mathematic conjectures or the conjecture of solvable state of a Rubik's cube (by providing valid moves of the cube as the axioms, the theorem prover can tell whether a given state of a cube can be solvable or not). However, the technology associated with ATP development has also been shown to be applicable in other more general areas of interest: ranging from hardware verification to program analysis.

The theorem provers (both ATPs and ITPs) can be further categorized by the language in which the axioms and conjectures are written with. In the case of classical first-order logic, well known theorem provers are E [Sch02], SPASS [Wei+09] and Vampire; whereas in the case of higher order logic one can think of systems like Coq [The04], Isabelle/HOL [Pau94] and Agda [Nor09].

2.5 The First-Order Theorem Prover Vampire

Vampire [KV13] is one of the automated first-order theorem prover. Specifically, it is a saturation-based theorem prover using superposition calculus. The conjecture is proved by refutation. By first negating the given conjecture and combining the negated conjecture with the axioms, Vampire tries to show the unsatisfiability of the bundled first-order clauses (axioms + the negated conjecture). This process is known as refutation because the negated conjecture is refuted by the theorem prover. Vampire is sound, in other words, the logical consequences produced are valid. However, the completeness of Vampire can be traded for other practical reasons, such as limited resources strategy [RV03] or ignoring the large and heavy clauses. The outcomes of a Vampire run can be either: (1) Refutation found with the proof steps of refutation process shown, meaning the conjecture is satisfiable under the axioms; (2) Refutation not found but time limit reached; (3) Saturation reached. The last outcome is rarely found in realistic problems, but it means the search of proof has reached a point called saturation. A set of clauses S reaches to its saturation closure,with respect to the inference system Inf, if for all inferences in Inf the resulting conclusions still belong to the set S. In other words, one cannot produce any new conclusion to add into the set S. The saturated outcome is only possible under a complete strategy, and if such outcome is reached, the saturated set is a witness of satisfiability of the original problem.

2.5.1 Alternating Quantifier and Skolemization

One of the special feature of Vampire is the capacity of efficiently handling formulae containing alternating quantifier ($\forall i, \exists j \Rightarrow ...$). This is achieved via a process called Skolemization. This process belongs to early stage of Vampire's problem handling mechanism and we do not explicitly alter this behaviour in our project. The idea behind Skolemization is using the Skolem functions to replace the existential quantified clauses. This way, the formulae containing alternating quantifiers still can be handled as if there exists only the one universal quantifier. The formulae without existential quantifiers are called Skolem Normal Form. Coupled with the Skolemization technique is the de-Skolemization, which recovers the existential quantifier in the formulae. De-Skolemized formulae are more human readable comparing to the Skolemized ones. However, it is shown to be beneficial for the goal of invariant generation in previous literature [KV09a] for keeping the Skolelmization functions as it is during invariant generation.

3 Literature Review

3.1 Verification of the Loops

Pioneered by the work of Floyd and Hoare [Hoa83] [Flo67], researches in computer science have been working on the verification of programs using formal methods. Verification is the formal approach to show a program is correct. A program is considered correct only if it satisfies the properties which deliver its intention. Like the proofs in math, the problem is twofold: one must first establish a formal notion which is expressive enough to encapsulate the underlining logic and then provide a systematic and reproducible proof using the formal notion. In [Hoa83], Hoare showed the correctness of a program can be deductively constructed with axioms and inference rules. Connecting formal logical assertions with the program constructs, one can reason about the program with Hoare-style verification. The simplest formulation can be:

```
Q { B } R
```

where

- `B` denotes the program body.
- `Q` denotes the assertions before executing the program body `B`.
- `R` denotes the assertions after the execution of program body `B`. `R` can be seen as the intention of the program.

This notion of combining logical assertions and program body was later widely adopted by the formal verification community and known as the "Hoare triple". On top of this notation, the corresponding inference rules are also introduced, including rules of consequences, composition and iteration. The concept of loop invariant was also introduced as loop invariants are needed for reasoning about the correctness of the loop and used in the rules of iteration. Also, the insightful idea of separating proof of partial correctness and proof of termination helps the focus of verification of loops.

Apart from the deductive reasoning, there exists another branch of formal verification: model checking. The verification processes work just like proofs as they connect the subject to its correctness. However, since the program verification targets only programs which are to be executed in computation machines (here we avoided the term "computers" for it is also an interest of research to seek for verification over large clusters of parallel computation devices), the "domain" of verification can be in favor of the prover. By exploring all the possible elements in the domain, one can effectively ensure the specification of a program is respected. The possible elements are corresponding to all the reachable states of a program. If a program satisfies its specification, one can also say this program is a model of such specification.

Whether taking the model checking approach or the deduction based reasoning, the process of program verification is verbose and tedious throughout most of the proof. This motivated the search of automatic program verification, a program which checks the correctness of other programs. This idea of automation collided head-on into the theory of computability as it is in general not decidable; even just to create a program to determine whether another program will eventually terminate is impossible in general, as stated by the famous halting problem. Yet this hard boundary does not prevent researchers from devising automatic program verification tools, the benefit of formally verified programs is simply critical as more and more technologies depends on the control of programs. Emerson and Clarke [CES86] and Quell and Sifakis [QS82] were the first to devise tools aiming to automatically check a program with its specification which is encoded in temporal logic. Their works got around the computability problem by aiming solely the finite-state fragment of all possible programs. This finite-state fragment effectively constrained the subject programs such that automatic reasoning always terminates. However, the finite-state fragment also limited the data types available. Only bounded data types can be contained in this fragment. These tools took the model checking approach and basically iterate throughout all reachable states of the program. The computation complexity inevitably grows as the reachable states expand, this is a problem so-called state space explosion. This potentially exponential growth hinders the use of model checking algorithms in larger examples, counter measures such as bounded model checking by Clarke et al. and abstract interpretation by Cousot and Cousot [CC77] are proposed.

3.2 Literatures of Invariant Generation using Vampire

While automated program verification is the ultimate goal, let us first focus on the automated loop invariant generation. The idea of using a theorem prover for invariant generation was first introduced in [KV09a]. The work in 2009 combines static program analysis techniques from previous works, such as the `Aligator` [Kov08], with the first-order theorem prover Vampire. The consequences generated can contain auxiliary symbols introduced for program states description and should not be treated as valid invariants. This need leads to a technique called ”symbol elimination” [KV09b]. The terms with auxiliary symbols are assigned with a large Knuth-Bendix [KB83] ordering weight. Resolution theorem provers are geared toward performing inferences with heavy symbol premises. This way, the heavy symbols can be removed from the search space faster, leaving lighter search space and hence faster proof search. Although the weighting was initially introduced for the purpose of efficiency, it also works well for the purpose of gradual elimination of designated symbols. In [AKR15], this approach was further extended for general syntax and better leveraging of the refutation proof system. Apart from these cited literatures, our project also benefits directly from [Kot+16] for the polymorphic array implementation and built-in array theories. The extensionality of array theory, which specifies the equality of two arrays, is originated from the work of Gupta et al. [Gup+14].

3.3 Other Approaches for Invariant Generation

Automated invariant generation has long been a popular topic of research. Apart from using theorem prover for logical inferences, other approaches are proposed under different reasoning frameworks. Based on the abstract interpretation framework proposed by the work of Cousot and Cousot [CC77], [Cou03], [FQ02], [Bla+03] infer the loop invariants by soundly approximating the semantics of a given program. This approach focuses on the abstract domains, providing invariants over different kinds of aims such as violation of specific properties (i.e. division by zero).

Another approach for automated invariant generation is based on constraints. By solving the constraints over a set of pre-defined templates, this approach transforms the inferencing problem into a constraint solving problem. One of the representing work is by Gupta and Rybalchenko [GR09]. This approach works with decidable logic fragments, and the templates involving quantification is not fully supported.

Another interesting insight regarding invariant generation can be found in the work of Furia et al. [FMV14] [FM10]. The intrinsic value of loop invariant is the connections to post-condition during verification. Similar to the approach of a human user writing the loop invariants manually, their approach mutates the post-condition and filter out the valid invariants from the mutations. This idea of post-condition mutation was further extended in the tool DynaMate [Gal+14a] [Gal+14b]. DynaMate represents the counterpart of our approach, instead of static analysis, DynaMate utilizes the dynamic analysis for invariant generation. During the analysis, Daikon [Ern+07] is used for invariant detection.

4 Program Analysis

4.1 Invariant Generation in Vampire

Using saturation theorem provers like Vampire, one can automatically generate loop invariants which are logical consequences of properties discovered by the static program analysis. This novel approach has been demonstrated both efficient in computation time and powerful in invariant derivation by the works of [AKR15] and [KV09a] . Particularly, using Vampire as the logic consequences generating engine provides the possibility of generating invariants with quantifier alternations. As Vampire itself is a fully automatic theorem prover, the invariant generation requires no user guidance such as templates. In this chapter, we introduce the hierarchy of invariant generation using Vampire in system level, followed by detailed explanation of each component steps.

The invariant generation process can be split into two main phases: the static analysis of programs and the logical inferencing of loop invariants. The two phases are functionally separated, hence it is possible to replace each phase with another equivalent process, such as replacing Vampire with other provers or enhance the preprocessing with other domain-specific knowledges. In the following chapters we will look into the detail of each phases separately. In Chapter 4, we focus on the preprocessing phase of our system, namely the static program analysis and property extraction, along with detailed introduction to our simple guarded command language and its semantics. **Specifically, we present the proposed new static property in Section 4.5.4**. In Chapter 5, we explain the logical inferencing process of Vampire and how one can reason about imperative loops over arrays in the Vampire framework.

4.2 Preprocessing of Invariant Generation

Prior to the logical inferences with saturation theorem prover, one requires to analyze the given program instance and create the logical equivalence of the program instance's semantics using the formal logic of the saturation theorem prover. In our particular case, Vampire performs logic inferences on first-order logic formulae, hence we must be able to express the semantics using first-order logic formulae. From the formulae formulated, Vampire can then infer more logic consequences on its own, hoping to generate the loop invariants or the consequences implying the post-conditions. This preprocessing step looks into the program instance and formulate the relations between program variables and the program states using first-order formulae. This step is often referred as program analysis in a correctness verification context (while program analysis in general can also discuss the optimization of programs, we only focus on the program correctness here). The goal of this preprocessing step is to automatically analyze and extract properties of the given program. Program analysis can be performed either dynamically or statically. Unlike dynamic analysis, our approach does not actually execute program statements, hence belongs to static program analysis. The phase of program analysis includes static program analysis for the properties extraction. The aim is to construct the connection between syntax (variable symbols) and the semantics (the semantics of array index operations are captured by the static properties). Specifically, our preprocessing can be further separated into the following tasks.

- Lex and parse the input program. In our implementation, we used standard C++ scanner generator flex and parser generator Bison for the simple guarded command language. A detailed definition of the syntax can be found in a later section.

- Formulate the pre- and post-conditions into first-order formulae if they are specified by the user. These formulae are created as first-order formulas (using Vampire terms). Associate the variable symbols with corresponding sorts while encode the program variables into *extended expressions.* Extended expression language is an extension on the assertion language, which helps us capturing the program state in which the update of a particular variable occurs. More detailed introduction can be found in the later section of this chapter.

- Perform static analysis over the input program instance and extract static properties of the variables. In our approach, we do not execute any part of the actual code from the input program instance. These properties are encoded using the extended expressions.

- Bundle the extracted properties and send to theorem prover for invariant inferencing.

  From the work of [AKR15], we learned that the formulated pre- and post-conditions can be further exploited during later stage of invariant generation. Also, the program analysis step described here can be extended with additional static property capture. One of our contribution in this master thesis work is the extension of a new static property capture, the *monotonic indexing* property which will be introduced in later section. The entire program analysis and the idea of capturing static program properties can be implemented as a stand-alone process; but in our approach, the program analysis shares the same code base with Vampire itself, making the development more convenient. Finally, the type of static properties captured can be adjusted as the different interests of program instances one would like to analyze; in our case, these static properties are targeting the operations and behaviors or array indices, allowing the logical inferences to reason about program variables and the array indices.

4.3 Syntax: Simple Guarded Command Language

The input of our system is in a simple guarded command language. Differing from the famous guarded command language discussed in the work of Dijkstra [Dij75], our simple guarded command language is deterministic. We support both scalar variables and array variables with two primitive types, int for the integers and bool for the booleans. Standard arithmetical functions symbols such as +, - ,* , / and predicate symbols $\leq$ and $\geq$ are used. Given an array variable `A`, we denote `A[p]` as accessing the array element of `A` at position `p`, corresponding to array read/select. The loop consists of a loop condition and an ordered collection of guarded statements as the loop body. A loop condition is a quantifier-free boolean formula, while each guarded statement consists of a guard and a non-ordered collection of parallel assignment statements. The ordered collection of guarded statement is checked in sequential order and each guard is assumed to be mutually exclusive to other guards. Further, the non-ordered collection of parallel assignment statements are assumed to respect the following rules:

1. In case of scalar variable assignment, there cannot exist two parallel assignments to the same scalar variable within the collection.

2. In case of array variable assignment (which corresponds to array write/store), the same position cannot be assigned to two values, i.e., if two array assignments `A[i] := e` and `A[j] := f` occur in a guarded statement, the condition i $\neq$ j is added to the guard.

Pre- and post-condition can be specified apart from the loop, using requires and ensures keywords respectively. Conditions are boolean formulae over program variables and quantifiers are allowed. Finally, all types must be declared upfront of the program.

The semantics of our simple guarded language is defined using the notion of state. Each scalar and array variable is mapped to a value of correct type by the program states. In our setting, a single program state corresponds to each loop iteration. Assuming n denotes the upper bound of loop iterations, at any loop iteration i which $0 \leq i < n$, we have σ_i as the program state. σ_0 and σ_n represent the initial state and final state of the loop respectively. Once the guard is valid, the associated collection of parallel assignment statements will be applied simultaneously to the program state. For example, executing the guarded statement

```
true -> x = 0; y = x;
```

in a program state where `x = 1` holds will result in a new program state with `x = 0 & y = 1`.

The syntax of our simple guarded command language can be best explained by Figure 4.1. All variables, including arrays and scalars, are declared upfront. The declaration is followed by user specified pre-conditions using the keyword `requires`. In this example, all indexing variables `a`, `b` and `c` are limited to be initialized with their values equal (value equality is denoted by `==`)

to zero. The last scalar variable, `alength`, is limited to positive values only. The pre-conditions are followed by the post-conditions, also user specified. The keyword for denoting post-conditions is `ensures`. While in this particular example, both pre- and post-conditions are specified by the user, their presences are not required for our approach to generate loop invariants. Our approach can utilize these contracts to provide better user experience, such as invariant filtering and direct proof of correctness. For further detail regarding the treatments of the pre- and post-conditions, we refer to **Section 5.5** of this thesis. While most numerical operations take the usual representing syntax, logical implication uses the `==>` syntax. Inside the loop body, each guarded command starts with double-colon followed by the quantifier-free boolean guard. The guard is separated with the collection of parallel assignments by `->`. Finally, each assignment is terminated by the semicolon.

```
int [] A, B, C;
int a, b, c, alength;

requires a == 0;
requires b == 0;
requires c == 0;
requires alength > 0;

ensures forall int i, 0 <= i & i < b ==> B[i] >= 0;
ensures forall int i,
          exists int j, 0 <= i & i < b ==>
                          0 <= j & j < a & B[i] == A[j];

while (a <= alength) do
  :: A[a] >= 0 -> B[b] = A[a]; a = a + 1; b = b + 1;
  :: true -> C[c] = A[a]; a = a + 1; c = c + 1;
od
```

Figure 4.1: *The running example of our thesis, `partition`, is expressed using our simple guarded command language. The second guard true functions as the final conditional guard, similiar to the otherwise construct found in other languages.*

4.4 Extended Expressions

While the goal of invariant finding is to generate properties that hold at arbitrary iteration of the loop (think of the definition of invariants), the process of reasoning and analyzing the program instance could be improved with the help of indicators marking the program states. These program states are the critical states in which variable values get updated. With this intuition, we found the need to formulate a stronger language for describing the intermediate program states. This language must be also capable to facilitate the expressions relating program variable values in respect of the program state. With the extended language, which we called the extended expressions, one can express richer properties of the loops. Still, it is important to remind ourselves that the invariants we are after are meant to be the properties which hold at arbitrary iteration of the loop. This explains the need of the symbol elimination technique we used in our approach, which will be formally introduced in the next chapter. In this following section, we first introduce the basic assertion language, which is followed by the introduction of the extended expressions.

4.4.1 Assertion Language

The assertion language is meant to express the properties as classical first-order logic formulae, hence one should be able to correlate the two without additional knowledge. For each scalar variable v of type τ used in the program instance, we create two corresponding symbols in our assertion language: $v : \tau$ and $v_{init} : \tau$. The introduction of $v_{init} : \tau$ allows us to state the initial value of a scalar variable, while the $v : \tau$ allows us to state the general value of v. Like the interpretations in first-order logic, the interpretations in our assertion language also gives the value of each symbol. In the assertion language, the interpretations depends on a given program state α. Assuming the initial program state is represented by α_0 and the final program state is represented by α_n, we can already know the following:

$$\begin{aligned} &\forall i, \quad 0 \leq i \leq n \Rightarrow \\ &\qquad v_{init} : \tau \quad of \quad \alpha_i == v : \tau \quad of \quad \alpha_0 \end{aligned} \tag{4.1}$$

An example of assertion language encoding can be drawn from the running example in Figure 4.1, since the value of a, b, c are assumed by the `requires` statements, one can translate the assumptions into their initial values:

$$a_{init} = b_{init} = c_{init} = 0$$

In the previous work [AKR15], the array variables are captured in our assertion language using the function symbols of type $\mathbb{Z} \rightarrow \tau$, where $\mathbb{Z}$ stands for integers which is used to capture the indexed access. This is no longer the case in our new implementation since we do not treat the arrays as functions taking indices anymore. The array variables are now captured using the same way as in scalar variables described above. This change also improves the readability in the codebase since now all the variables are equally handled. Finally, since the assertion language follows classic first-order logic, the pre- and post-conditions, which are in the first-order logic, are trivially describable using the assertion language.

4.4.2 Extended Expression Language

In order to allow the theorem prover leverage on the findings of each program states and draw invariant conclusion from these findings, we must enrich the underlying language for this expressivity. In other words, the program state α described in the assertion language must be incorporated into the extended expressions directly, making the reasoning possible without additional interpretation needed. The extended expressions encode the program state directly into the symbols. For each variable v of type τ, the extended expressions includes a function symbol of type $\mathbb{Z} \rightarrow \tau$, where the first argument of the function denotes the program state. Although the function type takes an integer $\mathbb{Z}$, the actual implementation has a non-negative program state limitation. The program state represents the number of iteration in which an update takes place. v^i denotes the application of the introduced functions, and is interpreted as the value of variable v in the program state α_i. In the previous implementation, array variables are expressed with an additional arity denoting the index accessed. This is no longer in our new implementation as the array is only accessed via `select(array, index)`. Finally, the extended expressions includes an additional symbol n, denoting the upper bound of loop iteration count. Notice the introduction of iteration upper bound n does not correlate to a defined number of iteration limit, the symbol n is introduced merely for the symbolical reasoning. One can substitute the symbol with another such as $final$. Examples of extended expressions are listed as follows:

$$v^{i+1} = v^i + 1$$

One can reason about the value of a variable between two program states. Two program states (first iteration and last iteration) are given special symbols ($init$ and n). The following example shows the final value of v remains the same as the initial value of itself.

$$v^n = v^{init}$$

Following the semantics of extended expressions, the following equalities between the assertion language terms and the extended expressions are naturally true:

$$v_{init} \equiv v^0$$

$$v_n \equiv v^n$$

$$\texttt{select(}\ A^{init}\ \texttt{, p)} \equiv \texttt{select(}\ A^0\ \texttt{, p)}$$
$$\texttt{select(}\ A\ \texttt{, p)} \equiv \texttt{select(}\ A^n\ \texttt{, p)}$$

Properties expressed using the extended expressions are called extended loop properties. These extended loop properties are the central idea of the process symbol elimination, which will be introduced in the next chapter. The idea is to first extract the static properties using this extended expressions and then ask the theorem prover for logical consequences which can be expressed in assertion language. Although the extended expressions can represent stronger properties, we only regard those invariants in the assertion language valid. This is due to the fact that the pre- and post-conditions are in the assertion language and the invariant in extended expressions can describe properties of intermediate steps. With the extended expressions introduced, we now look into the static property extraction and how the properties can be written in extended expressions.

4.5 Extracting Loop Properties

With the extended expressions, one can now express the variable value at a particular program state. The next step is to statically examine the input program and try to extract useful consequences from the program itself. We denote these consequences from static analysis the loop properties. Loop properties came from mathematical observations of the updates made to each variable. Properties of each variable are extracted and formulated in first-order formulae (in our implementation, these properties are formulated as additional axioms for the theorem prover) before sending to the first-order theorem prover. The properties extracted can be significantly helpful for deciding the final invariant generation as certain static properties (and the form of these properties) ease the work of the theorem prover. In this section, we overview the existing static properties extracted and introduce the new static property added in this master project, the monotonic indexing property. Asides from the static properties collected, if the user provides the pre-conditions, the pre-conditions are directly translated into extended expressions. Since any expression in the assertion language is also in the extended expressions with minor translation needed. Here is a concrete translation example:

```
requires forall int i, 0 <= i & i < alength ==> A[i] > 0
```

is translated into the following property in extended expressions and added into the Vampire problem instance:

$$(\forall i)(0 \leq i < \mathit{alength} \Rightarrow \mathit{select}(A(0), i) > 0)$$

4.5.1 Static Properties of Scalar Variable

Given a scalar variable `v`, we call it increasing or decreasing if the following extended property holds:

$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} \geq v^i \text{ /* v is increasing */}$$
$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} \leq v^i \text{ /* v is decreasing */}$$

For a scalar variable which is either increasing or decreasing, we also call it monotonic. This analysis for monotonicity can be achieved via light-weight analysis, such as verifying that every assignment to `v` is of the form `v = v + e`, where `e` is a non-negative integer.

Apart from monotonicity checking, a scalar variable is called strict if the following extended properties holds:

$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} > v^i \text{ /* v is strictly increasing */}$$
$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} < v^i \text{ /* v is strictly decreasing */}$$

Since our guarded command language only supports integer sort for numerical scalars, we further call an increasing scalar variable dense if the following property holds:

$$\forall i \in \textit{iteration}, 0 \leq i < n \Rightarrow |v^{i+1} - v^i| \leq 1 \text{ /* v is dense*/}$$

These static analyses are geared toward reasoning about the relation between indices and the array. However, some simple arithmetic properties can be derived by these analyses. The discovered extended properties and the added static properties are summarized in the following table:

Table 4.1: Added static properties for scalar variables

if scalar variable v is ...	we add the property ...
[increasing, strict, dense]	$\forall i, v^i = v^0 + i$
[increasing, strict]	$\forall i, \forall j, j > i \Rightarrow v^j > v^i$
[increasing, dense]	$\forall i, \forall j, j \geq i \Rightarrow v^i + j = v^j + i$
[increasing]	$\forall i, \forall j, j \geq i \Rightarrow v^j \geq v^i$

Finally, if a variable is never updated by the loop, one can simply treat it as a constant symbol throughout all program states.

4.5.2 Update Properties of Array Variable

For the array variables, our approach analyzes the conditions which trigger the update of the array at position `p` by the value `v` during iteration `i`. These predicates are denoted as $\textit{upd}_A(i, p, v)$. Another more general predicate is denoted as $\textit{upd}_A(i, p)$, which states the update during iteration `i` at index `p` with any value. These predicate conditions are then collected for the following property of array variables:

- If the array `A` is only updated once throughout all possible program states, and the update happens at index `p` with the value `v`, then this value is associated with the final value at the same index during last iteration.

$$\begin{aligned}&(\forall i \in \textit{iteration}, j \in \textit{iteration}, p \in \textit{index}, v)\\ &\quad (\textit{upd}_A(i, p, v) \wedge (\textit{upd}_A(j, p) \Rightarrow j \leq i)) \Rightarrow\\ &\quad \textit{select}(A(n), p(n)) = v)\end{aligned} \tag{4.2}$$

4.5.3 Array Non-update

Up until this point, all the previous static properties can be reused (with some translations) for the new implementation. However, one of the previous property must be removed from the static analysis. Given an array variable `A` which has not be updated at any location during iteration `i`, the existing approach adds the following property:

$$\forall j, A(i + 1, j) = A(i, j)$$

A naive translation of this property into the new framework would look like the following:

$$\forall j, \textit{select}(A(i), j) = \textit{select}(A(i + 1), j)$$

However such property should not be added as it is already assumed in the theory of array. Without any specific update via function `store`, the elements in an array are assumed to be unchanged from one program state to the next. While most static properties of scalar variables are directly reusable from previous work, the properties of array variables should be reconsidered under the new reasoning framework.

4.5.4 Monotonic Indexing

The process of static analysis can be improved separately from the logical inferencing system. **This monotonic indexing property is one of the improvement we made in this master project**. The intuition behind this monotonic indexing property originated from the monotonicity of indexing scalars. If a scalar variable is monotonic, the value of this scalar will only develop into larger (in case of monotonic increasing) or smaller (monotonic decreasing) value in later iterations. Therefore, once the update of this scalar variable occurs, the scalar variable will never have the same value again. Combining with the strictness property, if the scalar variable is strictly monotonic, we know for sure the variable will not hold the same value in later iterations. Applying this knowledge to the array indices, we can learn more about the traverse of the array.

Suppose the array variable `A` is only accessed by indexing variable x throughout the entire program, and the variable x is a monotonic variable, the monotonic indexing property can be derived from the following derivation:

1. Assuming x is *monotonic* and *strictly increasing.* From the properties described above, we know x has following property (expressed in the extended expressions) added during the program analysis.

   $$(\forall\, \alpha \in program\, state)\ (\forall\, \beta \in\ program\, state)$$
   $$(\beta > \alpha \Rightarrow x^{\beta} > x^{\alpha})$$

2. Suppose the array `A` is updated by the program statement `A[x] := val` at iteration α. We know for any iteration $\beta > \alpha \Rightarrow x^{\beta} > x^{\alpha}$. In other words, we never revisit array `A` at index x^{α} in later iterations. This means `val` is the final value of `A[x]`, since there will not be any update at the same location in later iterations. Hence one can safely assert the following property:

   $$A^n[x^{\alpha}] == A^{\alpha}[x^{\alpha}] == val$$

   given the array `A` is updated at index x at iteration α and the final iteration denoted by `n`.

   This property can be further translated into array theory based property:

   $$select(A(n), x(\alpha)) == select(A(\alpha), x(\alpha))$$

   The same derivation also hold for monotonic strictly decreasing, this property only relies on the fact that the loop never revisit any index in later iterations. Hence this monotonic indexing property is added for all array updates with uniquely-accessing monotonic strict indices.

An example based on our running example `partition` can show this monotonic indexing property in action:

```
while (a <= alength) do
  :: A[a] >= 0 -> B[b] = A[a]; a = a + 1; b = b + 1;
  :: true -> C[c] = A[a]; a = a + 1; c = c + 1;
od
```

In both branches of the guards, the scalar variable `a` is increased by one. Hence one can learn that variable `a` is monotonic and strictly increasing. Further, we know that the array `A` is only accessed by `a`. All of the premises of our monotonic indexing property is satisfied in this analysis, hence the property is added to the array variable `A`.

$$select(A(n), a(\alpha)) == select(A(\alpha), a(\alpha))$$

5 Reasoning with Vampire

5.1 Logical Inferences using Vampire

In the second invariant generation phase, we perform logical inferencing on the problem instance produced by the static analysis step. With the theorem prover Vampire, one can perform logical inferencing in following modes:

1. Allow Vampire to embark the consequence finding process on the problem instance, the results are sound logical consequences, hence the valid invariants for the loop. This process is referred as the symbol elimination as the auxiliary symbols such as symbols containing extended expressions are eliminated by Vampire.
2. The invariants generated from the first operation mode can be large in quantity and may not be of the interest to the user who is proving certain post-conditions of the loop. Hence the user can, by invoking our system with a special operation option, apply a filter based on the post-conditions to restrain the invariants inferred by Vampire.
3. Finally, based on the idea of refutation, the user can also directly supply the post-condition to Vampire. If the post-condition is a logical consequence of the invariants generated, including the intermediate ones expressed using extended expressions, Vampire can prove the post-condition by refutation.

In the work of Ahrendt et al.[AKR15], the last two operation modes are introduced to extend the original work of [KV09a]. Together with the user provided post-conditions, one can utilize the result of the symbol elimination method or even constructing a novel automation for the proof of correctness differing from the classic Hoare-rule based approach.

In this following chapter, we explain the logical inferences performed by Vampire and the symbol elimination method for invariant generation. Vampire is a fully automatic first-order theorem prover, hence it requires no user guidance during the theorem proving. This allows our invariant generation approach to be fully automated, the user is not required to provide any input or steer the generation during runtime. Also, Vampire is a logically sound inference system. This means all logical consequences generated would be valid consequence of the result of program analysis. Similar to other first-order theorem provers, the final performance of Vampire is significantly affected by the input. Additionally, Vampire has a sophisticated internal architecture for its unparalleled proving speed. This internal architecture is however subjected to the choice of options provide to Vampire. As we describe the process of logical inferencing in Vampire, we will also briefly explain critical concepts related to the choice of options.

5.2 Vampire and the FOOL: Polymorphic Arrays

At the time of the initiation of our project, one of the developing branches of Vampire has been published in the work of Kotelnikov et al. [Kot+16]. This particular branch supports the need of our previous implementation. The new introduction of first-class boolean variables and polymorphic arrays made the foundation needed for our project. The FOOL branch aims at mitigating the complexity of translation into first-order logic. This translation is normally performed by the Vampire users, yet the translation process requires substantial background knowledge of the internal Vampire operations. The result of the translation could significantly affect the performance of Vampire, as theorem provers are highly sensitive to their inputs. Since such a translation is so performance-critical, the reasonable solution is to let the theorem prover choose how to translate the inputs itself. The FOOL branch provided different new expressions such as if-then-else, let-in and the direct support of polymorphic arrays. The needed optimisations during translation are built-in as part of the FOOL extension, including the optimisation over the array axioms inclusion process. Our work benefits largely by these new features and optimisation. Also, given the minimum implementation needed for our extension, our experience can be regarded as one successful story of the FOOL branch.

5.3 First-Order Reasoning about Array Properties

Our main contribution to this new extension for the logical inference step is the translation from previous encoding into array theory encoding. Previously, the array variables are encoded as uninterpreted functions taking the indices as their argument. In the case of extended expression, the additional argument representing the program state can be included. In the previous encoding, one cannot explicitly express the differences between array reads and array writes. In our new reasoning framework, arrays are encoded as constant symbols instead of function taking indices. We also differentiate the operations of reading and writing in the program analysis phase. This results in a clearer semantics of array operations and disambiguates the equalities from array assignments. The concept can be more precisely explained using the following example:

- Previously, the array assignment of `a[i] = b[j]` at iteration `k` is internally encoded as:

  ```
  equals( a(k+1,i(k)),
          b(k,j(k)) )
  ```

  where the function `equals()` creates internal equality over two terms for Vampire.

- Now with the array encoding, we represent the same array assignment statement as:

  ```
  store( a(k+1),
         i(k),
         select(b(k),j(k)) )
  ```

The array theory axioms (read-over-write and extensionality) are automatically and optimally added by Vampire once the `select` and `store` functions are invoked. The polymorphic array introduced in the FOOL extension is more general than our guarded command language in terms of array sorts. In our language, arrays are only indexed by non-negative integers and contains either integers or booleans. But the more general underlining framework means possibility for future extension of our guarded command language.

Another alternative approach is to insert the needed array axioms into Vampire directly, as this alternative was used in the original work in [KV09a]. In the previous versions of Vampire, the theories of integer arrays and arrays of integer arrays are coded internally using this approach. Apart from providing less support for arrays of different sorts, this approach also lose the chance of potential optimization. During the automated inclusion of axioms, Vampire optimally avoids the axioms which are not necessary. For instance, the read-over-write axioms are not added in cases where the `store` function never occurs.

5.4 Logical Inferences

The logical inferencing step is carried out using the superposition inference system in Vampire. While the actual Vampire inference system is parametrized over the selection functions and ordering, the basic inference is defined using the superposition rules. Here is one simple example of resolution:

$$\frac{\underline{A} \vee B \qquad \underline{\neg A} \vee C}{B \vee C}\ \text{resolution}$$

with the underline representing selected terms from the selection function. The actual implementation requires an additional unifier over terms, further detail is referred to [KV13].

Superposition rules are meant for handling resolution between equalities, here we demonstrate one of the superposition rule:

$$\frac{\underline{a = b} \vee Q \qquad \underline{A[a] = c} \vee P}{A[b] = c \vee Q \vee P}\ \text{superposition}$$

More advanced resolution rules such as extensionality resolution are also implemented in Vampire. In our project, the extensionality resolution is critical for the array theory reasoning. A detailed description regarding extensionality resolution is referred to [Gup+14].

The logical inferences produce conclusions from the premises. These newly produced conclusions are added to the search space of the proof. As the inference system continues, more logical conclusions can be drawn from the search space. However, the growth of search space of first-order logic formulae with equalities is very fast in reality. One requires a very well-organized inference system (including well-behaved selection and ordering) and efficient redundancy elimination. As mentioned, the inference system is actually parametrized over selection functions and ordering. Vampire supports different selection functions and choose an appropriate one according to the execution options from these selection functions. On top of these carefully crafted efficiency, Vampire has yet another novel internal architecture called AVATAR [Vor14]. In AVATAR architecture, resolutions are controlled not to resolve into heavy conclusions and some of the proof steps are delegated to an internal SAT solver for further efficiency optimization. All these works contribute to the award-winning [Sut14] speed of logical inferences.

5.5 Symbol Elimination

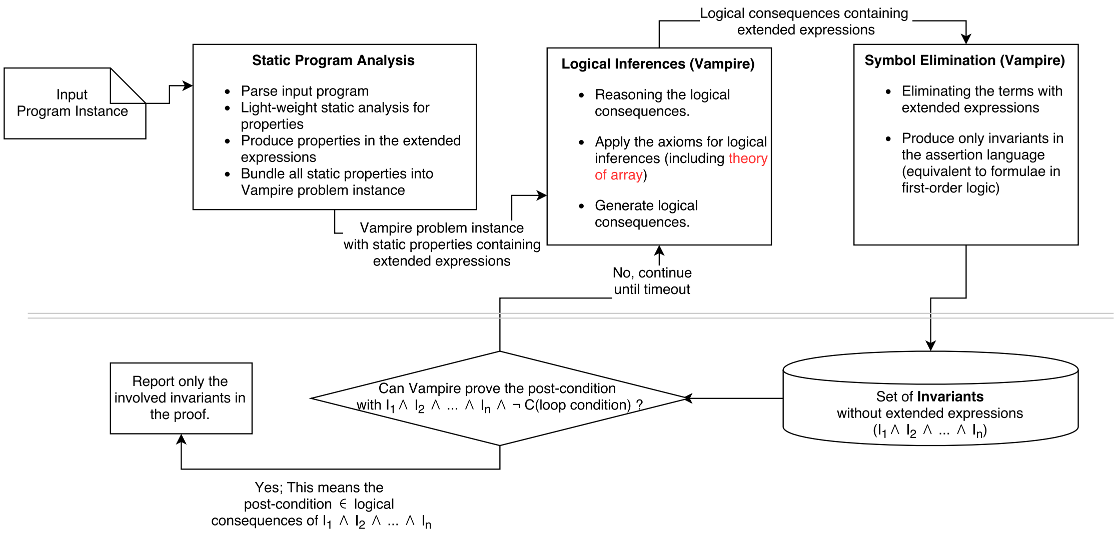
Running Vampire over the set of static properties can provide valid consequences via the logical inferencing. However, as the static properties are expressed using the extended expression, so are the logical consequences inferred. The loop invariants must be expressed only in the assertion language. While one can seek for similiar inverse translation rules as we demonstrated in the case of user-provided pre-conditions, the general translation from the extended expression into the assertion language is not possible. This is because some of the symbols in extended expression have no corresponding semantics in the assertion language. For the purpose of invariant generation, we need a directed inference system to get rid of the symbols in the extended expression. These technique is called the "symbol elimination" and was first introduced in the work of Kovács and Voronkov [KV09b]. The main idea is to make sure the conclusion drawn from the inference rule is either (1) a conclusion without any premises or (2) the premises contains at least one undesirable symbol. This way, the number of undesirable symbol would decrease as the inferences continue. For the detailed theoretical regarding symbol elimination, we refer to [KV09b].

5.6 Reaching Correctness in Different Ways

With the possibility for expressing post-conditions in our simple guarded command language, the verification of correctness is now more flexible. Figure 5.1 shows one of the verification methods called *invariant filtering.* The idea is to continuously attempt to prove the post-condition with the newly generated invariants in parallel to the actual invariant generation process (generated invariants are indicated as $I_1 \wedge I_2 \wedge ...I_n$ in the diagram) . If the post-condition is proved by Vampire, the system can report on the subset of invariants which actually participated during the proof. Otherwise, the inferences continue.

This approach can effectively reduce the total reported invariant and provide only the "useful" invariants based on the final step of relavant invariant filtering.

Another alternative approach toward proof of correctness is to *directly prove the post-condition* from the set of intermediate extended properties. This approach would omit the symbol elimination and perform the refutation proof directly. If one only seeks for the correctness under post-condition without the need for explicit invariants, this approach provides a novel and alternative solution to the classic Floyd-Hoare verification approach. Comparing to invariant filtering, the direct proof of correctness is faster and can benefit from the fact that the extended properties are stronger than the implied invariants. All proofs reachable by the invariant filtering should be also reachable by this approach, while in some cases the converse is not true.

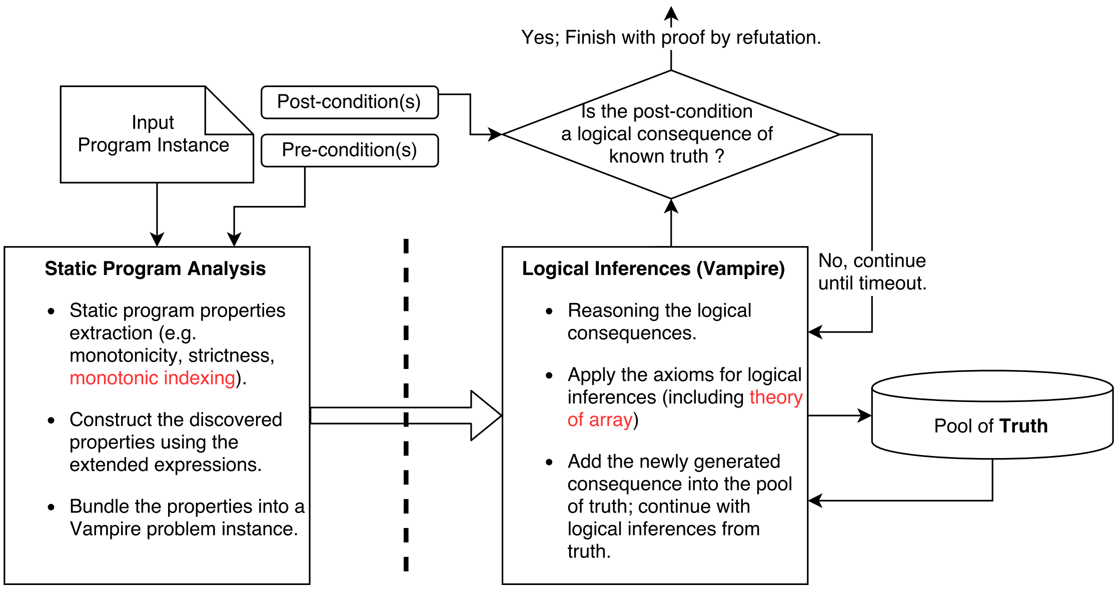


Input Program Instance
Static Program Analysis
Parse input program
Light-weight static analysis for properties
Produce properties in the extended expressions
Bundle all static properties into Vampire problem instance
Vampire problem instance with static properties containing extended expressions
Logical Inferences (Vampire)
Reasoning the logical consequences.
Apply the axioms for logical inferences (including theory of array)
Generate logical consequences.
Logical consequences containing extended expressions
Symbol Elimination (Vampire)
Eliminating the terms with extended expressions
Produce only invariants in the assertion language (equivalent to formulae in first-order logic)
No, continue until timeout
Report only the involved invariants in the proof.
Can Vampire prove the post-condition with $I_1 \wedge I_2 \wedge \ldots \wedge I_n \wedge \neg$C(loop condition) ?
Set of Invariants without extended expressions ($I_1 \wedge I_2 \wedge \ldots \wedge I_n$)
Yes; This means the post-condition $\in$ logical consequences of $I_1 \wedge I_2 \wedge \ldots \wedge I_n$

Figure 5.1: *Flow diagram of correctness proving using post-condition as invariant filter, we refer to this operation mode as the* invariant filtering

5.7 Overview of Invariant Generation Workflow

Our approach, similiar to the origin approach from [AKR15], is illustrated by the flow diagram shown in Figure 5.2. The entire approach is split into two phases by the dashed line with our contributions highlighted with red fonts. Particularly, Figure 5.2 shows the operation mode of *direct proof.* In this operation mode, the symbol elimination is omitted and Vampire will keep producing the logical consequences until the post-condition is derivable or timeout. This diagram also shows the two phases are decoupled and replaceable by other equivalent processes.



Yes; Finish with proof by refutation.
Input Program Instance
Post-condition(s)
Pre-condition(s)
Is the post-condition a logical consequence of known truth ?
Static Program Analysis
Static program properties extraction (e.g. monotonicity, strictness, monotonic indexing).
Construct the discovered properties using the extended expressions.
Bundle the properties into a Vampire problem instance.
Logical Inferences (Vampire)
Reasoning the logical consequences.
Apply the axioms for logical inferences (including theory of array)
Add the newly generated consequence into the pool of truth; continue with logical inferences from truth.
No, continue until timeout.
Pool of Truth

Figure 5.2: *Entire workflow of our invariant generation workflow, based on the* direct proof *operation option.*

6 Results

In this chapter we present the experiment results from our implementation. In this first section, we present the overall experiment result of all 20 test cases, the exact same set of test cases as in [AKR15]. The numerical experiment results are followed by detail examination of critical proof steps of the newly proved examples. Finally, this chapter ends with the generated invariants of other test cases, showing the capacity of our approach.

All the results are collected using a computer with quad-core i7 CPU equipped with 16GB of RAM. For better comparison, we have reproduced the same experiments as in previous paper and listed the result here.

6.1 Results of all test cases

On the left side, the table shows the experiment result without array theory reasoning, while the right table shows the final result after our array theory extension. All test cases are performed with the time limit set to 300 seconds. This time limit ensures sufficient time for the harder proofs. Other options such as splitting strategies are left as their default settings. The readings Δ_{direct} stands for the required time for Vampire to directly proof the desired post-condition from the extended properties, with unit in second. In the case where Δ_{direct} reading is missing, Vampire fails to prove the post-condition within the 300 seconds time limit and ends with refutation not found. As none of the examples reach to saturation within our time limits, we cannot formally conclude the answer to satisfiability. However, reaching to the saturation for any of the test cases is virtually impossible in our particular application. Apart from required proving time, the total number of created clauses is also featured in the table. In the cases where Δ_{direct} reading is missing, the created clauses count total created clauses within 300 seconds.

Table 6.2 shows the four newly proved test cases: copyPositive, partition, partitionInit, and swap. These cases witness the enhancement in reasoning about arrays in loops of the new implementation. While the newly proved cases show the improvement, the experiment also showed all the previously provable cases are still provable in array theory reasoning. With these two observations combined, we demonstrated the overall improvement over existing approach. By reasoning in the domain specific theory, in our case the theory of array, the theorem prover is capable of deriving much more complex invariants automatically.

Besides showing the improvement in reasoning ability, the two tables also provide a closer comparison between two approaches. One can observe the general trend of less time used for the invariant generation *without* array theory. In other words, the implementation of treating array variables as uninterpreted functions has marginal advantage in computation complexity. This observation can be witnessed by the fact that in all commonly provable cases the time consumed in the original implementation are marginally shorter than the array theory reasoning implementation. Also, the total number of created clauses agrees on such trend. However, in all the commonly provable cases, both implementations manage to derive the post-condition as the logical consequence within fraction of a second, making the new implementation still competitive solution for automatic invariant generation.

Although the newly proved cases cannot be compared with the results of previous implementation in terms of computation complexity, we found a reference time from the work of Kovács et al. [KV09a]. In the result section of [KV09a], the test case similiar to our `partition` took 56 seconds to generate the same invariant. This shows the agreement of increasing computation complexity of array theory reasoning.

Table 6.1: Reasoning **without** array theory

Testcase	$\Delta_{direct}(sec)$	created clauses
absolute	0.374	2095
copy	0.057	495
copyOdd	0.208	1571
copyPartial	0.047	426
copyPositive		530669
find		412821
findMax		324456
init	0.052	415
initEven		430518
initNonConstant	0.117	909
initPartial	0.060	495
inPlaceMax		362783
max	0.348	2140
mergeInterleave		376322
partition		622830
partitionInit		488387
reverse	0.079	593
strcpy	0.048	373
strlen	0.019	139
swap		812284

Table 6.2: Test cases reasoning with array theory

Testcase	$\Delta_{direct}(sec)$	created clauses
absolute	0.484	2614
copy	0.079	654
copyOdd	0.181	1098
copyPartial	0.104	800
copyPositive	46.238	89280
find		413352
findMax		398548
init	0.069	592
initEven		391735
initNonConstant	0.128	940
initPartial	0.069	593
inPlaceMax		530098
max	0.481	2634
mergeInterleave		543746
partition	97.519	210837
partitionInit	28.217	72989
reverse	0.098	733
strcpy	0.081	538
strlen	0.031	168
swap	11.218	61786

6.2 Case study: swap

```
int [] a, b, olda, oldb;
int i, alength, blength;

requires blength == alength;
requires i == 0;
requires forall int i, 0 <= i & i < alength ==> a[i] == olda[i];
requires forall int i, 0 <= i & i < blength ==> b[i] == oldb[i];

ensures forall int i, 0 <= i & i < blength ==> a[i] == oldb[i];
ensures forall int i, 0 <= i & i < blength ==> b[i] == olda[i];

while (i < alength) do
  :: true -> a[i] = b[i]; b[i] = a[i]; i = i + 1;
od
```

In this section we look into one of the newly proved test cases, `swap`. The program takes two integer arrays of equal length and element-wise swapping the two arrays. The pre-conditions ensure equal length of the input arrays and keep a copy of the values inside each array (`olda` and `oldb`). The post-condition for correctness of the loop checks if the element in the modified array `b` is indeed element-wise equal to `olda`. This test case has rather straightforward semantics for human reader, yet its algorithm heavily relies on the array operations `select` and `store`.

Some critical steps in the successful proof of correctness are contributed by the array theory axioms, as shown in Figure 6.1. This shows the array theory derived inferences are indeed used in the final proof by refutation. The previous implementation without array theory cannot prove this example, this suggests the improvement in reasoning ability of the new extension. Also, the steps show that complex invariant over array `select` and `store` with quantifier alternation can be derived using our extension. The final step `63634.` is a unification of AVATAR splitting.

```
38747.  C1 $select(oldb,sK6) = $select(a,sK6)
          | $lesseq(alength,$sum(-1,sK6))
            <- {80, 185, 192, 194}
            [subsumption resolution 38698,30268]

38749.  C1 $lesseq(alength,$sum(-1,sK6))
         <- {19, 80, 185, 190, 192,194}
           [subsumption resolution 38747,30169]

38750. 262 | 19 | ~80 | 185 | ~190 | ~192 | ~194
          [AVATAR split clause 38749,30139,30132
          ,30125,30101,1660,307,38738]

63634.  C0 $false [AVATAR sat refutation
      48074,47723,371,180,186,30241,340,341,3414,
      63200,339,38750,30127,30141,30134,381,208,
      214,44449,309,1780,1681,330,48112,45944,48885,
      51852,47081,3177,3247,332,323,48525,55659,331,
      63319,1668,1696,316]

\caption{Critical proof steps in the test case \inlinecode{partition}.}
```

During our invariant generation, Skolemized functions are not de-Skolemized and kept in original form for the entire derivation. Skolemization captures the alternating quantifiers, in the demonstrated steps above, the Skolemization funciton sk6 is defined as:

```
Skolemising: sK6 for X0 in ? [X0 : $int] :
    ($select(olda,X0) != $select(b,X0)
     & ~$lesseq(blength,X0)
     & $lesseq(0,X0)) in formula 87.
```

While de-Skolemization can arguably produce more human readable invariants, the Skolem functions are kept in Skolemized form for better reasoning ability. For the technical details we refer to the orignal paper in 2009 [KV09a], in which the reason for not performing de-Skolemization during the reasoning was first introduced and motivated.

With our approach, the system can automatically generate the following loop invariants:

```
! [X2 : $int] : ($select(oldb,X2) = $select(a,X2) |
  ~($less(X2,blength) & $lesseq(0,X2)))
```

which can be further de-Skolemized into more human-reader friendly form:

$$\forall i \in \mathit{Int}\,, \mathit{oldb}[i] = a[i] \lor \neg((i < \mathit{blength}) \,\land\, (0 \leq i))$$

by applying the rule of inference on the righthand side of OR, one can further derive:

$$\forall i \in \mathit{Int}\,, (0 \leq i < \mathit{blength}) \;\Rightarrow\; \mathit{oldb}[i] = a[i]$$

6.3 Other Invariants

Apart from the case study `swap`, we also demonstrate other invariants generated from the newly proved cases in this section.

Figure 6.1: *Test case `partition`.*

```
int [] A, B, C;
int a, b, c, alength;

requires a == 0;
requires b == 0;
requires c == 0;
requires alength > 0;

while (a <= alength) do
:: A[a] >= 0 -> B[b] = A[a]; a = a + 1 ; b = b + 1;
:: true -> C[c] = A[a]; a = a + 1; c = c + 1;
od
```

In this test case, our system can automatically derives the following invariant:

$$\begin{aligned}\forall\, \mathit{int}\; i,\ (0 \leq i \land i < b \Rightarrow (B[i] \geq 0)\land \\ (\exists\, \mathit{int}\, j,\ 0 \leq j \land j < a \land B[i] == A[j]))\end{aligned}$$

Figure 6.2: *Test case* `copyPositive`*.*

```
int [] a, b;
int i, j, alength, blength;

requires blength <= alength;
requires i == 0;
requires j == 0;

while (i < blength) do
:: b[i] >= 0 -> a[j] = b[i]; i = i + 1; j = j + 1;
:: true      -> i = i + 1;
od
```

In this test case, our system can automatically derives the following invariant:

$$\forall\, int\ k,\ 0 \leq k \wedge k < j \Rightarrow a[k] \geq 0;$$

Figure 6.3: *Test case* `partitionInit`*.*

```
int [] A, B, C;
int a, b, alength;

requires a == 0;
requires b == 0;
requires alength >= 0;

while (a < alength) do
:: A[a] == C[a] -> B[b] = a; a = a + 1; b = b + 1;
:: true         -> a = a + 1;
od
```

In this test case, our system can automatically derives the following invariant:

$$\forall\, int\ i,\ 0 \leq i \wedge i < b \Rightarrow A[B[i]] == C[B[i]];$$

---

7 Conclusion and Future Work

7.1 Conclusion

The inclusion of array theory in loop invariant generation has shown to be improving the invariant reasoning ability. In our experiment results, the test case `swap` works as the witness of such improvement comparing to the previous implementation. As the nature of `swap` is heavily dependent on the array operations `select` and `store`, the array theory pays off for the derivation of correctness. Apart from the improvement brought by the array theory, we also demonstrated that the process of invariant reasoning can be enhanced by the static program analysis. The monotonic indexing property we introduced helps the theorem prover by dropping the additional premises dealing with index bounds. This makes the theorem prover's task much simpler hence the successful proof of test cases such as `partition` and `partitionInit`. This experiment result shows the light-weight static analysis can be simple yet critical in the later logical inference process. Crafting the suitable static properties can be non-trivial. In our case the monotonic indexing property was devised after observing the additional premises during inferencing steps, which lead to difficulties for the theorem prover to derive the post-condition. These static properties are largely originated from a mathematical context. A single proper property can be crucial for the loop verification. It is very interesting to further explore more static analyses and experiment with their impact on other problem instances.

The improvement in reasoning power is certainly promising, however, we also observed marginal increase in both reasoning time and the internal clauses generated for the set of already proved cases. Comparing to the implementation back in 2009, the test case `partition` takes around twice as much computation time to derive, however, the new approach now directly handles the array operations. Despite the fact that most of the proving time required are longer than the standard timeout for Vampire in the newly proved cases, the result still indicates improvement in invariant reasoning, especially with the array-specific invariants.

On top of the enhanced reasoning power, we also found the encoding based on the array theory could benefit the human readability of the proof steps. Instead of equality over two general array terms, the new proof step clearly indicates the difference between reading from an array and writing into an array. This improvement gives a much better trace for user to examine the derivation.

Lastly, in this experiment we showed the idea of using a theorem prover for automated invariant generation is promising. The capacity to derive complex invariants including both quantifier alternations and domain specific theory reasoning can be used to ease the manual efforts needed for loop annotation. Particularly, our approach requires no user guidance nor any predefined templates of invariants.

7.2 Future Work

1. **Boolean array test cases and experiments**: with the foundation of FOOL and the polymorphic arrays, boolean arrays can be incorporated into our array theory reasoning approach. Due to project time limit, we have not yet included test cases containing boolean arrays. In the future work, boolean array can be used to explore interesting properties such as partial sortedness of the array. Furthermore, since Vampire with the FOOL extension already support first class boolean reasoning, it is interesting to experiment with the boolean array invariants.

2. **Vampire with SMT solvers**: The AVATAR structure of Vampire makes it possible to delegate satisfiability problems to a SAT solver. This structure enables the superposition-based Vampire to reason about logical consequences much faster hence solving the problem within the time constraint. Furthermore, the AVATAR structure can accomodate other types of solvers such as SMT solvers. Despite the fact that Vampire has built-in approximation of arithmetic axiomatization, complex properties involving arithmetic require dedicated

solvers. Problems involving arithmetic properties such as even / odd indices can possibly benefit from the arithmetic reasoning ability of SMT solvers such as Z3.

3. **More language constructs**: Apart from the polymorphic arrays, the paper of Vampire and the FOOL [Kot+16] also introduced the entire first-class boolean and corresponding constructs into the latest Vampire branch. Among the constructs introduced, the "if then else" construct can be directly benefiting program analysis. Our input language syntax can be extended with the condition construct and internally mapped to the new feature from the FOOL infrastructure. This would also mean new static properties can be explored specifically for the condition construct.

References

[Ahr+05] W. Ahrendt et al. The KeY tool. *Software and System Modeling* **4**.1 (2005), 32–54. URL: `http://www.springerlink.com/index/10.1007/s10270-004-0058-x`.

[AKR15] W. Ahrendt, L. Kovács, and S. Robillard. "Reasoning About Loops Using Vampire in KeY". *Lecture Notes in Computer Science. 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2015, Suva, Fiji, 24-28 November 2015.* Springer Berlin Heidelberg, 2015, pp. 434–443. ISBN: 978-3-662-48898-0. DOI: `10.1007/978-3-662-48899-7_30`. URL: `http://dx.doi.org/10.1007/978-3-662-48899-7_30`.

[Bla+03] B. Blanchet et al. "A static analyzer for large safety-critical software". *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003.* ACM. 2003, pp. 196–207. DOI: `10.1145/781131.781153`. URL: `http://doi.acm.org/10.1145/781131.781153`.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.* ACM. 1977, pp. 238–252. DOI: `10.1145/512950.512973`. URL: `http://doi.acm.org/10.1145/512950.512973`.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* **8**.2 (1986), 244–263. DOI: `10.1145/5397.5399`. URL: `http://doi.acm.org/10.1145/5397.5399`.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics* **58**.2 (1936), 345–363.

[Cou03] P. Cousot. "Verification by Abstract Interpretation". *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday.* Springer, 2003, pp. 243–268. DOI: `10.1007/978-3-540-39910-0_11`. URL: `http://dx.doi.org/10.1007/978-3-540-39910-0_11`.

[Dij75] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* **18**.8 (1975), 453–457. DOI: `10.1145/360933.360975`. URL: `http://doi.acm.org/10.1145/360933.360975`.

[Ern+07] M. D. Ernst et al. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* **69**.1–3 (Dec. 2007), 35–45.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science* **19**.19-32 (1967), 1.

[FM10] C. A. Furia and B. Meyer. *Inferring Loop Invariants Using Postconditions.* 2010. DOI: `10.1007/978-3-642-15025-8_15`. URL: `http://dx.doi.org/10.1007/978-3-642-15025-8_15`.

[FMV14] C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys* **46**.3 (2014), 34:1–34:51. DOI: `10.1145/2506375`. URL: `http://doi.acm.org/10.1145/2506375`.

[FQ02] C. Flanagan and S. Qadeer. "Predicate abstraction for software verification". *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002.* ACM. 2002, pp. 191–202. DOI: `10.1145/503272.503291`. URL: `http://doi.acm.org/10.1145/503272.503291`.

[Gal+14a] J. P. Galeotti et al. *Automating Full Functional Verification of Programs with Loops.* 2014. URL: `http://arxiv.org/abs/1407.5286`.

[Gal+14b] J. P. Galeotti et al. "DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification". *Proceedings of the 10th Haifa Verification Conference (HVC).* Ed. by E. Yahav. Vol. 8855. Lecture Notes in Computer Science. Tool paper. Springer, Nov. 2014, pp. 48–53.

[GR09] A. Gupta and A. Rybalchenko. “InvGen: An Efficient Invariant Generator”. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings.* 2009, pp. 634–640. DOI: 10.1007/978-3-642-02658-4_48. URL: http://dx.doi.org/10.1007/978-3-642-02658-4_48.

[Gup+14] A. Gupta et al. “Extensional Crisis and Proving Identity”. *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings.* Springer, 2014, pp. 185–200. DOI: 10.1007/978-3-319-11936-6_14. URL: http://dx.doi.org/10.1007/978-3-319-11936-6_14.

[HKV11] K. Hoder, L. Kovács, and A. Voronkov. “Invariant Generation in Vampire”. *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings.* 2011, pp. 60–64. DOI: 10.1007/978-3-642-19835-9_7. URL: http://dx.doi.org/10.1007/978-3-642-19835-9_7.

[Hoa83] C. A. R. Hoare. An Axiomatic Basis for Computer Programming (Reprint). *Commun. ACM* **26**.1 (1983), 53–56. DOI: 10.1145/357980.358001. URL: http://doi.acm.org/10.1145/357980.358001.

[KB83] D. E. Knuth and P. B. Bendix. “Simple word problems in universal algebras”. *Automation of Reasoning.* Springer, 1983, pp. 342–376.

[Kot+16] E. Kotelnikov et al. The vampire and the FOOL (2016), 37–48. DOI: 10.1145/2854065.2854071. URL: http://doi.acm.org/10.1145/2854065.2854071.

[Kov08] L. Kovács. “Aligator: A Mathematica Package for Invariant Generation (System Description)”. *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings.* Springer, 2008, pp. 275–282. DOI: 10.1007/978-3-540-71070-7_22. URL: http://dx.doi.org/10.1007/978-3-540-71070-7_22.

[KR16] L. Kovács and S. Robillard. “Reasoning About Loops Using Vampire”. *Proceedings of the 1st and 2nd Vampire Workshops.* Vol. 38. 2016, pp. 52–62.

[KV09a] L. Kovács and A. Voronkov. “Finding loop invariants for programs over arrays using a theorem prover”. *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* 2009, pp. 470–485. DOI: 10.1007/978-3-642-00593-0_33. URL: http://dx.doi.org/10.1007/978-3-642-00593-0_33.

[KV09b] L. Kovács and A. Voronkov. “Interpolation and Symbol Elimination”. *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings.* 2009, pp. 199–213. DOI: 10.1007/978-3-642-02959-2_17. URL: http://dx.doi.org/10.1007/978-3-642-02959-2_17.

[KV09c] L. Kovács and A. Voronkov. “Interpolation and symbol elimination”. *Automated Deduction–CADE-22.* Springer, 2009, pp. 199–213.

[KV13] L. Kovács and A. Voronkov. “First-Order Theorem Proving and Vampire”. *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 2013, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1. URL: http://dx.doi.org/10.1007/978-3-642-39799-8_1.

[Nor09] U. Norell. “Dependently Typed Programming in Agda”. *Proceedings of the 4th International Workshop on Types in Language Design and Implementation.* TLDI ’09. Savannah, GA, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862. URL: http://doi.acm.org/10.1145/1481861.1481862.

[Pau94] L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow).* Vol. 828. Lecture Notes in Computer Science. Springer Science & Business Media, 1994. ISBN: 3-540-58244-4. DOI: 10.1007/BFb0030541. URL: http://dx.doi.org/10.1007/BFb0030541.

[QS82] J. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR”. *International Symposium on Programming, 5th Colloquium, Torino, Italy,* *April 6-8, 1982, Proceedings.* 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22. URL: http://dx.doi.org/10.1007/3-540-11494-7_22.

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.* **36**.1-2 (2003), 101–115. DOI: 10.1016/S0747-7171(03)00040-3. URL: http://dx.doi.org/10.1016/S0747-7171(03)00040-3.

[Sch02] S. Schulz. E - a brainiac theorem prover. *AI Commun.* **15**.2-3 (2002), 111–126. URL: http://content.iospress.com/articles/ai-communications/aic260.

[SS04] G. Sutcliffe and C. B. Suttner. "The CADE ATP System Competition". *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings.* Springer, 2004, pp. 490–491. DOI: 10.1007/978-3-540-25984-8_36. URL: http://dx.doi.org/10.1007/978-3-540-25984-8_36.

[Sut14] G. Sutcliffe. The CADE-24 automated theorem proving system competition - CASC-24. *AI Commun.* **27**.4 (2014), 405–416. DOI: 10.3233/AIC-140606. URL: http://dx.doi.org/10.3233/AIC-140606.

[The04] The Coq development team. *The Coq proof assistant reference manual.* Version 8.0. LogiCal Project. 2004. URL: http://coq.inria.fr.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* **58**.345-363 (1936), 5.

[Vor01] A. Voronkov. How to optimize proof-search in modal logics: new methods of proving redundancy criteria for sequent calculi. *ACM Transactions on Computational Logic (TOCL)* **2**.2 (2001), 182–215. DOI: 10.1145/371316.371511. URL: http://doi.acm.org/10.1145/371316.371511.

[Vor14] A. Voronkov. "AVATAR: The Architecture for First-Order Theorem Provers". *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings.* Springer. 2014, pp. 696–710. DOI: 10.1007/978-3-319-08867-9_46. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_46.

[Wei+09] C. Weidenbach et al. "SPASS Version 3.5". *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings.* Springer, 2009, pp. 140–145. DOI: 10.1007/978-3-642-02959-2_10. URL: http://dx.doi.org/10.1007/978-3-642-02959-2_10.