CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

QuickInv: Program Invariant Discovery from Execution Traces

Dynamic invariant discovery system built on top of a theory exploration system

Master's thesis in Computer Science and Engineering

EDVIN NILSSON
ENAYATULLAH NOROZI

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2025

Master's thesis 2025

QuickInv: Program Invariant Discovery from Execution Traces

Dynamic invariant discovery system built on top of a theory exploration system

EDVIN NILSSON
ENAYATULLAH NOROZI

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2025

QuickInv: Program Invariant Discovery from Execution Traces
Dynamic invariant discovery system built on top of a theory exploration system
EDVIN NILSSON
ENAYATULLAH NOROZI

Supervisor: Nicholas Smallbone, Department of Computer Science and Engineering
Examiner: Yehia Abd Alrahman, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31-772 10 00

Typeset in Typst
Gothenburg, Sweden, 2025

QuickInv: Program Invariant Discovery from Execution Traces
Dynamic invariant discovery system built on top of a theory exploration system
EDVIN NILSSON
ENAYATULLAH NOROZI
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Automatic loop invariant inference is a fundamental problem in program verification, yet it is challenging due to the complex behavior of program loops, and the problem is undecidable. Traditional approaches are often limited by predefined templates or are specialized to specific types of loop invariants.

This thesis presents QuickInv, a dynamic invariant discovery system that is built upon the principles of theory exploration to overcome these limitations. QuickInv discovers high-level invariants by systematically exploring potential invariants by testing against recorded execution traces. We demonstrate QuickInv on several common array programs, such as binary search, selection sort, and merge sort, successfully recovering many key loop invariants.

Keywords: Invariant inference, Dynamic program analysis, Theory exploration, Random testing, Program correctness

Acknowledgements

We would like to thank our supervisor Nicholas Smallbone for his support, guidance and encouragement throughout this project.

We would also like to thank our examiner Yehia Abd Alrahman and our opponents Loke Gustafsson and Erik Magnusson for their valuable feedback.

Edvin Nilsson and Enayatullah Norozi, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

Writing bug-free and correct computer programs is hard. The most common way of finding software bugs is through testing, but testing still misses many bugs. A more rigorous way of preventing bugs is to use program verification tools, which allow programmers to formally prove that a program meets a specification. A fundamental problem in program verification, when verifying the behavior of program loops is to find suitable loop invariants. A loop invariant is a property, or an assertion of a program loop that holds before loop entry, between each loop iteration, and after loop termination.

In Floyd-Hoare logic [1], [2], a widely used formal system for reasoning about the correctness of computer programs, loop invariants must be specified in order to reason about program loops. In general, it can be tricky for programmers to come up with suitable loop invariants to verify loop behavior. The loop invariant needs to be weak enough to satisfy every loop iteration and strong enough to satisfy the post-condition when the loop terminates.

An alternative to human-specified loop invariants is to find them using automated tools. Automating loop invariant inference is also challenging. The problem of automatically inferring loop invariants is undecidable [3], meaning it is impossible to construct an algorithm that finds the correct loop invariants for any arbitrary loop. Instead, loop invariant inference will have to rely on heuristics, approximations, or restricted domains to work.

For example, consider verifying that the implementation of binary search in Listing 1 in C is correct. The function's return value, idx, is the index where the value of key exists in the array, or the special value -1 if the value does not appear in the array. This can be specified using two post-conditions: $idx \neq -1 \implies arr[idx] = key$ and $idx = -1 \implies key \notin arr$. To verify that this implementation fulfills its post-conditions using a program verifier such as Dafny [4], two loop invariants are required: $0 \leq lo \leq hi \leq |\, arr|$ and $key \in arr \iff key \in arr[lo \,..\, hi)$.

```
int binary_search(int* arr, int arr_len, int key)
    // pre: sorted arr
    // post: -1 <= idx < arr_len
    // post: idx != -1 ==> arr[idx] == key
    // post: idx == -1 ==> not (key in arr)
{
    int lo = 0;
    int hi = arr_len;
    int idx = -1;
    while (lo < hi)
        // inv: 0 <= lo <= hi <= arr_len
        // inv: key in arr[lo..hi) <=> key in arr
    {
        int mid = (lo + hi) / 2;
        if (arr[mid] < key) {
            lo = mid + 1;
        } else if (arr[mid] > key) {
            hi = mid;
        } else if (arr[mid] == key) {
            idx = mid;
            break;
        }
    }
    return idx;
}
```

Listing 1 : Implementation of binary search.

The most common way of automatically inferring loop invariants is by using static methods. Static methods analyze the program without executing it, relying on techniques like abstract interpretation [5] and symbolic execution [6] to find sound loop invariants. These methods are often specialized to specific types of invariants that they can infer, and are not easily extended to other types. The second loop invariant of binary search, $key \in arr \iff key \in arr[lo\,..\,hi)$ is about array membership, and these static methods most often express invariants using low-level logic. When using this low-level logic, array membership requires more complicated formulas with quantifiers, which are harder to understand and less intuitive for programmers. For example, this loop invariant would be expressed as $\exists i.\ 0 \leq i < |\,arr| \land arr[i] = key \iff \exists i.\ lo \leq i < hi \land arr[i] = key$.

The other general approach to inferring loop invariants is by using dynamic methods, which analyze the program by executing it with different inputs. The invariants found by dynamic methods are not guaranteed to be sound, but are often very effective in practice. One such dynamic tool is Daikon [7], which can find invariants using predefined templates such as $x = y$, $x \leq y$, and $y = ax + b$. These predefined templates restrict which invariants can be found, and the loop invariant $key \in arr \iff key \in arr[lo\,..\,hi)$ has a very specific shape. There is no template for it, so Daikon cannot find it. Dynamate [8], a system built on top of Daikon, can derive shapes of invariants from the post-conditions. If we write one of the post-conditions as $idx = -1 \implies key \notin arr[0..|\,arr|)$, the system can find the loop invariants $key \notin arr[0..\,lo)$ and $key \notin arr[hi\,..|\,arr|)$. These

two invariants can verify this implementation of binary search, but they are more restrictive than $key \in arr \iff key \in arr[lo\,..\,hi)$. For example, the latter can verify the variant of binary search that returns the first occurrence, which the first two invariants cannot.

Loop invariants inferred from existing systems are often expressed in low-level logic, which can be less intuitive than high-level loop invariants. We also see a need for a flexible system that can find loop invariants without being restricted by predefined templates or specific shapes, and that is easily extensible by the user with new domain-specific operators.

Our approach to dynamic loop invariant inference with QuickInv is to search for equational laws expressed in a high-level term language that hold for recorded program states captured while running the program. QuickInv uses a theory exploration system to search for loop invariants and is not limited by predefined templates.

To discover these loop invariants of binary search with this approach, we first run the program multiple times with randomized input to record program states. Binary search expects its input array to be sorted, and this is a pre-condition that the input generation must respect. The program states are captured at the beginning of every loop iteration and after loop termination, which is where the loop invariants should hold, and the program states contain the values of all program variables in scope. An example program state log of binary search is shown in Figure 1.

`arr` ↦ `[1,2,3,4,5]`	`key` ↦ `4`	`lo` ↦ `0`	`hi` ↦ `5`	`idx` ↦ `-1`
`arr` ↦ `[1,2,3,4,5]`	`key` ↦ `4`	`lo` ↦ `3`	`hi` ↦ `5`	`idx` ↦ `-1`
`arr` ↦ `[1,2,3,4,5]`	`key` ↦ `4`	`lo` ↦ `3`	`hi` ↦ `4`	`idx` ↦ `-1`
`arr` ↦ `[1,2,3,4,5]`	`key` ↦ `4`	`lo` ↦ `3`	`hi` ↦ `4`	`idx` ↦ `3`
`arr` ↦ `[1,2,8]`	`key` ↦ `6`	`lo` ↦ `0`	`hi` ↦ `3`	`idx` ↦ `-1`
`arr` ↦ `[1,2,8]`	`key` ↦ `6`	`lo` ↦ `2`	`hi` ↦ `3`	`idx` ↦ `-1`
`arr` ↦ `[1,2,8]`	`key` ↦ `6`	`lo` ↦ `2`	`hi` ↦ `2`	`idx` ↦ `-1`

Figure 1 : Example program state log of binary search.

The next step is to explore possible invariant candidates that could hold for the captured program states. In QuickInv, invariants are expressed in an expressive high-level term language using provided domain-specific functions and constants, rather than low-level first-order logic like in many other systems. For example, array membership can simply be expressed as $key \in arr$ where $\in$ is a provided function, instead of $\exists i.\ 0 \leq i < |arr| \wedge arr[i] = key$.

QuickInv is a Haskell program that generates the randomized input and collects the recorded program states for a C program. The grammar used in QuickInv to express invariants is defined in the functional programming language Haskell, and the user can add any Haskell function to the grammar. Given a grammar with functions and constants like `slice`, `∈`, `≤`, `length`, `0`, `1` and `True`, where `slice` is an array slicing function defined as $slice\ arr\ i\ j := arr[i..j)$, we can generate invariant candidates by enumerating type-correct terms from the given grammar and, through testing, determine which terms are equivalent to each other. For example, we can find that `key ∈ arr` is equivalent to

`key ∈ slice arr lo hi` when testing against the captured program states, and therefore discover one of the loop invariants with the law `key ∈ arr = key ∈ slice arr lo hi`. We can also find the other loop invariant $0 \leq lo \leq hi \leq |arr|$ by discovering that the terms `0 ≤ lo`, `lo ≤ hi` and `hi ≤ length arr` are equivalent to `True`. A fundamental challenge with this approach is that the search space for possible terms is very large and grows exponentially when the term size increases.

Compared to other systems like Daikon, QuickInv is very expressive and is not limited by predefined templates. To reduce the search space, Daikon uses many hand-crafted optimizations, while QuickInv can reduce the search space by using discovered properties of the given grammar. Letting the user add any domain-specific function to the grammar makes QuickInv easily extendable to algorithms from various domains, and the system will automatically adapt to the new grammar.

1.1 Contributions

This thesis presents QuickInv, a dynamic invariant discovery system built on top of a theory exploration system. Additionally, we build upon previous work by Reynolds [9] and present a grammar for expressing high-level program invariants for finite array programs with a theory exploration system. We evaluate the system by rediscovering the required loop invariants for a set of example array programs: binary search, linear search, selection sort, and merge sort. Finally, we present problems and limitations of the system observed in these examples and suggest possible solutions to the problems.

The research questions we address are:

- What is a suitable grammar for expressing high-level program invariants for finite array programs?
- How effective is a dynamic, theory-exploration-based approach to discovering program invariants?

1.2 Roadmap

This thesis is organized as follows: Section 2 provides the necessary background, introducing the principles of Floyd-Hoare logic, the invariant inference problem and theory exploration. Section 3 presents our core conceptual contribution, reframing the problem of invariant discovery as a theory exploration problem. Section 4 details the implementation of our system QuickInv; explaining how it captures program states and discovers program invariants. Section 5 focuses on the challenge of designing an effective and expressive grammar for these invariants. Section 6 evaluates the system's performance on a set of common array algorithms. Section 7 discusses related work in the field of program analysis and invariant inference. Finally, Section 8 concludes the thesis by discussing the research questions, our contributions, and future work.

2

Background

Program invariants are a fundamental concept in program verification, but also useful for program documentation, regression testing, and reasoning about program behavior. While useful in other contexts, this thesis motivates automatic program invariants discovery as a way to assist program verification using Floyd-Hoare logic.

2.1 Floyd-Hoare Logic and Invariant Inference

Floyd-Hoare logic [1], [2], is a foundational formal system for reasoning about the functional correctness of computer programs. At its core is the *Hoare triple* $\{P\}C\{Q\}$. This notation expresses that if the *pre-condition* P holds before executing the command C, then the *post-condition* Q will hold afterwards assuming that C terminates. This is known as *partial correctness*. A stronger notion of correctness is *total correctness* which, in addition, requires that C terminates. Floyd-Hoare logic includes a set of axioms and inference rules that support the inductive derivation of valid Hoare triples over the structure of program commands. This compositional approach allows the correctness of complex programs to be established from the correctness of their parts. Termination is often proved using a *variant*, a well-founded measure that decreases in loops and recursive functions, ensuring that the program will eventually terminate.

A particularly important rule is the *while rule* (Equation 1) which states that a loop is partially correct if it satisfies three conditions: (1) a property we call the *loop invariant* I holds before the loop starts – it follows from the pre-condition P, (2) the loop invariant I is maintained over each iteration of the loop, (3) when the loop terminates, the invariant and the negation of the loop guard G imply the post-condition Q. In particular, the loop invariant I is a property that holds before the loop starts, is maintained over each iteration of the loop, and holds after the loop terminates. The loop invariant I is a property that needs to be specified for the proof and needs to satisfy the three conditions of the while rule.

$$\frac{P \Rightarrow I \quad \{I \wedge G\}C\{I\} \quad (I \wedge \neg G) \Rightarrow Q}{\{P\}\ \texttt{while}\ G\ \texttt{do}\ C\{Q\}} \tag{1}$$

The while rule provides a systematic way to reason about loops, which are inherently complex due to their potentially unbounded execution. It allows verification of partial correctness by using a loop invariant I, without having to unfold the loop – it is

enough to proof three simpler conditions. The while rule bridges the gap between static reasoning and dynamic behavior, making program verification practical.

Program verifiers such as Dafny [4], based on Floyd-Hoare logic, can automatically and efficiently verify functional correctness of programs by checking the validity of Hoare triples: given suitable pre-conditions, post-conditions, and loop invariants. Post-conditions – describing the expected behavior of the program after execution – and pre-conditions – describing the valid inputs to the program – are often easy to specify. For example the post-condition of the binary search program in Listing 1 is that the return value is the index where the searched value exists in the array, or the special value -1 if the value does not appear in the array, and the pre-condition is that the input array is sorted. Proving termination is also easy; for binary search, it is sufficient to choose the variant $lo - hi$ as the measure that decreases with each loop iteration.

However, inventing loop invariants remains a major bottleneck; it needs to be strong enough to ensure the post-condition holds after the loop, yet weak enough to hold before and after every loop iteration. This is known as the invariant inference problem. It is a non-trivial task, especially for complex loops or entire programs, and often requires deep understanding of the program's behavior and structure. Manual loop invariant inference is often a trial-and-error process. Therefore, automating invariant discovery is highly desirable.

Existing methods for automatically inferring loop invariants work with low-level first-order logic, which is often very verbose, hard to read, and laborious to reason with. For example, Equation 2 is the loop invariant for the binary search program (Listing 1) expressed in low-level first-order logic[1].

$$(\exists i.\ 0 \leq i < |arr| \land arr[i] = key) \iff (\exists i.\ lo \leq i < hi \land arr[i] = key) \tag{2}$$

Previous work [10] suggests a need for expressing invariants as high-level properties based on what they call domain theory. By abstracting common patterns and concepts as high-level constructs, we allow for clarity and expressiveness in the invariants. By introducing domain-specific operators $\in$ and array slicing we can express the binary search loop invariant as in Equation 3. We call the set of constants and operators used to express invariants the *invariant grammar*.

$$key \in arr \iff key \in arr[lo..hi) \tag{3}$$

A common class of programs that are less supported by existing methods are array programs, which are programs that manipulate or examine arrays, e.g. the binary search program in Listing 1. These programs are often more complex and requires invariants that are large and awkward to express in a low-level invariant grammar. Previous work

[1]It is possible to verify binary search in Listing 1 with the invariant expressed in other ways that can be shorter, e.g. the invariant $\forall i.\ 0 \leq i < |arr| \land \neg(lo \leq i < hi) \implies arr[i] \neq key$ is a bit shorter. This invariant is also enough to verify this implementation of binary search, although not equivalent to Equation 2. This invariant would for example not hold for a variation of binary search that return the index of the first occurrence of *key*, while Equation 2 would still be valid. This invariant, although shorter, is still less intuitive than Equation 3 and complex as it has a quantifier.

[9] presents high-level concepts for reasoning about array programs, which is shown to be effective for constructing manual formal proofs for some array programs. This thesis takes inspiration from these concepts to construct a high-level grammar for expressing program invariants for finite array programs. We use a theory exploration system to discover invariants with a flexible grammar that is mutable by the user, allowing the user to add domain-specific functions and constants to the grammar.

2.2 Theory Exploration and QuickSpec

Theory exploration is a method for systematically discovering interesting lemmas, axioms, or properties within a given formal theory. It involves generating conjectures and attempting to validate them using automated reasoning and testing techniques.

QuickSpec [11] is a theory exploration system that discovers equational properties of a set of definitions in Haskell – useful for documentation, program understanding, and regression testing of functional programs. It systematically explores the behavior of a given set of functions and constants and generates conjectures (in the form of an equation) about their relationships.

The input to QuickSpec is a set of functions and constants, and their type signatures, e.g. some common list functions in Haskell:

$$\texttt{(++)} : [Int] \times [Int] \to [Int] \text{ -- list append}$$
$$\texttt{(:)} : Int \times [Int] \to [Int] \text{ -- list cons}$$
$$\texttt{[]} : [Int]$$
$$\texttt{reverse} : [Int] \to [Int].$$

QuickSpec discovers and reports the following properties about these functions and constants:

```
1. reverse [] = []
2. xs ++ [] = xs
3. [] ++ xs = xs
4. reverse (reverse xs) = xs
5. reverse (x : []) = x : []
6. (x : xs) ++ ys = x : (xs ++ ys)
7. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
8. reverse xs ++ reverse ys = reverse (ys ++ xs)
9. xs ++ (x : (y : [])) = reverse (y : (x : reverse xs))
```

where `xs`, `ys`, and `zs` are variables of type `[Int]`, and `x` and `y` are variables of type `Int`. QuickSpec automatically discovers the associativity and unit laws for append and all properties of the `reverse` function.

The variables `x`, `y`, `xs`, `ys`, `zs` in the equations, found by QuickSpec, are bound by the universal quantifier. The closed form of property 4 in the output of QuickSpec above, for example, is

$$\forall\ xs : [Int].\ reverse\ (reverse\ xs) = xs \tag{4}$$

QuickSpec explores the equational theory of a set of functions by systematically generating terms through the application of constants and functions to variables and smaller terms, bounded by a configurable term size limit. These terms are then evaluated on randomly sampled values for the universally quantified variables, and terms that produce identical results are grouped into equivalence classes. By leveraging the mathematical properties of these classes, QuickSpec efficiently identifies candidate equations

that capture functional relationships. Importantly, it doesn't rely solely on brute-force testing: QuickSpec integrates lightweight reasoning – often faster than testing – to prune redundant terms and eliminate trivial or derivable equations using rewrite rules. These and other optimizations make its search space surprisingly tractable, enabling QuickSpec to quickly generate meaningful and minimal sets of equations, even for complex programs.

QuickSpec uses the input functions and constants to build type-correct terms. Starting with the simplest terms (e.g. $[]$, xs), it applies the functions to these terms, generating larger terms (e.g. $x : []$, $xs +\!\!+ ys$, $reverse(xs)$), and searches for equations that hold between these terms incrementally. The term size limit controls how large terms can be, and thus how vast the search space is. The term $xs +\!\!+ ys$, for example, has size 3.

A key strength of QuickSpec is to perform automated reasoning to prune properties that are derivable from previously discovered properties. The pruner uses rewrite rules – built from previously discovered equations – to simplify terms on both sides of the equation, to eliminate equations between two terms that rewrite to the same term. Suppose QuickSpec has discovered the 9 properties above. Now consider the candidate equation $reverse(reverse([] +\!\!+ xs)) = xs +\!\!+ reverse([])$ This equation is redundant, and the pruner can eliminate it by applying previously discovered rules:

$$
\begin{aligned}
&reverse(reverse([] +\!\!+ xs)) = xs +\!\!+ reverse([]) && \{\text{law 3 to left-hand side}\}\\
&reverse(reverse(xs)) = xs +\!\!+ reverse([]) && \{\text{law 4 to right-hand side}\}\\
&xs = xs +\!\!+ reverse([]) && \{\text{law 1 to right-hand side}\}\\
&xs = xs +\!\!+ [] && \{\text{law 2 to right-hand side}\}\\
&xs = xs\,.
\end{aligned}
$$

It is possible to sequence multiple runs of QuickSpec by feeding the laws discovered in one run into the next. In this way, the equations from the first run serve as a background theory for subsequent runs. QuickSpec uses this background theory during pruning: it simplifies terms and eliminates equations that are derivable from the previously discovered laws. Additionally, QuickSpec can be configured to use a user-defined set of equations as background theory, enabling more controlled and modular exploration.

A notable limitation of QuickSpec is the limited support for discovering conditional equations – that is, equations that hold only under certain conditions. Many general properties such as $x \leq y \wedge y \leq z \Rightarrow x \leq z$ (transitivity) expressed as an equation $x \leq y = True = y \leq z = True \Rightarrow x \leq z = True$ are not discovered by QuickSpec. QuickSpec can discover some conditional equations, but it requires the user to provide a *predicate* and a *generator* for generating random input for which the predicates holds. Given the predicate $P(x, y, z) = x \leq y \wedge y \leq z$, QuickSpec can discover the conditional equation $P(x, y, z) = True \Rightarrow x \leq z = True$. To do this, QuickSpec expresses the conditional equation as a simple equation by introducing variables for the arguments of $P(x, y, z)$: P_0, P_1, and P_2 denoting x, y, and z respectively. The equation that QuickSpec discovers is $P_0 \leq P_2 = True$; because these variables are prefixed with P, QuickSpec knows they are special variables and samples random values for them from the generator provided

by the user. This will ensure that the values for which the equation is tested satisfy the predicate $P(x, y, z)$, and thus the equation holds only when the predicate holds.

This encoding of conditional equations imposes a limitation that the predicates in the condition cannot share arguments. Otherwise QuickSpec would not know how to generate a value for the shared variable that fulfills both predicates at the same time. If we use the predicate $P(x, y) = x \leq y$, QuickSpec will not discover the conditional equation $P(x, y) = \mathit{True} \wedge P(y, z) = \mathit{True} \Rightarrow P(x, z) = \mathit{True}$ because the two predicates share the variable y.

3

A New Lens on Invariant Discovery – Theory Exploration

The fundamental challenge in invariant discovery is the vast search space of potential invariants. Consider the elements present in a program invariant: program variables, constants, operators, and quantifiers. The number of possible properties (equations, inequalities, logical statements) that can be formed from these elements is combinatorially explosive. A naive "generate-and-test" approach, where one blindly constructs and checks all possible expressions, would be computationally infeasible for anything beyond trivial programs or grammars.

Even if we could test many properties, a significant portion would be:

- Trivial: e.g., $x = x$, $0 \leq 0$.
- Redundant: Logically implied by simpler, already known properties (e.g., if $t = u$ is an invariant, then $f(t) = f(u)$ is also an invariant for any function f, but less fundamental).
- False positive: True only by coincidence for a few test cases but not a general property.

We are interested in fundamental, concise, and non-obvious properties that characterize the program's behavior.

This thesis suggests theory exploration as a solution strategy for high-level invariant discovery. The key strengths of theory exploration, relevant for invariant discovery, are: (1) Flexible grammar with user-provided functions and constants, (2) systematic and efficient exploration of the search space, (3) automated reasoning to prune redundant and derivable properties from previously discovered properties. These strengths of theory exploration make it a suitable approach for discovering program invariants, as it allows effective navigation of the vast search space of potential invariants, while focusing computational effort on potentially new and interesting properties.

To understand how theory exploration can be applied to invariant discovery, we imagine the program's variables, and the operations applicable to them (as defined in the invariant grammar) as forming a small "mathematical theory." The program's execution, at specific points where we want to find invariants, provides concrete instances or models of this theory. An invariant is essentially a "theorem" that holds true across

all these observed instances (and ideally, all possible valid instances). The challenge of finding invariants then becomes analogous to exploring this implicit theory to uncover its inherent properties that are consistently satisfied by the program's runtime behavior.

By framing invariant discovery as a theory exploration problem, we can leverage the powerful machinery of existing theory exploration systems. This forms the basis of QuickInv: we reduce the task of inferring program invariants to a theory exploration problem, which we then solve using the theory exploration system QuickSpec. To achieve this, we must express program invariants as closed logical formulas in a form suitable for theory exploration, and determine the appropriate input for theory exploration.

3.1 Program Invariants as Closed Logical Formulas

We aim to find program invariants using theory exploration, but the closed form of a program invariant is not always clear from the notation. In this section, we derive a closed form suitable for theory exploration to illustrate how this technique can be used to discover program invariants.

Consider the loop invariant in Equation 5 for the `binary search` program in Listing 1. Invariants are logical formulas that involves program variables[2], constants, quantifiers, and operators which are pure functions. In this example the program variables are arr, lo, hi, and key, and the operators are $\in$, $\iff$, and the array slicing operator.

$$key \in arr \iff key \in arr[lo..hi) \tag{5}$$

In this notation, the program variables are free variables, but it is clear that the invariant does not hold for any arbitrary values assigned to the program variables. For example the assignment $key \mapsto 0, arr \mapsto [0,1,2], lo \mapsto 1, hi \mapsto 3$ does not satisfy the formula, since 0 is in the whole array but not in the slice $[1..3)$ of the array $[0,1,2]$. This state, however, is not obtainable at any of the points in the program where the loop invariant should hold, because for the state to be reached, the loop would have to set the lo variable to mid when $arr[mid] < key$. This is the opposite of how the program is implemented in Listing 1. The invariant only need to hold for obtainable program states; this should be reflected in the closed form of the invariant.

Let $\mathbb{S}$ denote the set of all program states s and $\mathcal{S}_{\mathcal{P}} \subseteq \mathbb{S}$ denote the set of all program states s obtainable by the program execution at a specific set of point $\mathcal{P}$ in the program. In this example a program state s over the loop can be represented by the tuple (key, arr, lo, hi) consisting of input to the program and loop variables. A closed form of the invariant without free variables is:

$$\forall (key, arr, lo, hi) \in \mathcal{S}_{\mathcal{P}}.\ key \in arr \iff key \in arr[lo..hi) \tag{6}$$

[2]Invariants (specifically post-conditions and loop invariants) can also refer to original values of program input, e.g. the post-condition for a sorting algorithm needs to express that the output array is a permutation of the input array with the invariant $multiset(arr) = multiset(old(arr))$. We can always treat invariants as a one-state function by recording a copy of the original value as part of the program state. Only mutable program inputs would require this. We prefix the original value with *old* in the program state entry [10].

We can express the invariant in a more theory-exploration-friendly way by modeling program variables as functions from program states to values, where the type of program states is denoted by S. In this representation we have: $key : S \to Bool$, $arr : S \to [Int]$, $lo : S \to Int$, and $hi : S \to Int$ as used in Equation 7.

$$\forall s \in \mathcal{S}_{\mathcal{P}}.\ key(s) \in arr(s) \iff key(s) \in arr(s)[lo(s)..hi(s)) \tag{7}$$

We can also optimize this formula for theory exploration with QuickSpec by replacing the logical implication operator with an equality. Furthermore, we can use the prefix operator $slice : [a] \times Int \times Int \to [a]$ to denote the array slicing operation. We thereby obtain a universally quantified formula in equational form (Equation 8).

$$\forall s \in \mathcal{S}_{\mathcal{P}}.\ key(s) \in arr(s) = key(s) \in slice(arr(s), lo(s), hi(s)) \tag{8}$$

We now have a closed logical formula that is suitable for theory exploration using QuickSpec – the formula is closed and only contains operators, constants, and variables bound by a universal quantifier . These operators and constants can serve as input to QuickSpec, which can then discover the invariant.

Although this form can be discovered by QuickSpec, we can make it nicer to present by getting rid of the function applications of the program variables to s. To do so we lift the operators:

$$(\in) : a \times [a] \to Bool$$
$$slice : [a] \times Int \times Int \to [a]$$

to take a function from program state S to values as arguments and result:

$$(\in) : (S \to a) \times (S \to [a]) \to S \to Bool$$
$$slice : (S \to [a]) \times (S \to Int) \times (S \to Int) \to S \to [a]$$

Now the expression $key \in arr = key \in slice(arr, lo, hi)$ has a type $S \to Bool$ in Equation 9. Given a program state s, the expression can be evaluated to a Boolean.

$$\forall s \in \mathcal{S}_{\mathcal{P}}.[key \in arr = key \in slice(arr, lo, hi)](s) \tag{9}$$

Omitting the universal quantifier (as is done by QuickSpec when presenting properties) give us the final form of the invariant in Equation 10.

$$key \in arr = key \in slice(arr, lo, hi) \tag{10}$$

This is an equivalent form of the original invariant in Equation 5 (albeit with different types), which now can be discovered by QuickSpec. In other words, by running QuickSpec with the input functions and constants from Equation 8 and allowing it to evaluate terms internally on the obtainable program states $\mathcal{S}_{\mathcal{P}}$, we can discover the invariant in Equation 10.

4

How QuickInv Works

QuickInv discovers program invariants by reframing the problem as a theory exploration task. The core methodology involves a three-step pipeline: first, it captures a diverse set of program states by executing the target program; second, it represents program variables and constants in a format suitable for the theory exploration engine QuickSpec; and third, it uses a sequenced, two-phase QuickSpec exploration process to distinguish general mathematical truths and discover a smaller set of specific program invariants.

The entire workflow, from code to invariant, is illustrated in Figure 2. It begins by targeting a specific part of a program for observation. For instance, the while loop in a binary search implementation, which is highlighted in the Figure 2. The user instruments the program to record a log of its state at each point of interest. In this case, before the loop and after each iteration. The program is executed many times with random inputs. The result is a collection of captured execution traces, represented in the figure as a table of variable values (arr, key, lo, hi, idx). This table of observed states serves as the empirical evidence for the discovery process.

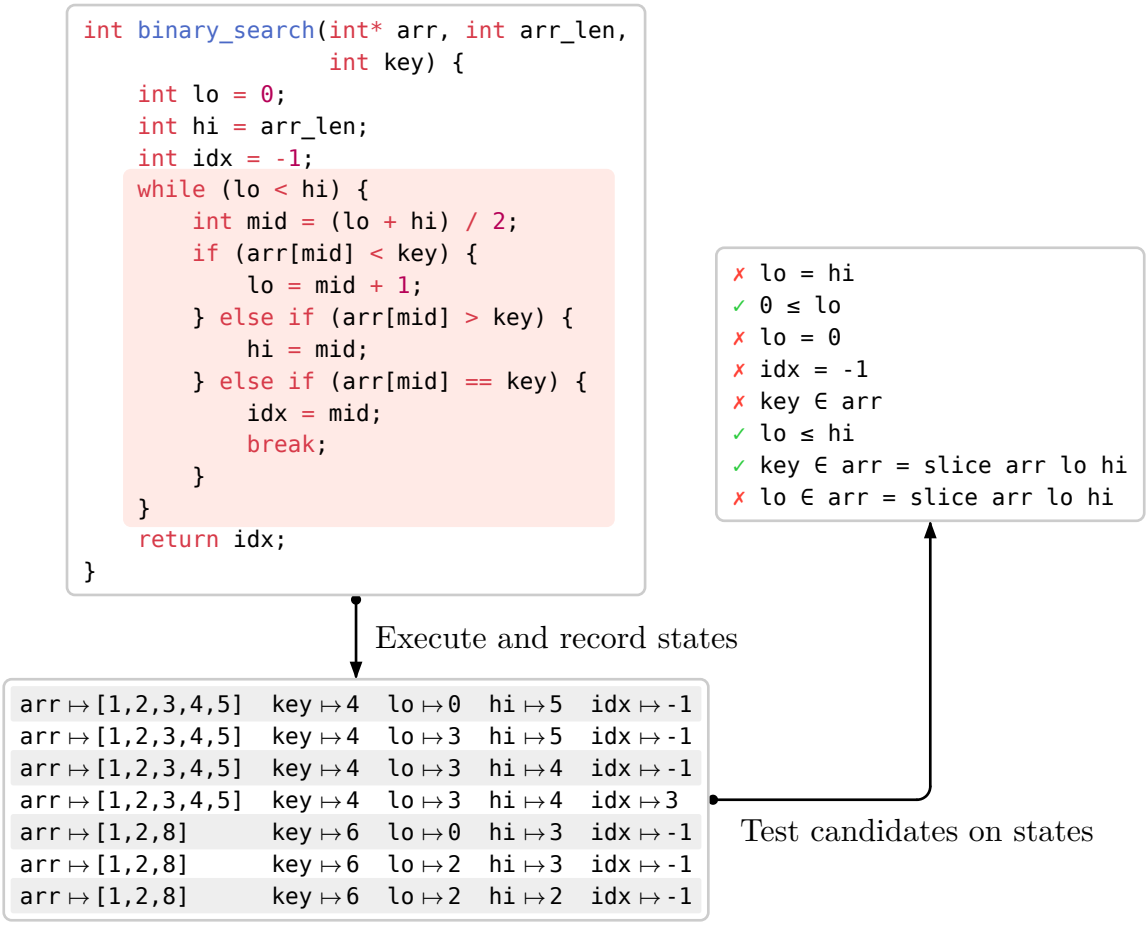

Figure 2 : Illustration of QuickInvs pipeline: targeting a program invariant, program state data collection and testing candidate invariants.

With this data in hand, QuickInv systematically generates and tests candidate invariants using QuickSpec. The figure depicts this as a list of potential properties, where each one is checked against the recorded states. Properties that fail for even one state (e.g., $lo = hi$) are falsified and crossed out. Those that hold true for the tested observed states (e.g., $0 \leq lo$ and $key \in arr = key \in slice\ arr\ lo\ hi$) and are not derivable from previously discovered properties by simple reasoning are accepted (marked with a tick), and presented as likely invariants.

The following sections detail the mechanics behind each part of this pipeline: how states are captured and represented, and how the two-phase exploration process effectively distinguishes program-specific invariants from general truths, while discussing key configuration options for QuickInv.

4.1 Capturing Program States from Execution Traces

The foundation of any dynamic analysis is the data it observes. In this thesis, the target program is manually instrumented with logging statements to capture program states at a specific set of points $\mathcal{P}$ of interest. For a loop invariant, this is before the loop, after each iteration, and after the loop. For a pre-condition and post-condition, it is

before and after the program execution respectively. The target program is executed many times with random inputs to collect the states data.

QuickInv needs a user-defined representation of a program state of a target program. In the examples we represent a program state with a Haskell data type with named fields for each program variable and input of interest. For example, the program state for the binary search program in Listing 1 is represented by:

```
data State = State
    { key :: Int
    , arr :: [Int]
    , lo  :: Int
    , hi  :: Int
    , idx :: Int
    }
```

In Haskell, defining a data type like this automatically generates field access functions (also known as getters) for each field in the record. That means after this definition, the following functions are available:

```
key :: State -> Int
arr :: State -> [Int]
lo  :: State -> Int
hi  :: State -> Int
idx :: State -> Int
```

To gather data, QuickInv executes the user-instrumented target program multiple times with randomized inputs. This data is sampled by QuickInv to randomly test candidates on by evaluation. A naive approach of using a uniform distribution for inputs can be problematic, as it may overrepresented states from large inputs (e.g., large arrays) and miss crucial edge-case behaviors (e.g., empty arrays, single-element arrays).

Running the program with the input $key \mapsto 1, arr \mapsto [1,2,3,4]$ captures the program states:

```
[State {key = 1, arr = [1,2,3,4], lo = 0, hi = 4, idx = -1},
 State {key = 1, arr = [1,2,3,4], lo = 0, hi = 2, idx = -1},
 State {key = 1, arr = [1,2,3,4], lo = 2, hi = 2, idx = 2}]
```

while running the program with the input $key \mapsto 5, arr \mapsto []$ captures the program states:

```
[State {key = 5, arr = [], lo = 0, hi = 0, idx = -1}]
```

The naive approach of collecting program states in a list is not sufficient to ensure that the program states are representative of the program's behavior. In this case there are far more program states where the array is non-empty than there are program states where the array is empty; the likelihood of randomly sampling the state with empty array is $\frac{1}{4}$ in this example. We want all the inputs that we run the program with to be equally likely to be sampled, but some inputs generates many more program states. Because we are targeting the loop, we capture far more program states where *arr* is non-empty (from the first run) than where it is empty (from the second run). This can

lead to missing edge cases, and as a result, discover false invariants that do not hold for all program states. For example, the invariant `0 < length arr` would be more likely to be discovered, which is not true for all program states in $\mathcal{S}_{\mathcal{P}}$.

To mitigate this, QuickInv stores program states in a nested array. The outer array is indexed by different inputs to the program being recorded, while the inner array contains the program states for each input. We sample a program state from this nested array by first sampling a random input and then sampling a random program state from the inner array. This approach ensures that we can sample states generated by small inputs more evenly, and states generated by large inputs are not overrepresented.

In the binary search example, this nested array would look like:

```
[ [State {key = 1, arr = [1,2,3,4], lo = 0, hi = 4, idx = -1},
   State {key = 1, arr = [1,2,3,4], lo = 0, hi = 2, idx = -1},
   State {key = 1, arr = [1,2,3,4], lo = 2, hi = 2, idx = 2}]
, [State {key = 5, arr = [], lo = 0, hi = 0, idx = -1}]
]
```

Sampling a program state from this nested array would first sample an index from the outer array, and then sample a random index from the inner array, making the likelihood of sampling the state where *arr* is empty $\frac{1}{2}$.

We have used QuickCheck [12], a random testing library for Haskell, to generate random inputs for the target program. QuickCheck provides a powerful way to generate random inputs based on the types of the program variables, ensuring that the inputs are valid and cover a wide range of possible values. In our evaluation, we found that this approach significantly improves the quality of the discovered invariants, minimizing the discovery of false positives. For the example programs in this thesis, the target programs were executed with 5 000 random inputs to collect program states.

Although there are no strict requirements on how program states are collected – for example, they could simply be printed to a file – we use Haskell's foreign function interface (FFI) to call C programs (the implementation language of our example programs) from Haskell. A callback function is provided to collect the program states during execution.

4.2 Program States and Variables

QuickInv represents *program values* as functions from program states to values.

```
newtype Prog s a = Prog (s -> a)
```

QuickInv exports the `con` and `var` functions to construct program variables and constants:

```
con :: a -> Prog s a
con = Prog . const

var :: (s -> a) -> Prog s a
var = Prog
```

We can now construct program variables and constants for binary search example as follows:

```
var key :: Prog State Int
var arr :: Prog State [Int]
var lo  :: Prog State Int
var hi  :: Prog State Int
var idx :: Prog State Int

con 0 :: Prog State Int
con False :: Prog State Bool
con [] :: Prog State [Int]
```

From this point onward in the thesis, we overload program variables and constants such as `key`, `arr`, `0`, `[]`, etc., to refer to program values of type `Prog s a`. For example, in the expression `key ∈ arr`, we have `key :: Prog s Int` and `arr :: Prog s [Int]`.

Operations on program variables Functions on program variables are lifted versions of functions on normal values. For example, the `length` function is defined as a lifted function that takes a program value for an array and returns an integer program variable that is the length of the original program array. The lifted `length` function is defined as follows:

```
plength :: Prog s [a] -> Prog s Int
plength (Prog arr) = Prog (\s -> length (arr s))
```

With an Applicative Functor instance for `Prog`, we can lift functions easily.

```
plength :: Prog s [a] -> Prog s Int
plength = fmap length

pslice :: Prog s [a] -> Prog s Int -> Prog s Int -> Prog s [a]
pslice = liftA3 slice
```

Comparing program variables and constants Expressions involving lifted operators and constants are of type `Prog` (a wrapped function `s -> a`) and cannot be compared directly. To compare program values we use observational equality; Two program values are equal if they are equal in all states. This is done in QuickSpec by providing a function `observe` that takes a program state and a program value and returns the value of the program value in that state. Program variables are now compared by comparing their values in a given state (`t == u` iff `∀s. observe s t == observe s u`).

```
instance (Ord a, Eq a, Arbitrary s) => Observe s a (Prog s a) where
    observe :: s -> Prog s a -> a
    observe state (Prog f) = f state
```

To test whether the invariant `key ∈ arr = key ∈ (slice arr lo hi)` holds, QuickSpec will sample program states `s` randomly from the collected program states and evaluates whether `observe s (key ∈ arr) = observe s (key ∈ (slice arr lo hi))` holds. This is an equality over Boolean values, which can be compared directly. If it samples a state where the left-hand side differs from the right-hand side, then the invariant is falsified.

Otherwise, if it is equal for all sampled states then the invariant is considered true and presented to the user.

4.3 Sequencing Theory Exploration

Running QuickSpec with the program variables and constants defined above will result in discovering many laws about the program variables and constants. However, many of these laws are not useful as program invariants. For example, laws such as `0 = length []`, `x ≤ x`, and `0 ≤ 1` are true for any program state – general mathematical truths. They are not entirely useless though; they are useful for reasoning about the program and pruning laws in QuickSpec. Knowing that `x ≤ x`, we do not need to test and report laws such as `lo ≤ lo` and `hi ≤ hi`, since they follow from `x ≤ x`. QuickSpec prunes laws that follow from previously discovered laws and avoid testing and reporting them to the user.

QuickInv sequences two phases of theory exploration with QuickSpec. First, QuickSpec is run with functions and constants in the invariant grammar – discovering general laws about the grammar (e.g. `sorted []` and `slice a x x = []`), which are general mathematical truths. We call this phase the *background phase*. The laws discovered in this first phase are used as background theory in the second phase, which is run with functions and constants in the invariant grammar, and in addition, the program variables. We call the second phase the *invariant discovery phase*. The pruner in QuickSpec uses the background theory to prune derivable laws, avoiding testing and reporting them to the user. For example, since the laws `sorted []` and `slice a x x = []` were discovered in the first run, the pruner will not test and report the law `sorted (slice arr lo lo)` in the second run, since it can be rewritten as `sorted []` using `slice a x x = []`. We already know that `sorted []` is true. The second phase discovers laws that involves program variables, e.g. `key in arr = key in slice arr lo hi`, which do not follow from any laws discovered previously.

In QuickSpec, Testing effort is spent on final laws that are presented to the user, reducing the number of spurious laws leads to better performance as well, as testing a true law is the most expensive part of the process.

The background phase for the binary search example discovers 98 laws, some of which being:

```
 1. sorted []
 2. 0 = length []
 3. x ≤ x
 4. x ∈ [] = False
```

The invariant discovery phase of theory exploration discovers 22 laws, some of which being:

```
 5. 0 ≤ lo
 7. lo ≤ hi
 9. hi ≤ length arr
13. key ∈ slice arr lo hi = key ∈ arr
```

These are the required loop invariants for the verification of the binary search program in Listing 1.

4.4 Configuring QuickInv

QuickInv is highly configurable, allowing users to tailor the invariant discovery process to their program domain, balance performance and completeness, and shape the final output. These configurations are inherited from QuickSpec. This section outlines the key configuration options, useful for tuning the invariant discovery process.

Invariant Grammar The core configuration is the invariant grammar – the set of Haskell functions and constants that defines the space of possible invariants. QuickInv provides a standard prelude of common functions (see Section 5.1), but users can modify this by adding or removing definitions, such as domain-specific operators or arithmetic functions. The design of this grammar is crucial and is discussed in detail in Section 5.

Background Theory QuickInv supports the use of a background theory – a set of known equational laws used to eliminate redundant properties. Users can provide their own axioms, especially for properties that QuickSpec cannot discover, such as the transitivity of $\leq$ ($x \leq y \land y \leq z \Rightarrow x \leq z$). These help reduce spurious and redundant invariants and improve output quality.

Filtering Terms and Properties To reduce search complexity and irrelevant results, QuickInv allows custom filters. *Enumerator filters* discard syntactically unhelpful terms – like nested `slice` calls (`slice (slice ...)`) which are equivalent to slicing with an offset – which can speed up the search. *Print filters* remove discovered laws that are technically valid but not meaningful in context from being presented to the user.

Tuning the Exploration Engine QuickInv inherits key tuning parameters from QuickSpec. The most important is the *maximum term size*, which controls the complexity of discoverable invariants. Larger limits uncover more complex laws but expand the search space rapidly. Users can also configure the *number of test cases* used to validate properties – higher values increase confidence but also slow down execution.

5

Designing a Grammar

This section describes the grammar used in QuickInv, followed by some of the challenges in designing an expressive grammar that works well for finding invariants using theory exploration; for example, how to deal with partial functions and how to avoid redundant and false laws.

5.1 Program Invariant Grammar

Designing a good set of constants and functions to be able to discover program invariants is essential. If an invariant cannot be expressed with the functions and constants and program variables, we will not be able to discover it. Additionally, some functions and constants are useful during reasoning and pruning. To obtain a compact set of laws presented to the user, we also need to discover such general mathematical truths.

An obvious grammar could include low-level functions and constants such as $\wedge$, $\vee$, $\Rightarrow$, $\leq$, $<$, $\neg$, $\forall k$, $\neq$, $-$, $arr[?]$, 0, 1, $true$. We can express that `arr` is sorted between index 0 and i with $(\forall k.0 < k < i \implies arr[k-1] \leq arr[k]) = true$ in equation form.

In our method, we instead include high-level functions for arrays such as `slice`, `∈`, `arr[?]`, `length`, `sorted`, `0`, `True`, `≤`. We can express that `arr` is sorted between 0 and i with `sorted (slice arr 0 i) = True` as an equation.

The latter grammar has two advantages. Firstly, it is more intuitive and easier to understand compared to the former, which needs some deciphering to understand. Secondly, the term size needed to express the former is much larger than the latter. The equation in the former grammar has a left-hand side term size of 14. The equation in the latter grammar has an expression size of 5. Since, QuickSpec iterates all terms up to a term size, with the former grammar many useful terms would quickly become infeasible to generate because of the explosive blowup in the number of combinations needed to consider.

We specialize the grammar to the domain of array programs and, in that way minimize the expression size while allowing expression of intuitive relationships. We also avoid quantifiers like $\forall$ and $\exists$ by limiting our domain to finite array programs – we can test finite arrays instead of relying on quantifiers. Although QuickSpec can discover invariants with universal quantification, we avoid them in the invariants we present to

the user. With this method, we limit the expressivity, but with a well-designed grammar for the target domain, all useful relationships can hopefully be expressed. QuickInv can be specialized for other domains by choosing a different set of functions and constants.

Reasoning about arrays requires common operations on arrays and ways to compare arrays, their elements, and their properties. Previous work by Reynolds [9] has found useful operations and their properties for reasoning about arrays. Although Reynolds' work is not directly applicable to our problem, it provides a good starting point for designing our grammar. We can use the operations and properties found by Reynolds as a basis for our grammar. Reynolds treats arrays as functions from indices to values. We instead treat arrays as a list of values, which is more convenient for our purpose.

We have defined a grammar prelude for QuickInv with useful functions and constants for expressing invariants common in many array programs. The prelude can be seen as a good starting point, but functions and constants can easily be added or removed as needed.

QuickInv Prelude

- `True :: Bool` - true
- `False :: Bool` - false
- `0 :: Int` - zero
- `1 :: Int` - one
- `[] :: [Int]` - empty array

The prelude contains the elementary constants like `True` and `False`, `0`, `1` and `[]`. The constant `True` lets us discover predicates as equational laws, for example, `sorted arr = True`, but these laws will be presented as just `sorted arr` to the user.

- `slice :: [Int] -> Int -> Int -> [Int]` - slicing an array
- `!! :: [Int] -> Int -> Int` - array indexing
- `length :: [Int] -> Int` - length of an array
- `∈ :: Int -> [Int] -> Bool` - membership of an element in an array
- `sorted :: [Int] -> Bool` - whether an array is sorted

Our grammar includes array operations such as slicing, indexing, and array length. The constant `0` and `length arr` are useful for slicing from the beginning or to the end of the array, respectively. The grammar also includes the predicates `∈` and `sorted`, which are useful for algorithms that reason about array containment or sortedness, like searching or sorting algorithms.

- `≤ :: Int -> Int -> Bool` - less than or equal to

Moreover, the grammar includes operators to express inequalities, so we can find invariants like $0 \leq i \leq |arr|$ with the laws `0 ≤ i = True` and `i ≤ length arr = True`. The relation $<$ is excluded from the grammar because `x < y` can be expressed with the equation `y ≤ x = False` and having multiple ways to express the same thing is generally undesired. A final post-processing step is used to present `y ≤ x = False` as `x < y` to the user.

- `≤* :: [Int] -> [Int] -> Bool` - point-wise extension of `≤` to arrays
- `≤* :: Int -> [Int] -> Bool` - point-wise extension of `≤` to arrays with single argument on the left-hand side
- `≤* :: [Int] -> Int -> Bool` - point-wise extension of `≤` to arrays with single argument on the right-hand side

We also include the $\leq^*$ operator that is the point-wise extension of $\leq$ for arrays as described by Reynolds [9]. This lets us express that elements in one array are less than or equal to all the elements in another array; for example, `slice arr 0 i ≤* slice arr i (length arr)`. Additionally, we have the $\leq^*$ operator with a single element on either side of the operator, which lets us compare a single element with an array. For example, that `x` is the smallest element of an array can be expressed as `x ≤* arr = True`.

- `multiset :: [Int] -> IntMultiSet` - array to multiset conversion
- `∪ :: IntMultiSet -> IntMultiSet` - union of multisets

Lastly, the prelude also includes functions for multisets, which is useful for expressing invariants that an algorithm preserve elements in an array. For example, that a sorting algorithm returns an array with the same elements as the input can be expressed as `multiset arr = multiset old_arr`. An alternative way of expressing this would be to have a `permutation` function in the grammar, but the downside of this would be larger term sizes; for example, the law `multiset (slice arr p r) = multiset (slice old_arr p r)` with a term size of 5 would instead be `permutation (slice arr p r) (slice old_arr p r) = True` with a term size of 9.

Arithmetic Extension

- `+ :: Int -> Int -> Int` - addition
- `- :: Int -> Int -> Int` - subtraction

Arithmetic operations will significantly increase the number of possible terms and number of invariants found, and is therefore offered as an extension. It is recommended to first try to search for invariants without arithmetic operations and then add them if needed.

The grammar can easily be customized to work in different domains by adding any custom Haskell function to the grammar or exclude any of the predefined functions.

5.2 Dealing with Partial Functions (Slice and Indexing)

One of the challenges of designing a grammar that works well with theory exploration is how to deal with partial functions. Partial functions are not defined for all possible inputs. For example, the `slice` function is not defined for negative indices, indices that are out of array bounds and indices where the first index is larger than the second index. The definition of `slice` in Haskell would look something like:

```
slice :: [a] -> Int -> Int -> [a]
slice a x y
  | not (0 <= x && x <= y && y <= length a)
      = error "slice: invalid input"
```

```
  | otherwise
      = [a !! i | i <- [x..y - 1]]
```

Using this definition of `slice` will result in unwanted trivial loop invariants like `slice arr lo lo = []` and `slice arr hi hi = []` to be discovered. This is because there are many general laws about `slice` that need valid inputs to hold, for example, `slice a x x = []`. When this law is absent from the background laws will result in these unwanted invariants to be discovered. This general law cannot be found by QuickSpec unless we provide a predicate `0 ≤ x ≤ length a`. Otherwise, during testing, QuickSpec will try to QuickCheck the law `slice a x x = []` which will error out when it tries to test with an invalid input for `x` (e.g. `x = -1`) and the test will fail with a runtime error. A runtime error is treated as a falsification of the law in QuickSpec.

Generally, the condition required for a law involving a function is that the input is valid and that the input is in the domain of the function. For total functions this condition is always true, but for partial functions this is not the case. Therefore, we need to be able to express the condition that the input is valid.

A general way to do this is to use a predicate for valid inputs for the partial function. For example, we can define a predicate:

```
slice_dom a x y = 0 <= x && x <= y && y <= length a
```

Now QuickSpec can find more laws such as `slice_dom a x y => slice a x x = []`.

There is, however, a problem with this approach. We cannot find laws with other conditions that reuse the variables `a`, `x`, and `y` due to the limitation of how conditionals are handled in QuickSpec. Any law that needs additional conditions on any of the input to `slice` cannot be expressed in QuickSpec. For example, the law `slice_dom a x y && sorted a => sorted (slice a x y)` cannot be found this way because `a` is used in the predicate `slice_dom` and `sorted`. We can find more conditional laws about total functions than we can about partial functions since one condition almost always required for partial functions is that the input is valid.

The question is if we can find laws about partial functions without needing the valid input condition. If we can, then we will be able to find conditional laws just like total functions.

5.2.1 Combined Predicates

One potential solution to this problem is to combine the predicates in the grammar and generate more predicates conjuncted with `slice_dom`. If the grammar has the predicate `sorted` we can also generate the predicate `slice_dom_and_sorted a x y = slice_dom a x y && sorted a`. This way we can find laws such as `slice_dom_and_sorted a x y => sorted (slice a x y)`, but the number of predicates now is at least twice the number of predicates without these combined predicates. This slows down QuickSpec substantially because conditional laws are slow in QuickSpec, due to the way they are checked during pruning.

5.2.2 Generating Valid Input

Another, possibly better, method is to restrict the input to the valid range during testing. This method is used for predicates, where a separate generator is used for generating random values fulfilling the predicate. The same method could help with partial functions, if it is possible.

For example, the law `length (slice a x y) = y - x` will hold for all `x`, `y` and `a` if the inputs tested are in the domain of `slice`. Similarly, the law `sorted a => sorted (slice a x y)` can be discovered without the domain condition.

The difficulty in this method lies in the implementation. How can we handle a property with multiple instances of partial functions? Can we lift the restriction that the inputs are all variables?

For example, how do we generate input for the property `slice (slice a x y) y z = slice a x z`?

The input validity conditions for the property are:

```
0 <= x <= y <= length a &&
0 <= y <= z <= length (slice a x y) &&
0 <= x <= z <= length a
```

How can we generate valid input for this property? We need to combine multiple generators and generate values that all the generators would be able to generate. Is it possible to generate valid indices for the outer `slice` in the left-hand side, since the array input is a term (`slice a x y`), not a variable? These challenges highlight that implementing this method is difficult.

5.2.3 Making Partial Functions Total

Another approach to dealing with partial functions is to extend their definition to handle invalid inputs, making them total functions.

Clamping

One way to make `slice` total is to define it to clamp the input indices to the valid range:

```
slice xs i j
  | not (0 <= i && i <= j && j <= length xs)
        = slice xs (max 0 i) (min (length xs) j)
  | otherwise
        = [xs !! k | k <- [i..j - 1]]
```

This definition of `slice` allows discovering of the laws `slice a x x = []` and `sorted a => sorted (slice a x y)`, but this hinders laws about the length of the slice because of the clamping behavior; for example the law `x ≤ y => length (slice a1 x y) = y - x` since `length (slice [1,2,3] 0 5) = length [1,2,3] = 3 ≠ 5 - 0`.

Randomness

A `slice` definition that preserves the length property is one that uses random values for invalid input:

```
index_dom :: [a] -> Int -> Bool
index_dom a x = 0 <= x && x < length xs

slice :: (Int -> a) -> [a] -> Int -> Int -> [a]
slice def a x y
    | y <= x = []
    | otherwise = [if index_dom a i then a !! i else def i | i <- [x..y - 1]]
```

The argument `def` is a random function that provides a value for indices outside the original array's bounds. The idea is to make the function total, without creating any spurious laws about the structure of the array for invalid input. Using random values is a way to model undefindness during testing, and for a law to hold, the law needs to hold for all `def` functions.

With this definition of `slice`, we can find the laws about length. However, we can no longer find `sorted a => sorted (slice a x y)` without the domain constraint. For example `slice def [1,2,3] (-1) 5` could evaluate to `[8,1,2,3,0,7]`, which is no longer sorted. The absence of this law will result in unwanted redundant loop invariants like `sorted (slice arr 0 lo)`, `sorted (slice 0 hi)` and `sorted (slice lo hi)` for binary search.

Replicate

A total definition of slice that preserves both the length property and the sorted subslice property extends the array by replicating the values at its ends; for example, `slice def [1,2,3] (-1) 5 = [1,1,2,3,3,3]`.

```
slice :: a -> [a] -> Int -> Int -> [a]
slice def [] x y = slice def [def] x y
slice def a x y
    | y <= x       = []
    | x < 0        = replicate (min (-x) y - x) (head a) ++ slice def a 0 y
    | y > length a = slice def a x (length a) ++
                     replicate (min (y - length a) y - x) (last a)
    | otherwise    = [a !! i | i <- [x..y - 1]]
```

The benefit of this definition of `slice` is that it can find both `x ≤ y => length (slice a x y) = y - x` and `sorted a => sorted (slice a x y)`.

When slicing an empty array, there are no values to replicate. Returning an empty array in this case would invalidate the law about the length of the slice. In this case, we instead use a single random value that is replicated for the array slice.

This is the definition of `slice` that is used by QuickInv, and the index function is also based on this definition. However, this definition is still not perfect. It finds some laws about how it behaves for invalid inputs, like `sorted (a x 0)` and `sorted (slice a (length a) x)`. If array concatenation was included in the grammar, this definition

would not work for laws like `slice a 0 x ++ slice a x (length a) = a`; for example, `slice [1,2,3] 0 5 ++ slice [1,2,3] 5 3 = [1,2,3,3,3]`.

5.2.4 Too General Invariants

Unfortunately, when using the extended total function of `slice`, we often discovered invariants with array slicing in a more general form with free variables instead of program invariants. For example, when evaluated on selection sort implemented in Listing 3, we would like to discover the loop invariant $sorted\ arr[0..i)$, but this loop invariant was discarded because it was already known from the more general law $\forall x.\ sorted\ arr[x..i)$. This more general form follows from the fact that all sub-array slices of a sorted array slice are themselves sorted.

Another example is the loop invariant $key \in arr[lo..hi) = key \in arr$ for binary search. This law was also discarded because it was already known since it could be derived from two other laws: $\forall x.\ key \in arr[lo..x) = key \in arr[0..x)$ and $key \in arr[0..hi) = key \in arr$. The derivation steps are shown in Figure 3. The more general forms of this invariant follow from the fact that if an array slice contains an element, then a larger slice containing the original slice will also contain that element.

$$\begin{aligned} &\forall x.\ key \in arr[lo..x) = key \in arr[0..x) \\ &\quad key \in arr[lo..hi) = key \in arr[0..hi) && \text{Instantiate with } x \mapsto hi \\ &\quad key \in arr[lo..hi) = key \in arr && \text{Apply } key \in arr[0..hi) = key \in arr \end{aligned} \tag{11}$$

Figure 3 : Derivation of the invariant $key \in arr[lo..hi) = key \in arr$ of binary search.

To circumvent this undesired behavior, we use the extended total function definition of slice in the background phase, but in the invariant discovery phase, we instead use the partial definition of slice. The partial definition in the invariant discovery phase will cause laws like $\forall x.\ sorted\ arr[x..i)$ to result in a runtime error for invalid values of x during testing and therefore be treated as a falsification of the law by QuickSpec. Laws like $sorted\ arr[0..i)$ will now be discovered because $\forall x.\ sorted\ arr[x..i)$ is discarded and the array slice from 0 to i is always valid during testing. We will also avoid laws like $sorted\ arr[1..i)$ since the array slice will be invalid when testing against a program state where i is 0.

5.3 Too Many Laws

Many of the laws for loop invariants are not necessary for verifying the program, and we would like to avoid as many redundant laws as possible.

A strong background theory allows us to prune away laws that are trivially true. Our slice definition lets us remove trivial laws for binary search, such as `slice arr lo lo = []`, `length (slice arr 0 hi) = hi`, and `sorted (slice arr 0 lo)`, but we find other examples of redundant laws for the example programs we tested.

For example, in selection sort (implemented in Listing 3), we have the loop invariant `multiset old_arr = multiset arr`, but we also get the law `length old_arr = length arr`,

which should follow from the first law. To prune away this law, a background law like `multiset a1 = multiset a2 => length a1 = length a2` is needed, but we do not discover it because QuickSpec cannot find this implication with an equation as the condition. Another redundant law in selection sort that is harder to prune away is `sorted (slice arr i (length arr)) = sorted arr`, and this law follows from the loop invariants `sorted (slice arr 0 i)` and `slice arr 0 i ≤* slice arr i (length arr)`. To get rid of it, we would need the background laws `sorted a1 & a1 ≤* a2 => sorted a2 = sorted (a1 ++ a2)` and `slice a 0 x ++ slice a x (length a) = a`, but the first law has reused variables in the condition and is therefore undiscoverable, and the second law does not hold for our `slice` definition when `x` is outside its valid range.

Sometimes, there will be laws that are not needed to verify the algorithm, but they are not redundant in the sense that they follow from other laws. For example, if we add arithmetic operations to binary search, we discover the law `lo ≤ (hi - lo) = lo ≤ 0`, which given $0 \leq lo \leq hi$ can be reformulated as $lo \neq 0 \implies hi < 2 \cdot lo$. This law follows from what lo and hi states are reachable in binary search, but it is not a loop invariant required to verify the algorithm. This means that there can be laws about the behavior of the implementation that are of no interest when verifying its correctness.

In binary search, we also discover redundant laws like `key ∈ slice arr 0 hi` and `key ∈ slice arr lo (length arr)`, which follow from `key ∈ slice arr lo hi` and `0 ≤ lo ≤ hi ≤ length arr`. These laws follow from that if an array slice contains an element, then a larger slice containing the original slice will also contain that element, which is a type of monotonicity. Pruning away these redundant laws would need a complex background law like `x0 ∈ slice a x1 x2 & x3 ≤ x1 & x2 ≤ x4 => x0 ∈ slice a x3 x4`. This law is undiscoverable because it reuses variables in the conditions and uses more than the four variables that QuickSpec can handle by default; increasing this value would have a performance cost. An alternative, more general approach for redundant laws like this is to reason about monotonicity and antitonicity.

5.3.1 Monotonicity and Antitonicity

We have noticed that the conditional laws needed to reason about some of the behavior of array (especially for reasoning about `slice`) is that we need some laws following from monotonicity and antitonicity. For example to reason about `∈` we need the law `a ⊑ b => (x ∈ a ==> x ∈ b)` which is monotonicity of `x ∈` with the partial order `⊑` on arrays (`infixOf` operator) and `==>` as the partial order on Booleans (implication). For this to work, we need to have implication as a function in the grammar, which is inefficient.

Another one is about sortedness `a ⊑ b => (sorted b ==> sorted a)` which says that `sorted` is an antitone function with respect to the partial order `⊑` on arrays.

We also have that `a ⊑ b => (length a ≤ length b)` which is monotonicity of `length`.

It is potentially interesting to consider that monotonicity and antitonicity properties can be used to embed order for one type in another type. Thus, this fact can be used to avoid order information about one type if one has the order in another type. The

information for one type can be used to reason about the order in another type (order embedding).

5.3.2 Semilattices

We have also looked at how we can use functions that have the structure of semilattices in the grammar, in order to reason about redundant laws. A semilattice consists of a set with an associated binary operation that is associative, commutative, and idempotent.

One example is the longest common subsequence (LCS) operation on a set of all integer arrays. LCS returns the longest subsequence common of two sequences, for example, `lcs [1,2,3,4,5] [1,3,5,7] = [1,3,5]`. Sometimes, there will be multiple subsequences that are the longest, for example, `[1]` and `[2]` are both valid results for `lcs [1,2] [2,1]`. In order for the commutativity property (`lcs x y = lcs y x`) to hold, the returned subsequence cannot depend on the argument order. This can be done by, for example, implementing LCS, so it returns the longest subsequence that is the lexicographically smallest.

LCS can be used to reason about sortedness about sub-slices. We can find the background law `sorted a1 => sorted (lcs a1 a2)`. For example, in binary search, we have the redundant law `sorted (slice arr lo hi)`, which follows from the pre-condition `sorted arr`. If we discover that `lcs arr (slice arr lo hi) = slice arr lo hi`, we can apply the law `sorted a1 => sorted (lcs a1 a2)` where `a1` is `arr` and `a2` is `(slice arr lo hi)` to get `sorted arr => sorted (slice arr lo hi)` and therefore know that the original law was redundant. A limitation with this approach is that we discover `sorted (slice arr lo hi)` before we know it is redundant because `lcs arr (slice arr lo hi) = slice arr lo hi` is discovered later due to its larger term size. Compared to the background law `sorted a => sorted (slice a x y)` that we find with the clamping and replicate version of slice, a benefit with this approach is that it will also work for sub-slices of slices. For example, in merge sort, we have two sorted slices before merging them, and any laws about sub-slices of them being sorted are redundant.

5.4 Undesirable Terms in the Grammar

In the grammar, we have the operator $\leq^*$ that is the point-wise extension of $\leq$ on arrays, which lets us express that elements in one array are less than or equal to all the elements in another array. It may also be useful to have invariants that a single element is less than or equal to all elements in an array slice. There are two ways of expressing this; we could add a version of the operator that takes a single integer on the left-hand side, or we could add the `singleton` function that takes an integer and returns a singleton array.

This second approach has an undesirable effect on the grammar that equality `x = y`, can be expressed with the term `x ∈ singleton y`, which we do not want. QuickSpec is designed to find equalities efficiently without having to test every combination, and having equality expressible in the grammar would lead to inefficiencies and undesired laws like `idx ∈ singleton (length arr) = False`. However, this effect can easily be

avoided by adding an enumeration filter that filters out terms containing `∈ singleton`, as described in Section 4.4.

5.5 False Laws

Another problem is the discovery of false laws. The laws are discovered through random testing, which allows incorrect laws to pass through. Discovered false laws can affect which laws are explored next, potentially leading to the discovery of more false laws and preventing the discovery of true laws as they are pruned away by false laws.

An example of a false law is `length arr ≤ 1 = arr ≤* arr` that sometimes appears. The expression `arr ≤* arr` is only true when all the elements in `arr` are equal to each other, and this is always true when `arr` is empty or contains a single element. This coincides also with `length arr ≤ 1`. When `arr` is a larger array, it is unlikely that all elements are generated to the same value. Therefore, to falsify this law, the generated `arr` needs to have a length of at least two, and all elements need to have the same value, for example `arr = [7,7]`. This law can pass thousands of test cases before being falsified.

The number of false laws can be reduced by increasing the number of test cases and improving the quality of program states. Increasing the number of test cases will reduce performance, and the number of times the program is executed to capture states will also need adjustments to get enough states. To achieve higher quality program states, they should generally have a good distribution between small and large arrays. The distribution of values within arrays should not be too similar to the ranges of other program variables. The generated program input should result in a good distribution of different execution paths of the algorithm; for example, the input for binary search should cover both cases when the element is found and when it is not.

6

Evaluation on Example Algorithms

The following section summarizes how QuickInv performs on some example algorithms with the loop invariants needed to verify the correctness of the algorithms using a verification tool such as Dafny [4]. QuickInv is evaluated on a system with an AMD Ryzen 9700X processor and 32 GB of RAM. The algorithms are executed 5 000 times with randomized inputs to collect the program states, and a law is required to pass 10 000 test-cases to be accepted.

6.1 Binary Search

For binary search from the introduction implemented in Listing 1, the loop invariants are $0 \leq lo \leq hi \leq |arr|$ and $key \in arr[lo..hi) = key \in arr$, and QuickInv can find both of them. The grammar used is the full **QuickInv Prelude** defined in Section 5.1, and the constant -1 is added because it appear in the program. For this grammar, QuickInv will find 98 background laws in the background phase, and 22 loop invariant candidates was found in the invariant discovery phase. Four out of these 22 loop invariant candidates were required to verify the algorithm's correctness, namely `0 ≤ lo`, `lo ≤ hi`, `hi ≤ length arr`, and `key ∈ slice arr lo hi = key ∈ arr`. The invariant candidates `key ∉ slice arr 0 lo` and `key ∉ slice arr hi (length arr)` are also discovered, which can be used instead of `key ∈ slice arr lo hi = key ∈ arr` to verify the algorithm. For this example, QuickInv is configured to search for laws up to a term size of 7, which is the default for QuickInv. The term size is determined by the maximum number of functions, constants and variables on one side of the equal sign; for example, the law `key ∈ slice arr lo hi = key ∈ arr` has a term size of 6. A summary of the QuickInv results is presented in Table 1.

Table 1 : QuickInv results for binary search.

Grammar	**QuickInv Prelude** `-1 :: Int`	
Loop Invariants	**Is Found**	**Law(s) Found by QuickInv**
$0 \leq lo \leq hi \leq \lvert arr \rvert$	Yes	`0 ≤ lo`, `lo ≤ hi`, `hi ≤ length arr`
$key \in arr[lo..hi) = key \in arr$	Yes	`key ∈ slice arr lo hi = key ∈ arr`
Number of Background Laws	98	
Number of Invariant Laws	22	
Invariant Laws of Interest	4/22	
Max Term Size	7 (default)	
Execution Time	38 seconds	

6.2 Linear Search

The second example algorithm is linear search, implemented in Listing 2. If the searched value *key* does not appear in the array, then the special value -1 is returned. This special value is also added to the grammar. The example has the loop invariants $idx \neq -1 \implies key = arr[idx]$ and $idx = -1 \implies key \notin arr[0..i)$. These invariants have a condition on whether idx is equal to -1 or not, and such conditional invariants can currently not be found by QuickInv. However, QuickInv could find an alternative invariant without an implication for $idx = -1 \implies key \notin arr[0..i)$, which is $key \in arr[0..i) = 0 \leq idx$. For linear search, five of six invariant candidates were required to verify the algorithm. The one that was not required was $idx < i$. A summary of the test results is presented in Table 2.

Table 2 : QuickInv results for linear search.

Grammar	**QuickInv Prelude** `-1 :: Int`	
Loop Invariants	**Is Found**	**Law(s) Found by QuickInv**
$0 \leq i \leq \lvert arr \rvert$	Yes	`0 ≤ i, i ≤ length arr`
$-1 \leq idx < \lvert arr \rvert$	Yes	`(-1) ≤ idx, idx < length arr`
$idx \neq -1 \implies key = arr[idx]$	No (inexpressible)	
$idx = -1 \implies key \notin arr[0..i)$	Yes (alternative invariant)	`key ∈ slice arr 0 i = 0 ≤ idx`
Number of Background Laws	98	
Number of Invariant Laws	6	
Invariant Laws of Interest	5/6	
Max Term Size	7 (default)	
Execution Time	34 seconds	

```
int linear_search(int* arr, int arr_len, int key)
    // post: -1 <= idx < arr_len
    // post: idx != -1 ==> key == arr[idx]
    // post: idx == -1 ==> not (key in arr)
{
    int idx = -1;
    int i = 0;
    while (i < arr_len)
        // inv : 0 <= i <= arr_len
        // inv: -1 <= idx < arr_len
        // inv: idx != -1 ==> key == arr[idx]
        // inv: idx == -1 ==> not (key in arr[0..i))
    {
        if (arr[i] == key) {
            idx = i;
        }
        i = i + 1;
    }
    return idx;
}
```

Listing 2 : Implementation of linear search.

6.3 Selection Sort

For selection sort implemented in Listing 3, QuickInv finds all three loop invariants. In order to discover the invariant $arr[0..i) \leq^* arr[i..|arr|)$, we needed to increase the term size to 10, which resulted in QuickInv taking around over an hour to execute due to the exponentially larger search space.

Table 3 : QuickInv results for selection sort.

Grammar	**QuickInv Prelude**	
Loop Invariants	**Is Found**	**Law(s) Found by QuickInv**
$0 \leq i \leq \lvert arr \rvert$	Yes	`0 ≤ i, i ≤ length arr`
$sorted\ arr[0..i)$	Yes	`sorted (slice arr 0 i)`
$arr[0..i) \leq^* arr[i..\lvert arr \rvert)$	Yes	`slice arr 0 i ≤* slice arr i (length arr)`
$multiset(arr) = multiset(old(arr))$	Yes	`multiset old_arr = multiset arr`
Number of Background Laws	503	
Number of Invariant Laws	16	
Invariant Laws of Interest	5/16	
Max Term Size	10	
Execution Time	1 hour, 8 minutes and 28 seconds	

Our implementation of selection sort has a helper function `find_min` that finds where the minimum element is in the unsorted part of the array. To find the invariants for this helper function, the program states are captured by running the selection sort function and capturing the states in the helper function. It would also be possible to run the helper function directly. The QuickInv results for the helper function are shown in Table 4.

Table 4 : QuickInv results for the helper function `find_min` of selection sort.

Grammar	**QuickInv Prelude**	
Loop Invariants	**Is Found**	**Law(s) Found by QuickInv**
$i \leq j \leq \lvert arr \rvert$	Yes	`i ≤ j, j ≤ length arr`
$i \leq m < \lvert arr \rvert$	Yes	`i ≤ m, m < length arr`
$arr[m] \leq^* arr[i..j)$	Yes	`(arr !! m) ≤* slice arr i j`
Number of Background Laws	130	
Number of Invariant Laws	44	
Invariant Laws of Interest	4/44	
Max Term Size	8	
Execution Time	1 minute and 40 seconds	

```
void selection_sort(int* arr, int arr_len)
    // post: sorted arr
    // post: multiset(arr) == multiset(old(arr))
{
    int i = 0;
    while (i < arr_len)
        // inv: 0 <= i <= arr_len
        // inv: sorted arr[0..i)
        // inv: arr[0..i) <=* arr[i..arr_len)
        // inv: multiset(arr) == multiset(old(arr))
    {
        int m = find_min(arr, arr_len, i);
        int t = arr[m];
        arr[m] = arr[i];
        arr[i] = t;
        i = i + 1;
    }
}

int find_min(int* arr, int arr_len, int i)
    // pre: 0 <= i < arr_len
    // post: i <= m < arr_len
    // post: arr[m] <= arr[i..arr_len)
{
    int j = i;
    int m = i;
    while (j < arr_len)
        // inv: i <= j <= arr_len
        // inv: i <= m < arr_len
        // inv: arr[m] <=* arr[i..j)
    {
        if (arr[j] < arr[m]) {
            m = j;
        }
        j = j + 1;
    }
    return m;
}
```

Listing 3 : Implementation of selection sort.

6.4 Merge Sort

For merge sort implemented in Listing 4, QuickInv finds 11 of 14 loop invariants. Of the three invariants that we cannot find, two of them have inexpressible implications and one is too large.

The loop invariant $multiset(arr[p..(p+k))) = multiset(u[0..i)) \cup multiset(v[0..j))$ has a term size of 11, and it caused performance and memory issues when trying to discover it. When the maximum term size was increased to 11, QuickInv searched for background laws for over two hours and then terminated because it ran out of system memory.

For this algorithm, we discover way too many invariant candidates, 756 laws and only a few are of interest. The amount of program variables and that addition and subtraction are in the grammar is one contributing factor to why there are so many laws found for this algorithm. The results are summarized in Table 5.

Table 5 : QuickInv results for merge sort.

Grammar	**QuickInv Prelude Arithmetic Extension**	
Loop Invariants	**Is Found**	**Law(s) Found by QuickInv**
$0 \leq k \leq r - p$	Yes	`0 ≤ k, k ≤ (r - p)`
$0 \leq i \leq q - p$	Yes	`0 ≤ i, i ≤ length u, q - p = length u`
$0 \leq j \leq r - q$	Yes	`0 ≤ j, j ≤ length v, r - q = length v`
$k = i + j$	Yes	`k = i + j`
$sorted\ u$	Yes	`sorted u`
$sorted\ v$	Yes	`sorted v`
$sorted\ arr[p..(p + k))$	Yes	`sorted (slice arr p (k + p))`
$i < q - p \implies arr[p..(p + k)) \leq^* u[i]$	No (inexpressible)	
$j < r - q \implies arr[p..(p + k)) \leq^* v[j]$	No (inexpressible)	
$arr[0..p) = old(arr[0..p))$	Yes	`slice old_arr 0 p = slice arr 0 p`
$arr[r..\|arr\|) = old(arr[r..\|arr\|))$	Yes	`slice old_arr r (length arr) = slice arr r (length arr)`
$multiset(arr[p..(p + k))) = multiset(u[0..i)) \cup multiset(v[0..j))$	No (too large term)	
$u = old(arr[p..q))$	Yes	`u = slice old_arr p q`
$v = old(arr[q..r))$	Yes	`v = slice old_arr q r`
Number of Background Laws	118	
Number of Invariant Laws	756	
Invariant Laws of Interest	16/756	
Max Term Size	7 (default)	
Execution Time	12 minutes and 2 seconds	

```
void merge(int* arr, int arr_len, int p, int q, int r)
    // pre: 0 <= p < q < r <= arr_len
    // pre: sorted arr[p..q) && sorted arr[q..r)
    // post: sorted arr[p..r)
    // post: multiset(arr[p..r)) == multiset(old(arr[p..r)))
```

```
    // post: arr[0..p) == old(arr[0..p))
    // post: arr[r..arr_len) == old(arr[r..arr_len])
{
    int u[q - p];
    copy(a, u, p, q);
    int v[r - q];
    copy(a, v, q, r);
    int i = 0;
    int j = 0;
    int k = 0;
    while (k < r - p)
        // inv: 0 <= k <= r - p
        // inv: 0 <= i <= q - p
        // inv: 0 <= j <= r - q
        // inv: k == i + j
        // inv: sorted u
        // inv: sorted v
        // inv: sorted arr[p..p + k)
        // inv: i < q - p ==> arr[p..(p + k)) <=* u[i]
        // inv: j < r - q ==> arr[p..(p + k)) <=* v[j]
        // inv: arr[0..p) = old(arr[0..p))
        // inv: arr[r..arr_len) = old(arr[r..arr_len))
        // inv: multiset(arr[p..p + k)) == multiset(u[0..i))
        //      ++ multiset(v[0..j))
        // inv: u == old(arr[p..q))
        // inv: v == old(arr[q..r))
    {
        if ((i < q - p && j < r - q && u[i] <= v[j]) || j == r - q) {
            arr[p + k] = u[i];
            i = i + 1;
        } else {
            arr[p + k] = v[j];
            j = j + 1;
        }
        k = k + 1;
    }
}

void merge_sort(int* arr, int arr_len, int p, int r)
    // pre: 0 <= p < r <= arr_len
    // post: sorted arr[p..r)
    // post: multiset(arr[p..r)) == multiset(old(arr[p..r)))
    // post: arr[0..p) == old(arr[0..p))
    // post: arr[r..arr_len) == old(arr[r..arr_len])
{
    if (r - p > 1) {
        int q = (p + r) / 2;
        merge_sort(arr, arr_len, p, q);
        merge_sort(arr, arr_len, q, r);
        merge(arr, arr_len, p, q, r);
    }
}
```

```
int* copy(int* a, int* b, int p, int q)
    // pre: 0 <= p < q <= a_len
    // post: b == a[p..q)
{

    for (int k = p; k < q; ++k)
        // inv: p <= k <= q
        // inv: a[p..k) == b[0..k - p)
    {
        b[k - p] = a[k];
    }
    return b;
}
```

Listing 4 : Implementation of merge sort.

7

Related Work

Loop invariant inference methods can be divided into two different categories: static and dynamic. Static methods infer invariants directly from the code by reasoning. Dynamic methods observe runtime behavior through execution traces to identify invariants. This thesis is concerned with dynamic loop invariant inference.

Static methods include abstract interpretation [5], constraint solving [13], [14], predicate abstraction [15], and more recently constraint Horn clause solving [16], [17]. M. Karr [18] showed that it is possible to discover linear relationships between program variables of the form $a_1 \times x_1 + a_2 \times x_2 + ... + a_n \times x_n + c = 0$ where $x_1, x_2, ..., x_n$ are program variables and $a_1, a_2, ..., a_n, c$ are constants. This was done by approximating program behavior by an abstract domain. Constraint-solving-based systems translate invariant inference to constraint solving and have been used to discover polynomial relationships between program variables [13]. Finally, predicate abstraction has been used to find classes of invariants by constructing Boolean formulas given a set of predicates [19]. It has been shown that program verification can be reduced to constraint Horn clauses solving [16] – the problem of finding an unknown formula (e.g. a loop invariant) that satisfies a set of constraints (e.g. the verification conditions in the while rule of Hoare logic). A few such Horn clause solvers have been developed recently [16], [17]. Static methods are powerful and can discover sound invariants, but their designs are often complex and often scales poorly with program complexity.

Dynamic methods scale better due to their rather simplistic approach. A popular dynamic invariant inference system is Daikon [7]. Daikon collects execution traces, which are sequences of variable values recorded during the program's execution. It then analyzes these traces to discover patterns or relationships that hold true. Daikon has been integrated with automatic test case generators to produce better execution traces to catch edge cases, eliminating many false positives [20]. Daikon considers a set of defined templates for kinds of invariants to search for. Some examples are constant ($x = a$), non-zero ($x \neq 0$), being in a range ($a \leq x \leq b$) and linear relationships ($y = ax + b$) among others. The user can also extend Daikon with more templates. It can, however, be hard to know what template might be needed without knowing the invariant we are searching for.

An algorithmic difference between Daikons approach and QuickInv is that Daikon is more efficient with how it uses the collected states data. Daikon starts off with the assumption that all invariant candidates are true. It iterates over recorded states, incrementally falsifying candidates. QuickInv, on the other hand, starts off with the assumption that all candidates (except `t = t` for all `t`) are false. It then samples thousands of states to test each candidate, unless falsified quickly. It is possible for QuickInv to miss a counter-example due to sampling – even if the data includes the counter-example, which can lead to false positives.

An alternative to QuickSpec is Speculate [21], which is a theory exploration system inspired by QuickSpec. Speculate discovers universally quantified equations, and in addition, it can also discover conditional equations and inequalities as a post-processing step after equation discovery. Speculate lacks built-in support for observational equality, and is less optimized than QuickSpec.

QuickSpec has been used in HipSpec [22] to discover lemmas for inductive theorem proving of recursive functions. It was also used for conjecture discovery in the interactive proof assistant Isabelle/HOL [23]. QuickSpec was extended to support guided theory exploration using user-defined templates in RoughSpec [24], to allow searching for larger terms with templates. QuickSpec added support for reasoning with conditional equations towards the end of this thesis via an encoding [25]. It enabled adding conditional laws manually to the pruner, however, QuickSpec still has the same limited support for automatically discovering conditional laws.

8

Discussion and Conclusion

This thesis presented QuickInv, a system for dynamic program invariant discovery built upon the theory exploration system QuickSpec. Our central aim was to leverage theory exploration to discover high-level, easy-to-understand invariants from program execution traces, particularly for array programs. We reframed invariant discovery as a theory exploration problem, developed a high-level grammar in Haskell for expression such invariants, and implemented the QuickInv system. We captured program states by executing C programs and employed a two-phase QuickSpec process to distinguish general mathematical truths from program-specific properties. The evaluation was on the binary search, linear search, selection sort, and merge sort algorithms demonstrated QuickInv's ability to rediscover many essential loop invariants through systematic exploration of possible invariants up to a certain term size.

The thesis sought to address several research questions. Regarding a suitable grammar for high-level invariants in array programs, we listed a set of functions and constants in a prelude and additionally some arithmetic functions required for some of the algorithms. The performance of the system depends highly on what is included in the grammar, and the number of functions and constants. The addition and subtraction operators lead to significantly worse performance and more discovered laws – many of them spurious. QuickSpecs effeciency has enabled systematic exploration of large search spaces, however, exploring larger terms becomes infeasible quite quickly beyond term size 7 due to the exponential growth of the search space.

The grammar is able to express the invariants needed for the example algorithms; however, there is an additional requirement for the grammar to fulfill: the grammar needs to have nice properties that are simple and discoverable automatically by QuickSpec. Due to the limitations of conditional support in QuickSpec, laws about partial functions such as array slicing and indexing cannot be discovered automatically. Total functions have simpler, often non-conditional properties that can be discovered automatically. One solution is to manually add background properties as axioms to the QuickSpec pruner, which allows reasoning about the properties that otherwise would not be discovered by QuickSpec. This, however, is laborious and requires domain knowledge and is problematic for a user-configurable grammar.

Concerning the effectiveness of a dynamic, theory-exploration-based approach, QuickInv demonstrated encouraging potential. It successfully rediscovered many loop invariants for the example algorithms, while struggling with invariants with large term sizes and conditional invariants. The system's systematic exploration is a significant advantage over template-based approaches like Daikon, which can miss important invariants due to their reliance on predefined templates. The system's ability to leverage reasoning and pruning of redundant laws, informed by background theory discovered in its first phase, reduces the number of reported laws following from reasoning. There are, however, still many laws that the simple reasoning based on rewriting up to a certain term size cannot prune away, and the possibility of discovering false laws that can only be falsified by certain corner cases – an issue with all dynamic invariant discovery systems – remains a challenge.

The main limitations of QuickInv are the inability to discover conditional invariants and background laws, and to handle the large search space of invariants more efficiently to allow for discovery of larger term sizes.

Addressing these limitations naturally points towards several key directions for future work. A primary focus is enhancing the discovery of conditional invariants. One approach within QuickInv could involve a pre-filtering step on the collected program states, allowing the system to test potential consequent properties only on states satisfying a user-defined antecedent, thereby enabling the inference of specific $P \Rightarrow Q$ style invariants. Concurrently, advancing QuickSpec's support for more intricate conditional laws would be beneficial, particularly for discovering laws involving partial functions, which often are conditional.

Furthermore, improving the performance and efficiency of the discovery process is crucial. One enhancement would be the native support for inequalities (e.g. $\leq$, $\leq^*$) within QuickSpec. This could potentially be achieved by leveraging the mathematical properties of partial orders, similar to how QuickSpec utilizes equivalence classes for equational reasoning. Given that relations like $\leq$ are fundamental to many invariants, such as index bounds and properties establishing sortedness, native support for inequalities could significantly enhance the system's efficiency and effectiveness. Beyond these enhancements, exploring methods to guide the theory exploration process could also yield performance improvements – perhaps through template-based exploration for larger terms through RoughSpec or through heuristics that suggest candidates based on mutations of the post-condition as utilized by many static methods.

In this thesis, we evaluated QuickInv by testing whether it can successfully rediscover known loop invariants required for verification across a set of algorithms. A natural next step is to assess its real-world usefulness in assisting program verification of algorithms with unknown invariants. Evaluating QuickInv on other array algorithms, or program domains, would measure effectiveness of the grammar and of the system as a domain-independent invariant discovery tool. It would also be valuable to verify the discovered invariants using an external program verifier to eliminate false positives. Another

promising direction is to minimize the set of discovered invariants to include only those necessary for program verification, filtering out valid but irrelevant invariants.

9

References

[1] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 19–32, p. 1, 1967.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, doi: 10.1145/363235.363259.

[3] A. Blass and Y. Gurevich, "Inadequacy of computable loop invariants," *ACM Transactions on Computational Logic*, vol. 2, no. 1, pp. 1–11, Jan. 2001, doi: 10.1145/371282.371285.

[4] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., Berlin, Heidelberg: Springer, 2010, pp. 348–370. doi: 10.1007/978-3-642-17511-4_20.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, in POPL '77. New York, NY, USA: Association for Computing Machinery, Jan. 1977, pp. 238–252. doi: 10.1145/512950.512973.

[6] P. H. Schmitt and B. Weiss, "Inferring Invariants by Symbolic Execution," in *Proceedings of 4th International Verification Workshop in connection with CADE-21, Bremen, Germany, July 15-16, 2007*, B. Beckert, Ed., in CEUR Workshop Proceedings, vol. 259. CEUR-WS.org, 2007. [Online]. Available: http://ceur-ws.org/Vol-259/paper16.pdf

[7] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007, doi: 10.1016/j.scico.2007.01.015.

[8] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller, "Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 1019–1037, Oct. 2015, doi: 10.1109/TSE.2015.2431688.

[9] J. C. Reynolds, "Reasoning about arrays," *Commun. ACM*, vol. 22, no. 5, pp. 290–299, May 1979, doi: 10.1145/359104.359110.

[10] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, pp. 1–51, 2014.

[11] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed, "Quick specifications for the busy programmer," *Journal of Functional Programming*, vol. 27, p. e18, 2017, doi: 10.1017/S0956796817000090.

[12] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, in ICFP '00. New York, NY, USA: Association for Computing Machinery, Sep. 2000, pp. 268–279. doi: 10.1145/351240.351266.

[13] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using Gröbner bases," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004, pp. 318–329.

[14] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, 2003, pp. 420–432.

[15] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 191–202.

[16] K. L. McMillan and A. Rybalchenko, "Solving constrained Horn clauses using interpolation," *Tech. Rep. MSR-TR-2013-6*, 2013.

[17] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Springer, pp. 24–51, 2015.

[18] M. Karr, "Affine relationships among variables of a program," *Acta Informatica*, vol. 6, no. 2, pp. 133–151, 1976.

[19] K. Kalyanasundaram and C. Marché, "Automated generation of loop invariants using predicate abstraction," 2011.

[20] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller, "Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 1019–1037, 2015, doi: 10.1109/TSE.2015.2431688.

[21] R. Braquehais and C. Runciman, "Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Oxford UK: ACM, Sep. 2017, pp. 40–51. doi: 10.1145/3122955.3122961.

[22] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *Automated Deduction–CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*, 2013, pp. 392–406.

[23] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," in *International Conference on Intelligent Computer Mathematics*, 2014, pp. 108–122.

[24] S. H. Einarsdóttir, N. Smallbone, and M. Johansson, "Template-based theory exploration: Discovering properties of functional programs by testing," in *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, 2020, pp. 67–78.

[25] K. Claessen and N. Smallbone, "Efficient encodings of first-order Horn formulas in equational logic," in *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings 9*, 2018, pp. 388–404.