

---

# Automated Verification of While-Programs: Empirically Comparing Dafny and Caesar

---

*By*  
Hanbit Chang

*Supervisor*  
Philipp Schroer

*Examiners*  
Prof. Dr. ir. Dr. h. c. Joost-Pieter Katoen  
apl. Prof. Dr. rer. nat. Thomas Noll

*a thesis presented for the degree of  
Bachelor of Science*

*in the*

Chair for Software Modeling and Verification  
RWTH Aachen University



# *Abstract*

Dafny and Caesar are verification tools, whereas Caesar is a newly developed tool for verifying probabilistic programs. It came to question whether this tool can verify reasoning deterministic programs and what problems appear during verification. This study compares the implementation and verification of the algorithms focusing on the Left Pad function, Bubble Sort, and Binary Search Tree.

Dafny requires the implementation of the specification in predicates and functions, which guarantees the verification of the algorithms. Caesar requires implementing user-defined data types for the input of the algorithms and additional functions to substitute the features in Dafny for attempting to verify the algorithms. However, Dafny and Caesar have different verification results than expected in the Left Pad function and Bubble Sort. Caesar failed to verify the algorithms, as it could not find the error during the debugging process or implement functions due to its quantified instances that stay in a loop. In contrast, both verifiers ensure the Binary Search Tree with its features. The study specifies why these problems appeared in Caesar during the implementation and verification of each algorithm. Moreover, it suggests what improvements are required to solve these problems.



# Contents

<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries</b>	<b>3</b>
2.1 Infrastructure of Dafny and Caesar . . . . .	3
2.2 Hoare Logic . . . . .	4
2.2.1 Rule of Statements . . . . .	4
2.3 Verification Condition . . . . .	5
2.3.1 Weakest Liberal Pre-condition . . . . .	5
2.3.2 Guarded Commands . . . . .	6
2.3.3 Conditional Statement . . . . .	7
2.3.4 While Statement . . . . .	7
<b>3 Comparing Features in Dafny and Caesar</b>	<b>9</b>
3.1 Comparing Supported Features . . . . .	9
3.1.1 Types . . . . .	9
3.1.2 Collections . . . . .	10
3.1.3 Methods, Functions, Procedures, and Domains . . . . .	11
3.1.4 Assumption and Assertions . . . . .	12
3.1.5 If and While Statements . . . . .	12
3.1.6 Pre-conditions and Post-conditions . . . . .	13
3.1.7 Predicates . . . . .	14
3.2 Additional Features in Dafny . . . . .	14
3.2.1 Decreases Clause . . . . .	14
3.2.2 Print Statement . . . . .	15
3.2.3 Trigger Selection . . . . .	15
3.2.4 Finding Errors . . . . .	16
<b>4 Comparison Using Left Pad Function</b>	<b>17</b>
4.1 Verification of Left Pad Function in Dafny . . . . .	18
4.1.1 Predicates of <i>LeftPad</i> in Dafny . . . . .	18
4.1.2 Verification of <i>LeftPad</i> in Dafny . . . . .	19
4.2 Verification Failure of Left Pad Function in Built-in <i>Lists</i> in Caesar	21
4.3 Verification of the Left Pad Function in Datatype <i>List</i> in Caesar	24
4.3.1 Datatype <i>List</i> and Axiom of Extensionality in Caesar . .	24
4.3.2 Verification of <i>LeftPad</i> in Datatype <i>List</i> in Caesar . . . .	27

<b>5</b>	<b>Comparison Using Bubble Sort</b>	<b>29</b>
5.1	Verification of Bubble Sort in Dafny . . . . .	30
5.1.1	Predicates of <i>BubbleSort</i> in Dafny . . . . .	30
5.1.2	Verification of <i>BubbleSort</i> in Dafny . . . . .	32
5.2	Verification of Bubble Sort in Caesar . . . . .	34
5.2.1	Verification of <i>BubbleSort</i> in Built-in <i>Lists</i> in Caesar . . .	34
5.2.2	Verification of <i>BubbleSort</i> in Datatype <i>List</i> in Caesar . . .	36
5.3	Verification Failure of Multiset in Caesar . . . . .	36
5.3.1	Verification of <i>multiplicity</i> in Dafny . . . . .	36
5.3.2	Implementation Failure of <i>multiplicity</i> in Built-in <i>Lists</i> in Caesar . . . . .	37
5.3.3	Implementation Failure of <i>multiplicity</i> in Datatype <i>List</i> in Caesar . . . . .	38
<b>6</b>	<b>Comparison Using Binary Search Tree</b>	<b>41</b>
6.1	Verification of Binary Search Tree in Dafny . . . . .	41
6.2	Verification of Insertion in Dafny . . . . .	43
6.2.1	Verification of <i>Insert</i> in Dafny . . . . .	44
6.3	Verification of Deletion in Dafny . . . . .	44
6.3.1	Verification of <i>Delete</i> in Dafny . . . . .	46
6.4	Verification of Binary Search Tree in Caesar . . . . .	47
6.4.1	Verification of <i>Insert</i> in Caesar . . . . .	49
6.4.2	Verification of <i>Delete</i> in Caesar . . . . .	49
<b>7</b>	<b>Results and Discussion</b>	<b>51</b>
7.1	Comparison of the Results . . . . .	51
7.2	Discussion . . . . .	52
7.2.1	Finding Errors . . . . .	52
7.2.2	Trigger Selection . . . . .	53
<b>8</b>	<b>Conclusion</b>	<b>55</b>
<b>A</b>	<b>Code</b>	<b>57</b>
A.1	Dafny . . . . .	57
A.1.1	Left Pad Function . . . . .	57
A.1.2	Bubble Sort Algorithm . . . . .	57
A.1.3	Insertion . . . . .	58
A.1.4	Deletion . . . . .	58
A.2	Caesar . . . . .	59
A.2.1	Left Pad Function in Built-in <i>Lists</i> . . . . .	59
A.2.2	Bubble Sort Algorithm in Built-in <i>Lists</i> . . . . .	60
A.2.3	Bubble Sort Algorithm in Datatype <i>List</i> . . . . .	60
A.2.4	Insertion . . . . .	61
A.2.5	Deletion . . . . .	62
<b>B</b>	<b>Counter-example</b>	<b>63</b>
B.1	Caesar . . . . .	63







# List of Figures

2.1	The infrastructure of Dafny and Caesar . . . . .	3
3.1	Dafny lines up the error code in red . . . . .	16
4.1	Example of the Left Pad function . . . . .	17
4.2	The method <i>LeftPad</i> fails to verify the correctness of the string <i>a</i> . . . . .	20
4.3	After correcting <i>suffix</i> , Dafny verifies the padded string . . . . .	21
4.4	Counter-examples of <i>LeftPad</i> in Built-in <i>Lists</i> . . . . .	23
4.5	Caesar verifies the Left Pad function using the procedure <i>Extensionality</i> . . . . .	28
5.1	Example of a Bubble Sort . . . . .	29
5.2	The predicate <i>arraySortedWrong</i> ensures the array <i>arr</i> is sorted . . . . .	31
5.3	Dafny verifies Bubble Sort by calling the method <i>BubbleSort</i> . . . . .	34
5.4	The procedure <i>Main</i> calls <i>BubbleSort</i> . . . . .	36
5.5	Examples of calling the function <i>multiplicity</i> . . . . .	38
6.1	Example of a Binary Search Tree . . . . .	41
6.2	The predicate <i>BST</i> has the properties of a Binary Search Tree . . . . .	43
6.3	Add value 6 in Binary Search Tree . . . . .	43
6.4	Case 1: Delete node with no sub-trees . . . . .	45
6.5	Case 2: Delete node with one sub-tree . . . . .	45
6.6	Case 3: Delete node with both sub-trees . . . . .	45
7.1	Caesar verifies that multiset of two lists are equal . . . . .	53



# List of Tables

7.1	Code statistic of the Left Pad function in Dafny and Caesar . . .	51
7.2	Code statistic of Bubble Sort in Dafny and Caesar . . . . .	52
7.3	Code statistic of Binary Search Tree in Dafny and Caesar . . . .	52



# Listings

3.1	Example of a datatype <i>List</i> in Dafny and Caesar . . . . .	9
3.2	Example of <i>array</i> and built-in <i>Lists</i> in Dafny and Caesar . . . . .	10
3.3	Example of <i>seq</i> in Dafny . . . . .	10
3.4	Example of <i>set</i> in Dafny . . . . .	11
3.5	Example of <i>multiset</i> in Dafny . . . . .	11
3.6	Example of <i>method</i> and <i>proc</i> in Dafny and Caesar . . . . .	11
3.7	Example of <i>function</i> in Dafny . . . . .	11
3.8	Example of <i>domain</i> in Caesar . . . . .	12
3.9	Example of <i>assume</i> and <i>assert</i> in Dafny and Caesar . . . . .	12
3.10	Example of <i>if</i> statement in Dafny and Caesar . . . . .	13
3.11	Example of <i>while</i> statement in Dafny and Caesar . . . . .	13
3.12	Example of pre-condition and post-condition in Dafny and Caesar . . . . .	14
3.13	Example of <i>predicate</i> in Dafny and its translation in Caesar . . . . .	14
3.14	Example of <i>decreases</i> in Dafny . . . . .	15
3.15	Example of <i>print</i> in Dafny . . . . .	15
3.16	Example of trigger selection in Dafny . . . . .	16
3.17	Debugging of trigger selection in Dafny . . . . .	16
4.1	Implementation of the Left Pad function in Dafny . . . . .	17
4.2	Predicate <i>prefix</i> of the Left Pad function in Dafny . . . . .	18
4.3	Predicate <i>suffix</i> of the Left Pad function in Dafny . . . . .	18
4.4	Annotations of <i>LeftPad</i> in Dafny . . . . .	19
4.5	Invariants of <i>LeftPad</i> in Dafny . . . . .	19
4.6	Improvement of the predicate <i>suffix</i> . . . . .	20
4.7	Annotations of <i>LeftPad</i> in Caesar . . . . .	21
4.8	The first loop invariant of <i>LeftPad</i> in Caesar . . . . .	22
4.9	The second loop invariant of <i>LeftPad</i> in Caesar . . . . .	23
4.10	Domain of the datatype <i>List</i> in Caesar . . . . .	24
4.11	Length of the datatype <i>List</i> in Caesar . . . . .	25
4.12	Select of the datatype <i>List</i> in Caesar . . . . .	25
4.13	Annotations of the procedure <i>Extensionality</i> . . . . .	26
4.14	Proof of the extensionality for an empty list in Caesar . . . . .	26
4.15	Proof of the extensionality for a list in Caesar . . . . .	26
4.16	Annotations of <i>LeftPad</i> in datatype <i>List</i> in Caesar . . . . .	27
5.1	Implementation of Bubble Sort in Dafny . . . . .	29
5.2	Predicate <i>arraySorted</i> of Bubble Sort in Dafny . . . . .	31
5.3	Predicate <i>arraySortedWrong</i> in Dafny . . . . .	31
5.4	Predicate <i>bubblesSorted</i> of Bubble Sort in Dafny . . . . .	32
5.5	Predicate <i>bubbleStepFinished</i> of Bubble Sort in Dafny . . . . .	32
5.6	Annotations of <i>BubbleSort</i> in Dafny . . . . .	32
5.7	Outer loop invariants of <i>BubbleSort</i> in Dafny . . . . .	33

5.8	Inner loop invariants of <i>BubbleSort</i> in Dafny . . . . .	33
5.9	Annotations of <i>BubbleSort</i> in Caesar . . . . .	34
5.10	Multiplicity in Dafny . . . . .	36
5.11	Multiplicity in built-in type <i>Lists</i> in Caesar . . . . .	37
5.12	Multiplicity in datatype <i>List</i> in Caesar . . . . .	38
5.13	Multiplicity in datatype <i>Lst</i> in Z3 . . . . .	39
6.1	Datatype <i>Tree</i> in Dafny . . . . .	41
6.2	Tree set in Dafny . . . . .	42
6.3	Predicate <i>BST</i> in Dafny . . . . .	42
6.4	Implementation of Insertion in Dafny . . . . .	43
6.5	Annotations of <i>Insert</i> in Dafny . . . . .	44
6.6	Implementation of <i>getMin</i> in Dafny . . . . .	45
6.7	Implementation of Deletion in Dafny . . . . .	46
6.8	Annotations of <i>Delete</i> in Dafny . . . . .	46
6.9	Annotations of <i>getMin</i> in Dafny . . . . .	47
6.10	Domain of the datatype <i>Tree</i> in Caesar . . . . .	47
6.11	Implementation of <i>contains</i> in Caesar . . . . .	48
6.12	Binary Search Tree in datatype <i>Tree</i> in Caesar . . . . .	48
6.13	Annotations of <i>Insert</i> in Caesar . . . . .	49
6.14	Annotations of <i>Delete</i> in Caesar . . . . .	49
6.15	Implementation of <i>getMin</i> in Caesar . . . . .	50
7.1	Example of selecting triggers in Caesar . . . . .	53
A.1	Verification of Left Pad function in Dafny . . . . .	57
A.2	Verification of Bubble Sort algorithm in Dafny . . . . .	57
A.3	Verification of Insertion in Dafny . . . . .	58
A.4	Verification of Deletion in Dafny . . . . .	58
A.5	Verification of Left Pad function in built-in <i>Lists</i> in Caesar . . .	59
A.6	Verification of Bubble Sort algorithm in built-in <i>Lists</i> in Caesar .	60
A.7	Verification of Bubble Sort algorithm in built-in <i>Lists</i> in Caesar .	60
A.8	Verification of Insertion in data type <i>Tree</i> in Caesar . . . . .	61
A.9	Verification of Deletion in data type <i>Tree</i> in Caesar . . . . .	62
B.1	Counter-example of <i>LeftPad</i> in built-in <i>Lists</i> in Caesar . . . . .	63

# Chapter 1

## Introduction

Program verification ensures a computer program's correctness and reduces errors by specifying its properties. Verification tools are developed to automate this process to disprove or prove a computer program. Dafny [Lei10] is a programming language developed by Microsoft to verify a program's functional correctness by writing the code and its specifications. Moreover, Dafny supports various features, as shown in [KL12; LFC21]. Recently, a new verification platform called Caesar [Sch22] to verify probabilistic programs with expected values. Probabilistic programming can be explained by flipping a dice. Flipping the dice each time it does not have a fixed output but has a random output. Moreover, Caesar outputs boolean embeddings that return true or false. However, there have not been verification reasoning deterministic programs in Caesar. So, it came to the question of Caesar's ability to verify these programs. Therefore, this research compares Dafny and Caesar using deterministic algorithms such as the Left Pad function, Bubble Sort, and Binary Search Tree, as they represent fundamental examples of strings, lists, and trees.

This study begins by introducing the infrastructure of Dafny and Caesar and defines the formal verification method, the Hoare logic [Hoa69], which is intuitive and the weakest liberal pre-conditions [Wag85], that Dafny and Caesar use to verify computer programs. In Chapter 3, the research focuses on which each tool supports different features, such as types, loop conditions, and executable functions. Chapter 4 shows the implementation and verification of the Left Pad function in Dafny and displays the translation of the specification in Caesar. Caesar fails to verify the Left Pad function in built-in *Lists* and requires another solution by implementing the datatype *List*. Chapter 5 shows the implementation and verification of Bubble Sort in Dafny and Caesar, successfully verifying that the list is sorted. However, Caesar fails to implement the substitute function for verifying the list's multiset. Chapter 6 shows the implementation of the Binary Search Tree and verifies the insertion and deletion function in Dafny and Caesar. Chapter 7 shows the implemented specification, presents the verification results of the algorithms in Dafny and Caesar, and discusses the problems during the verification process in Caesar. The conclusion summarises and rounds off this bachelor thesis and suggests implementing new features for a better verification process.





## Chapter 2

# Preliminaries

This section introduces the preliminaries regarding the infrastructure of Dafny and Caesar to show what layer this study compares. Then, it shows the formal verification method of programs by Hoare logic in [Hoa69] and the weakest liberal pre-condition (*wlp*) in [Wag85]. It then introduces the verification conditions of HeyVL and Boogie using guarded-commands language by Dijkstra[Dij75]. Moreover, it introduces ensuring the statements of *wlp* summarized in [Mül19].

### 2.1 Infrastructure of Dafny and Caesar

The infrastructure between Dafny and Caesar differs, whereas Caesar is on the same layer with its intermediate verification language, HeyVL. Figure 2.1 outlines the infrastructure of Dafny and Caesar and the layer to compare these languages.

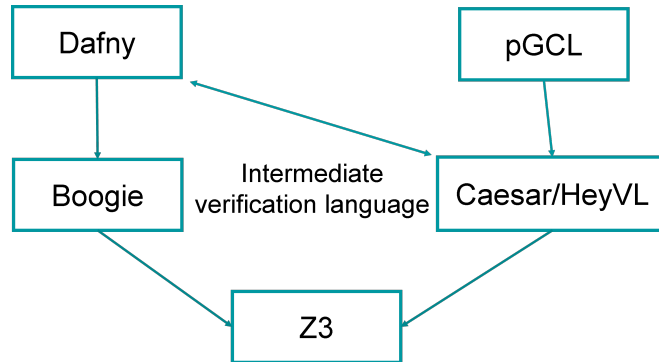


FIGURE 2.1: The infrastructure of Dafny and Caesar

Dafny is on the top of the layer and is used to verify the functional correctness of programs [Lei10]. The programming language uses built-in specification constraints, allowing users to specify the program's behavior. Dafny uses the intermediate verification language Boogie [Lei08] for the program verification.

Caesar is on the same layer as HeyVL[Sch22] as the verifier allows the inputs of the intermediate verification language HeyVL. For verification conditions of probabilistic programming, HeyVL encodes the probabilistic programming language (pGCL) [MM05], which is a probabilistic extension of Dijkstra's guarded-commands language (GCL) [Dij75]. This guarded-commands

language is also the basis structure for Boogie. Moreover, Caesar allows to have boolean embeddings that return true or false.

Verifying a program requires formal specification of its behavior, which can become difficult. Intermediate verification languages support the middle layer of the program that requires verification and the needed verification conditions. These languages aim to simplify the verification condition by encoding its program's specifications.

Boogie and HeyVL generate the verification conditions and use the automated theorem prover Z3 [MB08] to prove or disprove the program's correctness. Also, it outputs unknown, that can not determine if the program is true or false.

## 2.2 Hoare Logic

Hoare logic provides a set of logical rules to prove the correctness of computer programs using formal methods. The Hoare triple is the feature that progresses the computation, which is written as follows:

$$\{P\}S\{Q\}$$

The triple consists of condition  $P$ , which refers to the pre-condition, and condition  $Q$ , which is known as the post-condition[AO19]. The pre-condition signifies the properties that must be satisfied whenever function  $S$  is called. On the other hand, the post-condition states the properties that the function ensures when it returns[Mey97]. Using the standard Hoare logic, the Hoare triple is valid under the partially correct condition:

if  $P$  holds  
 then  $S$  terminates in condition  $Q$   
 or  $S$  does not terminate at all.

### 2.2.1 Rule of Statements

**Assignment** The rule of assignment is the basic proof rule for Hoare logic. The axiom is written as follows:

$$\{P[t/x]\}x := t\{P\}$$

The code of the Hoare triplet is formulated with a variable  $x$ , which assigns itself a new value.  $P[t/x]$  is the pre-condition obtained from  $P$ , and it substitutes  $x$  with the expression  $t$ . This condition represents the program's state before executing the assignment statement by substituting  $t$  for  $x$ .

**Conditional** In Hoare logic, the conditional statement presents if the condition holds, then it allows to be written as the if-then-else statement:

$$\frac{\{P \wedge C\}S1\{Q\}, \{P \wedge \neg C\}S2\{Q\}}{\{P\} \text{ if } C \text{ then } S1 \text{ else } S2\{Q\}}$$

The statement has two Hoare triples, where the first triple holds if,  $P$  and  $C$  holds then terminate  $S1$  and holds  $Q$ . The other triple terminates  $S2$  and ensures  $Q$  when the pre-condition holds  $P$  and  $\neg C$ .

**While loop** The while statement in Hoare logic executes its code until the condition is true. The statement verifies the loop invariant, which is expressed as follows:

$$\frac{\{P \wedge C\}S\{P\}}{\{P\} \text{ while } C \text{ do } S\{P \wedge \neg C\}}$$

The while statement in the Hoare triple starts with a pre-condition  $P$  and loop condition  $C$ . The loop invariant represents the condition at the beginning of each iteration. The loop body  $S$  is computed repeatedly in the loop condition until  $C$  remains true. The post-condition asserts that the loop invariant  $P$  is valid when the loop terminates, and therefore, the loop condition  $C$  is false.

## 2.3 Verification Condition

### 2.3.1 Weakest Liberal Pre-condition

The intermediate verification languages Boogie and HeyVL use the weakest liberal pre-condition for the verification condition of the programs that reasons partial correctness. The weakest liberal pre-condition  $P$  is defined with the code block  $S$  and a post-condition  $Q$ , and for all the pre-conditions  $P'$ , the following should hold:

$$\forall P' : \{P'\}S\{Q\} \text{ if } P' \implies wlp(S, Q)$$

This means that  $P$  is the least restrictive verification condition. The definition of  $P = wlp(S, Q)$  indicates that  $P$  is the weakest liberal pre-condition for statement  $S$  and post-condition  $Q$ .

**Backward Reasoning** Backward reasoning defines the weakest liberal pre-condition with the given program and a post-condition. By searching for the appropriate pre-condition, backward reasoning determines the proof of the expected post-condition. Backward reasoning begins by defining the post-condition that the code wants to reach. Then, it moves backward and defines the pre-condition of the last statement that holds for the post-condition. This process repeats until it reaches the top of the statement list, and this condition ensures the weakest liberal pre-condition of the statements respecting the post-condition.

For instance, defining the weakest liberal pre-condition in a basic code is described as follows:

$$\begin{aligned} &\{t - 1 < 0\} \\ &x := t; \\ &\{x - 1 < 0\} \\ &x := x - 1; \\ &\{x < 0\} \end{aligned}$$

If the post-condition of this code is  $x < 0$ , then the second statement of the value assigned to  $x$  must also hold, meaning the condition must be  $x - 1 < 0$ . The first statement assigned to  $x$  proves the pre-condition  $\{t - 1 < 0\}$ , which then holds the  $\{x < 0\}$ .

### 2.3.2 Guarded Commands

The basis structure of an intermediate verification language as Boogie and HeyVL approach the work in [Mül19]. This paper describes the correctness of guarded commands with the weakest pre-condition. However, it is important to note that this paper uses the weakest liberal pre-condition for the correctness of the intermediate verification languages. Therefore, this study writes *wlp* instead of *wp*. The syntax of this language, known as guarded-commands language, is as follows:

$$\begin{aligned} S ::= & x := t \\ & | \text{havoc } x \\ & | \text{assert } P \\ & | \text{assume } P \\ & | S; S \\ & | S \parallel S \end{aligned}$$

The first grammar assigns the value of  $t$  to  $x$ . The havoc statement updates the values of the variable  $x$  and deletes all the information from previous values by assigning a non-deterministic value. The *assert* command specifies the condition that the program must hold, whereas the similarly *assume* command ensures the condition  $P$  is correct by making an assumption. The following command has the sequential composition  $S1; S2$ , where the program  $S1$  executes before  $S2$ . The last command presents the non-deterministic choice between  $S1$  or  $S2$  that either one of them is executed.

To ensure the correctness of a guarded command  $S$ , one can verify its validity by proving the verification pre-condition  $wlp(S, Q)$  under the condition that  $Q$  is satisfied. The weakest liberal pre-condition of guarded command is written as  $wlp(S, Q)$ , defined as  $Q[t/x]$  that represents  $Q$  with the expression  $t$

substituted for the variable  $x$ .

$$\begin{aligned}
wlp(x := t, Q) &= Q[t/x] \\
wlp(\mathbf{havoc} \ x, Q) &= \forall x * Q \\
wlp(\mathbf{assert} \ P, Q) &= P \wedge Q \\
wlp(\mathbf{assume} \ P, Q) &= P \implies Q \\
wlp(S1; S2, Q) &= wlp(S1, wlp(S2, Q)) \\
wlp(S1 \parallel S2, Q) &= wlp(S1, Q) \wedge wlp(S2, Q)
\end{aligned}$$

The  $wlp(\mathbf{havoc} \ x, Q)$  represents the universal quantifier for  $x$  by ensuring the  $Q$ . The pre-condition  $wlp(\mathbf{assert} \ P, Q)$  verifies the  $Q$  and  $P$  with the conjunction. The  $wlp(\mathbf{assume} \ P, Q)$  establishes an implication of  $Q \implies P$ . The  $wlp$  of sequential execution computes for the second statement  $S2$  first and then the first statement  $S1$ , which verifies in order. Last, in the case of a non-deterministic choice, verification conditions of both  $S1$  and  $S2$  are valid, and the correctness of the program holds for all possible choices.

### 2.3.3 Conditional Statement

The conditional statement introduced in Section 2.2.1 is not included in basic intermediate verification language. However, the statement can be encoded conveniently with guarded commands using the *assume* statement and the non-deterministic choice. Guarded commands of the conditional statement follows:

$$(\mathbf{assume} \ C; \llbracket S_1 \rrbracket) \parallel (\mathbf{assume} \ \neg C; \llbracket S_2 \rrbracket)$$

The presented notation  $\llbracket S \rrbracket$  encodes the statement of  $S$ . The encoding of guarded commands is considered correct under the following conditions. If the condition  $C$  is valid, then the left side of the non-deterministic choice of  $S1$  is correct. Conversely, if the condition  $C$  is false, the right side of  $S2$  is correct. This encoding is the conditional statement of the weakest liberal pre-condition verification that holds for the post-condition  $Q$ :

$$(C \implies wlp(\llbracket S_1 \rrbracket, Q)) \wedge (\neg C \implies wlp(\llbracket S_2 \rrbracket, Q))$$

### 2.3.4 While Statement

Guarded-commands language allows encoding while statement. The encoding follows the verification of  $wlp$  as shown in [Kam19], or it can be analyzed by the while statement of Hoare logic as shown in 2.2.1. The encoding of the

while statement is introduced as follows:

```

assert  $P$ ;
havoc  $x_i$ ; assume  $P$ ;
(assume  $C$ ;  $\llbracket S \rrbracket$ ; assert  $P$ ; assume  $false$ )
 $\square$ 
assume  $\neg C$ 

```

Encoding begins with an *assert* statement representing the loop invariant. Guarded commands use a *havoc* statement assigning non-deterministic values to  $x_i$  and assuming the loop invariant. This process clears previous knowledge about these variables and satisfies the loop invariant. The non-deterministic choice accounts, if the loop condition holds, execute the body  $S$  and assert that the loop invariant  $P$  holds. The *assume false* statement ensures that any code following holds and guarantees the other non-deterministic choice. Last, assume the loop condition is incorrect with  $\neg C$ .

## Chapter 3

# Comparing Features in Dafny and Caesar

This chapter provides an overview of Dafny and Caesar, presenting their features to compare the possibility of translating and constructing Dafny programs into Caesar. Additionally, it shows the difference in supported features between Dafny and Caesar, where Caesar is limited in its features compared to Dafny, as Caesar is still a new verification program in development. Examining the supported features of both languages helps the reader understand the proof techniques.

### 3.1 Comparing Supported Features

This section introduces the supported types for constructing the code in Dafny and Caesar. It also compares and demonstrates the implementation of the statements. A complete overview of Dafny is provided in the official Dafny reference manual [LFC21].

#### 3.1.1 Types

**Numeric and Boolean Types** Dafny and Caesar support standard types as a boolean type returns a true or false value. Additionally, both languages support numeric types, including integers, natural numbers, and real numbers. Caesar includes a signed and unsigned value that makes the numeric types different. For instance, a signed integer contains positive and negative integers and zero, whereas an unsigned integer only contains natural numbers. Dafny supports characters, but Caesar does not support this feature.

**Datatypes** Dafny offers datatypes to create data structures defined by the users as recursive lists or trees. A datatype is declared with constructors as a single value or a value with a parameter. Recursive data structures are created using a constructor with parameters of the same datatype. For example, Listing 3.1 illustrates the datatype *List* in Dafny and Caesar.

```
1 datatype List = Null | Cons(head:int, tail:List)
3 domain List {
4   func null(): List
5   func cons(head:Int, tail:List): List
```

6 }

LISTING 3.1: Example of a datatype *List* in Dafny and Caesar

Dafny constructs a *Null* list to represent that the list is empty or a *Cons* that contains an integer *head* and its list *tail*. On the other hand, Caesar uses the domain to construct the datatype *List*.

### 3.1.2 Collections

The represented collection types of Dafny and Caesar are used to verify the codes.

**Arrays and Lists** Dafny has a mutable type called *array*, which is declared with the type  $T$  as  $\text{array}\langle T \rangle$ . Arrays in Dafny use indices in the  $[0, a.Length - 1]$  range. Caesar has a built-in type for lists  $[]T$  of type  $T$ . The verifier uses functions as *len*, *select*, and *store* to manipulate them. The operations of built-in *Lists* are comparable with the type *array* in Dafny, as illustrated in Listing 3.2.

<pre> 1 //Create a new array length of 3 2 var arr:array&lt;int&gt;:= new int[3];  4 //Get element in index 2 5 var res := arr[2];  7 //Get the length of the array 8 var res := arr.Length;  10 //Store element in index 1 11 arr[1] := 2; </pre>	<pre> 1 //Create a new list length of 3 2 assume ? len(arr) = 3  4 //Get element in index 2 5 var res = select(arr, 2)  7 //Get the length of the list 8 var res = len(arr)  10 //Store element in index 1 11 var arr = store(arr, 1, 2) </pre>
--	---

LISTING 3.2: Example of *array* and built-in *Lists* in Dafny and Caesar

**Sequences** Dafny supports type *seq*, an ordered list of immutable elements; type  $T$  is written as  $\text{seq}\langle T \rangle$ . This type helps analyze the index of the elements in an array. For instance, var *sq*, defined as  $\text{seq}\langle \text{int} \rangle$ , contains integer variables in square brackets. Dafny provides the type *string*, represented as  $\text{seq}\langle \text{char} \rangle$ , equivalent to a sequence of characters [LFC21]. The *string* types have the same properties as sequences and display string literals. For instance, Listing 3.3 shows the sequences.

```

1 var sq :seq<int> := [1, 2, 3, 4];

3 // Seq<char>
4 assert "test" == ['t','e','s','t'];

```

LISTING 3.3: Example of *seq* in Dafny

**Sets** Dafny has the type *set*, which represents a collection of elements without any particular order that does not contain duplicates. This type is used in annotations to determine if the variables contain the desired elements. Listing 3.4 illustrates an example of operating with the set variables *st\_1* and *st\_2*.



```

1 var st_1 :set<int> := {1,1,1,1,2};
2 var st_2 :set<int> := {1,2};

4 // Sets are equal.
5 assert st_1 == st_2;

```

LISTING 3.4: Example of *set* in Dafny

The type *multiset* shares the same properties as *set*, with the difference that *multiset* consists of multiple instances of each element. This specification is helpful to compare the elements of the array after sorting or updating with new elements. For instance, the variables of multisets *ms\_1* and *ms\_2* verify that they are unequal, as shown in Listing 3.5.

```

1 var ms_1 :multiset<int> := multiset([1,1,1,2]);
2 var ms_2 :multiset<int> := multiset([1,2]);

4 // Multisets are not equal.
5 assert ms_1 != ms_2;

```

LISTING 3.5: Example of *multiset* in Dafny

### 3.1.3 Methods, Functions, Procedures, and Domains

Dafny supports *method* statements, which are imperative and executable codes [KL12]. This unit consists of its name, input parameters, return value, body, and specification that defines the program's behavior. By defining the code's behavior, Dafny verifies the validation of the program. Caesar supports a procedure named *proc* to compute HeyVL statements. A procedure has an equal structure to Dafny, which defines a program's specifications and statements. For instance, the method and procedure in Listing 3.6 demonstrates an executable code that takes a parameter *x* and returns an integer value *y*. Here, the variable *x* assigns all occurrences to the variable *y*, which substitutes *x*.

<pre> 1 method assign_x(x:int) returns (y:int) { 2   y := x; 3 } </pre>	<pre> 1 proc assign_x(x:Int) -&gt; (y:Int) { 2   y = x 3 } </pre>
---	---

LISTING 3.6: Example of *method* and *proc* in Dafny and Caesar

Dafny supports *function*, which is a concept of a mathematical function [KL12]. A function can only consist of an expression with only one type and return an unnamed type. The input parameter in the example code in Listing 3.7 is an integer value and returns an unnamed boolean.

```

1 function even(x:int): bool {
2   if (x%2 == 0) then true else false
3 }

```

LISTING 3.7: Example of *function* in Dafny

Caesar supports *domain* statements to create user-defined types. To specify the properties of the type, the *domain* block defines a list of functions and axioms. The axioms specify the properties of the function. Listing 3.8 uses the domain *Calc* to specify a function *even* that returns true if the input value *x* is even. Additionally, write the axiom when the function has odd numbers, which is false.

```

1 domain Calc{
2   func even(x:Int): Bool
3   axiom even_t forall x:Int. (x%2 == 0) ==> even(x) == true
4   axiom even_f forall x:Int. (x%2 != 0) ==> even(x) == false
5 }

```

LISTING 3.8: Example of *domain* in Caesar

### 3.1.4 Assumption and Assertions

In Dafny, *assert* statements check if the logical expression is valid. If the expression can not be proven, then it outputs an error. The verifier states that a logical expression is valid using the *assume* statement, even though the expression is not verified. This statement even allows the verification of an invalid proposition, which leads to invalid conclusions. The *assume* statement guides the verification process to prove desired properties by assuming that the program holds after execution. Dafny does not compile files with the assumed statements and alerts whenever the verifier finds an assumed statement.

Caesar uses the assertion and assumption statements to validate quantitative specifications, which do not assign true or false but have a larger set of valuation[Sch23]. The verifier requires for boolean embedding inserting ? next to the *assert* and *assume* statements. Caesar compiles the code with the *assume* statement; for example, the while statements in 3.11 require the assumption to prove their invariants. Listing 3.9 demonstrates the relation between *assume* and *assert* statements.

<pre> 1 method AA() { 2   var x :int := 1; 3   assume 0 &gt; x; 4   assert 0 &gt; x; 5 } </pre>	<pre> 1 proc AA () -&gt; () { 2   var x :Int = 1 3   assume ? (0 &gt; x) 4   assert ? (0 &gt; x) 5 } </pre>
---	---

LISTING 3.9: Example of *assume* and *assert* in Dafny and Caesar

The variable  $x$  is assigned the value 1. Nevertheless, after assuming that the variable  $x$  is smaller than 0, the program verifies that this property is valid using the assertion statement.

### 3.1.5 If and While Statements

Dafny and Caesar feature an *if* statement, a boolean expression that computes their bodies depending on their condition. Dafny can remove the *else* statement, meaning it is an empty body. The *if* statement in Caesar contains an *else* block to define both cases. However, Caesar allows ignoring the *else* statement with an empty body. The method and the procedure *Min* demonstrate a simple implementation of the *if* statement in Dafny and Caesar, and it is acknowledged that the codes have a similar implementation.

<pre> 1 method Min(x:int, y:int) returns (res:int){ 2   if (x &lt; y) { 3     res := x; 4   } else { 5     res := y; 6   } 7 } </pre>	<pre> 1 proc Min(x:Int, y:Int) -&gt; (res:Int) { 2   if (x &lt; y){ 3     res = x 4   } else { 5     res = y 6   } 7 } </pre>
---	---

LISTING 3.10: Example of *if* statement in Dafny and Caesar

In Dafny, the *while* loop statement executes its body repeatedly if its condition is true. The loop sets a finite iteration and terminates the code if the condition is false. The code sticks in an infinite loop if the condition is always true. In Caesar, the loop is not directly supported and requires the implementation of guarded commands of the while statement, as shown in Section 2.3.4. The rule can be viewed as a quantitative version of the loop rule from Hoare logic[Hoa69]. Listing 3.11 illustrates the difference between Dafny and Caesar in implementing the while loop.

<pre> 1 method Mul(b:int, e:int) returns (x:int){ 2   x := 0; 3   var i := e; 4   while (0 &lt; i) 5     invariant x == b*(e-i) 6   { 7     x := x+b; 8     i := i-1; 9   } 10 } </pre>	<pre> 1 proc Mul(b:Int, e:Int) -&gt; (x:Int) { 2   x = 0 3   i :Int = e 4   assert ? ((x == b * (e-i))) 5   havoc x, i 6   assume ? ((x == b * (e-i))) 7   if (0 &lt; i) { 8     x = x + b 9     i = i - 1 10    assert ? ((x == b * (e-i))) 11    assume ? (false) 12  } else {} 13 } </pre>
---	---

LISTING 3.11: Example of *while* statement in Dafny and Caesar

In the example, it is recognizable that Caesar needs more lines of code to compute a while loop. First, assert that the invariant verifies the properties of the while loop. Havoc *x* and *i* that checks how the variable *x* and *i* performs in the code with each iteration by removing its previous value. Then, assume the invariant holds the loop specification. When the condition satisfies  $0 < i$ , compute the loop body and assert that the invariant is true. Last, assume it is false to ensure the body is computed regarding the invariant.

### 3.1.6 Pre-conditions and Post-conditions

Dafny uses annotations of specifications to verify *method* and *function* definitions, just as Caesar verifies *procedure* statements. The *domain* does not have annotations and consists of a *func* statement defined with axioms. Dafny declares pre-conditions with the keyword *requires* and post-conditions with the keyword *ensures*. They are annotated with multiple clauses or *&&* operator for additional specifications. Caesar annotates pre-condition as *pre ?* and post-condition *post ?*, which annotates *?* to set as the boolean embedding. However, Caesar does not allow multiple clauses in the procedure, requiring the operator *&&*. Listing 3.12 demonstrates the correct implementation of the pre-condition and post-condition of the previous method *Mul* in Dafny and Caesar.

<pre> 1  method Mul(b:int, e:int) returns (x:int) 2    requires 0 &lt;= e 3    ensures x == b*e 4  { 5    x := 0; 6    var i := e; 7    while (0 &lt; i) 8      invariant x == b * (e-i) 9    { 10     x := x + b; 11     i := i - 1; 12   } 13 }</pre>	<pre> 1  proc Mul(b:Int, e:Int) -&gt; (x:Int) 2    pre ? (0 &lt;= e) 3    post ? (x == b * e) 4  { 5    x = 0 6    var i :Int = e 7    assert ? (x == b * (e-i)) 8    havoc x, i 9    assume ? (x == b * (e-i)) 10   if (0 &lt; i) { 11     x = x + b 12     i = i - 1 13     assert ? (x == b * (e-i)) 14     assume ? (false) 15   } else {} 16 }</pre>
---	---

LISTING 3.12: Example of pre-condition and post-condition in Dafny and Caesar

Annotate variable  $e$  as a non-negative variable declaring  $0 \leq e$ , which is the quantifier of the multiplication. The method ensures that the result  $x$  equals  $b * (e - i)$ . In the following, annotate  $i$  to 0 to ensure the correct iteration process. Caesar does not support multiple condition clauses. Therefore, write the specifications after `&&` in the following line.

### 3.1.7 Predicates

In Dafny, a predicate is a function that returns a boolean that composes specifications to verify the programs. Caesar does not support the predicate statements, but two ways exist to define its specification. One way is to define the properties in a domain with the function that returns a boolean with its specifications as axioms. Another way is to simplify the translation by annotating the exact specification of predicates directly on the procedure. However, this approach might challenge to have an overview of the code. Listing 3.13 shows the translation of the predicate *odd* into the domain in Caesar.

<pre> 1  predicate odd(x:int) { 2    (x%2) == 1 3  }</pre>	<pre> 1  domain Calc{ 2    func odd(x:Int): Bool 3    axiom o_t forall x:Int. 4      (x%2 == 1) ==&gt; (odd(x) == true) 5  }</pre>
--	--

LISTING 3.13: Example of *predicate* in Dafny and its translation in Caesar

## 3.2 Additional Features in Dafny

### 3.2.1 Decreases Clause

Dafny provides the *decreases* clause for loop termination of the programs under its specification and the code. Dafny automatically generates this clause or is annotated by the user for the program's termination by taking the variables

listed in the function parameter. When the method fails to decrease its iteration, Dafny does not allow it to terminate its code. For instance, Listing 3.14 demonstrates annotating the *decreases* clause in a while loop.

```

1  method Mul(b:int, e:int) returns (x:int, i:int)
2      requires 0 <= e
3      ensures x == b * (e-i) && i == 0
4  {
5      x := 0;
6      i := e;
7      while (0 < i)
8          invariant x == b * (e-i)
9          decreases i
10     {
11         x := x + b;
12         i := i - 1;
13     }
14 }
```

LISTING 3.14: Example of *decreases* in Dafny

### 3.2.2 Print Statement

Dafny supports the *print* statement to display the values, where it takes multiple expressions to the console, separated by commas. Dafny converts the value types as integers, booleans, sets, arrays, or self-created datatypes into a string to display in the console. The printing feature does not verify the properties of the method or function statement, but it outputs the values of the variables without verifying their correctness. Listing 3.15 provides an example of calling the *print* statement.

```

1  datatype List = Null | Cons(head:int, tail:List)

3  method Main() {
4      var ls :List := Cons(1, Cons(2, Null));
5      print "ls: ", ls, "\n";
6  }
```

LISTING 3.15: Example of *print* in Dafny

The provided code example has a datatype *List*, and after initiating *ls* with elements of 1 and 2, Dafny prints the list *ls* that outputs a string in the console as "ls: List.Cons(1, List.Cons(2, List.Null))". The string "`\n`" adds a new output line.

### 3.2.3 Trigger Selection

Dafny has the feature of manual and automatic trigger selection. The triggers are patterns to compute the instantiation of quantifiers to solve the process and identify the triggered value. Dafny automatically identifies and selects *matching triggers* to instantiate the quantified statements. Matching triggers are specific instances of quantified expressions to verify their process. With the trigger selection, Dafny tries to avoid trigger instantiation that stays in a loop or takes a long time to verify the quantified expressions. inefficient[ALR14]. The example code of Listing 3.16 demonstrates the trigger's use case by manually selecting the trigger.

```

1 function P(x:int): bool
2 function Q(y:int): bool

4 method TriggerQ()
5   requires forall i {:trigger Q(i)} :: P(i) ==> P(i-1) && Q(i)
6   {
7     assume P(0);
8     assert Q(0);
9     assert P(-1);
10    assert P(-2);
11  }

```

LISTING 3.16: Example of trigger selection in Dafny

The code defines two functions,  $P(x)$  and  $Q(y)$ , that return a boolean. The quantifier specifies  $P(i)$  implies  $P(i - 1)$  and  $Q(i)$ . By selecting the trigger as  $Q(i)$ , the function  $Q$  instantiates with the value  $i$ . The method *TriggerQ()* assumes that  $P$  holds for 0. Then Dafny verifies  $P(-1)$  and  $Q(0)$ . However, the  $P(i)$  is not selected as the trigger, and the verifier does not verify that the function  $P$  holds for  $P(-2)$ . In order to verify the method, include an additional  $P(i)$  as the trigger, as shown in Listing 3.17.

```

1 method TriggerQ()
2   requires forall i{:trigger Q(i) && P(i)} ::
3     P(i) ==> P(i-1) && Q(i)
4   {
5     assume P(0);
6     assert Q(0);
7     assert P(-1);
8     assert P(-2);
9   }

```

LISTING 3.17: Debugging of trigger selection in Dafny

With this adjustment, Dafny verifies that  $P$  holds for each iteration decreasing  $i - 1$  and, therefore, also  $P(-2)$  without any problem. With this configuration, the method terminates successfully.

### 3.2.4 Finding Errors

Dafny provides a feature to find errors in the program by processing the verifier in the background [LW14]. Whenever there is an error, the verifier shows which part of the code requires adjustment. Figure 3.1 lines up the specification error by lining the code in red. Lining the code allows the user to directly recognize the error in the post-condition and make it possible to correct the code.

```

method LeftPad(str: string, ln: int, c: char) returns (res: string)
  requires 0 <= |str| && 0 <= ln
  ensures max(ln, |str|) == |res|
  ensures prefix(str, ln+1, c, res)
  ensures suffix(str, ln, res)
  decreases ln

```

FIGURE 3.1: Dafny lines up the error code in red

## Chapter 4

# Comparison Using Left Pad Function

The comparison of Dafny and Caesar starts with the implementation of an incident involving a Node.js package named "Left Pad", which was published on the package management platform as Node Package Manager (NPM) [Bog+16]. After the package owner removed the package from NPM, it caused disruptions to internet sites that depend on it, such as Facebook and Netflix[Abd+20]. The Left Pad function represents an example of proof techniques of string, which is a function that requires verifying the aspects of the string behavior, such as its length and alignment. Figure 4.1 demonstrates an example of the Left Pad function.

*"test"*  
↓    length = 7  
*"000test"*

FIGURE 4.1: Example of the Left Pad function

A string is given that is called *"test"* with a length of 4. The desired length of the padded string is 7. The function adds the padding, the number zero, to the left side of the input string, which returns the padded string with the desired length.

Listing 4.1 shows the implementation of the Left Pad function in Dafny.

```
1 method LeftPad(str:string, ln:nat, c:char) returns (res:string)
2 {
3   if (ln <= |str|) {
4     res := str;
5   } else if (|str| < ln) {
6     var i := 0;
7     var pads := ln - |str|;
8     res := str;
9     while(i < pads)
10    {
11      res := [c] + res;
12      i := i + 1;
13    }
14  }
15 }
```

LISTING 4.1: Implementation of the Left Pad function in Dafny

The Left Pad function performs in the following two cases. Lines 3 and 4 show the first case for the condition  $ln \leq |str|$ , and the body sets *res* to *str*, which does not require padding. In the second case, as shown in lines 5-12, padding is necessary for the condition  $|str| < ln$ . The integer *i* is initialized to 0 as the loop iteration, and the variable *pads* is initialized to  $\max(ln - |str|, 0)$  as the upper bound of the iteration. Add the pads to the string while the condition  $i < pads$  holds. A padding character *c* is added in each iteration to the left side of the resulting string *res*. The variable *i* is incremented by 1 in each loop. Once the loop is complete, Dafny outputs *res* with the desired length *ln*, with the padding character *c*.

## 4.1 Verification of Left Pad Function in Dafny

This section verifies the method *LeftPad* in Listing 4.1 with appropriate specifications. Therefore, two predicates called *prefix* and *suffix* are implemented, as each predicate specifies the structure of the padded string and helps to prove that the method returns a string consisting of padding characters and the original input string.

### 4.1.1 Predicates of *LeftPad* in Dafny

The first predicate ensures that the output string *res* contains the expected padding characters. Furthermore, the second predicate guarantees that the original string *str* elements align correctly with the padded string *res* elements.

**Predicate *prefix*** The predicate *prefix* described in Listing 4.2 ensures that the output string has been correctly padded with the pads.

```

1 predicate prefix(str:string, ln:int, c:char, res:string)
2   requires ln-|str| <= |res|
3   {
4     forall k :: 0 <= k < ln-|str| ==> c == res[k]
5   }
```

LISTING 4.2: Predicate *prefix* of the Left Pad function in Dafny

The pre-condition is annotated as  $ln - |str| \leq |res|$ , ensuring the index of the padded characters of *res* is not outside the range. The predicate defines the quantified expression that asserts for all *k* indices from 0 to  $ln - |str|$ , as the element at index *k* in *res* corresponds to the padding character *c*.

**Predicate *suffix*** The second predicate, called *suffix* in Listing 4.3, determines whether elements of the input string *str* are consistent with elements in an output string *res* within the correct index.

```

1 predicate suffix(str:string, ln:int, res:string)
2   requires max(ln, |str|) == |res|
3   {
4     forall k :: 0 <= k < |str| ==> str[k] == res[max(ln-|str|, 0)+k]
5   }
```

LISTING 4.3: Predicate *suffix* of the Left Pad function in Dafny



The pre-condition specifies that the length of *res* is either *ln* or  $|str|$ , as the function returns *str* if the desired length is smaller than the length of the input string. The body of the predicate contains the *forall* quantifier with its expression over indices *k* from  $[0, |str|]$ . It asserts that for every index of *k*, the elements at position  $\max(ln - |str|, 0) + k$  in *res* are equal to the element in *str*, at position *k*. The index  $\max(ln - |str|, 0)$  ensures that the element of *res* starts after padding characters.

### 4.1.2 Verification of *LeftPad* in Dafny

Verification of the Left Pad function involves specifying the necessary annotations. The complete code of the verified function is in Appendix A.1. Listing 4.4 illustrates the annotations to ensure the correctness of the Left Pad function.

```

1 method LeftPad(str:string, ln:int, c:char) returns (res:string)
2   requires 0 <= |str| && 0 <= ln
3   ensures max(ln, |str|) == |res|
4   ensures prefix(str, ln, c, res)
5   ensures suffix(str, ln, res)
6   {...}

```

LISTING 4.4: Annotations of *LeftPad* in Dafny

The method requires the length of the input string  $|str|$  and *ln* to be non-negative values. The annotation in line 3 provides that the output length should equal the maximum value between *ln* and  $|str|$  as if *ln* is smaller than  $|str|$ , the method must return *str*. To ensure that the prefix of the padded string *res* consists of padding characters, annotate with the predicate *prefix* in Listing 4.2 as *prefix(str, ln, c, res)*. Then annotate the suffix of the padded string *res* equals the original input string with the predicate *suffix* in Listing 4.3 as *suffix(str, ln, res)*. To ensure the while loop, the result string *res* contains the padding characters and the characters of *str*. The invariant of the while loop, as shown in Listing 4.5, annotates the correctness of its body.

```

1 var i := 0;
2 var pads := max(ln - |str|, 0);
3 res := str;
4 while(i < pads)
5   invariant 0 <= i <= pads
6   invariant |str| + i == |res|
7   invariant prefix(str, i+|str|, c, res)
8   invariant suffix(str, i+|str|, res)
9   {...}

```

LISTING 4.5: Invariants of *LeftPad* in Dafny

In the while loop, the invariant  $0 \leq i \leq pads$  ensures the loop repeats until *i* reaches the length of padding characters. Additionally, the invariant  $|str| + i = |res|$  ensures that *res* equals the sum of the input string *str* and the length of the added padding characters *i*. The invariant in line 7 uses the predicate *prefix* in Listing 4.2, which takes the parameter of the desired result length  $i + |str|$  to ensure all the characters in *k* between  $0 \leq k < i$  of *res* equals the padding character *c*. The second loop invariant in line 8 is the predicate *suffix* in Listing 4.3, where it takes variable  $i + |str|$  as the parameter of the resulting length, which then *res* maintains the input string *str* characters starting from index *i*. Even though the method *LeftPad* in Appendix A.1 verifies the Left

Pad function, it is not enough to assert the correctness of the padded string, as shown in Figure 4.2.

```
method Main() {
  var a: string := LeftPad("test", 5, '0');
  assert a[...] == ['0', 't', 'e', 's', 't'];
}
```

FIGURE 4.2: The method *LeftPad* fails to verify the correctness of the string *a*.

The method *Main* declares the variable *a* that stores the result of calling the method *LeftPad* with three arguments: the string "test", the original string, an integer 5, the desired length of the output string after padding, and the padding character 0. Then, the method *Main* fails to assert that the output string *a* corresponds to the padded string.

The result emerges as Dafny does not know how to instantiate the post-condition *suffix(str, ln, res)* of the method *LeftPad*. The *Main* method wants to prove for all *i*, *a*[*i*] is correct, and therefore, the quantifier of the predicate *suffix* must prove that  $k = i - \max(\ln - |\text{str}|, 0)$ , which means that the elements of the input string *str* are allocated in the index *k* of *res*. Therefore, an improvement of the predicate *suffix* is required by modifying the quantifier as follows:

- The range of the index *k* is defined as  $\max(\ln - |\text{str}|, 0) \leq k < |\text{res}|$  to verify the allocation of the input string in *res*.
- To prove that the characters of the index *k* of the result string *res* are in the correct positions, change the index of *res* from  $\max(\ln - |\text{str}|)$  to *k*.
- To ensure that the index of the string *str* is not out of the range, change The index of input string *str* from *k* to  $k - \max(\ln - |\text{str}|, 0)$ .

Listing 4.6 demonstrates the improvement of the predicate *suffix* to assert the result of the Left Pad function. Moreover, the predicate *suffix* confirms to the verifier how to instantiate the quantifier to retrieve information about *res*[*i*].

```
1 predicate suffix(str:string, ln:int, res:string)
2   requires max(ln, |str|) == |res|
3   {
4     forall k :: max(ln-|str|, 0) <= k < |res| ==> str[k-max(ln-|str|, 0)] == res[k]
5   }
```

LISTING 4.6: Improvement of the predicate *suffix*

After adjusting the predicate *suffix*, Dafny verifies that the result of the method *LeftPad* is correct, as shown in Figure 4.3.

```
method Main() {
  var a: string := LeftPad("test", 5, '0');
  assert a[..] == ['0', 't', 'e', 's', 't'];
}
```

FIGURE 4.3: After correcting *suffix*, Dafny verifies the padded string

## 4.2 Verification Failure of Left Pad Function in Built-in Lists in Caesar

This section demonstrates and analyses the verification failure of the Left Pad function with the built-in type *Lists* in Caesar using the same verification annotations from Dafny. Caesar does not support a sequence type, so it uses the built-in type *Lists*. Another limitation of Caesar is that it does not support a type of character. However, characters supported in Dafny are represented in UTF-16 code unit[LFC21]. UTF-16 encodes Unicode in which each character comprises one or two 16-bit elements. These binary values can be translated into decimal numbers[Muk90]. So, instead of using type *string*, Caesar implements the Left Pad with the list of integer values.

In Dafny, implementing the Left Pad function requires adding pads on the left side of the string, iterating within a while loop. However, Caesar is not satisfied with implementing the Left Pad function correctly only by adding pads to the list. The result string does not have information on the elements of the input list after the added pads. Instead, Caesar must declare the functional specifications of the iteration of adding pads and the elements of the input list. First, declare the annotations of the procedure of the *LeftPad* with its parameters similar to Dafny as in Listing 4.4.

```
1  proc LeftPad(str:[]Int, ln:UInt, c:Int) -> (res:[]Int)
2    pre ? (0 <= len(str) && 0 <= ln)
3    post ? (
4      (ln(res) == ite(ln < len(str), len(str), ln)) &&
5      (forall k:UInt. ((k < ln - len(str))) ==> (select(res, k) == c)) &&
6      (forall k:UInt. ((ite(0 < ln-len(str), ln-len(str), 0) <= k) &&
7      (k < len(str))) ==> (select(res, k) == select(str, (k - ite(0 < ln-len(str), ln-len(str),
8      0))))))
9    {...}
```

LISTING 4.7: Annotations of *LeftPad* in Caesar

The pre-condition states that the input list *str* and integer *ln* must be greater than or equal to zero:  $0 \leq \text{len}(\text{str})$  and  $0 \leq \text{ln}$ . In the post-condition, Caesar utilizes the expression  $\text{len}(\text{res}) = \text{ite}(\text{ln} < \text{len}(\text{str}), \text{len}(\text{str}), \text{ln})$ , to annotate the output length is the maximum between the required *length* or length of *str*. The function  $\text{ite}(a, b, c)$  represents a conditional choice that evaluates to *b* if *a* is true and to *c* otherwise[Sch23], which then verifies the maximum between *a* and *b*, with the expression  $\text{ite}(a < b, b, a)$ . Additionally, post-condition ensures both expressions of quantifiers of the predicate *prefix* in Listing 4.2 to verify the allocation of the padding characters in the executing list and the predicate *suffix* in Listing 4.6 to verify the allocation of the elements of the input list in executing list.

The procedure *LeftPad* requires a while loop to ensure the code inserts the pads correctly in the list *res*. To ensure the correctness of the while loop, define the invariant  $I_1$ :

$$\begin{aligned} I_1 : & 0 \leq ln \wedge 0 \leq i \leq pads \wedge \\ & len(res) = ln \wedge \\ & \forall k : \mathbf{N} :: k < i \implies select(res, k) = c \end{aligned}$$

The variable  $i$  represents the loop iteration, is initialized to 0, and is less or equal to *pads*. The variable *pads* is  $ln - len(str)$ , which is the length of the pads. The length of *res* equals *ln* as the pads are added. During the iteration of  $i$ , the invariant ensures the character  $c$  is the Left Padded in *res*.

The invariant  $I_2$  is consistent with the invariants of the while-loop of the Left Pad method in Dafny in Listing 4.5, and it includes the specification of the predicates *prefix* in Listing 4.2 and *suffix* in Listing 4.6 from the previous section:

$$\begin{aligned} I_2 : & 0 \leq j \leq len(str) \wedge \\ & len(res) = ln \wedge \\ & \forall k : \mathbf{N} :: k < i \implies select(res, k) = c \wedge \\ & \forall k : \mathbf{N} :: 0 \leq k < j \implies \\ & select(res, ln - len(str) + k) = select(str, k) \end{aligned}$$

The non-negative variable  $j$  represents the loop iteration and is initialized to 0. The invariant ensures the iteration  $j \leq len(str)$ . Additionally, the invariant  $I_2$  ensures that the length of *res* equals *ln*. Furthermore,  $I_2$  contains invariant from  $I_1$ , which ensures the character  $c$  is the Left Padded in the resulting list *res*. At last, during the iteration of  $j$ , the invariant guarantees that the list *str* copies its values in the Left Pad *res*.

Appendix A.5 exhibits the failed verification of the procedure *LeftPad* in datatype *Lists* with its annotations and the body with the invariants of the while loops. The first loop is declared in Listing 4.8.

```

1  var i :UInt = 0
2  var pads :Int = ln-len(str)
3  assert ? I1
4  havoc res, i
5  assume ? I1
6  if (i < pads){
7    res = store(res, i, c)
8    i = i + 1
9    assert ? I1
10   assume ? (false)
11 } else {}

```

LISTING 4.8: The first loop invariant of *LeftPad* in Caesar

The variable *pads* is an integer variable of  $ln - len(str)$ , and  $i$  is 0. By each iteration, the values of *res* and  $j$  change, which makes them havoc variables. The first loop holds for the condition  $i < pads$  with the invariant  $I_1$ , where The body stores character  $c$  to *res* in each iteration  $i$ . Then, increase the iterator  $i$  by one.

The second loop ensures the placement of the elements of *res* as shown in Listing 4.9.

```

1      var j :UInt = 0
2      assert ? I2
3      havoc res, j
4      assume ? I2
5      if (j < len(str)) {
6          res = store(res, ln-len(str) + j, select(str, j))
7          j = j + 1
8          assert ? I2
9          assume ? (false)
10     } else {}
    
```

LISTING 4.9: The second loop invariant of *LeftPad* in Caesar

The iterator *j* is declared as 0. The variables *res* and *j* are set as havoc, as their value changes by each iteration. The second loop uses the invariant *I<sub>2</sub>*, holding for *j* < *len(str)*. The body copies the values of *str* to *res* in each iteration *j*.

Running the code of Appendix A.5, the verifier fails to ensure the correctness of the procedure *LeftPad*. The complete counter-example is shown in Appendix B.1. Figure 5.4 illustrates the part of the counter-examples of procedure *LeftPad* to analyze the problems of the properties of *res* and *str*.

```

1  j_0!21 -> 3
2  k!20 -> 0
3  ln!9 -> 18022
4  c!12 -> 9
5  res_!116 -> (let ((a!1 (store (store (store ((as const (Array Int Int)) 6) 0 9) 1575 29)
6                                     (- 9943)
7                                     30)))
8          (|List[Int]_list| 18022 a!1))
9  str!8 -> (let ((a!1 (store (store (store ((as const (Array Int Int)) 5) (- 5982) 33) 1 9)
10                                     3
11                                     (- 2332))))
    
```

FIGURE 4.4: Counter-examples of *LeftPad* in Built-in Lists

The document of SMT-LIB [BST+10] helps to analyze the counter-examples, which proposed the *ArrayEx* and its extensionality, used for the built-in type *Lists* feature of Caesar. The expression (*x*!*n*) annotates every term of a variable *x* with its term attribute *n*. Term attributes are meta-logical information that does not affect the logical meaning of the variable[BST+10]. So in Figure 5.4, the variable *j* in term 21 is inserted with 3, which means the second loop is the iteration of 3. The variable *ln* in term 9 has a length of 18022. The padding character *c*!12 is inserted with 9. Caesar assigns *res*!16 a list of type integers, and the value *a*!1 is the identifier for the list. The list is constructed using the *store* operation on an array of 6. The store function has on the first parameter the index and the second parameter the stored element. For instance, the value 9 is stored at index 0. Line 8 shows that the list *a*!1 has a length of 18022. The variable *str* is a list of integers with the size of 5. For instance, 33 is stored in index -5982. The counter-examples show that the input and output lists of the procedure *LeftPad* with the built-in type *List* do not guarantee equal length or store the same value.

### 4.3 Verification of the Left Pad Function in Datatype *List* in Caesar

The built-in type *Lists* in Caesar fails to verify and implement the Left Pad function that does not guarantee the length and stored elements of the executed list. This section is divided into two parts to verify and implement the Left Pad function in Caesar. The first part represents a new datatype *List* and explains the proof method of extensionality. The second part verifies the Left Pad function by implementing the datatype *List* by proving the extensionality of its return list and the desired padded list.

#### 4.3.1 Datatype *List* and Axiom of Extensionality in Caesar

Verifying the Left Pad function in Caesar requires that the return list has the desired length and contains the elements of the input list. The axiom of extensionality proves these properties by ensuring that the return list of the Left Pad function equals the desired padded list. The extensionality of a list is defined as [Stu+01]:

$$\forall a : \text{List}, \forall b : \text{List} :: \forall i : a_i \Leftrightarrow b_i \implies a = b.$$

The axiom of extensionality of a list means that for any list  $a$  and  $b$ , if they have the same elements in the same index, meaning they have the same length, these lists are equal. However, Caesar cannot automatically accept the axiom of extensionality, and it requires proof so that the statement is correct. The paper [LP13] introduces a datatype called *List* in Dafny and its proof of extensionality by defining it in a method. Hence, this section focuses on adapting and implementing the proposed solution of this paper in Caesar to ensure the correctness of the allocation of the elements and the length of the return list after calling the procedure *LeftPad*.

**Domain *List*** The *domain* statement declares the properties of the *List* in Caesar with its functions and axioms as shown in Listing 4.10.

```

1  domain List {
2      func null(): List
3      func cons(head:int, tail:List): List

4
5      func get_value(ls:List): Int
6      axiom g_v forall h:Int, t:List. get_value(cons(h, t)) == h

7
8      func get_tail(ls:List): List
9      axiom g_t forall h:Int, t:List. get_tail(cons(h, t)) == t

10
11     func is_null(ls:List): Bool
12     axiom i_n forall ls: List. is_null(ls) == (ls == null())

13
14     func is_list(ls: List): Bool
15     axiom i_l forall ls:List, h:Int, t:List.
16     ((get_tail(ls) == t) && (get_value(ls) == h)) ==> (is_list(ls) == (ls == (cons(h, t))))

17
18     axiom either_list_or_null forall ls:List. (is_null(ls) || is_list(ls))
19     ...
20 }
```

LISTING 4.10: Domain of the datatype *List* in Caesar

The datatype *List* is null, meaning it is empty or contains an integer value at the top of the list named *head* and its sub-list called *tail*, which includes the remaining elements. To retrieve the elements in the list, declare the function *get\_value*, which returns the value at the top of the list, and if the list is empty, it is undefined. The axiom in line 6 states that for any integer *h* and list *t*, the value returned by *get\_value* for the list *cons(h, t)* equals *h* itself. To obtain the rest of the list, declare the function *get\_tail*, which follows a similar structure of the axiom of *get\_value*, but the axiom returns the list *t* itself. The function *is\_null* returns an empty list, with its axiom that declares *ls = null()*. Declare the function *is\_list(ls : List)* to define that the list *ls* contains elements. The axiom in lines 15 and 16 defines that *t* is the rest of *ls* and *h* is the element on top of *ls*. This implies that *ls* is listed when constructed as *cons(h, t)*. To clarify that the list can only be null or either, specify the axiom as in line 18.

**Func length** Upon completing the implementation of the axioms for the datatype *List*, it is required to define its operations. The first step begins with defining the length of the list as a function in domain *List* with its axioms in Listing 4.10. Listing 4.11 demonstrates the implementation.

```

1  domain List {
2  ...
3  func length(ls:List): UInt
4      axiom l_n forall ls:List. is_null(ls) ==> (length(ls) == 0)
5      axiom l_r forall ls:List. is_list(ls) ==> (length(ls) == 1 + length(get_tail(ls)))
6  ...
7  }
```

 LISTING 4.11: Length of the datatype *List* in Caesar

The function *length(ls)* takes input from a list *ls* and returns an unsigned integer value. The first axiom accepts that the length of the list *ls* is 0 if the list is null. The other axiom expresses that the length of *ls* computes recursively, adding 1 to its length until the list is null.

**Func select** The datatype *List* requires a function that selects the element in each list index. The following Listing 4.12 illustrates the implementation of the function *select* with its axioms in domain *List*.

```

1  domain List {
2  ...
3  func select(ls:List, i:UInt): Int
4      axiom s_n forall ls:List, i:UInt. is_null(ls) ==> (select(ls, i) == 0)
5      axiom s_l_v forall ls:List, i:UInt.
6          (is_list(ls) && (i == 0)) ==> (select(ls, i) == get_value(ls))
7      axiom s_l_r forall ls:List, h:Int, t:List, i:UInt.
8          (is_list(ls) && (0 < i) && (ls == cons(h, t))) ==> (select(ls, i) == select(t, i-1))
9  ...
10 }
```

 LISTING 4.12: Select of the datatype *List* in Caesar

The function *select(ls, i)* takes a list *ls* and an unsigned integer *i* as the parameters. The first axiom accepts that if the list is empty, it returns the value 0. Then, declare additional axioms in case the list contains elements. If the index is *i = 0*, the function returns the value of the list. If the index *i* is greater than 0, the function recursively computes the list *ls* until *i* reaches 0.

**Proof of the extensionality in procedure *Extensionality*** The proof of extensionality in Caesar requires translating the proof in Dafny step by step. In the paper [LP13], Dafny uses a *calc* statement, which can be transformed into a *assert* statement, to prove the induction of the extensionality in Caesar. Implementing the procedure *Extensionality* behaves like a lemma, which does not return any value. Listing 4.13 shows the annotations that specify the extensionality.

```

1 procedure Extensionality(ls:List, ts:List) -> ()
2   pre ? (
3     (length(ls) == length(ts)) &&
4     (forall i:UInt. (i < length(ls)) ==> (select(ls, i) == select(ts, i)))
5   )
6   post ? (ls == ts)
7   {...}

```

LISTING 4.13: Annotations of the procedure *Extensionality*

The procedure takes two lists, *ls* and *ts*, as parameters and does not return any type. The procedure begins with annotating the pre-condition and the post-condition, the axiom of extensionality 4.3.1 as shown in lines 3-6. The pre-condition requires the length of both lists to be equal, and the element must be equal in index *i*. Then, the post-condition ensures that *ls* and *ts* are equal after the execution.

The body of the code corresponds to a proof of the lemma that is structured into two cases. The first case is shown in Listing 4.14 and the second case in Listing 4.15.

```

1 if is_null(ls) {
2   assert ? (length(ts) == 0)
3   assert ? (length(ls) == length(ts))
4   assert ? (ts == null())
5   assert ? (ls == ts)
6 }

```

LISTING 4.14: Proof of the extensionality for an empty list in Caesar

In the first case, if the list *ls* is empty, the *assert* statements in lines 2-4 prove that *ls* = *ts*. The first statement defines *length(ts)* = 0, indicating that the lengths of both lists are equal. Consequently, this ensures that *ts* is empty and verifies *ls* = *ts*.

```

1 if is_list(ls){
2   if is_list(ts){
3     var h :Int = get_value(ls)
4     var t :List = get_tail(ls)
5     var g :Int = get_value(ts)
6     var u :List = get_tail(ts)
7     assert ? (ls == cons(h, t))
8     assert ? (h == select(ls, 0))
9     assert ? (select(ls, 0) == select(ts, 0))
10    assert ? (select(ts, 0) == g)
11    assert ? (h == g)
12    assert ? (cons(h, t) == cons(g, t))
13    assert ? (forall j:UInt. (j < length(t)) ==>
14      ((select(t, j) == select(ls, j+1)) &&
15      (select(ls, j+1) == select(ts, j+1)) &&
16      (select(ts, j+1) == select(u, j)) &&
17      (select(t, j) == select(u, j))
18    ))
19    Extensionality(t, u)
20    assert ? (cons(g, t) == cons(g, u))

```



```

21   assert ? (cons(g, u) == ts)
22   assert ? (ls == ts)
23 }

```

LISTING 4.15: Proof of the extensionality for a list in Caesar

In the second case, if  $ls$  contains elements, it requires declaring the variables as shown in lines 3-6. The variable  $h$  is the value at the top of the list  $ls$ , and  $t$  is the rest of the list  $ls$ . The variable  $g$  represents the value at the top of the list  $ts$  and  $u$  the rest of the list  $ts$ . After that, the *assert* statements of the body ensure the proof process in induction in lines 7-22.

The first statement ensures that  $ls = cons(h, t)$  due to the variable declarations. Then, assert that the list  $ls$  contains variable  $h$ , stored in index 0. Then check if  $ts$  also contains the same element in index 0. By confirming the initial value is equal, prove with the quantified index  $j$  that the entire elements of the list  $ls$  and  $ts$  are equal in index  $j$  as shown in lines 13-17. After that, call recursively the procedure *Extensionality* with the parameters  $t$  and  $u$  to prove that these lists prove the extensionality. The final steps of the assertions in lines 20-22 ensure that the element of the tail of both lists  $t$  and  $u$  are equal with the variable  $g$ , which then verifies that  $ts = cons(g, u)$  and at the last  $ls = ts$ . The procedure *Extensionality* verifies the extensionality property between the two lists that asserts  $ls = ts$  when the list is empty or contains elements.

### 4.3.2 Verification of *LeftPad* in Datatype List in Caesar

After implementing the datatype *List* and successfully defining the axiom of extensionality in Caesar, the verifier allows the Left Pad function validation by utilizing this datatype. The Left Pad function is implemented in a recursive function as the datatype *List* is defined in a recursive form [Cor23]. This allows us to avoid implementing additional procedures to implement and verify the Left Pad function.

The procedure *LeftPad* verifies the properties of the Left Pad function as shown in Listing 4.16.

```

1  proc LeftPad(str:List, ln:UInt, c:Int) -> (res:List)
2    pre ? (0 <= length(str))
3    post ? (
4      (length(res) == ite(ln < length(str), length(str), ln)) &&
5      (forall i:UInt. (i < ln-length(str)) ==> (select(res, i) == c)) &&
6      (forall j:UInt. ((ite(0 < ln-length(str), ln-length(str), 0) <= j) && (j < length(res)))
7        ==>
8        (select(res, j) == select(str, j- ite(0 < ln-length(str), ln-length(str), 0))))
9    )
10   {
11     if (ln <= length(str)) {
12       res = str
13     } else{
14       if (length(str) <= ln)
15       {
16         res = LeftPad(cons(c, str), ln, c)
17       } else {}
18     }
19   }

```

LISTING 4.16: Annotations of *LeftPad* in datatype *List* in Caesar

The pre-condition and post-condition remain consistent with the previous annotation from the built-in list. Therefore, the post-condition has equal annotations using the functions created in domain *List*. As mentioned before, in contrast to a while-loop, the procedure's body is implemented in a recursive function, in which the return list of the Left Pad function is straightforward. The body is declared with two conditions. Lines 10-11 describe if  $ln \leq str$  the procedure returns the list *str*. If the length of the input list *str* is less than the desired length *ln*, then the procedure recursively calls the input list of *cons(c, str)*, with the length *ln* and the character *c* as shown in lines 13-15. As there are no other conditions, the else statement in line 16 is not defined.

Figure 4.5 shows the procedure *Main* to demonstrate the behavior of the procedure *LeftPad* and the axiom of extensionality.

```

proc Main () -> ()
{
  var ls : List = cons(0, cons(1, cons(3, null())))
  var res: List = LeftPad(cons(1, cons(3, null())), 3, 0)
  Extensionality(ls, res)
  assert? (length(res) == length(ls))
  assert? (forall i: UInt. select(res, i) == select(ls, i))
}

```

FIGURE 4.5: Caesar verifies the Left Pad function using the procedure *Extensionality*

The code declares the list *ls*, containing elements 0,1,3, to compare with the padded list. The list *res* calls the procedure *LeftPad* with parameters of the list containing elements 1,3, the length 3, and the character 0. Then prove two lists *ls* and *res* are extensionally equal. Then Caesar asserts *res* and has the same length as *ls*, which verifies that the list *res* is padded with the desired length. Additionally, the verifier asserts that the padded list *res* contains the same values as *ls* in the same order, which ensures the list *res* contains the correct number of the padding characters and the list elements *ls*. After compiling the program, it verifies all implemented procedures, verifying the Left Pad and proof of extensionality in Caesar.

## Chapter 5

# Comparison Using Bubble Sort

Bubble sort is an elementary sorting algorithm [Ive62], which sorts an array with element  $n$  and has the average case time complexity of  $O(n^2)$ . Furthermore, it has a worst-case time complexity of  $O(n^2)$ . The array sorts the elements with "bubble up" to their proper position[Knu73]. The main goal of this section is to present the verified implementation of Bubble Sort in Dafny and also examine the difference in encoding and output of the results between Dafny and Caesar. This algorithm example introduces the essential concepts for successful Dafny programming, including the correct use of predicates, type arrays, and mutable states. By declaring necessary functions, the readers understand the implementation of Bubble Sort in Dafny. It also declares key directives for implementing Bubble Sort in Caesar to display the output of the results. Additionally, this section discusses the challenges during the algorithm verification process. Figure 5.1 shows an example of Bubble Sort.

<u>3</u>	<u>2</u>	5	1	4	swap 3, 2
2	3	<u>5</u>	<u>1</u>	4	swap 5, 1
2	3	1	<u>5</u>	<u>4</u>	swap 5, 4
2	3	1	4	5	
2	<u>3</u>	<u>1</u>	4	5	swap 3, 1
2	1	3	4	5	
<u>2</u>	<u>1</u>	3	4	5	swap 2, 1
1	2	3	4	5	

FIGURE 5.1: Example of a Bubble Sort

The list is initiated with the elements 3,2,5,1,4. The algorithm starts comparing from left to right. The element 3 is larger than the 2, which is out of order, so it swaps them. Swapping two elements in a list means that they change their positions in the correct order. Then the list becomes 2,3,5,1,4. Swapping the elements continues until the greatest number of the list is in the correct position at the end. After that, the algorithm repeats the same process of swapping the elements from the beginning. The list is sorted when the whole process is finished.

Bubble Sort is implemented in Dafny according to Listing 5.1.

```
1 method BubbleSort(arr:array?<int>)
2   requires arr != null
3   modifies arr
4   {
```

```

5   var i := 0;
6   var n := arr.Length-1;
7   while (0 < n-i)
8   {
9       var j := 0;
10      while (j < n-i)
11      {
12          if(arr[j] > arr[j+1])
13          {
14              var ind1 := arr[j];
15              var ind2 := arr[j+1];
16              arr[j] := ind2;
17              arr[j+1] := ind1 ;
18          }
19          j := j+1;
20      }
21      i := i+1;
22  }
23  }

```

LISTING 5.1: Implementation of Bubble Sort in Dafny

The method *BubbleSort* takes a parameter of the array *arr*. It is necessary to set the array for input data in method or predicate functions as not null by writing the condition in the method annotation, as shown in line 2. Otherwise, Dafny emits an error message that the target object might be null in the method. An array stores its value directly in memory, which does not sort the elements on a null array [KL12]. The method requires annotating the *modifies* clause that the method modifies the input array *arr*, which then changes the values in the memory. Lines 6 and 7 initialize the variable *i* as the outer loop iteration and the variable *n* with the value *arr.Length* − 1, where *arr.Length* represents the length of the array *arr*. These initializations ensure the condition  $0 < n - i$ , which decreases the *n* by *i* at each iteration and holds until the array elements are sorted. Declare the variable *j* with 0 for the inner while loop iteration. The inner loop traverses the array and swaps the elements until the greatest element of the array is at the end. Lines 12-17 describe the condition of the inner loop and the body for the swap process. The condition compares the adjacent element if  $arr[j] > arr[j + 1]$ . If the condition holds, the variable *ind1* is initiated with *arr[j]* and *ind2* with *arr[j + 1]*. Then the body swaps the values by storing *ind2* in *arr[j]* and *ind1* in *arr[j + 1]*. This process continues until the condition holds by incrementing the iteration *j* in line 20. After each complete swap process of the inner loop, the outer loop increases the iteration *i*, as shown in line 22.

## 5.1 Verification of Bubble Sort in Dafny

Verifying algorithms in Dafny requires declaring the necessary properties in the predicates. This section proposes implementing three predicates for Bubble Sort to verify that the input array is sorted in order.

### 5.1.1 Predicates of *BubbleSort* in Dafny

**Predicate *arraySorted*** Listing 5.2 illustrates the implementation of the predicate *arraySorted* to guarantee the properties of the sorted array.

```

1 predicate arraySorted(arr:array<int>, low:int, up:int)
2   requires arr != null
3   reads arr
4   {
5     forall p, q :: low <= p <= q <= up && 0 <= p <= q < arr.Length ==> arr[p] <= arr[q]
6   }

```

LISTING 5.2: Predicate *arraySorted* of Bubble Sort in Dafny

Type *array* is mutable, which allows it to change its state during program execution. To specify which objects a predicate depends on, the predicate function uses the *reads* clause. The annotation *reads arr* must be implemented in the predicate that allows reading the array's content *arr*.

The predicate denotes a *forall* quantifier over all pairs of indices *p* and *q* that satisfy the condition that both *p* and *q* are indices within the range of the array's lower bound and upper bound. The predicate checks if the elements at indices *p* and *q* are sorted in ascending order, meaning *arr[p]* is less than or equal to *arr[q]*. The body of the predicate *arraySorted* is valid for the specified condition and meets all pairs of indices, implying the array sorts in ascending order.

However, selecting the correct predicate to verify the sorting algorithm is crucial. A predicate with an incorrect body is possible to accept a sorted array. As an example, the predicate *arraySortedWrong* has the same parameter given as by *arraySorted*, but the quantified expression is specified for all *p* and *q* such as in Listing 5.3

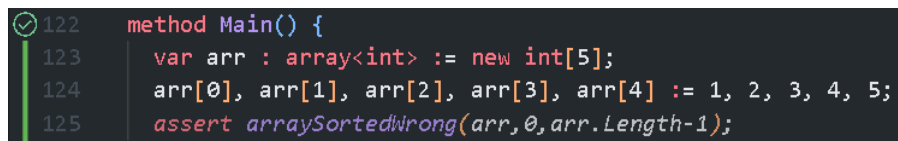
```

1 predicate arraySortedWrong(arr:array<int>, low:int, up:int)
2   reads arr
3   requires arr != null
4   {
5     forall p, q :: low >= p >= q >= up ==> 0 >= p >= q > arr.Length ==> arr[p] <= arr[q]
6   }

```

LISTING 5.3: Predicate *arraySortedWrong* in Dafny

The equation says if the quantified variables *p* and *q* are in the range of *low* and *up* and 0 is greater than equal *p* and *q*. However, they are larger than the length of the array *arr*, which is wrong. This quantifier has a wrong expression to prove the array sorts in ascending order. However, the predicate verifies the algorithm regardless of considering the order of the elements in the array. Using this quantifier results in Dafny failing to capture the intended behavior and allows verification of a sorted array with the wrong predicate, as shown in Figure 5.2.



```

122 method Main() {
123   var arr : array<int> := new int[5];
124   arr[0], arr[1], arr[2], arr[3], arr[4] := 1, 2, 3, 4, 5;
125   assert arraySortedWrong(arr, 0, arr.Length-1);

```

FIGURE 5.2: The predicate *arraySortedWrong* ensures the array *arr* is sorted

**Predicate *bubblesSorted*** The predicate *bubblesSorted* is required to ensure that the algorithm is correct in each step, as shown in Listing 5.4

```

1 predicate bubblesSorted(arr:array?<int>, index:int)
2   requires arr != null
3   reads arr
4   {
5     forall p, q :: p < index < q && 0 <= p < q < arr.Length ==> arr[p] <= arr[q]
6   }

```

LISTING 5.4: Predicate *bubblesSorted* of Bubble Sort in Dafny

The predicate requires that the array *arr*! = *null* and to read the values in the array *arr* annotate *reads arr*. The body has the quantified expression with the variable *p* and *q*, where  $p < index$  and  $index < q$  is less than the length of the *arr*. Then it implies that  $arr[p] \leq arr[q]$ , meaning that the elements in range  $[0, p]$  are smaller than  $(index, q]$ . Then, this ensures that Bubble Sort completes each sorting step correctly.

**Predicate *bubbleStepFinished*** Verifying Bubble Sort in Dafny requires checking if the largest element of the array traversed to the last index with the bubble steps. This expression is defined in the predicate *bubbleStepFinish* as shown in Listing 5.5.

```

1 predicate bubbleStepFinish(arr:array?<int>, up:int)
2   requires arr != null
3   reads arr
4   {
5     forall k :: 0 <= k < up < arr.Length ==> arr[k] <= arr[up]
6   }

```

LISTING 5.5: Predicate *bubbleStepFinished* of Bubble Sort in Dafny

The predicate requires reading the array *arr* and that it is not null. The body has the quantified expression with the variable *k* that has the range of  $[0, up)$  which is less than the length of the *arr*. Then, the array *arr* in index *k* must have less or equal value than *arr* in index *up*. This expression guarantees that when the bubble steps are finished in iteration *up*, the largest element bubbled up to the last position of index *up*. If the predicate holds for each step of the algorithm, then it ensures the array is sorted.

### 5.1.2 Verification of *BubbleSort* in Dafny

Verifying Bubble Sort requires specifying that the return array is in order. It requires ensuring the process by declaring the specifications as in the post-condition and invariants of the loops. The complete code of the method *BubbleSort* is shown in Appendix A.2. Listing 5.6 describes annotations of the pre-conditions and post-conditions of the method *BubbleSort* with its parameter.

```

1 method BubbleSort(arr:array?<int>)
2   requires arr != null
3   ensures arr != null
4   ensures |arr[..]| == old(|arr[..|)
5   ensures arraySorted(arr, 0, arr.Length-1)
6   ensures multiset(arr[..]) == multiset(old(arr[..]))
7   modifies arr
8   {...}
9 }

```

LISTING 5.6: Annotations of *BubbleSort* in Dafny

The method *BubbleSort* takes input array *arr*, which requires the array is not null and modified. Annotate in the post-condition to confirm that the array *arr* is not null after it is sorted and the sequence of *arr* equals the old sequence of *arr* that these sequences have the same length as shown in line 4. The old state references the state where *arr* is not changed, and the definition *arr[..]* is a sequence of the array *arr*. In the next post-condition, the predicate *arraySorted* in Listing 5.2 ensures that the return value is sorted at index  $[0, arr.Length - 1]$ . Verifying the multiset of a sorted array is an important factor in proof techniques. In Dafny, the built-in type *multiset* ensures the same elements of the arrays of the method *BubbleSort* by annotating as *multiset(arr[..]) = multiset(old(arr[..]))*.

The invariants of the outer loop ensure that each sorting step is correct until its condition holds. Listing 5.7 shows the specifications of the loop invariants.

```

1  var i := 0;
2  var n := arr.Length-1;
3  while (0 < n-i)
4    invariant 0 <= i <= n || n == -1
5    invariant arraySorted(arr, n-i, n)
6    invariant bubblesSorted(arr, n-i)
7    invariant multiset(arr[..]) == multiset(old(arr[..]))
8    decreases n-i
9  {...}

```

LISTING 5.7: Outer loop invariants of *BubbleSort* in Dafny

The first invariant ensures the while loop iteration as  $0 \leq i \leq n || n = -1$ . Even though the while condition does not hold for the length of *arr* is 0, and the iteration of  $0 \leq i \leq n$  is correct, the loop does not execute, and it requires for *n* a proper adjustment. The variable *n* becomes  $-1$  when the length of *arr* is 0, which means the range of *n* is  $[-1, arr.Length - 1]$ . Therefore, adjust the range of the *n* annotating  $n = -1$ . This change ensures that the invariant verifies the variable *n* by the loop entry. Next, invariant describes the annotation of the predicate *arraySorted* in Listing 5.2, which checks the sorted array *arr* in the  $[n - i, n]$  range. However, this predicate cannot verify the algorithm as the  $n - i$  can be any value of  $[0, arr.Length - 1]$ . It requires to set when index  $n - i$  ensures the sorted array. This can be then annotated by the predicate *bubblesSorted(arr, n - i)* in Listing 5.4. For all *p, q* has the range of  $0 \leq p < n - i < q < arr.Length$ . The elements in index *p* must be smaller than in index *q*. Then, set, the outer loop decreases by the iteration of  $n - i$ , which ensures the loop termination.

The invariants of the inner loop ensure the swapping process that swaps until the greatest number of the array is at the end of the array index. The annotations of the invariants are shown in Listing 5.8.

```

1  var j := 0;
2  while (j < n-i)
3    invariant 0 <= j <= n-i
4    invariant arraySorted(arr, n-i, n)
5    invariant bubbleStepFinish(arr, j)
6    invariant multiset(arr[..]) == multiset(old(arr[..]))
7    decreases n-j
8  {...}

```

LISTING 5.8: Inner loop invariants of *BubbleSort* in Dafny

The swapping process begins with the iteration  $j$ , which increases by at most  $0 \leq j \leq n - i$ . The predicates  $\text{arraySorted}(arr, n - i, n)$  and  $\text{bubblesSorted}(arr, n - i)$  are annotated to ensure Bubble Sort. Nevertheless, these conditions do not verify Bubble Sort as the inner loop has the same problem with the iteration  $j$ . The verifier does not know when to accept iteration  $j$  to have the greatest element in  $res$ . Therefore, it requires the annotation of the predicate  $\text{bubbleStepFinish}$  in Listing 5.5, which takes the parameters of  $res$  and  $j$ . The body of the predicate ensures that for all  $k$ , that is  $0 \leq k < j < res.Length$  implies that  $res[j]$  contains the greatest element.

The correctness of the method *BubbleSort* is checked by calling the method in *Main* as shown in Figure 5.3.



```

122 method Main() {
123     var arr : array<int> := new int[5];
124     arr[0], arr[1], arr[2], arr[3], arr[4] := 3, 2, 5, 1, 4;
125     var res := BubbleSort(arr);
126     assert arraySorted(res, 0, res.Length-1);
127 }

```

FIGURE 5.3: Dafny verifies Bubble Sort by calling the method *BubbleSort*

In the method, *Main*, the array *arr* is declared with a length of 5, which assigns the values of 3, 2, 1, 5 and 4. The variable *res* calls the method *BubbleSort* with its parameter *arr*. Then Dafny verifies with the predicate *arraySorted* that the array *res* is sorted in the range of  $[0, res.Length - 1]$ .

## 5.2 Verification of Bubble Sort in Caesar

This section demonstrates the verification of Bubble Sort with the built-in type *Lists* and the datatype *List* in Caesar using the same verification annotations from Dafny.

### 5.2.1 Verification of *BubbleSort* in Built-in *Lists* in Caesar

The verification of Bubble Sort with the built-in type *Lists* in Caesar has the same verification annotations from Dafny. The quantified expressions of the predicates in Dafny are annotated directly as post-conditions or invariants. The complete code of the verified procedure *BubbleSort* is shown in A.7. Listing 5.9 describes the annotations that the procedure *BubbleSort* must hold after executing its return list *res* with its parameters.

```

1 proc BubbleSort(arr: []Int) -> (res: []Int)
2   pre ? (0 <= len(arr))
3   post ? (0 <= len(res)
4     && (len(arr) == len(res))
5     && (forall p:UInt, q:UInt. ((0 <= p) && (p < q) && (q < len(res))) ==> (select(res, p) <=
6       select(res, q))))

```

LISTING 5.9: Annotations of *BubbleSort* in Caesar



The procedure *BubbleSort* takes the integer list *arr* as a parameter and returns the list *res*, which sorts the elements after the execution. The pre-condition for the procedure *BubbleSort* sets the length of *arr* as  $0 \leq \text{len}(\text{arr})$  so that the list *arr* can be empty or contain elements. The post-condition ensures that the length of the return list *res* is  $0 \leq \text{len}(\text{res})$  and the length of *arr* is equal to the length of *res*, which ensures that the list did not change its length during the process. Line 5 describes the procedure taking the same quantified expression as 5.2. When the quantified variables *p* and *q*, which  $p < q$  and has the range of  $[0, \text{len}(\text{res}))$  then  $\text{select}(\text{res}, p) \leq \text{select}(\text{res}, q)$  is valid. This expression verifies that the return list *res* elements are in order.

The procedure *BubbleSort* requires an outer loop and inner loop to ensure the process of Bubble Sort. The body and the invariants are the same as in Dafny, as shown in Appendix A.2. The invariant  $I_3$  defines the correctness of the outer loop that guarantees the properties of Bubble Sort, which is written as:

$$\begin{aligned} I_3 : & 0 \leq i \leq n \wedge \\ & \text{len}(\text{arr}) = \text{len}(\text{res}) \wedge \\ & \forall p, q : \mathbb{N} :: n - i \leq p < q \leq n < \text{len}(\text{res}) \implies \text{select}(\text{res}, p) \leq \text{select}(\text{res}, q) \wedge \\ & \forall p, q : \mathbb{N} :: 0 \leq p < n - i < q < \text{len}(\text{res}) \implies \text{select}(\text{res}, p) \leq \text{select}(\text{res}, q) \end{aligned}$$

The iteration *i* is initialized to 0 and has the index of  $0 \leq i \leq n$ , where *n* is the  $\text{len}(\text{res}) - 1$ . Caesar does not require to adjust  $n = -1$  as it sets *n* automatically to 0 when the value gets less than 0. In the invariant, the expression  $\text{len}(\text{arr}) = \text{len}(\text{res})$  guarantees that the length of *res* did not change. The first quantified expression is from the predicate *arraySorted* in Listing 5.2 and the second expression from the predicate *bubblesSorted* in Listing 5.4

The inner loop invariant is defined in  $I_4$ , which is written as:

$$\begin{aligned} I_4 : & 0 \leq j \leq n - i \wedge \\ & \text{len}(\text{arr}) = \text{len}(\text{res}) \wedge \\ & \forall p, q : \mathbb{N} :: n - i \leq p < q \leq n < \text{len}(\text{res}) \implies \text{select}(\text{res}, p) \leq \text{select}(\text{res}, q) \wedge \\ & \forall p, q : \mathbb{N} :: 0 \leq p < n - i < q < \text{len}(\text{res}) \implies \text{select}(\text{res}, p) \leq \text{select}(\text{res}, q) \\ & \forall k : \mathbb{N} : 0 \leq k \leq j \implies (\text{select}(\text{res}, k) \leq \text{select}(\text{res}, j)) \end{aligned}$$

The iteration *j* is initialized as 0 and has the range of  $[0, n - i]$ , and also, the length of *arr* and *res* must be equal. The invariant  $I_4$  uses quantified expressions of the implemented predicates in Listing 5.2, 5.4 and 5.5.

Figure 5.4 demonstrates calling the procedure *BubbleSort* in *Main*. The list *arr* is initialized with the length 5 and the elements as in Figure 5.1. The list *res* is the sorted list from the call of the procedure *BubbleSort* with *arr*. Compiling this code, Caesar verifies that the elements of *res* are in order.

```

proc Main () -> ()
{
  var arr: []Int;
  assume ? (len(arr) == 5);
  arr = store(arr, 0, 3);
  arr = store(arr, 1, 2);
  arr = store(arr, 2, 5);
  arr = store(arr, 3, 1);
  arr = store(arr, 4, 4);
  var res: []Int = BubbleSort(arr)
  assert ? (
    forall p: UInt, q: UInt. ((0 <= p) && (p < q) && (q < len(res))) ==>
    (select(res, p) <= select(res, q))
  )
  assert ? (len(res) == 5)
}

```

FIGURE 5.4: The procedure *Main* calls *BubbleSort*

## 5.2.2 Verification of *BubbleSort* in Datatype *List* in Caesar

Caesar verifies Bubble Sort in datatype *List* with the same annotations from built-in *Lists* from the previous section. The verifier uses the same datatype *List* from the Left Pad function in Section 4.3.1.

## 5.3 Verification Failure of Multiset in Caesar

Caesar does not have built-in type *multiset* as in Dafny to prove the multiset of Bubble Sort, and therefore, it requires implementing a function to verify the multiset. Verifying the multiset of Bubble Sort using an implemented function in Dafny is proposed in [Fre16], [SD13]. The proposed function returns the multiplicity, which is the incident of the counted element in the multiset [Bli89]. The multiset of Bubble Sort is verified by comparing the sum of all the multiplicity of the elements in the input array and the sorted array.

This section introduces the implementation and verification of the proposed function to compute the multiplicity in Dafny with the type *array*. Then, it demonstrates the problem of implementing the function with the built-in type *Lists* and the datatype *List* in Caesar.

### 5.3.1 Verification of *multiplicity* in Dafny

The multiplicity of an element in the array is computed by comparing the input element with the element of the array at each index. If the compared values are equal, the function increases its return value by one. Listing 5.10 describes the implementation and verification of the function *multiplicity* that returns the multiplicity of an array element.

```

1 function multiplicity(n:nat, arr:array?<int>, el:int): nat
2   requires arr != null
3   ensures (0 < n <= arr.Length) && (arr[n-1] != el) ==>
4     multiplicity(n, arr, el) == multiplicity(n-1, arr, el)
5   ensures (0 < n <= arr.Length) && (arr[n-1] == el) ==>
6     multiplicity(n, arr, el) == 1 + multiplicity(n-1, arr, el)
7   ensures (0 == n) ==> multiplicity(n, arr, el) == 0

```

```

8   reads arr
9   {
10  if (0 < n <= arr.Length) then
11    if (arr[n-1] != el) then
12      multiplicity(n-1, arr, el)
13    else if (arr[n-1] == el) then
14      1 + multiplicity(n-1, arr, el)
15    else 0
16  else 0
17 }

```

LISTING 5.10: Multiplicity in Dafny

Line 10-16 describes the body of the function. The multiplicity of the element  $el$  is counted if the condition holds for  $0 < n \leq arr.Length$ . The condition for  $n$  is set as  $0 < n \leq arr.Length$  so that  $n$  satisfies the natural number by decreasing its value in a recursive call. If the condition  $(arr[n-1] \neq el)$  holds, the function computes recursively decreasing the  $n$  as  $multiplicity(n-1, arr, el)$ . The index of  $arr$  is defined as  $n-1$  to ensure the index is not out of the range of  $[0, arr.Length)$ . If the condition holds for  $(arr[n-1] = el)$ , then the function increases its value by one and computes recursively, decreasing the  $n$  as  $multiplicity(n-1, arr, e)$ . In other conditions, it does nothing and returns 0, as shown in lines 13-14. In the annotation, the function requires that the input  $arr$  is not null. Also, it requires that the function reads the elements of  $arr$ , as shown in line 6. Line 3-5 annotates each condition of the body, which then verifies the multiplicity of the element  $el$  in the array.

### 5.3.2 Implementation Failure of *multiplicity* in Built-in Lists in Caesar

Caesar fails to implement and ensure the function to compute recursively and execute the multiplicity of an element of the multiset in built-in type *Lists*. Listing 5.11 describes the function *multiplicity* in domain *Mul* with its function and axioms that have the same properties of Dafny function 5.10.

```

1  domain Mul {
2    func multiplicity(n:UInt, arr:[]Int, el:Int): UInt
3    axiom c_n forall n:UInt, arr:[]Int, el:Int. (0 == n) ==> (multiplicity(n, arr, el) == 0)
4    axiom c_e forall n:UInt, arr:[]Int, el:Int. ((0 < n) && (n <= len(arr)) && (select(arr, n-1)
5      != el)) ==> (multiplicity(n, arr, el) == multiplicity(n-1, arr, el))
6    axiom c_ne forall n:UInt, arr:[]Int, el:Int. ((0 < n) && (n <= len(arr)) && (select(arr, n-1)
7      == el)) ==> (multiplicity(n, arr, el) == 1 + multiplicity(n-1, arr, el))
8  }

```

LISTING 5.11: Multiplicity in built-in type *Lists* in Caesar

The first axiom describes that if the index  $n$  is out of the range, the function finishes traversing the elements and returns the value 0. On the one hand, the second axiom describes that while  $n$  is in the range  $(0, len(arr)]$  and the list contains the element  $el$  in index  $n-1$ , then it returns recursively decreasing  $n$  and adding one as  $multiplicity(n-1, arr, el)$ . On the other hand, The third axiom describes that while  $n$  is in the range  $(0, len(arr)]$  and the list does not contain the element  $el$  in index  $n-1$ , then it returns recursively decreasing  $n$ , as  $multiplicity(n-1, arr, el)$ .

After compiling the code, Caesar can not decide if the function is satisfied or unsatisfied. Also, tracking the triggered variables or debugging is impossible as it gives no information about the errors in quantified expressions.

### 5.3.3 Implementation Failure of *multiplicity* in Datatype *List* in Caesar

Caesar accepts the axioms of the *multiplicity* function in datatype *List* that compares the element and recursively calls the rest of the list. Listing 5.12 describes the implementation of the function *multiplicity* with its axiom in domain *List*.

```

1 domain List {
2   ...
3   func multiplicity(ls:List, el:Int): UInt
4     axiom count_n forall ls:List, el:Int. (is_null(ls)) ==> (multiplicity(ls, el) == 0)
5     axiom count_l forall ls:List, h:Int, t:List, el:Int.
6       (is_list(ls) && cons(h, t) && (h == el)) ==> (multiplicity(ls, el) == (1 + multiplicity(t, el)))
7     axiom count_l_n forall ls:List, h:Int, t:List, el:Int.
8       (is_list(ls) && cons(h, t) && (h != el)) ==> (multiplicity(ls, el) == (0 + multiplicity(t, el)))
9   ...
10 }
```

LISTING 5.12: Multiplicity in datatype *List* in Caesar

The function *multiplicity* takes parameters of the list *ls* and the element *el* to check the number of the elements. The first axiom describes that if the list is empty, the function returns 0, as it contains no elements. The second axiom describes all the list *ls* that has the form of *cons(h, t)*, and if *h = el*, then the function increases its result by one and returns recursively its sublist *t* as  $1 + \text{multiplicity}(t, el)$ . The third axiom describes if the list *ls* when  $h \neq el$  returns recursively *multiplicity(t, el)*.

Even though Caesar accepts compiling the code, calling the function *multiplicity* could not verify the sum of the element for all possible lists, as shown in Figure 5.5.

```

var ls : List = cons(2, cons(4, cons(4, cons(4, cons(3, null())))))
assert ? (forall k: UInt. (multiplicity(ls, 4) == 3))
var ls : List = cons(2, cons(4, cons(2, cons(4, cons(3, null())))))
assert ? (forall k: UInt. (multiplicity(ls, 4) == 2))
```

FIGURE 5.5: Examples of calling the function *multiplicity*

Caesar could verify by calling the function *multiplicity* that list *ls* has the multiplicity of the element 4 of 3. However, the verifier could not verify that the list has the multiplicity of 2 of the element 4.

**Axiom Profiler** The Axiom Profiler is used to debug the quantified expressions that illustrate the quantifier instantiations, which are instances created by the pattern of the quantified expressions[BMS19]. The tool takes a log file generated by the Z3 and runs the file. It illustrates a graph of the quantifier instantiation to show the problem of the quantified expression. The tool does not

support the log data of Caesar. It requires translating the function *multiplicity* in Z3 with the same properties as Listing 5.12. Listing 5.13 describes the translation of the function *multiplicity* in Z3.

```

1 (declare-datatypes (T) ((Lst nil (cons (hd T) (tl Lst)))))
2 (declare-const ls (Lst Int))
3 (declare-const el Int)
4 (declare-fun Mul ((Lst Int) Int) Int)
5 (assert (forall ((ls (Lst Int)) (el Int)) (=> (= ls nil) (= (Mul ls el) 0))))
6 (assert (forall ((ls (Lst Int)) (el Int) (h Int) (t (Lst Int)))
7   (=> (and (= ls (cons h t)) (= h el)) (= (Mul ls el) (+ 1 (Mul t el))))))
8 (assert (forall ((ls (Lst Int)) (el Int) (h Int) (t (Lst Int)))
9   (=> (and (= ls (cons h t)) (not(= h el)) (= (Mul ls el) (Mul t el))))))
10 (check-sat)

```

LISTING 5.13: Multiplicity in datatype *Lst* in Z3

Lines 1-4 illustrate the declaration of the datatype *Lst* and the function *Mul* with its constants. Lines 5-9 have the same quantified expression of the axioms from Listing 5.12.

Obtaining a log from this Z3 code is crucial, as Z3 keeps creating logs that increase its log data to gigabytes. By setting the time limit, it could obtain a small data. However, loading the log file in Axiom Profiler could not produce the graph and closed the program automatically.



## Chapter 6

# Comparison Using Binary Search Tree

This Chapter investigates the verification of the Binary Search Tree in Dafny and Caesar, as it represents the basic algorithm of sorted binary trees. A Binary Search Tree is a sorted tree that starts with a root and its value with two sub-trees, the left sub-tree and the right sub-tree[Hib62]. The values on the left sub-tree are always smaller than its node, and those on the right sub-tree are greater than its node value. The Binary Search Tree allows one to add elements with Insertion or remove them with Deletion. On average, these operations have a time complexity of  $O(\log(n))$ , and in the worst case, it takes  $O(n)$ . Figure 6.1 demonstrates an example of a Binary Search Tree.

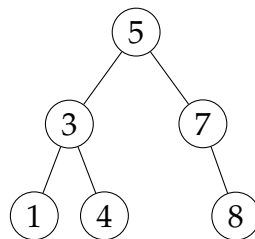


FIGURE 6.1: Example of a Binary Search Tree

The root has a value of 5 with its sub-trees, where its left sub-tree has a value of 3, which is smaller than its root, and the value of the right sub-tree, 7, is greater than its root. The node 3 has sub-trees containing the values 1 and 4, and the node 7 has a right sub-tree with a value 8.

### 6.1 Verification of Binary Search Tree in Dafny

Verifying the Binary Search Tree in Dafny, the verifier requires implementing the properties of the Binary Search Tree. First, declare a datatype *Tree* that is null, or it can be a node containing a left tree, the value of the node, and its right tree as shown in Listing 6.1.

```
1 datatype Tree = Null | Node(left:Tree, value:int, right:Tree)
```

LISTING 6.1: Datatype *Tree* in Dafny

**Function** *treeSet* In order to obtain the tree elements, use the type *set*. The function *treeSet* in Listing 6.2 defines the set of a tree.

```

1 function treeSet(t:Tree): set<int>
2 {
3   if(t.Null?) then {}
4   else if(t.Node?) then
5     var l:Tree := t.left;
6     var v:int := t.value;
7     var r:Tree := t.right;
8     treeSet(l) + {v} + treeSet(r)
9   else {}
10 }
```

LISTING 6.2: Tree set in Dafny

The function takes a tree value  $t$  as a parameter and returns  $\text{set}\langle\text{int}\rangle$  containing all the integer values of the tree. The function is used to define the properties of Binary Search Tree. In case the condition of the function is  $t.Null?$ , then the function returns that the set is empty. If the tree  $t$  contains nodes, the function recursively traverses the sub-tree  $l$  and  $r$  of the input tree  $t$  and adds the value  $v$  of the node to the set.

**Predicate** *BST* The predicate *BST* declares the properties of the Binary Search Tree. The predicate specification consists of the function *treeSet*, which ensures the left sub-tree contains a smaller value than the right sub-tree and the right sub-tree contains a greater value than its left sub-tree. Listing 6.3 demonstrates the implementation of the predicate *BST*.

```

1 predicate BST(t:Tree)
2 {
3   if(t.Null?) then
4     true
5   else if (t.Node?) then
6     var l:Tree := t.left;
7     var v:int := t.value;
8     var r:Tree := t.right;
9     (forall z:: z in treeSet(l) ==> z < v) &&
10    (forall z:: z in treeSet(r) ==> v < z) && BST(l) && BST(r)
11   else true
12 }
```

LISTING 6.3: Predicate *BST* in Dafny

If the tree is empty, then it returns that it is true. Otherwise, the predicate checks two conditions using *forall* quantifiers. The first quantifier ensures that all elements of  $z$  obtained from the set of the sub-tree  $l$  are smaller than the value  $v$  of the current node. The second quantifier ensures that all elements of  $z$  obtained from the set of the right sub-tree  $r$  are greater than the value  $v$  of the current node. The predicate calls recursively to ensure the left and right sub-trees are valid Binary Search Trees.

Figure 6.2 illustrates two examples calling the predicate *BST* that ensures the properties of a Binary Search Tree. In the first example, Dafny ensures that  $t$  is a Binary Search Tree. However, Dafny fails to confirm that  $t2$  is a Binary Search Tree since the left sub-tree has the node 5, which is greater than its root.



```

var t := Node(Node(Null, 2, Null), 4, Node(Null, 8, Null));
assert BST(t);

var t2 := Node(Node(Null, 5, Null), 4, Node(Null, 8, Null));
assert BST(t2);

```

FIGURE 6.2: The predicate *BST* has the properties of a Binary Search Tree

## 6.2 Verification of Insertion in Dafny

The insert function constructs the Binary Search Tree by adding a new node with a given value. This construction proceeds in two cases. A new node is added as the root when the tree is empty. The tree adds value when it contains elements by traversing its sub-trees in the correct order. Figure 6.3 demonstrates that the value six is added to the given tree in the correct position.

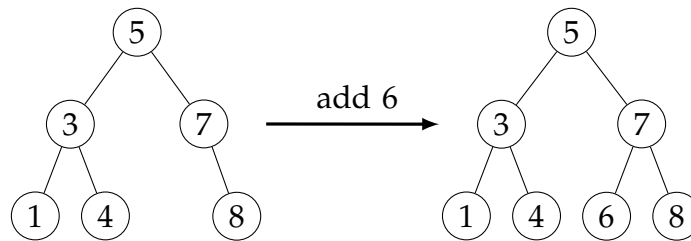


FIGURE 6.3: Add value 6 in Binary Search Tree

The insertion function is implemented as a method in Dafny according to Listing 6.4.

```

1 method Insert(t:Tree, val:int) returns (res:Tree)
2 {
3   if (t.Null?) {
4     res := Node(Null, val, Null);
5   }
6   else if (t.Node?) {
7     var l:Tree := t.left;
8     var v:int := t.value;
9     var r:Tree := t.right;
10    if (val < v) {
11      var ll:Tree := Insert(l, val);
12      res := (Node(ll, v, r));
13    }
14    else if (val > v) {
15      var rr:Tree := Insert(r, val);
16      res := (Node(l, v, rr));
17    }
18    else {
19      res := t;
20    }
21  }
22 }

```

LISTING 6.4: Implementation of Insertion in Dafny

The method *Insert* takes parameters from the tree *t*, and an integer *val* is added to the tree *res*. If the tree contains no values, the root node will be

the value described in lines 3 and 4. Line 6-20 shows that in case  $t$  contains elements, the method adds  $val$  under three conditions.

If the condition  $val < v$  holds, the function moves the added value  $val$  to the left tree by recursively calling  $Insert(l, val)$ . Then, the method creates a new left tree  $ll$  containing the value  $val$ , and  $res$  returns a Node with  $ll$  as  $Node(ll, v, r)$ . Suppose  $val > v$ , the value  $val$  is added to the right sub-tree of  $t$ , which is described as  $rr$ . Then return  $res$ , the tree of  $Node(l, v, rr)$ . In the third condition,  $res$  is initialized to  $t$ , so the method returns the original tree.

### 6.2.1 Verification of *Insert* in Dafny

Verifying the insertion function requires specifying that the returned tree is a Binary Search Tree that contains the same nodes from the previous tree with the added node. Appendix A.3 provides the complete code of *Insert*. Listing 6.5 demonstrates the annotations of the method *Insert*.

```

1  method Insert(t:Tree, val:int) returns (res:Tree)
2    requires BST(t)
3    ensures forall x :: x in treeSet(t) && x < val ==> x in treeSet(res)
4    ensures forall x :: x in treeSet(t) && x > val ==> x in treeSet(res)
5    ensures treeSet(res) == treeSet(t) + {val}
6    ensures BST(res)
7    decreases t
8    {...}

```

LISTING 6.5: Annotations of *Insert* in Dafny

The method *Insert* takes an input of the tree  $t$  with properties of a Binary Search Tree and returns a Binary Search Tree  $res$  annotate *requires*  $BST(t)$  and *ensures*  $BST(res)$  as on lines 2 and 6. The correct position of the added node is verified by the quantified expressions on lines 3 and 4. These expressions check that if all the elements  $x$  in the set of  $t$  are  $x < val$  or  $x > val$ , then  $res$  must contain  $x$ .

The method *Insert* requires that the body of each if condition returns the property of a Binary Search Tree. As shown in Listing 6.4, the method returns  $res = Node(Null, val, Null)$  if  $t$  is empty. Otherwise, if the tree contains nodes, the return value of the third condition equals  $t$ . These return values ensure that the return tree  $res$  is a Binary Search Tree. The annotation verifies the other conditions that recursively call the method, as shown in line 5, which specifies that the set of the tree changes after the method *Insert* returns  $res$ . At last, annotate *decreases*  $t$  to prove the method *Insert* computes recursively and terminates after the call.

## 6.3 Verification of Deletion in Dafny

The delete function deletes the node in a Binary Search Tree by removing the desired node with a given value. This construction proceeds in three cases when the tree contains elements, as shown in the following Figures.

In the first case, Binary Search Tree deletes the node when it has no sub-trees. In Figure 6.4, the node 6 is deleted after assigning the node as null.

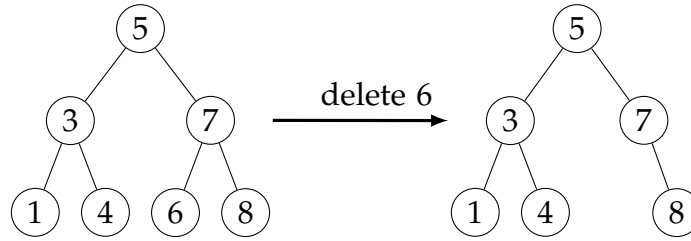


FIGURE 6.4: Case 1: Delete node with no sub-trees

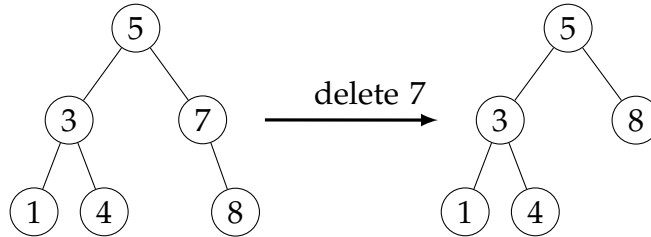


FIGURE 6.5: Case 2: Delete node with one sub-tree

In the second case, Binary Search Tree deletes the node when it has one sub-tree. The node has a sub-tree on the left or the right side. In Figure 6.5, the node 6 is deleted after assigning the node with its right sub-tree.

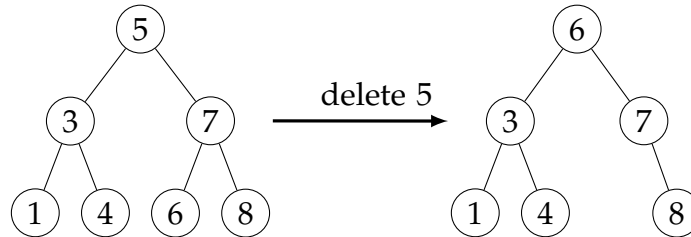


FIGURE 6.6: Case 3: Delete node with both sub-trees

In the last case, the tree deletes a node with both sub-trees. The node finds the minimum value in its right sub-tree, substituting it with its value. Figure 6.6 illustrates the Deletion of the root 5, which substitutes with node 6.

The last case of Deletion requires a function that finds a minimum value. Listing 6.6 describes the function *getMin*.

```

1 function getMin(t:Tree) : int
2 {
3   if(t.Node?) then
4     var l :Tree := t.left;
5     var v :int := t.value;
6     var r :Tree := t.right;
7     if (l != Null) then getMin(l)
8     else v
9   else 0
10 }
```

LISTING 6.6: Implementation of *getMin* in Dafny

The function *getMin* takes a parameter *t* and returns an integer value. If *t* and its left sub-tree *l* contain nodes, it calls recursively with *l*. Otherwise, it returns the value *v* of its node. If *t* is empty, then it returns 0.

Listing 6.7 shows the implementation of the Deletion in Binary Search Tree as method *Delete*.

```

1  method Delete(t:Tree, val:int) returns (res:Tree)
2  {
3      if (t.Null?) {
4          res := t;
5      }
6      else if (t.Node?) {
7          var l :Tree := t.left;
8          var v :int := t.value;
9          var r :Tree := t.right;
10         if (val < v) {
11             var ll := Delete(l, val);
12             res := Node(ll, v, r);
13         }
14         else if (val > v) {
15             var rr := Delete(r, val);
16             res := Node(l, v, rr);
17         }
18         else if (val == v) {
19             if (l == Null && r == Null) {res := Null;}
20             else if (l == Null) {res := r;}
21             else if (r == Null) {res := l;}
22             else {
23                 var minV := getMin(r);
24                 var rm := Delete(r, minV);
25                 res := Node(l, minV, rm);
26             }
27         }
28     }
29 }

```

LISTING 6.7: Implementation of Deletion in Dafny

The method *Delete* takes parameters of the tree  $t$  and integer value  $val$ , which returns a tree  $res$ . When  $t$  is empty, the method removes no elements and returns an empty tree. Lines 10-17 describe that if the desired value  $val$  differs from  $v$ , the method recursively calls its sub-trees. If  $val = v$ , the method processes Deletion in three cases, as shown in lines 18-25. In the first case, if the node has no sub-tree, it returns *Null*, which removes the desired value from  $res$ . In the second case, the node contains one sub-tree. The method deletes the desired node by returning its sub-tree, as shown in lines 20 and 21. In the last case, both sub-trees contain the nonempty tree. The method calls the function *getMin* and initializes its value in  $minV$ . The element  $minV$  is deleted in its right sub-tree and substituted with  $v$ .

### 6.3.1 Verification of *Delete* in Dafny

Verifying the Deletion in a Binary Search Tree requires specifying that the returned tree does not contain the desired node from the previous tree. Appendix A.4 provides the complete code of verified deletion function in Dafny. Listing 1.4 demonstrates the annotations of the method *Delete*.

```

1  method Delete(t:Tree, val:int) returns (res:Tree)
2      requires BST(t)
3      ensures forall x :: x in treeSet(t) && x < val ==> x in treeSet(res)
4      ensures forall x :: x in treeSet(t) && x > val ==> x in treeSet(res)
5      ensures treeSet(res) == treeSet(t) - {val}
6      ensures BST(res)
7      decreases t

```

8 {...}

LISTING 6.8: Annotations of *Delete* in Dafny

Annotate that the method requires the input tree to be a Binary Search Tree as *requiresBST(t)* and ensures the result tree is a Binary Search Tree as *ensuresBST(res)* as shown on lines 2 and 6. The correct position of the deleted node is verified by the quantified expressions on lines 3 and 4. These expressions check that if all the elements  $x$  in the set of  $t$  are  $x < val$  or  $x > val$ , then  $res$  must contain  $x$ . The method *Delete* requires that the body of each if condition returns the property of a Binary Search Tree, as shown in Listing 6.7. In line 5, the annotation verifies that the tree  $res$  equals the tree  $t$  without the value  $val$ , which ensures the return value of if conditions with the recursive call of the method. At last, annotate *decreases t* to prove the method *Delete* computes recursively and terminates.

The function *getMin* aims to find and return the minimum value of the Binary Search Tree. As shown in Listing 6.9, the annotations verify the return value after method *Delete* calls the function.

```

1 function getMin(t:Tree): int
2   requires BST(t)
3   requires t != Null
4   ensures getMin(t) in treeSet(t)
5   ensures forall x:int :: (x in treeSet(t) && x != getMin(t)) ==> getMin(t) < x
6   decreases t
7 {...}

```

LISTING 6.9: Annotations of *getMin* in Dafny

The function requires that the input tree is a Binary Search Tree and is not empty, annotated as  $t! = \text{Null}$  in the pre-condition. The problem is that when the function computes an empty tree, it fails to ensure that the tree set contains its minimum value, as it returns 0. Nevertheless, an empty tree cannot contain any value. First, the function ensures that the value *getMin(t)* is in the tree set  $t$ . Line 5 describes that *getMin(t)* is the minimum value from all  $x$  in the tree set  $t$ .

## 6.4 Verification of Binary Search Tree in Caesar

This section translates the implementation of the Binary Search Tree and verifies the Insertion and Deletion of Dafny into Caesar. Caesar does not support built-in type *set*, so it requires a function to substitute and use to verify the properties of a Binary Search Tree. The function called *contains* is implemented, which does not store elements the same as type *set* but manages to define a Binary Search Tree and its properties.

**domain *Tree*** Caesar uses the domain to define datatype *Tree* and specifies the properties with the functions and their axioms. Listing 6.10 defines the implementation of domain *Tree*.

```

1 domain Tree {
2   func null():Tree
3   func node(left:Tree, val:Int, right:Tree): Tree

```

```

5  func get_value(t:Tree): Int
6  axiom g_v forall l:Tree, v:Int, r:Tree. get_value(node(l, v, r)) == v

8  func get_left(t:Tree): Tree
9  axiom g_l forall l:Tree, v:Int, r:Tree. get_left(node(l, v, r)) == l

11 func get_right(t:Tree): Tree
12 axiom g_r forall l:Tree, v:Int, r:Tree. get_right(node(l, v, r)) == r

14 func is_null(t:Tree): Bool
15 axiom i_n forall t:Tree. is_null(t) == (t == null())

17 func is_tree(t:Tree): Bool
18 axiom i_t forall t:Tree, l:Tree, v:Int, r:Tree. (
19 ((get_left(t) == l) && (get_right(t) == r) && (get_value(t) == v)) ==>
20 (is_tree(t) == (t == node(l, v, r))))
21 )

23 axiom tree_or_null forall t:Tree. (is_null(t) || is_tree(t))

```

LISTING 6.10: Domain of the datatype *Tree* in Caesar

The tree is empty or contains a node with sub-tree *left*, integer value *val*, and sub-tree *right*. Lines 5 and 6 describe function *get\_value* that returns the value of the node.

The datatype *Tree* can only be null or contain nodes.

**func contains** Caesar does not have built-in type *set*. Therefore, the function *contains* computes the properties of a Binary Search Tree. Listing 6.11 describes the function *contains* with its axioms.

```

1  domain Tree {
2  ...
3  func contains(t:Tree, val:Int) :Bool
4  axiom ct_n forall t:Tree, v:Int. is_null(t) ==> (contains(t, v) == false)
5  axiom ct_t_l forall t:Tree, l:Tree, v:Int, r:Tree, val:Int. (
6  is_tree(t) && (t == node(l, v, r)) && (l == get_left(t)) && (r == get_right(t)) && (val < v))
7  ==> (contains(t, val) == contains(l, val))
8  axiom ct_t_g forall t:Tree, l:Tree, v:Int, r:Tree, val:Int. (
9  is_tree(t) && (t == node(l, v, r)) && (l == get_left(t)) && (r == get_right(t)) && (val > v))
10 ==> (contains(t, val) == contains(r, val))
11 axiom ct_t_v forall t:Tree, l:Tree, v:Int, r:Tree, val:Int. (
12 is_tree(t) && (t == node(l, v, r)) && (l == get_left(t)) && (r == get_right(t)) && (val == v))
13 ==> (contains(t, val) == true)
14 ...
15 }

```

LISTING 6.11: Implementation of *contains* in Caesar

The function takes two parameters: the tree *t* and integer value *val*. The first axiom describes the function as false if the tree is empty. The second and third axiom in lines 3-8 search through the nodes to check if the sub-trees contain the value *val*. If *val* is smaller than the value of the tree, search on the left tree of *t*. If *val* is larger than the value of the tree, then search on the right tree of *t*. The last axiom expresses that if the *val* = *v*, the function returns *t* containing the value *val*.

**func BST** The properties of the Binary Search Tree are implemented as the function *BST*. Define function *BST* with its properties in domain *Tree*, as shown in Listing 6.12.

```

2  domain Tree {
3  ...
4  func BST(t:Tree): Bool
5  axiom b_n forall t:Tree, min:Int. is_null(t) ==> (BST(t) == true)
6  axiom b_t forall t:Tree, l:Tree, v:Int, r:Tree. (
7    is_tree(t) && (l == get_left(t)) && (v == get_value(t)) &&
8    (r == get_right(t)) && (t == node(l,v,r))) ==>
9    (BST(t) == (BST(get_left(t)) && BST(get_right(t)) &&
10     (forall z:Int. contains(r, z) ==> (v < z)) && (forall z:Int. contains(l, z) ==> (v > z))))
11 )
12 ...
13 }

```

LISTING 6.12: Binary Search Tree in datatype *Tree* in Caesar

The function *BST* takes a parameter of *t* and returns a boolean type. The axioms have the same expression as in Listing 6.3, defined with the function *contains* in Listing 6.11 that ensures the inorder of the binary tree. The first quantified expression says that for all elements of *z* contained in *r*, *v* is less than *z*. The second expression ensures that all elements of *z* are contained in *l*, then *v* is greater than *z*.

### 6.4.1 Verification of *Insert* in Caesar

The insert function in Caesar has the same implementation as in Dafny in Listing 6.4. The complete code of verification of the procedure *Insert* is shown in A.8. Listing 6.13 provides the necessary annotations of procedure *Insert*.

```

1  proc Insert(t:Tree, val:Int) -> (res:Tree)
2  pre ? (BST(t))
3  post ? (
4    (forall x:Int. ((contains(t, x) && (x < val)) ==> contains(res, x))) &&
5    (forall x:Int. ((contains(t, x) && (x > val)) ==> contains(res, x))) &&
6    (forall v:Int. (contains(res, v) == contains(t,v) || (v == val))) &&
7    BST(res)
8  ) {...}

```

LISTING 6.13: Annotations of *Insert* in Caesar

The procedure defines the input tree *t*, and the output tree *res* is a Binary Search Tree. The annotation *BST(res)* is verified by specifying the quantified expressions in lines 4-6. The first and second quantified expressions ensure *res* did not lose any elements during the insertion process and that the added *val* is in the correct position. The third expression ensures *res* contains the same values in *t* with additional value *val*.

### 6.4.2 Verification of *Delete* in Caesar

In Casear, the delete function is implemented as in Dafny in Listing 6.7. For the verification, the procedure *Delete* requires the annotations of its pre-condition and post-condition. The complete code of verification of procedure *Delete* is shown in A.9. Listing 6.14 provides the necessary annotations of procedure *Delete*.

```

1  proc Delete(t:Tree, val:Int) -> (res:Tree)
2  pre ? (BST(t))
3  post ? (
4    (forall x:Int. ((contains(t, x) && (x < val)) ==> contains(res, x))) &&
5    (forall x:Int. ((contains(t, x) && (x > val)) ==> contains(res, x))) &&

```

```

6   (forall v:Int. (contains(res, v) == (contains(t,v) && (v != val)))) &&
7   BST(res)
8   )
9   {

```

LISTING 6.14: Annotations of *Delete* in Caesar

The procedure defines the input tree  $t$ , and the output tree  $res$  is a Binary Search Tree. The annotation  $BST(res)$  is verified by specifying the quantified expressions in lines 4-6. The first and second quantified expressions ensure  $res$  did not lose any elements during the deletion process and that for all  $x$ ,  $x < val$  or  $x > val$  is in the correct position. The third expression ensures  $res$  contains the same values in  $t$  without the value  $val$ .

**func *getMin*** Dafny uses the *getMin* function to remove the nodes with two sub-trees. The properties of *getMin* are translated as an axiom in Caesar. Listing 6.15 illustrates the implementation of the function *getMin* in domain *Tree*.

```

1  domain Tree {
2  ...
3  func getMin(t:Tree): Int
4  axiom gm_l forall t:Tree, l:Tree, v:Int, r:Tree. (
5  is_tree(t) && (l == get_left(t)) && (v == get_value(t)) && (r == get_right(t)) &&
6  (t == node(l,v,r)) && is_tree(l)) ==> (getMin(t) == getMin(l))
7  axiom gm_n forall t:Tree, l:Tree, v:Int, r:Tree. (
8  is_tree(t) && (l == get_left(t)) && (v == get_value(t)) && (r == get_right(t)) &&
9  (t == node(l,v,r)) && is_null(l)) ==> (getMin(t) == v)
10 axiom gm_d forall t:Tree, l:Tree, v:Int, r:Tree. (
11 (is_tree(t) && (l == get_left(t))) ==> (contains(t, getMin(t)) &&
12 (forall x :Int. (contains(t, x) && (x != getMin(t))) ==> (getMin(t) < x)))
13 )
14 ...
15 }

```

LISTING 6.15: Implementation of *getMin* in Caesar

The first axiom calls the function *getMin* recursively with parameter left sub-tree  $l$  when  $l$  contains nodes. The second axiom gets the node's value if the sub-tree  $l$  is empty. The last axiom has the same meaning as in Listing 6.9. It is defined with the function *contains* to specify the tree also contains the minimum value of the tree. Moreover, the quantified expression in line 14 ensures that the minimum value of the tree is the smallest value of the tree.

Caesar has difficulty translating the same properties of Dafny in functions. In Caesar, the axioms specify function properties but do not define pre- or post-conditions. The function cannot get the minimum value from the tree when the tree is specified as a Binary Search Tree. Specifying the axioms that the tree  $t$  is  $BST(t)$ , Caesar verifies *assert 0*, which should never terminate, as it means the definition of the axiom is false.



## Chapter 7

# Results and Discussion

### 7.1 Comparison of the Results

The following tables structure the verification results and implemented specifications of the programs in Dafny and Caesar.

the Left Pad			
	Dafny	Caesar (built-in)	Caesar (datatype)
<i>prefix</i>	predicate	pre/post	pre/post
<i>suffix</i>	predicate	pre/post	pre/post
<i>Extensionality</i>	-	-	procedure
Verified	-	-	yes
<i>LeftPad</i>	method	procedure	procedure
Verified	yes	no	yes

TABLE 7.1: Code statistic of the Left Pad function in Dafny and Caesar

Table 7.1 shows the results of the Left Pad function in Dafny and Caesar. The "pre/post" signifies direct annotations in pre-conditions and post-conditions. These specifications could be implemented as functions with the axioms, but this paper only annotates them as the outputs of the quantifiers in Caesar are unpredictable. The "Verified" row indicates whether the implemented algorithm is verified. Dafny uses the method to verify the Left Pad function. Caesar fails to verify the Left Pad function in built-in *Lists* but verifies it as datatype *List*.

Table 7.2 shows the results of Bubble Sort in Dafny and Caesar. Dafny declares three predicates of the Bubble Sort specifications, whereas Caesar annotates the specifications as pre-conditions and post-conditions directly in procedure *BubbleSort*. Caesar could not implement the function *multiplicity* in built-in *Lists* and datatype *List* as its axioms are stuck in a loop due to its quantified instantiations. "MulVer" indicates the verification of Bubble Sort with a multiset. Caesar could not verify the multiset of Bubble Sort in both types. "no MulVer" describes the verification of Bubble Sort without multiset.

Table 7.3 describes the required functions to implement a Binary Search Tree in Dafny and Caesar. Dafny uses the function *treeSet* for the specification

Bubble Sort			
	Dafny	Caesar (built-in)	Caesar (datatype)
<i>arraySorted</i>	predicate	pre/post	pre/post
<i>bubblesSorted</i>	predicate	pre/post	pre/post
<i>bubbleStepFinish</i>	predicate	pre/post	pre/post
<i>multiplicity</i> Implemented	function yes	function no	function no
<i>BubbleSort</i> MulVer no MulVer	method yes yes	procedure no yes	procedure no yes

TABLE 7.2: Code statistic of Bubble Sort in Dafny and Caesar

Binary Search Tree		
	Dafny (datatype)	Caesar (datatype)
<i>treeSet</i>	function	-
<i>contains</i>	-	function
<i>BST</i>	predicate	function
<i>getMin</i>	function	function
<i>Insert</i> Verified	method yes	procedure yes
<i>Delete</i> Verified	method yes	procedure yes

TABLE 7.3: Code statistic of Binary Search Tree in Dafny and Caesar

of *BST*, and Caesar uses func *contains* for *BST*. Both verifiers successfully verify *Insert* and *Delete*.

## 7.2 Discussion

### 7.2.1 Finding Errors

At the beginning of the study, it was expected that Caesar would verify the example functions without any problems. However, Caesar could not verify the Left Pad function in built-in *Lists*, which resulted in errors and counter-examples. The counter-example helps a little to find the error of the code as it shows many examples of the variables but does not explicitly select the variable with an error. In Caesar, it requires debugging the code by hand and investigating all possible errors such as semantics, code, specification, index of the list, or triggers. Therefore, supporting the feature that shows which part of the code has an error would bring an advantage to the debugging process in Caesar.

### 7.2.2 Trigger Selection

Caesar has the problem of implementing functions with the quantified expressions, the same as in Dafny. These expressions create quantified instantiations that break the code and never terminate. Caesar could not implement the function *multiplicity* to ensure a list contains the same elements due to its unexpected quantified instantiations. However, Caesar is still in development, and there will be an update with a new feature. The upcoming feature allows the user to select the triggers in axioms. For example, Listing 7.1 demonstrates selecting the triggers in the function *multiplicity*.

```

1 func multiplicity(ls:List, el:Int): UInt
2 axiom m_n forall ls:List, el:Int. is_null(ls) ==> (multiplicity(ls,el) == 0)
3 axiom m_l forall ls:List, h:Int, t:List, el:Int @trigger(cons(h,t), multiplicity(ls,el))
4 (is_list(ls) && (ls == cons(h,t)) && (el == h)) ==> (multiplicity(ls,el) == 1+multiplicity(t,el))
5 axiom m_l_n forall ls:List, h:Int, t:List, el:Int @trigger(cons(h,t), multiplicity(ls, el))
6 (is_list(ls) && (ls == cons(h,t)) && (el != h)) ==> (multiplicity(ls,el) == multiplicity(t,el))

```

LISTING 7.1: Example of selecting triggers in Caesar

The triggers for both axioms are selected as *cons(h,t)* and *multiplicity(ls,el)* covering all the quantified variables. Selecting the trigger, the function *multiplicity(ls,el)* allows to compute recursively. The trigger *cons(h,t)* verifies the changes of its lists during each computation of the function *multiplicity*. After selecting the triggers, the function *multiplicity* verifies each multiplicity of elements in a list. For instance, Figure 7.1 shows the multiset of both lists *ls* and *res* are equal. After implementing the selected

```

proc main () -> ()
{
  var ls : List = cons(2, cons(1, cons(4, cons(4, cons(5, null())))))
  var res : List = cons(5, cons(4, cons(1, cons(2, cons(4, null())))))
  assert ? (forall k: UInt. (multiplicity(ls, k) == multiplicity(res,k)))
}

```

FIGURE 7.1: Caesar verifies that multiset of two lists are equal

triggers, the quantified expressions are not stuck in the matching loop, which allows the verifier to compare the multiplicity of all the elements.



## Chapter 8

# Conclusion

The primary objective of this study is to compare and analyze the verification capabilities of deterministic programs in Dafny and Caesar. The thesis handles implemented codes such as the Left Pad function, Bubble Sort, and Binary Search Tree to examine the effectiveness of the verification process of these languages. Caesar and Dafny show different results in the Left Pad function, where Caesar requires additional specifications and implements a new data-type *List* to verify its correctness. Both languages verify the Bubble Sort, but Caesar could not implement multiset due to the limitation of quantified expressions in Caesar. Also, the languages verify Binary Search Trees, where Dafny uses type *set* and Caesar uses functional specifications.

The comparison of Caesar and Dafny contributes to the analysis of the verification capabilities of Caesar in deterministic programming. Also, it proposes new features for its ongoing development of the verification language Caesar. Dafny and Caesar are in development, and recently, Caesar updated a new feature of manually selecting triggers that improves the implementation and verification of quantified expressions, as shown in the discussion. Therefore, this study investigates the current state of these tools. In conclusion, Caesar and Dafny are actively evolving to improve the correct output of the programs. Including new built-in types such as multiset, sequence, and strings in Caesar would ease verifying programs of strings and lists. Additionally, it would be helpful to support a feature for the user to find which part of the code has an error during the debugging process. Also, supporting automatic triggers for quantified expressions would improve the implementation of the quantified expression to support the verification process in Caesar.



# Appendix A

## Code

### A.1 Dafny

#### A.1.1 Left Pad Function

```
1 method LeftPad(str: string, ln: int, c: char) returns (res: string)
2   requires 0 <= |str| && 0 <= ln
3   ensures max(ln, |str|) == |res|
4   ensures prefix(str, ln, c, res)
5   ensures suffix(str, ln, res)
6   {
7     if (ln <= |str|) {
8       res := str;
9     } else if (|str| < ln) {
10      var i := 0;
11      var pads := ln - |str|;
12      res := str;
13      while(i < pads)
14        invariant 0 <= i <= pads
15        invariant |str| + i == |res|
16        invariant prefix(str, i+|str|, c, res)
17        invariant suffix(str, i+|str|, res)
18      {
19        res := [c] + res;
20        i := i + 1;
21      }
22    }
23  }
```

LISTING A.1: Verification of Left Pad function in Dafny

#### A.1.2 Bubble Sort Algorithm

```
1 method BubbleSort(arr: array?<int>)
2   requires arr != null
3   ensures arr != null
4   ensures |arr[..]| == old(|arr[..|)
5   ensures arraySorted(arr, 0, arr.Length-1)
6   ensures multiset(arr[..]) == multiset(old(arr[..]))
7   modifies arr
8   {
9     var i := 0;
10    var n := arr.Length-1;
11    while (0 < n-i)
12      invariant 0 <= i <= n || n == -1
13      invariant arraySorted(arr, n-i, n)
14      invariant bubblesSorted(arr, n-i)
15      invariant multiset(arr[..]) == multiset(old(arr[..]))
16      decreases n-i
17    {
18      var j := 0;
```

```

19   while (j < n-i)
20     invariant 0 <= j <= n-i
21     invariant arraySorted(arr,n-i,n)
22     invariant bubblesSorted(arr,n-i)
23     invariant bubbleStepFinish(arr,j)
24     invariant multiset(arr[..]) == multiset(old(arr[..]))
25     decreases n-j
26     {
27       if(arr[j] > arr[j+1]) {
28         var ind1 := arr[j];
29         var ind2 := arr[j+1];
30         arr[j] := ind2;
31         arr[j+1] := ind1;
32       }
33       j := j+1;
34     }
35     i := i+1;
36   }
37 }

```

LISTING A.2: Verification of Bubble Sort algorithm in Dafny

### A.1.3 Insertion

```

1  method Insert(t:Tree, val:int) returns (res:Tree)
2    requires BST(t)
3    ensures forall x :: x in treeSet(t) && x < val ==> x in treeSet(res)
4    ensures forall x :: x in treeSet(t) && x > val ==> x in treeSet(res)
5    ensures treeSet(res) == treeSet(t) + {val}
6    ensures BST(res)
7    decreases t
8    {
9      if (t.Null?) {
10         res := Node(Null, val, Null);
11       } else if (t.Node?) {
12         var l:Tree := t.left;
13         var v:int := t.value;
14         var r:Tree := t.right;
15         if (val < v) {
16           var ll:Tree := Insert(l, val);
17           res := (Node(ll, v, r));
18         } else if (val > v) {
19           var rr:Tree := Insert(r, val);
20           res := (Node(l, v, rr));
21         } else {
22           res := t;
23         }
24       }
25     }

```

LISTING A.3: Verification of Insertion in Dafny

### A.1.4 Deletion

```

1  method Delete(t:Tree, val:int) returns (res:Tree)
2    requires BST(t)
3    ensures forall x :: x in treeSet(t) && x < val ==> x in treeSet(res)
4    ensures forall x :: x in treeSet(t) && x > val ==> x in treeSet(res)
5    ensures treeSet(res) == treeSet(t) - {val}
6    ensures BST(res)
7    decreases t
8    {
9      if (t.Null?) {
10         res := t;
11       } else if (t.Node?) {
12         var l : Tree := t.left;

```



```

13   var v : int := t.value;
14   var r : Tree := t.right;
15   if (val < v) {
16     var ll := Delete(l, val);
17     res := Node(ll, v, r);
18   } else if (val > v) {
19     var rr := Delete(r, val);
20     res := Node(l, v, rr);
21   } else if (val == v) {
22     if (l == Null && r == Null) {res := Null;}
23     else if (l == Null) {res := r;}
24     else if (r == Null) {res := l;}
25     else {
26       var minV := getMin(r);
27       var rm := Delete(r, minV);
28       res := Node(l, minV, rm);
29     }
30   }
31 }
32 }

```

LISTING A.4: Verification of Deletion in Dafny

## A.2 Caesar

### A.2.1 Left Pad Function in Built-in *Lists*

```

1  proc LeftPad(str: []Int, ln: UInt, c: Int) -> (res: []Int)
2    pre ? (0 <= len(str) && 0 <= ln)
3    post ? (
4      (len(res) == ite(ln < len(str), len(str), ln)) &&
5      (forall k: UInt. ((k < ln - len(str))) ==> (select(res, k) == c)) &&
6      (forall k: UInt. ((ite(0 < ln-len(str), ln-len(str), 0) <= k) &&
7      (k < len(str))) ==> (select(res, k) == select(str, (k-ite(0 < ln-len(str), ln-len(str), 0))
8      )))
9  {
10   assume ? (len(res) == ln)
11   if(ln <= len(str)) {
12     res = str
13   } else {
14     if (len(str) < ln){
15       var i: UInt = 0
16       var pads: Int = ln-len(str)
17       assert ? I1
18       havoc res, i
19       assume ? I1
20       if (i < pads){
21         res = store(res, i, c)
22         i = i + 1
23         assert ? I1
24         assume ? (false)
25       } else {}
26       var j : UInt = 0
27       assert ? I2
28       havoc res, j
29       assume ? I2
30       if (j < len(str)) {
31         res = store(res, ln-len(str)+ j, select(str, j))
32         j = j + 1
33         assert ? I2
34         assume ? (false)
35       } else {}
36     } else {}
37   }

```

38 }

LISTING A.5: Verification of Left Pad function in built-in *Lists* in Caesar

## A.2.2 Bubble Sort Algorithm in Built-in *Lists*

```

1  proc BubbleSort(arr: []Int) -> (res: []Int)
2    pre ? ((0 <= len(arr)))
3    post ? ((0 <= len(res)))
4    && (forall p: UInt, q: UInt. ((0 <= p) && (p < q) && (q < len(res))) ==> (select(res, p) <=
      select(res, q)))
5    && (len(arr) == len(res))
6    && (forall k: UInt. (multiplicity(len(arr), arr, k) == multiplicity(len(res), res, k)))
7  )
8  {
9    res = arr;
10   var i: UInt = 0;
11   var n: UInt = (len(res) - 1);
12   assert ? I3
13   havoc res
14   assume ? I3
15   if (0 < n-i) {
16     var j: UInt = 0
17     assert ? I4
18     havoc res
19     assume ? I4
20     if (j < n-i) {
21       if((select(res, j) > select(res, j+1)))
22       {
23         var ind1 : Int = select(res, j);
24         var ind2 : Int = select(res, j+1);
25         res = store(res, j, ind2);
26         res = store(res, j+1, ind1);
27       } else {}
28       j = j + 1;
29       assert ? I4
30       assume ? (false)
31     } else {}
32     i = i + 1
33     assert ? I3
34     assume ? (false)
35   } else {}
36 }
```

LISTING A.6: Verification of Bubble Sort algorithm in built-in *Lists* in Caesar

## A.2.3 Bubble Sort Algorithm in Datatype *List*

```

1  proc BubbleSort(ls: List) -> (res: List)
2    pre ? ((0 <= length(ls)))
3    post ? ((0 <= length(res)))
4    && (forall p: UInt, q: UInt. ((0 <= p) && (p <= q) && (q < length(res))) ==> (select(res, p)
      <= select(res, q)))
5    && (length(ls) == length(res))
6  )
7  {
8    res = ls;
9    var i: UInt = 0;
10   var n: UInt = (length(res) - 1);

12   assert ? I3
13   havoc res, i
14   assume ? I3
```

```

15   if (0 < n-i) {
16       var j: UInt = 0

18       assert ? I4
19       havoc res, j
20       assume ? I4
21       if (j < n-i) {
22           if((select(res, j) > select(res, j+1)))
23           {
24               var temp : Int = select(res, j);
25               var temp2 : Int = select(res, j+1);
26               res = Store(res, j, temp2);
27               res = Store(res, j+1, temp);
28           } else {}

30           j = j + 1;

32       assert ? I4
33       assume ? (false)
34       } else {}

36       i = i + 1

38       assert ? I3
39       assume ? (false)
40   } else {}
41 }

```

LISTING A.7: Verification of Bubble Sort algorithm in built-in *Lists* in Caesar

## A.2.4 Insertion

```

1  proc Insert(t: Tree, val: Int) -> (res: Tree)
2  pre? (BST(t))
3  post? (
4      (forall x: Int. ((contains(t, x) && (x < val)) ==> contains(res, x))) &&
5      (forall x: Int. ((contains(t, x) && (x > val)) ==> contains(res, x))) &&
6      (forall v: Int. (contains(res, v) == contains(t,v) || (v == val))) &&
7      BST(res) &&
8      contains(res, val)
9  )
10 {
11     if(is_null(t)){
12         res = node(null(), val, null())
13     }else{
14         if(is_tree(t)){
15             var temp: Tree;
16             if (val == get_value(t)){
17                 res = t
18             }else{
19                 if(val < get_value(t)){
20                     temp = Insert(get_left(t), val)
21                     res = node(temp, get_value(t), get_right(t))
22                 }else{
23                     if(val > get_value(t)){
24                         temp = Insert(get_right(t), val)
25                         res = node(get_left(t), get_value(t), temp)
26                     } else {}
27                 }
28             }
29         } else {}
30     }
31 }

```

LISTING A.8: Verification of Insertion in data type *Tree* in Caesar

## A.2.5 Deletion

```

1  proc Delete(t: Tree, val: Int) -> (res: Tree)
2    pre ? (BST(t))
3    post ? (
4      (forall x: Int. ((contains(t, x) && (x < val)) ==> contains(res, x))) &&
5      (forall x: Int. ((contains(t, x) && (x > val)) ==> contains(res, x))) &&
6      (forall v: Int. (contains(res, v) == (contains(t,v) && (v != val)))) &&
7      BST(res)
8    )
9  {
10   if(is_null(t)){
11     res = t
12   }else{
13     if (is_tree(t)) {
14       var l: Tree = get_left(t)
15       var v: Int = get_value(t)
16       var r: Tree = get_right(t)
17       if (val < v) {
18         var ll : Tree = Delete(l, val)
19         res = node(ll, v, r)
20       } else {
21         if (val > v) {
22           var rr : Tree = Delete(r, val)
23           res = node(l, v, rr)
24         } else {
25           if (val == v) {
26             if (l == null() && r == null()) {
27               res = null()
28             } else {
29               if (l == null()) {
30                 res = r
31               } else {
32                 if(r == null()) {
33                   res = l
34                 } else {
35                   if ((l != null()) && (r != null())){
36                     assert ? (BST(r))
37                     var minV : Int = getMin(r)
38                     var rm : Tree = Delete(r, minV)
39                     res = node(l, minV, rm)
40                   } else {}
41                 }
42               }
43             }
44           } else {}
45         }
46       }
47     } else {}
48   }
49 }

```

LISTING A.9: Verification of Deletion in data type *Tree* in Caesar

# Appendix B

## Counter-example

### B.1 Caesar

```
1 j_0!21 -> 3
2 k!20 -> 0
3 ln!9 -> 18022
4 res_1!16 -> (let ((a!1 (store (store (store ((as const (Array Int Int)) 6) 0 9) 1575 29)
5 (- 9943)
6 30)))
7 (|List[Int]_list| 18022 a!1))
8 res_0!22 -> (let ((a!1 (store (store (store ((as const (Array Int Int)) 7) 1 15)
9 18020
10 (- 335))
11 (- 5982)
12 34)))
13 (|List[Int]_list| 18022 (store (store a!1 0 9) (- 9943) 32)))
14 k!26 -> 0
15 k!17 -> 0
16 c!12 -> 9
17 i_0!15 -> 1
18 k!23 -> 0
19 k!11 -> 0
20 str!8 -> (let ((a!1 (store (store (store ((as const (Array Int Int)) 5) (- 5982) 33) 1 9)
21 3
22 (- 2332))))
23 (let ((a!2 (store (store (store (store a!1 0 15) 1575 9) 18019 9) 2 9)))
24 (|List[Int]_list| 18021 a!2)))
25 k!18 -> 0
26 k!14 -> 0
27 k!24 -> 0
28 k!13 -> 0
29 res!10 -> (|List[Int]_list| 18022 ((as const (Array Int Int)) 0))
30 k!19 -> 0
31 k!25 -> 0
32 k!551 -> {
33 (- 5982) -> 33
34 1 -> 9
35 3 -> (- 2332)
36 0 -> 15
37 1575 -> 9
38 18019 -> 9
39 2 -> 9
40 else -> 5
41 }
42 k!552 -> {
43 0 -> 9
44 1575 -> 29
45 (- 9943) -> 30
46 else -> 6
47 }
48 k!553 -> {
49 1 -> 15
50 18020 -> (- 335)
51 (- 5982) -> 34
52 0 -> 9
```

```
53   (- 9943) -> 32
54   else -> 7
55 }
56 array-ext -> {
57   (_ as-array k!551) (_ as-array k!553) -> (- 5982)
58   (_ as-array k!552) (_ as-array k!553) -> (- 9943)
59   else -> 1575
60 }
```

LISTING B.1: Counter-example of *LeftPad* in built-in *Lists* in Caesar

# References

- [Abd+20] R. Abdalkareem et al. “On the Impact of Using Trivial Packages: An Empirical Case Study on NPM and PyPI”. In: *Empir. Softw. Eng.* 25.2 (2020), pp. 1168–1204.
- [ALR14] N. Amin, K. R. M. Leino, and T. Rompf. “Computing with an SMT Solver”. In: *Tests and Proofs - 8th International Conference, TAP@STAF 2014*. Vol. 8570. Lecture Notes in Computer Science. Springer, 2014, pp. 20–35.
- [AO19] K. R. Apt and E. Olderog. “Fifty Years of Hoare’s Logic”. In: *Formal Aspects Comput.* 31.6 (2019), pp. 751–807.
- [Bli89] W. D. Blizard. “Multiset Theory”. In: *Notre Dame J. Formal Log.* 30.1 (1989), pp. 36–66.
- [BMS19] N. Becker, P. Müller, and A. J. Summers. “The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations”. In: *Tools and Algorithms for the Construction and Analysis of System*. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 99–116.
- [Bog+16] C. Bogart et al. “How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems”. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 109–120.
- [BST+10] C. Barrett, A. Stump, C. Tinelli, et al. “The SMT-LIB Standard - Version 2.0”. In: *Proceedings of the 8th international workshop on satisfiability modulo theories*. Vol. 13. 2010, p. 14.
- [Cor23] M. Corporation. *Z3 Guide*. 2023. URL: <https://microsoft.github.io/z3guide/>. (accessed: 22.08.2023).
- [Dij75] E. W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (1975), pp. 453–457.
- [Fre16] M. Fredrikson. *Incremental Proof Development in Dafny*. 2016. URL: <https://www.cs.cmu.edu/~mfredrik/15414/lectures/17-notes.pdf>. (accessed: 17.08.2023).
- [Hib62] T. N. Hibbard. “Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting”. In: *J. ACM* 9.1 (1962), pp. 13–28.
- [Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580.

- [Ive62] K. E. Iverson. “A Programming Language”. In: *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring)*. Association for Computing Machinery, 1962, pp. 345–351.
- [Kam19] B. L. Kaminski. “Advanced Weakest Precondition Calculi for Probabilistic Programs”. PhD thesis. RWTH Aachen University, Germany, 2019.
- [KL12] J. Koenig and K. R. M. Leino. “Getting Started with Dafny: A Guide”. In: *Software Safety and Security - Tools for Analysis and Verification*. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 152–181.
- [Knu73] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [Lei08] K. R. M. Leino. “This is Boogie 2”. In: *manuscript KRML 178.131* (2008), p. 9.
- [Lei10] K. R. M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.
- [LFC21] K. R. M. Leino, R. L. Ford, and D. R. Cok. *Dafny Reference Manual*. 2021.
- [LP13] K. R. M. Leino and N. Polikarpova. “Verified Calculations”. In: *Verified Software: Theories, Tools, Experiments - 5th International Conference*. Vol. 8164. Lecture Notes in Computer Science. Springer, 2013, pp. 170–190.
- [LW14] K. R. M. Leino and V. Wüstholtz. “The Dafny Integrated Development Environment”. In: *arXiv preprint arXiv:1404.6602* (2014).
- [MB08] L. M. de Moura and N. S. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.
- [Mey97] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.
- [MM05] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.
- [Muk90] S. Mukhopadhyay. “An Optical Conversion System: From Binary to Decimal and Decimal to Binary”. In: *Optics Communications* 76.5 (1990), pp. 309–312.
- [Mül19] P. Müller. “Building Deductive Program Verifiers - Lecture Notes”. In: vol. 53. *Engineering Secure and Dependable Software Systems*. 2019, pp. 189–206.
- [Sch22] P. Schroer. “A Deductive Verifier for Probabilistic Programs”. In: (2022).



- [Sch23] P. Schroer. “A Deductive Verification Infrastructure for Probabilistic Programs”. In: 7.OOPSLA1 (2023).
- [SD13] W. Sonnex and S. Drossopoulou. *Verified Programming in Dafny*. 2013. URL: [https://www.doc.ic.ac.uk/~scd/Dafny\\_Material/Lectures.pdf](https://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf). (accessed: 17.08.2023).
- [Stu+01] A. Stump et al. “A Decision Procedure for an Extensional Theory of Arrays”. In: *16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001, pp. 29–37.
- [Wag85] E. G. Wagner. “A Categorical View of Weakest Liberal Preconditions”. In: *Category Theory and Computer Programming, Tutorial and Workshop, Guildford, UK, September 16-20, 1985 Proceedings*. Ed. by D. H. Pitt et al. Vol. 240. Lecture Notes in Computer Science. Springer, 1985, pp. 198–205.