Towards Automated Verification of Object-Based Software with Reference Behavior

Yu-Shan Sun
Clemson University, yushan87@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Recommended Citation
https://tigerprints.clemson.edu/all_dissertations/2280

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
Towards Automated Verification of Object-Based Software with Reference Behavior

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Yu-Shan Sun
December 2018

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. John D. McGregor
Dr. Nigamanth Sridhar
Dr. Pradip K. Srimani
Abstract

Automated verification is critical for ensuring that an implementation is correct and meets the specified behavior on every valid input. Verification should be modular to promote reuse and to scale up. However, for code that involves explicit reference behavior, there is the added complexity of reasoning that only the intended objects are being affected.

This research focuses on simplifying automated verification and enabling modular verification using data abstractions that hide explicit reference behavior. While avoiding explicit reference behavior simplifies reasoning for a majority of software components, for capturing unavoidable reference behavior, such as that needed to implement classes of lower-level “linked” realizations such as for lists and trees, the research introduces and uses automation-friendly abstractions to capture acyclic reference behavior.

The overall research involves the development of specification and verification mechanisms for components where objects share a global state, along with a new prototype verification system that is designed to generate simplified verification conditions with automation in mind. Experimentation and evaluation involve a class of components with and without explicit reference behavior.
Dedication

To my parents and sister who supported me all throughout this process and for believing that I could finish. Also to entire Hiestand family who provided me with a home away from home.
Acknowledgments

First and foremost, I would like to thank my advisor, Murali Sitaraman, for his guidance, support and patience throughout this whole process. I would like to thank Joan Krone for acting as a co-advisor and for the countless meetings, revisions and feedback. Without both of them, none of this work would have been possible. Special thanks to my committee members: John D. McGregor, Nigamanth Sridhar and Pradip K. Srimani for providing feedback and encouraging me to improve this work.

I would like to acknowledge members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State. My special thanks to Bill Ogden, Bruce Weide and Joan Krone for their insights in developing the specifications given in this dissertation and suggestions for improvements. I gratefully acknowledge the numerous hours Charles “Chuck” Cook and Daniel Welch have spent helping me improve the implementations associated with this work. Also my thanks to all my friends from around the world, who listened to all my troubles and offered words of encouragement.

This research is supported in part by U.S. National Science Foundation (NSF) grants CCF-1161916, DUE-1022941 and DUE-1611714. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

On a personal note, I would like to thank my parents and sister for patiently waiting until I finished and offering unconditional support throughout this process. A special thanks to the Hiestand family for always being there for me when I needed it the most.
# Table of Contents

Title Page ................................................................. i
Abstract ................................................................. ii
Dedication ............................................................... iii
Acknowledgments ...................................................... iv
List of Tables ........................................................ vii
List of Figures ........................................................ viii
List of Listings ......................................................... ix

1 Introduction .......................................................... 1
   1.1 Facilitating Direct Reasoning through Data Abstraction .......... 4
   1.2 Reasoning about Reference Behavior Explicitly ................. 9
   1.3 Problem Statement ............................................. 10
   1.4 Research Approach and Evaluation ............................. 11
   1.5 Thesis Statement .............................................. 14
   1.6 Dissertation Organization .................................... 14

2 Verification Background and Related Work .......................... 15
   2.1 General Verification Background ................................ 15
   2.2 Separation Logic .............................................. 16
   2.3 Dynamic Frames .............................................. 19
   2.4 Region Logic .................................................. 22
   2.5 An Approach Using RESOLVE Principles ...................... 23
   2.6 Other Efforts ................................................ 24

3 RESOLVE Background and Prior Work ............................... 25
   3.1 Introduction to RESOLVE Specifications ...................... 25
   3.2 Recent Work in Automated Verification of RESOLVE Components . 28
   3.3 Previous Work in Formalizing Specifications to Capture Reference Behavior . 34

4 Generation of Simplified VCs and Sequent .......................... 36
   4.1 Reduced Sequents ............................................. 36
   4.2 Forming Parsimonious VCs .................................... 38
   4.3 Other VC Generator Updates .................................. 44

5 Data Abstractions with Shared State ............................... 46
   5.1 Communally_Bounded_List_Template .......................... 48
5.2 Example Client Programs and Evaluation ........................................ 50
5.3 Enhancement for Searching a List and Evaluation ............................ 55

6 A Concept to Capture Acyclic Reference Behavior .............................. 59
6.1 An Informal Description ............................................................... 60
6.2 New Mathematical Theories Needed To Describe Reference Behavior .... 65
6.3 Formal Specification of Communal_UVR_Template .......................... 69
6.4 Formal Specification for CBUVRT Operations ................................. 73
6.5 Alternative Versions of UVRT ...................................................... 76

7 VC Generation and Evaluation for Concept Realizations ...................... 78
7.1 Non-Sharing Implementations ...................................................... 79
7.2 Interference-Free Sharing Implementations with Independent Correspondence 88
7.3 Interference-Free Sharing Implementations with Independent Regional Correspondence 91

8 Conclusions and Future Research .................................................... 98

Appendices ......................................................................................... 101
A Proof Rules ................................................................................... 102
B Globally Bounded List Collection ............................................... 121
C Communally Bounded List Collection ...................................... 152
D Bounded Stack Collection .......................................................... 185
E Globally Bounded Stack Collection ............................................. 208
F Communally Bounded Stack Collection ...................................... 212
G Communally Bounded UVRT Collection ..................................... 240
H Mathematical Theories ............................................................... 247

Bibliography ...................................................................................... 256
List of Tables

4.1 Reduction Rules ................................................. 37
7.1 Categories of Concept Realization ............................ 78
List of Figures

1.1 Contrasting When Reasoning About Reference Behavior Is Needed . . . . . . . . . . 2
1.2 Illustration of Mathematical Abstraction for a List . . . . . . . . . . . . . . . . . . . 4
1.3 Illustration of List Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Illustration of Reverse_List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.5 Tracing of the Code in Listing 1.2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.6 Modular Reasoning Overview for the Recursive List Reversal in Listing 1.2 . . . . . 8
1.7 Reversing a List Using the Code in Listing 1.3 . . . . . . . . . . . . . . . . . . . . . . 10

3.1 Applying Prt_Btw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2 Verification Pipeline Process of Listing 3.5 . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Verification of Listing 3.8 Using an Automated Prover . . . . . . . . . . . . . . . . . 33

4.1 Sample Sequent Reduction Tree Produced by VC Generator . . . . . . . . . . . . . . 38

5.1 Comparing Reasoning Approaches (Reproduced from Chapter 1) . . . . . . . . . . . 46
5.2 Example of a Communal List Instantiation . . . . . . . . . . . . . . . . . . . . . . . . . 47

6.1 Sample UVRT Chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.2 Illustration for Selected Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.3 Illustration of the Effects of Operation Calls in Insert_Front . . . . . . . . . . . . . 63
6.4 Reversing a singly-linked location . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.5 Is_Closed_wrt and ¬ Is_Closed_wrt with One Function . . . . . . . . . . . . . . . . 66
6.6 Is_Closed_wrt and ¬ Is_Closed_wrt with Multiple Functions . . . . . . . . . . . . . 67
6.7 Closure_for . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.8 Is_Stable_wrt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.9 Terminal_Range . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.10 Global State Variables Ref and Content . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.11 Finalization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.12 Modified Global State Variable Ref . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

7.1 Relationship Between Type Constraint, Convention and Correspondence . . . . . 81

8.1 A cactus-like Stack Realization (Reproduced from [96]) . . . . . . . . . . . . . . . . 99
## List of Source Listings

1.1 Reasoning in the Presence of Aliased References ........................................... 2  
1.2 List Reversal (Without Explicit References) ................................................. 6  
1.3 List Reversal (Explicit References) ............................................................ 9  
2.1 List Reversal (Hoare Logic) ........................................................................... 17  
2.2 List Interface with Dynamic Frames from [99] ............................................... 20  
3.1 A Globally-Bounded List Concept .................................................................. 26  
3.2 Mathematical Model for List ...................................................................... 26  
3.3 Selected Operations for List ....................................................................... 27  
3.4 List Reversal Specifications ....................................................................... 28  
3.5 Recursive List Reversal .............................................................................. 29  
3.6 Assertive Program used to Generate VCs for Listing 3.5 ......................... 30  
3.7 Termination VC ......................................................................................... 31  
3.8 Iterative List Reversal ................................................................................. 32  
4.1 Termination VC for `Reverse_List` (Reproduced from Listing 3.7) ........ 39  
4.2 Termination VC for `Reverse_List` with only Relevant Givens ................. 40  
4.3 Confirm Rule ......................................................................................... 41  
4.4 Initial Assertive Program .......................................................................... 41  
4.5 Assertive Program After Applying the Confirm Rule ................................ 41  
4.6 Substitution Step ...................................................................................... 42  
4.7 Assertive Program After Applying the Substitution Step ......................... 42  
4.8 Assume Application Step ......................................................................... 42  
4.9 Final Resulting Sequents .......................................................................... 43  
4.10 Partial Assertive Program with Nested-If Statements .............................. 43  
4.11 Stipulate Substitution Step ..................................................................... 44  
4.12 Stipulate Application Step ..................................................................... 44  
5.1 A Communally-Bounded List Concept ...................................................... 48  
5.2 Which_Entails Rule .................................................................................. 48  
5.3 Resulting VC After Applying the Which_Entails Rule .............................. 49  
5.4 Shared Variables ..................................................................................... 49  
5.5 Communal List Specifications .................................................................... 49  
5.6 Selected Operations for Communal List ................................................... 50  
5.7 Example Client Program #1 ........................................................................ 50  
5.8 Facility Instantiation VC for `List_Fac` ...................................................... 51  
5.9 VC for Establishing the Pre-Condition of the Third Call to `Insert` ........ 52  
5.10 Example Client Program #2 ....................................................................... 52  
5.11 (Unprovable) VC for Establishing the Pre-Condition of the Third Call to `Insert` .................................................. 53  
5.12 VC for Establishing that `List_Fac::Total_Size` isn’t Modified ............. 53  
5.13 Simplifying the VC (Step #1) .................................................................. 53  
5.14 Simplifying the VC (Step #2) .................................................................. 54  
5.15 Simplifying the VC (Step #3) .................................................................. 54
Chapter 1

Introduction

Software that functions correctly, both from the point of view of validation and verification, plays a critical role in our current and future society. The software validation process checks that a software system meets its requirements and is done primarily through testing. The verification process checks that the system is properly engineered. A properly engineered system must be built from components and must include full behavioral specifications for those components. At present, most software engineers use testing as the means for developing trustworthy systems, because testing remains the most practical method. However, since it is impossible to test every possible data set for any given system, the software engineering community continues to explore the use of automated verification. Unlike testing, automated software verification can demonstrate that each possible execution path leads to the specified behavior on every valid input. Much progress in automated verification has been made and several systems have been used to specify and verify various software components [58].

There is added complexity in verifying software when the implementing code involves explicit reference behavior, no matter what language is being used. Reasoning must ensure only the intended objects are being affected and deal with any potential aliasing [48][97]. For example, consider the following code-snippet written in Java. In this example, u and v are local (reference) variables to some objects and modify is a method that affects u’s content. The specifications for modify will state the changes to u after the call. However, in this case v is also affected, but it is not explicitly mentioned in the call to modify. In a language with clean semantics, such as RESOLVE [60][61][87], it is not possible to have such side effects because there is only one reference per object. Aliasing
cannot be introduced through routine assignments or parameter passing. It is necessary to write and use operation specifications that make such side-effects explicit.

Listing 1.1: Reasoning in the Presence of Aliased References

```java
...
    v = u;
    ...
    modify(u);
    ...
```

Figure 1.1 compares what reasoning is necessary when using the RESOLVE approach versus when programming in Java-like languages [93]. In the RESOLVE approach, use of reference-hiding data abstractions with clean semantics (e.g., the list abstraction in Section 1.1) simplify most routine reasoning. Reasoning about reference behavior is needed only when doing so is unavoidable. On the other hand, because most of the programs written in Java or C contain explicit references, these programs require reasoning about references routinely.

Figure 1.1: Contrasting When Reasoning About Reference Behavior Is Needed

**Verification in the Absence of Sharing:** The white section in the Java-like approach represents simple code involving primitive types (i.e. Integers), where no indirection is involved. Reasoning about these programs only involve the operation’s explicit parameters with no side-effects and significant automation has been achieved by using various approaches [58].

In the approach based on RESOLVE, a language with clean semantics, reasoning for primitive types and non-sharing data abstractions, such as individually bounded stacks or queues, is straightforward. This class of programs is subsumed in the green section of the figure and significant automation has been achieved in prior research [1][42][91].
Verification with Reference-Hiding Data Abstractions: In reasoning about code with reference behavior, two complications arise. The first one is handling of a shared state space that is affected by operations on a collection of objects. The second one is how other objects are affected (or not affected) by changes to a given object (frame property).

Reference-hiding (or other) data abstractions with sharing proposed in this research address the shared state space complication (also included in the green section). They are designed to avoid aliasing. Operations are only allowed to modify the passed-in object and the shared state space. These data abstractions can capture, for example, lists and trees that share space. Specification constructs are necessary to capture sharing in these data abstractions. In their implementations, additional constructs are necessary to connect the shared data representation with abstraction. These data abstractions can be built by reusing other components with sharing, and reasoning can be done modularly. Development of a proof system for handling the class of data abstractions with sharing is a central contribution of this work.

Reference-Capturing Data Abstractions: Reasoning about explicit reference behavior cannot be ultimately avoided because lower-level linked implementations of lists and trees, for example, need that behavior, and this class of programs correspond to the purple section(s) of the figure. When reasoning about code involving (unavoidable) references, shared state space and other object modification complications are both present. Verification and automation about arbitrary reference behavior is hard. To minimize the complexity of specification and reasoning, we avoid aliasing and modifications to objects other than explicit parameters as far as possible. To achieve these objectives, we introduce a data abstraction that is more amenable to automated verification and that can be used to capture acyclic reference behavior, such as what is sufficient for implementing lists and trees for example. We distinguish uses of this data abstraction, with and without aliasing, to build components with reference behavior.

In order to build components, and evaluate and experiment with generation of verification assertions, this research has led to the development of a sequent-based verification condition generator that both simplifies the task of an automated prover and assists programmers to more easily track down faulty specification or code.
To emphasize the central importance of the role of data abstraction in mitigating reasoning complexity and to motivate the challenges in reasoning about components with explicit reference behavior, we consider two motivating examples in this introduction.

1.1 Facilitating Direct Reasoning through Data Abstraction

In order for verification to “scale-up” for software built from components, modular reasoning (verification of one component at a time) is indispensable. Modular reasoning requires that all components have formal behavioral specifications. Using these specifications, together with proof rules that extend mathematical logic, it is possible to generate verification conditions equivalent to the correctness of any given implementation of the specifications. Moreover, once a component has been verified, that component can be safely incorporated into larger components without reverifying. Although the development time increases with modular reasoning because all components need specifications, the cost is amortized throughout the software’s lifetime and through reuse of components in multiple systems.

Modular reasoning has been applied to object-based software [74]. Reference behavior is a central source of complexity and challenge in modular verification of object-based software. When passing arguments using references or manipulating a reference to memory directly, client components could inadvertently create modifications that are not captured by the mathematical specifications. When a component specification allows for the possibility of aliased references, verification would then need to do cross-boundary reasoning to ensure that all modifications are captured. This in turn can break modularity of verification and can preclude safe reuse of a given component.

The first example demonstrates how to use abstraction to reason about a list component without having to reason about explicit references. A suitable mathematical abstraction for a list is a pair of (mathematical) string of entries [88]. The current position of the list cursor is in between the two strings and that is where insertions and deletions can take place.

List Abstraction

\[
\begin{array}{c}
< \\
\text{ } \\
\text{ } \\
\text{ } \\
\text{ } \\
\text{ } \\
> \\
\text{ } \\
\text{ } \\
\text{ } \\
\text{ } \\
\text{ } \\
< \\
\end{array}
\]

Figure 1.2: Illustration of Mathematical Abstraction for a List
The effects of each operation call are illustrated in Figure 1.3. Note that the red bar represents a conceptual cursor separating the two strings. A call to `Advance` will move the conceptual cursor. When a new tree is inserted into the list using the `Insert` operation call, it is inserted after the cursor position, i.e., at the beginning of the remaining or second string. Similarly, the `Remove` operation call removes and returns the the entry immediately following the cursor.

<table>
<thead>
<tr>
<th>Operation Call</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advance</td>
<td>![ BEFORE IMAGE ]</td>
<td>![ AFTER IMAGE ]</td>
</tr>
<tr>
<td>Insert</td>
<td>![ BEFORE IMAGE ] (Tree Object to be Inserted)</td>
<td>![ AFTER IMAGE ]</td>
</tr>
<tr>
<td>Remove</td>
<td>![ BEFORE IMAGE ]</td>
<td>![ AFTER IMAGE ] (Tree Object Removed)</td>
</tr>
</tbody>
</table>

Figure 1.3: Illustration of List Operations

The desired behavior of a list reversal operation, a secondary operation that can be implemented using the primary operations in the `List`, is illustrated through an example in Figure 1.4. Notice that this operation requires that the first string preceding the cursor is empty and the list contents are in the remaining string before the call. After the call is completed, the first string contains the reverse of the original second string and second string is now empty.

<table>
<thead>
<tr>
<th>Operation Call</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse_List</td>
<td>![ BEFORE IMAGE ]</td>
<td>![ AFTER IMAGE ]</td>
</tr>
</tbody>
</table>

Figure 1.4: Illustration of `Reverse_List`

The `List` reversal code in Listing 1.2 is written in an integrated specification and programming language called RESOLVE [87], which has some similarities to other object-oriented program-
ming languages such as C++ or Java. Here, the List object is treated as any other parameter in an operation call, so `Advance(s)` is equivalent to `s.Advance()` in Java. Reasoning about the Reverse_List code listed below is straightforward using the mathematical abstraction for List as well as specifications for the Advance, Insert, Is_Empty and Remove operations. However, each implementation of List will also need to be verified, but can be done separately. Recursion is used in the implementation below, but reasoning can also be done on an iterative implementation of List reversal. An iterative solution is discussed in Chapter 3.

Listing 1.2: List Reversal (Without Explicit References)

```plaintext
Recursive Procedure Reverse_List(updates s: List);
  Var temp : Entry;
  if ( not Is_Empty(s) ) then
    Remove(temp, s);
    Reverse_List(s);
    Insert(temp, s);
    Advance(s);
  end;
end Reverse_List;
```

To illustrate how the code in Listing 1.2 works, Figure 1.5 shows a tracing table on a sample Integer list with the input value `<|<10, 20, 30|>`. The expected output is `<30, 20, 10|>`. Since `temp` in this case is an Integer, its initial value is 0 and will be subsequently updated by the Remove and Insert operations. When tracing through the code using a natural reasoning approach, the Facts column records the values of `s` and `temp` in each State [45].

There are a couple of interesting points that may not be clear from the tracing table. First, before the code can safely execute Remove, it must ensure that `s` is not empty. The if statement guarantees this. Therefore, Remove can be safely called and executed. Second, rather than unrolling recursion, tracing of the call to Reverse_List uses the specification of Reverse_List to assume the values in `s` have been put in reverse order. Third, the list that is supplied to the recursive call is strictly shorter than the initial list, ensuring recursion will terminate.
Figure 1.5: Tracing of the Code in Listing 1.2

Figure 1.6 depicts how modular reasoning for this implementation uses the specifications of List operations and Reverse_List operation. The dotted line in the middle of the figure separates mathematical specifications from programming implementations. Even if the underlying implementation of the List abstraction is based on linked references, the references are hidden in the List implementation and do not show up in Reverse_List. It’s specification-based reasoning that allows us to swap different implementations of List without having to reason about the correctness of the code for Reverse_List again. However, each implementation of List must be verified using the List specification and specifications of components reused in their underlying implementations. The reasoning of Linked List Implementation may use a specification that captures (acyclic) reference behavior, while reasoning of Fixed Array Implementation and Dynamic Array Implementation may use array specifications. In any case, once a component has been verified, it can be used to build other components, and reasoning about the larger component can be done using only the specifications of the previously verified components, thereby promoting modular verification.
1.1.1 Clean Semantics

In order for one to reason through the code as we have done in Listing 1.2, the language in question must have clean semantics. There are two properties of a clean language: it’s variable-based and effect-restricted [61]. Clean semantics ensure that the specifications only deal with abstract values of variables (not their internal references) that are accessible in the current state of the operation. Any operation call’s effect is restricted to the explicit parameter variables and any global variables declared to be affected thereby avoiding side-effects. A programming language with clean semantics where one can directly reason without involving references is the topic of [60].

1.1.2 Automated Verification

For the code shown in Listing 1.2 (or an equivalent implementation) the RESOLVE compiler has performed automated verification [1][42][87][91]. Using the mathematical theories that have been designed to be automation-friendly, the programmer provides behavioral specifications and additional correctness justifications to the code. The automated reasoning system generates verification conditions (VCs) that are both necessary and sufficient to prove the code’s correctness and an automated prover attempts to prove these VCs. A more in depth description of the RESOLVE automated reasoning system can be found in Chapter 3.
1.2 Reasoning about Reference Behavior Explicitly

Ideally if all objects were passed by value rather than by reference, then modular verification wouldn’t have to be concerned about references “leaking” to other components, but there are numerous algorithms and operations that simply cannot avoid using references, including implementations such as those underlying the list abstraction discussed in the last section. Therefore, a mechanism to reason about references is needed. Researchers have proposed using specialized logics (e.g., separation logic and region logic) to reason about code involving references [35]. This dissertation explores an alternative solution to the problem. The purpose of this subsection is to explain the problem and motivate the work presented in this dissertation.

Suppose that instead of using a list abstraction, an implementation with explicit references is used. This implementation can be viewed as a collection of nodes linked together and manipulated via references $i$, $j$ and $k$ to reverse the linked nodes. Initially, $i$ points to the starting node, $j$ points to a special nil reference and $k$ points to some location in memory. The next object inside the node points to the next reference in the chain.

Listing 1.3: List Reversal (Explicit References)

```java
j = nil;
while (i ≠ nil) {
  k = i.next;
  i.next = j;
  j = i;
  i = k;
}
```

Since there are no mathematical abstractions, the reasoning involves the references $i$, $j$ and $k$ as shown in Figure 1.7. Indeed, the references themselves are aliases to the original node locations created outside the current scope, and thus reasoning would need to ensure that other spaces in memory were not inadvertently modified. Furthermore, the reasoning for a piece of client code that uses Listing 1.3 may need to know the implementation details, thereby increasing coupling.

There are difficulties and challenges in reasoning about components with references and [34] provides a list of principles that must occur in the reasoning. However, the key challenge is reasoning about potential aliased references to mutable objects, i.e., objects whose values may be modified [97]. When encountering implementations such as the one shown in Listing 1.3, previous research efforts either use separation logic or define a frame rule to capture parts of the heap. Separation
logic is a specialized logic that allows specifications to state how to partition the heap into disjoint sections [83]. Modifications to the heap can only happen on disjoint sections. Dynamic frames and region logic define a region of the heap and use global states to define accessibility [6][53]. Chapter 2 contains a complete overview of these techniques.

From the two examples above, it is not surprising that specification and use of data abstractions ease reasoning and makes the approach outlined in Section 1.1 desirable, wherever such direct reasoning is possible [60][87][88][97].

Modular reasoning for a code based on a List abstraction, such as the one presented in Listing 1.2, does not require reasoning about references. However, the references in Listing 1.3 are part of a linked implementation of List, illustrating that reasoning about reference behavior ultimately cannot be avoided, at least for a core set of components at the bottom of the hierarchy that need to be built from scratch. Formal specification and subsequent reasoning of unavoidable references in component realizations are the topics of this dissertation.

1.3 Problem Statement

Reasoning about components that use a shared state implementation should not break modularity. The solution must work seamlessly in conjunction with automated and direct reasoning about components that don’t involve a shared state. Using concepts with reference behavior implemented using a shared state, this research will address the following specification and verification problems:
• **Shared State Specification**: The specification language should be augmented to capture a shared state through the use of global variables. This new construct should allow implementations to share resources and to define and use components with sharing, with and without explicit reference behavior. There needs to be a way to specify collections of objects and document how they might affect each other. For reference-hiding data abstractions such as a community of lists (or any other object collection) that is bounded by a shared maximum capacity, any changes that increases or decreases the remaining capacity will affect calls to other instantiated list objects. For concepts with reference behavior, it will be necessary to capture reference behavior to reason about what references are accessible and how accessibility of objects in a collection can be affected by the different operations.

• **Implementation Annotations**: In order to show correctness for code with shared representation, additional assertions need to be added to state properties about the shared state, such as correspondence and internal effects. For components with reference behavior, implicit or explicit assertions may be needed to indicate, for example, when an object representation is modified, representations of other shared objects are not compromised.

• **Verification System Extensions**: In light of the above key enhancements to the specification language, the automated verification system and its underlying formal proof system will need capabilities beyond the existing machinery to reason about code correctness. This dissertation is only concerned with automated generation of verification conditions (VCs). Automated proof of the VCs themselves is left as a future goal.

• **New Library Components**: New shared concept specifications must be defined to capture explicit reference behavior and facilitate automation in verification. Such specifications and verification of implementations based on those specifications will serve as a useful means of evaluating the proposed solutions to the problems listed above.

### 1.4 Research Approach and Evaluation

This research seeks to extend the RESOLVE language to address modular reasoning for components with a shared state. In order to preserve clean semantics, the specification must be able to capture the effects of explicit parameter variables and any global variables that may be affected.
Currently, the RESOLVE specifications only capture the effects of explicit parameter variables. There is no construct that allows global variables to be declared as affecting an operation, nor is there any mechanism to reason about modifications to the global state. In order to reason about changes to the shared global state, there must be a way to talk about how it affects any other objects that use this shared implementation. If no changes are made, there must also be a way to state that other objects are not affected.

The ideas of this research can be applied to any components with a shared state. However, since components that capture and use reference behavior require solutions to the problems proposed in the previous section, such components will be used to illustrate the research approach to addressing the shared state problem. To illustrate how this might occur, we introduce Ultimately Void Referencing Template (UVRT, for short), which is a component that encapsulates acyclic reference behavior. The global state of UVRT is shared among objects that use it, and its specification will need to ensure that operations affect shared objects only as specified. The specifications also provide a notion of accessibility, which is used to both to restrict the references from forming a cycle and to ensure that there are no side-effects that inadvertently affect other allocated objects of the same type. Moreover, a component that uses UVRT must be able to reason automatically that no “dangling references” are present and all references are “garbage collected” when they are no longer accessible.

Although there has been prior work in RESOLVE literature on components with reference behavior, in principle [49][63], UVRT is a formal specification that has been designed with automated verification as a goal.

While the current VC generator has served as a useful prototype of the formal proof system, major updates are needed to make automation for components with shared state possible. There is a need to add additional proof rules to address the Shared State Specification. It is also necessary that RESOLVE include necessary mathematical definitions and theories to support formal specifications that allow programmers to specify that clean semantics are preserved.

The VC generator must also be upgraded to handle generating VCs for nested function calls and avoid the need to separate each call and store the results in temporary variables. Lastly, there is a need to add additional simplification steps in order to reduce the number of givens that are needed to prove a VC. All these changes are done in order to automatically reason about components with reference behavior, because the assertions are notoriously complex.
It is possible that automated verification might require a component implemented using a shared global state to include additional specifications to help the automated prover. For example, the code for Insert for a link-based implementation of List must ensure that no other accessible lists are modified inadvertently. Since this post-condition only occurs in the link-based implementation, it is an additional requirement to the ensures clause stated in the operation specifications. The intricacies of this internal ensures clause will be addressed by this research.

Overall, this research will lead to the development of solutions to the problems discussed in the previous section. The solutions will be evaluated using a set of components with reference behavior. The complexity of reference behavior is such that in general, they should be avoided wherever possible through clean language design. However, the language would not be complete if there isn’t a way to specify and safely reason about components that use a shared global state, because ultimately at the lowest levels reference behavior is necessary to implement a class of implementations.

1.4.1 Contributions

This research aims to make the following contributions:

The first contribution is to specify concepts with shared state. Although this research motivates the ideas using concepts with explicit reference behavior, the new specification constructs allows any concepts with shared states to be specified.

The next contribution is for development of annotated implementations with shared state so that they are amenable to automated verification.

Third, the verification system will be extended to include simplification capabilities and new formal proof rules to reason about the shared state. All this is done as a prelude to enable proving of non-trivial assertions automatically, though this thesis is concerned with generating VCs only.

Lastly, \(\text{UVRT}\) will allow a core set of link-based components to be built from scratch. Since \(\text{UVRT}\) is designed to be automation-friendly, these components can be automatically verified.

Experimentation and evaluation concern VC generation for a host of realizations with and without explicit reference behavior.
1.5 Thesis Statement

It is possible to develop a uniform specification and verification machinery towards automated verification of object-based components with (and without) reference behavior, using standard mathematical logic.

1.6 Dissertation Organization

The chapters for this dissertation are as follows: Chapter 2 contains an overview of the different verification efforts for code involving reference behavior. Chapter 3 describes the RESOLVE language and prior work in more detail. The remaining chapters illustrate the proposed proof system and verification machinery using a variety of examples, but the formal proof rules are left to an Appendix. Some extensions to the formal verification system are presented in Chapter 4. A data abstraction with shared state and a discussion of how to reason about it is given in Chapter 5. Chapter 6 introduces a concept to capture acyclic reference behavior that is designed to be automation friendly. Chapter 7 presents an evaluation through the specification and verification of (shared) concept realizations. Chapter 8 summarizes our findings and includes a list of future research directions.
Chapter 2

Verification Background and Related Work

This chapter summarizes specification and verification work related to this dissertation.

2.1 General Verification Background

The idea of formal reasoning is often credited to Robert Floyd. In addition to using flowcharts to illustrate program execution, he proposed the use of propositions within each connection in a flowchart to indicate each condition that must be satisfied in order to transition between states [37]. In 1969, Tony Hoare proposed a set of axiomatic logic rules that can be used to reason about program correctness [46]. Rules are defined as triplets of the form: \( \{P\} C \{Q\} \), where \( P \) is the precondition, \( Q \) is the post-condition and \( C \) is a command or statement. The standard Hoare logic can be applied repeatedly to a sequence of commands to establish whether or not an assertion \( Q \) is satisfied at the end of a sequence, given an initial condition \( P \) before the statements to ensure partial correctness (i.e., code is correct if it terminates); Termination would need to be proved separately.

While the basic principles of formal verification for simple programs have been known for over 50 years, the task of automating verification for software composed from components continues to remain a grand challenge [47]. The verifying compiler grand challenge envisions checking the correctness of the implementations with respect to their specifications. Several of these efforts are
discussed in [58] and we summarize some of them here.

Jahob is a system for verifying programs written in a subset of Java [64][104] using Isabelle as its specification language [75]. In order to prove the generated VCs, Jahob uses an *integrated reasoning* approach where both interactive proof assistants and automated provers are used in conjunction. Examples of interactive provers include Isabelle, Coq [15] and PVS [79]. Some of the automated reasoning systems that can be interfaced with Jahob include MONA [57], SPASS [98] and Z3 [30].

Spec# is a programming/specification language built from an extension of C# [10]. Compiled Spec# byte-code is then translated into an intermediary language BoogiePL [9], which produces VCs. These VCs can then be dispatched to automatic theorem provers such as Simplify [31], Zap [3] and Z3. ACL2 is a dialect of Common Lisp [56] designed to support both software and hardware verification. Its automated prover was designed to be “industrial strength” and has been applied in a variety of settings.

While this research is in the area of formal reasoning about full behavioral specification [43][62], model checking is an alternative approach for reasoning and automation [26]. This approach relies on specifying a set of hardware and/or software properties and automatically checking to see if a model of a *finite-state* system satisfies the properties. Properties that are often checked include null pointer dereferences, buffer overflows and array index out of bounds [25][32]. Model checking efforts include Java Path Finder [95], PRISM [65], SLAM [4], SPIN [50] and UPAAL [66].

The rest of this chapter summarizes various techniques for reasoning about programs that contain references. In Section 2.2, we present the ideas of separation logic. This is a well known technique in the formal reasoning community and has been applied in several verification tools. Sections 2.3 and 2.4 present alternatives to separation logic by addressing the *frame problem*. Section 2.6 contains other related efforts.

### 2.2 Separation Logic

Separation logic is an extension of Hoare’s logical rules to address properties about the heap [83]. Like the name suggests, if the heap can be safely separated into disjoint sections and each reference only operates on a disjoint section, then it makes it possible to formally reason about the program. Recall from the example in Listing 1.3 from Section 1.2 that separation logic can be used
to formally reason about code. We now illustrate how this is done based on the content from [83].

We begin by applying the standard Hoare triple: \( \{ P \} \ C \{ Q \} \). In order to specify the pre- and post-conditions of list reversal, we define a linked list predicate \( \text{list} \ \alpha \ \iota \) by induction on the length of \( \alpha \). If the list with reference \( \iota \) is empty, then \( \iota \) points to \( \text{nil} \). If the list is non-empty with \( \iota \) pointing to the beginning of the list and the list contains the values \([a_0, a_1, \ldots, a_n]\), then there exists a reference \( \iota' \) such that \( \iota \) points to \( a_0 \) and \( \iota' \), and \( \text{list} [a_1, \ldots, a_n] \ \iota' \) is true. Using this definition, the pre- and post-condition for the list reversal example can be expressed as follows:

Listing 2.1: List Reversal (Hoare Logic)

\[
\{ \text{list} \ \alpha_0 \ \iota \} \\
\quad \quad j = \text{nil}; \quad \quad \text{while} \quad (i \neq \text{nil}) \quad \{
\quad \quad \quad k = i.\text{next}; \\
\quad \quad \quad i.\text{next} = j; \\
\quad \quad \quad j = i; \\
\quad \quad \quad i = k; \\
\quad \quad \} \\
\{ \text{list} \ \text{rev}(\alpha_0) \ \iota' \}
\]

The invariant of the loop must state a property that holds at the beginning of each iteration. In this case, we want to state that the reverse of the initial list \( \alpha_0 \) can be obtained from \( i \)'s sequence \( \alpha \) and \( j \)'s sequence \( \beta \). The invariant can be stated as follows (Note: \( \cdot \) here means concatenation):

\[
\exists \alpha, \beta. \ \text{list} \ \alpha \ \iota \ \land \ \text{list} \ \beta \ \iota' \ \land \ \text{rev}(\alpha_0) = \text{rev}(\alpha) \cdot \beta
\]

However, we want to state that \( i \) and \( j \) are disjoint, meaning that they are not sharing any references. Therefore, we will need to strengthen the invariant by stating that \( \text{nil} \) is the only thing reachable from \( i \) and \( j \) respectively.

\[
(\exists \alpha, \beta. \ \text{list} \ \alpha \ \iota \ \land \ \text{list} \ \beta \ \iota' \ \land \ \text{rev}(\alpha_0) = \text{rev}(\alpha) \cdot \beta) \land \\
(\forall k. \ \text{reachable}(i,k) \ \land \ \text{reachable}(j,k) \ \Rightarrow \ k = \text{nil})
\]

However, if there were other lists in the heap, we would have to state that the only thing that is reachable from all these lists is \( \text{nil} \). Rather than doing this for all possible references in the heap, separation logic uses a \textit{separating conjunction} of the form \( \{ P \} * \{ Q \} \) to indicate that \( P \) and \( Q \) are from disjoint regions in the heap. Therefore, even if there is a list with a starting reference \( x \)
and a sequence of $\gamma$, the invariant can be written as:

$$(\exists \alpha, \beta. \text{list } \alpha \ i \ast \text{list } \beta \ j \ast \text{list } \gamma \ x ) \land \text{rev}(\alpha_0) = \text{rev}(\alpha) \cdot \beta$$

Since list $\gamma \ x$ is not being modified in the list reversal code, we can simply use the separating conjunction to state that the pre-condition is $\{\text{list } \alpha_0 \ i \ast \text{list } \gamma \ x\}$ and the post-condition is $\{\text{list } \text{rev}(\alpha_0) \ j \ast \text{list } \gamma \ x\}$. This is also known as the frame rule and can be formally stated as:

$$\frac{\{P\} \ C \ {Q}\} \ {P \ast R}\ C \ {Q \ast R}$$

In this case since $P$ and $Q$ are both disjoint from $R$, the frame rule allows us to show that if $P$ is met, then $Q$ must be true after executing $C$.

Several formal reasoning systems apply separation logic for verifying programs with references. Coq is a mathematical system that can be used to state and interactively proof mathematical properties and assertions [15]. Bedrock is a framework that is used to verify low-level programs written in Coq [23]. In Bedrock, function specifications are written in terms of reference implementations in a pure functional language, therefore enabling the reasoning using separation logic to be computational. This means that a lot of the quantifiers used in the specifications can be replaced with the execution of programs written using the specification language. Most of the verification condition generation can be automated.

Coq has been able to verify high-order imperative programs [24] and provide a checkable proof to certify the FSCQ file system [22]. Similar to Coq, VeriFast has used separation logic in a custom specification language to verify Java and C programs. An example of this can be found in [51][52].

While separation logic can be used to formally reason about programs involving reference behavior, it is suitable typically only when all references are captured explicitly. This means that it can only be used to specify programs with explicit references manipulation, like that shown in Listing 2.1. When references are hidden, separation logic isn’t able to formally reason about these references. There has been attempts to address information hiding using separation logic [76][77][78], however it cannot be generalized to encompass many object instances.

Another problem with separation logic is that it is a specialized logic and cannot be fully
automated by the theorem provers. This means that theorem provers must either be extended to
include this new form of logic or the specifications needs to be translated from separation logic into
first-order logic. Smallfoot attempts to automate the proofs for loop-free code by applying symbolic
execution with separation logic [13][14]. This tool depends on rolling and unrolling inductive defini-
tions and the ability to make inferences about parts of the heap that are not modified. Other efforts
in this area can be found in: [16][18][19][81][82][103][104].

2.3 Dynamic Frames

When using modular reasoning, only the specifications of reused components are available
to the client and the internal representation is hidden. However, the specifications need to be strong
enough to assert that there is no reference aliasing without knowing about all the implementation
details. Dynamic frames are designed to addresses the frame problem for shared and encapsulated
references [53][54][55].

In dynamic frames the idea of an infinite set of locations, Loc, with each subset being a
“region” is introduced. For each of the locations in a region, there is a state σ that provides a
mapping between a location to its value. Σ is used to denote the set of all states. For a given
location it is either allocated/used (Used = in σ) or unallocated (Loc \ Used).

In the presence of references, each module defines a region. An expression E is framed
by this region if E only depends on the locations in this region. If all values corresponding to the
locations in the region are not modified, then the specifications can simply state that E is unchanged.
A region can also self-frame itself and state that the values corresponding to the locations in the
region are only modified by the location/value mapping in the frame.

In dynamic frames, a footprint is the set of fields that a method can modify. Dynamic
frames provide two new specification constructs: a reads mode to indicate a region is only being
read and a modifies mode to indicate the operation modifies the region. For each of the methods, it
either reads or modifies a footprint. The specification can also introduce additional invariants that
must hold as additional requirements. For example, the list object’s size is always greater or equal
to 0.
Listing 2.2: List Interface with Dynamic Frames from [99]

```java
public interface List {
    //@
    // public model instance \
    // locset footprint;
    //@
    // public accessible \inv : footprint;
    //@
    // public accessible footprint: footprint;
    //@
    public invariant 0 <= size();
    //@
    public accessible footprint;
    //@
    ensures \result == size();
    //@
    public /*@pure@ */ int size();
    //@
    public normal behaviour
    @ assignable footprint;
    @ requires 0 <= index && index < size();
    @ assignable footprint;
    @ ensures \result == get(index);
    @ also public exceptional behaviour
    @ requires index < 0 || size() <= index;
    @ signals_only IndexOutOfBoundsException;
    @
    public /*@pure@ */ Object get(int index);
    //@
    normal behaviour
    @ assignable footprint;
    @ ensures \result == (\exists int i; 0 <= i && i < size(); get(i) == o);
    @
    public /*@pure@ */ boolean contains(Object o);
    //@
    normal behaviour
    @ assignable footprint;
    @ ensures size() == \old(size()) + 1 && get(size() - 1) == o;
    @ assignable footprint;
    @ ensures \forall int i; 0 <= i && i < size() - 1; get(i) == \old(get(i));
    @ assignable footprint;
    @
    public void add(Object o);
}
```

Listing 2.2 is an example of dynamic frame written in JML* taken from [99]. JML* is an extension of Java Modeling Language (JML), a specification language built on top of Java [67][68]. JML specifications are written based on Java expressions, a decision influenced by Eiffel [73]. Despite this fact, JML uses the model-based specifications in Larch-style [40]. Specifications and invariants in JML are written in special annotation comments, allowing the Java code to be compiled using any compiler. In the specification, the behaviors of the operations are explained using assertions just above the operation heading. However, JML specifications are executable Java methods, therefore they need to be declared as pure to ensure that they do not contain side-effects.
In this example, the specification first defines a model instance for the location set called \textit{footprint}. The \textit{accessible} syntax defines that both \texttt{inv} and \textit{footprint} depend at most on the locations in \textit{footprint}. The size must be 0 or positive and is stated as an invariant. The invariant must hold before and after each method execution.

There are four methods shown in Listing 2.2: \texttt{size}, \texttt{get}, \texttt{contains} and \texttt{add}. All four methods define the normal behavior; Only the \texttt{get} method has an exception behavior where an index out of bounds exception can be thrown. Note that the methods \texttt{size}, \texttt{get} and \texttt{contains} only read the \textit{footprint} and do not make any modifications.

The standard set operators such as \texttt{\textbackslash intersect}, \texttt{\textbackslash set\_minus}, \texttt{\textbackslash set\_union}, \texttt{\textbackslash subset} and \texttt{\textbackslash disjoint} can be used on expressions of type \texttt{\textbackslash locset}. Although not shown in the example, JML* can also write specification relating a field $f$ of the object $o$ as a singleton set \texttt{\textbackslash singleton}(o,$f$) with another set.

The method \texttt{add} introduces the keyword \textit{assignable}. This means that the \textit{footprint} is being modified by the \texttt{add} method. In this case the new size of the list is the old size of the list + 1 and the new element is added at the last position. All the other elements in the list are not modified. This method further states that the object added is not an aliased reference to another existing object in the \textit{footprint}. If any other aliasing is present, the frame property might not hold.

In dynamic frames, a \textit{swinging pivots operator} (defined in JML* using the \texttt{\textbackslash new\_elements\_fresh} operator) states that if there is a location in the post-condition that wasn’t there before, it must be allocated during the method execution. The frame property holds for the \texttt{add} method, because the new item was not in \textit{footprint} before the call, got allocated during the method execution and is added to \textit{footprint} at the end of the method execution.

Similar to JML*, both Dafny and KeY uses dynamic frames in their reference specifications [21][70][86]. Dafny is an integrated programming and specification language [70][71]. The specifications are only used to verify the code and are omitted when producing executable code. Dafny specifications are translated into an intermediary language BoogiePL [9] and the generated VCs are sent to Z3 for automated proving [30]. KeY is a verifier that translates JML specifications to its inner representation (dynamic logic) to generate fully automated proofs [11][12]. This tool also has features for an interactive mode where the user can manually perform the proof.

Dynamic frames use higher order logic and as a result, are able to quantify over mathematical functions. Thus, in order to achieve automation, there must be a translation from higher-order to
first-order logic to use an automated prover’s specialized decision procedures. These efforts can be found in [89][90].

2.4 Region Logic

Region logic is similar to dynamic frames where the specification defines a region and global states [6]. However, rather than using higher-order logic, region logic formulates the specifications using only first-order logic in hopes of using automated provers.

A region expression $G$ of type $\texttt{rgn}$ is used to define a region in the heap. There are two special expressions: $\texttt{emp}$ that denotes the empty region and $\texttt{alloc}$ that denotes the allocated regions. The expression $\{E\}$ can be used to either represent a singleton set if $E$ is a reference or the empty set. Similar to dynamic frames, the region being modified is represented using the keyword $\texttt{wr}$ or write effects and a region that is simply read is represented using the keyword $\texttt{rd}$.

In region logic, the image expression $G'f$ (the reading of this is “$G$’s image under $f$”) indicates that the region $G$ depends on the type of $f$. If $f$ is a reference, then this returns the set of allocated references in $G$ that can be reached from $f$. Nevertheless, if $f$ is a region, then $G'f$ returns the union of all of $f$’s image expressions.

Since regions are mathematical sets, the standard set operators can be used to provide additional assertions. For example, $G \subseteq G'$ indicates that the region $G$ is a subset of $G'$. Similarly, the expression $G \cap G' \subseteq \texttt{emp}$ (or the short hand $G\#G'$) indicate that the empty region is the intersection of the contents of sets $G$ and $G'$. Most importantly, reachability can be written as $G'f \subseteq G$ (read as “$G$ is $f$-closed”) to express that for a reference $o \in G$, all the references that can be reached from $o$ by following $f$ must be in $G$ as well. If there is a root node root, then the assertion $\texttt{root} \in G \land G'f \subseteq G$ ensures that all allocated references that are reachable from root by following $f$ are in $G$.

The specifications written in region logic can be translated in to BoogiePL to generate VC and subsequently dispatched to Z3 for automated verification [5]. However, frequent usage of quantifiers in the specifications may present difficulty for automated provers. As such, there is a large ongoing effort to create specialized decision procedures capable of dealing with the quantifier free fragments of region logic using partial function and array theory [84][85]. This extension of the automated prover can reason about sets, while leaving the elements of the set to the general prover.
There are numerous other efforts that are similar or are extensions to those discussed in the sections above. Regions can be formulated using a simple, type based frame rule [92]. Others have attempted to use regions to specify parallel programs [17]. A recent effort presents a new logic for unifying separation and region logic for specifying programs with shared mutable data with mixture of styles [7][8].

2.5 An Approach Using RESOLVE Principles

When writing specifications, a variable’s value could have multiple meanings. For languages that have value semantics such as RESOLVE, a variable’s abstract value refers to a value from the mathematical abstraction. A variable’s concrete value refers to a variable’s internal data representation, but is not exposed to the client. For a language with reference semantics such as C or Java, a variable’s value could refer to it’s referenced memory address or to the referenced object’s value.

A variable’s value is important when implementing methods such as contains for Java List [101]. KeY’s verified implementation checks whether or not a reference exists in the List, while the default implementation in java.util.List depends on the object’s equals method implementation. Comparing memory address values has limited usability; Most of the time, the client is interested in finding objects that have the same value, which requires a proper implementation of equals. Any attempt to improve KeY’s implementation would require knowledge of objects being stored and the information being stored. Doing so would require expanding the framed region, which will decrease the ability to show that two frames are disjoint [101].

An alternative approach to apply RESOLVE principles\footnote{An overview of RESOLVE is presented in the following chapter.} to Java is discussed in [102]. The contract specifications are written using JavaDoc-style tags to provide a mathematical abstraction, specification parameter modes and the pre- and post-condition for a method. This work introduces a novel specification parameter mode (depletes) and encumbers relation notation for advertising aliasing. The depletes parameter mode indicates that the implementation of the method might create and store an alias in its internal representation. When a variable is passed to a method with this mode, it is no longer readable by the client after the call and behaves like an uninitialized Java variable. Any attempts to access the previously referenced object will result in an error. An encumbers notation indicates that the return value exposes part of the object’s inner represent-
tation. This is useful to advertise that the return value is an alias to an object stored inside an internal representation. Java programs annotated with these constructs provide a way to generate verification conditions in terms of abstract values and can be verified using a RESOLVE automated prover such as the one given in [1].

2.6 Other Efforts

Modular verification of components with shared realizations is a hard problem to solve. The authors of [34] present principles to achieve modular reasoning and outline the general difficulties and challenges in order to achieve this goal. Perhaps one of the most challenging problems is aliasing, where there are two or more references that point to the same mutable object. There have been several different proposals to address this problem and Hatcliff, et al. and Hogg, et al. both have presented summaries of these ideas in [44][48].

Nonetheless, all the efforts discussed can deal with references within a component, but do not present a solution for the references that are cross-boundary. Both Dafny and VeriFast have attempted to verify a generic map component with reference type parameters proposed in [20]. However, the map keys are references, which means that the user could inadvertently modify the map using an alias reference to the key. This creates the need to do cross-boundary inference, thus breaking modular reasoning. A potential solution is to make both the key and value objects immutable, but this results in a component that has limited usability.

The verification system must provide a way to deal with aliasing across components when the programming languages allow references to be aliased as stated by Leavens, et al. and O’Hearn, et al. [69][77]. The problems with the situation of when references are aliased and modified across components is also noted in [35].

Why3 is a system that takes in language specifications written in programming languages such as Ada, C and Java and converts them into an intermediary language, WhyML (a ML-like specification language) [36]. Using this output combined with logical declarations from theory files, Why3 can produce VCs that can be translated to a selection of both automated provers and proof assistants. The intent of Why3 designers is to facilitate verification of reference behavior without the use of separation logic [80]. However, there are no published results at the time of writing this dissertation.
Chapter 3

RESOLVE Background and Prior Work

The previous chapter discussed various automated and non-automated verification efforts, including efforts to deal with components with reference behavior. In preparation for a new RESOLVE approach to this problem, this chapter provides the necessary background information on the RESOLVE language and summarizes what has already been achieved relative to automation and verification of components with and without reference behavior.

3.1 Introduction to RESOLVE Specifications

RESOLVE is an imperative, object-oriented based language that has integrated executable code with mathematical specifications and extensible mathematical theories [87]. The goal of RESOLVE is to enable formal modular specification and verification of components that are designed to be efficient and reusable. The language has clean semantics and allows extensible (and reusable) mathematical theories to be used in the specifications.

As a way to illustrate these ideas, we return to the list reversal example in Section 1.2 (Listing 1.2) that uses a List abstraction to specify its correctness. For this example, we assume that lists are globally bounded by the system’s memory and do not contain a user supplied maximum size. Section 5 discusses an example with bounds. In order to better explain the specification of a
List abstraction, the concept (or interface specification) has been broken down into smaller sections. The complete specification is given in Appendix B.

**Listing 3.1: A Globally-Bounded List Concept**

```plaintext
Concept Globally_Bounded_List_Template(type Entry);
    uses String_Theory;

...
end Globally_Bounded_List_Template;
```

Notice that Listing 3.1 is generic and that the actual type for the entry elements is not known until the Concept is instantiated. To achieve this generality, the Concept has a parameter, Entry. Entry is a user-supplied generic type. The uses line is an import statement to String_Theory in the RESOLVE math library and it contains a formal definition for mathematical strings. Strings are used to specify the List abstraction and will be used in the verification process.

The Type Family clause below specifies that the abstract generic type List is modeled mathematically by a Cartesian product of two Entry strings: Prec and Rem. The Prec string contains all the elements of type Entry in front of the current cursor position, while those after the cursor are in the Rem string. The exemplar, P, that is chosen as a short representative in the initialization ensures clause, is used to assert that both the Prec and Rem strings are initially empty.

**Listing 3.2: Mathematical Model for List**

```plaintext
Type Family List is modeled by Cart_Prod
    Prec, Rem: Str(Entry);
end;
exemplar P;
initialization
    ensures P.Prec = Empty_String and P.Rem = Empty_String;
end;
```

Using the mathematical model for the List type, the Concept defines operations that allow us to access and modify a List object. Although the List concept provides multiple operations, only the operations used in this section are presented; Others are omitted for brevity.

The pre-condition (requires clause) and post-condition (ensures clauses) are strictly mathematical and they explain the behavior of each operation. Together they create a contract that must be adhered to when implementing or using the operation. When specifying a List object
such as \( P \), the specification always refers to the mathematical abstraction of \( P \), which is a Cartesian product of two mathematical strings.

Listing 3.3: Selected Operations for List

```plaintext
Operation Insert( alters New_Entry: Entry; updates P: List );
    ensures P.Prec = #P.Prec and P.Rem = <New_Entry> o #P.Rem;

Operation Remove( replaces Entry_Removed: Entry; updates P: List );
    requires not (P.Rem = Empty_String);
    ensures P.Prec = #P.Prec and
        Entry_Removed = DeString(Prt_Btwn(0, 1, #P.Rem)) and
        P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);

Operation Advance( updates P: List );
    requires not (P.Rem = Empty_String);
    ensures P.Prec = #P.Prec o Prt_Btwn(0, 1, #P.Rem) and
        P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);

Operation Advance_to_End( updates P: List );
    ensures P.Prec = #P.Prec o #P.Rem and P.Rem = Empty_String;

Operation Is_Rem_Empty( restores P: List ); Boolean;
    ensures Is_Rem_Empty = ( P.Rem = Empty_String );
```

The Insert operation concatenates New_Entry to the beginning of the incoming Rem string. Since the concatenation operator operates on two strings, the \(<...>\) notation casts a single entry to a singleton-string. The parameters for Insert use specification parameter modes, which explicitly state the effect of the operation on each parameter upon completion of the operation. The alters mode allows New_Entry to pass a meaningful value to the operation, but the value of it at the end of the operation is unspecified. The updates mode indicates that \( P \) had some meaningful value when passed to the operation and will be updated according to the ensures clause.

The Remove operation requires a List with a non-empty Rem string and removes the first entry from the incoming Rem string and returns it in Entry_Removed. The replaces mode is used to give Entry_Removed a meaningful value at the end of the call. Rather than requiring the parameter to be copied, using either the replaces mode or the alters mode discussed above, the specification can address the cross-boundary referencing problem by restricting effects of the parameters being passed to the local context [41]. The DeString function is the inverse of \(<...>\), where a singleton-string is cast to an entry. The Prec string of the parameter List is not modified.

Advance operation moves the first Entry from \( P \)'s Rem string to the end of \( P \)'s Prec string if the Rem string is not empty. The \#P in the ensures clause is used to specify the value of the list
At the beginning of the operation, while \( P \) represents the value of the list upon completion of the operation. In order to specify the changes to \( \text{Prec} \) and \( \text{Rem} \), the \textbf{ensures} clause uses the \texttt{Prt\_Btw} mathematical function that returns a substring of the original string over the specified interval. The \( \text{Prec} \) string is the incoming \( \text{Prec} \) string concatenated \((o)\) with the string between the interval \( 0 \) and \( 1 \) in the incoming \( \text{Rem} \) string, which is the string containing the first entry in \( \text{Rem} \). This will leave \( \text{Rem} \) with the string between \( 1 \) and the length of the incoming \( \text{Rem} \) string \((\lfloor\ldots\rfloor \text{ function returns the length of a string})\). The figure below illustrates the effects of \texttt{Prt\_Btw} with an incoming \( \text{Rem} \) string of \(<10, 20, 30>\).

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\text{\#Rem} & <10, 20, 30> & \text{Prt\_Btw}(0, 1, \#\text{Rem}) = <10> & \text{Prt\_Btw}(1, \#\text{Rem}, \#\text{Rem}) = <20, 30>
\end{array}
\]

Figure 3.1: Applying \texttt{Prt\_Btw}

\texttt{Advance\_to\_End} updates the \( \text{Prec} \) string by appending the incoming \( \text{Rem} \) to the end of the incoming \( \text{Prec} \) and leaves the outgoing \( \text{Rem} \) as the \texttt{Empty\_String}. \texttt{Is\_Rem\_Empty} returns a \texttt{Boolean} to indicate whether or not the \( \text{Rem} \) string is empty. It uses the \texttt{restores} mode to indicate that the parameter \( P \)'s value will be the same as what it was before the operation call.

\[\text{3.2 Recent Work in Automated Verification of RESOLVE Components}\]

RESOLVE components based on from formal specifications, like the one provided in the previous section, are meant to be automatically verifiable. In order to achieve this goal, the RESOLVE Verifying Compiler has made numerous efforts towards solving the grand challenge proposed by Tony Hoare in 2003 [47][87]. The goal of this section is to use a \texttt{List} reversal extension and present the prior work in RESOLVE automation for two distinct implementations of the extension. An extension, which we call an \texttt{Enhancement}, to the \texttt{Concept} can be created to add secondary operations that are not in the \texttt{Concept} and can be implemented using the \texttt{Concept}'s primary operations. In this case, \texttt{List\_Reversal\_Capability} provides an operation to reverse a \texttt{List} object.
Listing 3.4: List Reversal Specifications

Enhancement List_Reversal_Capability for Globally_Bounded_List_Template;

Operation Reverse_List(updates P: List);
  requires P.Prec = Empty_String;
  ensures P.Prec = Reverse(#P.Rem) and P.Rem = Empty_String;
end List_Reversal_Capability;

The pre- and post-conditions for the operation Reverse_List use the mathematical type abstraction provided by Globally_Bounded_List_Template and do not depend on any of its implementations. For the parameter P, the specification requires that Prec must be empty and ensures that the Reverse_List operation will reverse the incoming Rem string. Any implementation of Reverse_List must adhere to the specifications, but can implement the behavior in any way, thereby allowing for a variety of implementations with different efficiency characteristics. The actual implementation is transparent to the users who use Reverse_List and the implementations details are not used when VCs are generated for user code.

The listing below is an extension of Listing 1.2 presented in Section 1.1. This Realization implements List_Reversal_Capability using Globally_Bounded_List_Template's operations. The Recursive keyword is used to denote a recursive operation. When an operation is recursive, a decreasing progress metric must be provided by the programmer to justify that the recursive code terminates. In this case, Reverse_List terminates when P's Rem string is empty. Other than a name change, Is_Rem_Empty is the same operation as Is_Empty from Listing 1.2.

Listing 3.5: Recursive List Reversal

Realization Recursive_List_Reversal_Realiz for List_Reversal_Capability of Globally_Bounded_List_Template;

Recursive Procedure Reverse_List(updates P: List);
  decreasing |P.Rem|;
  Var E: Entry;
  If ( not Is_Rem_Empty(P) ) then
    Remove (E, P);
    Reverse_List(P);
    Insert(E, P);
    Advance(P);
  end;
end Reverse_List;
end Recursive_List_Reversal_Realiz;
To formally reason about the correctness of the code in Listing 3.5, the verifying compiler uses specifications from `Globally_Bounded_List_Template` (Listing 3.3) and `Reverse_List` (Listing 3.4) along with the code and generates an intermediate representation. The compiler then applies individual proof rules to these intermediate representations to generate verification conditions (VCs) [42]. Subsequently, the VCs can be sent to the automated prover for automated verification [91]. This process is shown in Figure 3.2.

![Figure 3.2: Verification Pipeline Process of Listing 3.5](image)

The specifications in RESOLVE facilitate direct reasoning, where reasoning does not introduce or require reasoning about references [60]. Direct reasoning is possible because of clean semantics. As discussed in Section 1.1.1, a clean language requires two properties: variable-based and effect-restricted. During the reasoning process, the variables are viewed as a conceptually direct abstraction and not as a reference. When reasoning about an operation call, the effects of the call are expressed using explicit parameter variables and global variables. In the case of global variables, the operation must explicitly specify which global variables are affected.

The original prototype VC Generator [42] uses an extended version of the proof rules defined in [59]. During this process, the specification and code are combined into what is known as an assertive program. Initially, the assertive program (shown below) contains all the assertions that can be assumed and all assertions for items that need to be confirmed. The system then uses a set of proof rules to process the assertive code one statement at a time starting with the last statement before the confirm assertion at the end. The proof rule application continues until one single confirm assertion statement, equivalent to the correctness of the program, remains. This process is known as the goal-oriented approach and it attempts to minimize the number of VCs generated and the number of new variables introduced in hopes of discharging the proof automatically. The VC Generator has also been extended to include proof rules that generate VCs for performance [100].
Listing 3.6: Assertive Program used to Generate VCs for Listing 3.5

Assume true;
Assume (((min_int <= 0) and (1 <= max_int)) and (1 <= Last_Char_Num)) and (1 <= Max_Char_Str_Len));
Assume L.Prec = Empty_String;
Remember;
Var E : Entry;
Assume P_val = |L.Rem|;
If Not(Is_Rem_Empty(L)) then
    Remove(E, L);
    Reverse_List(L);
    Insert(E, L);
    Advance(L);
end;
Confirm (L.Prec = Reverse(#L.Rem) and L.Rem = Empty_String);

The compiler generates eight VCs that are necessary and sufficient to prove the correctness of Listing 3.5. The pre-condition is the responsibility of the caller, therefore three of the VCs come from the requires clauses of Remove, Reverse_List and Advance. Since there is an If statement, the code could either execute the if-statement block if the condition is evaluated to true or simply skip it. This means that both post-conditions must be evaluated for both possible paths, which results in four VCs for the post-condition of Reverse_List. Lastly, there must be a VC that establishes that the recursion terminates. As an example, the VC establishing the termination of Reverse_List is shown below:

Listing 3.7: Termination VC

Goal:

((1 + |Prt_Btwn(1, |P.Rem|, P.Rem)|) <= |P.Rem|)

Given:
1. (P'''''.Prec = P.Prec)
2. (E’ = DeString(Prt_Btwn(0, 1, P.Rem)))
3. not((P.Rem = Empty_String))
4. Entry.Is_Initial(E)
5. (P.Prec = Empty_String)
6. (min_int <= 0)
7. (1 <= max_int)
8. (1 <= Last_Char_Num)
9. (1 <= Max_Char_Str_Len)

From Given #3, we know that P.Rem is not empty and the length must be a positive integer value. There is a theorem in String_Theory that states that for any non-empty string S, 1 + Prt_Btwn(1, |S|, S) = |S| and will allow us to prove this VC.
Another possible implementation for Reverse_List is an iterative approach. An example iterative implementation is shown in Listing 3.8. Unlike the iterative implementation presented in Section 1.2, this implementation does not use explicit references.

This implementation uses a temporary List to assist in the list reversal process. The loop simply removes an entry from \( P \) and inserts it to \( Temp \) until \( P \) is empty. The maintaining clause is the loop invariant. It is a programmer-supplied assertion to aid in verification and it must be true at the beginning and at the end of each iteration of the loop. In this case, it states that \( Temp \)'s \( Prec \) does not change and the incoming \( P.Rem \) string is the reverse of \( Temp.Rem \) string concatenated with the current \( P.Rem \) string. Once all the entries have been added to \( Temp \) in the reverse order, this implementation simply moves all the content to \( Temp.Prec \) string by advancing the cursor to the end. Lastly, it swap the contents of \( Temp \) with \( P \) by using the swap operator (\( :=: \)) that is available on every type in RESOLVE. Swapping allows data to be efficiently moved, even in the case of objects, without introducing aliasing. A comparison of (deep and shallow) copying and swapping can be found in [41].

Listing 3.8: Iterative List Reversal

\[
\text{Realization Iterative\_List\_Reversal\_Realiz for List\_Flipping\_Capability of Globally\_Bounded\_List\_Template;}
\]

\[
\text{Procedure Reverse\_List(updates P: List);}
\]
\[
\text{Var Temp: List;}
\]
\[
\text{Var Next\_Entry: Entry;}
\]
\[
\text{While ( not Is\_Rem\_Empty(P) )}
\]
\[
\text{maintaining Temp.Prec = Empty\_String and Reverse(Temp.Rem) o P.Rem = #P.Rem;}
\]
\[
\text{decreasing |P.Rem|;}
\]
\[
\text{do}
\]
\[
\text{Remove(Next\_Entry, P);}
\]
\[
\text{Insert(Next\_Entry, Temp);}\]
\[
\text{end;}
\]
\[
\text{Advance\_to\_End(Temp);}\]
\[
\text{Temp :=: P;}
\]
\[
\text{end Reverse\_List;}
\]
\[
\text{end Iterative\_List\_Reversal\_Realiz;}
\]

The prototype automated prover attempts to prove all the generated VCs and produce proof results. It rewrites terms to find a relationship between the goal and the givens [91]. The proof process is sound (meaning no false assertions would be proved), but necessarily incomplete (meaning some true assertions may not be proved). Incompleteness arises not from the RESOLVE
proof system, but from number theory incompleteness and current state of the automated prover which is a research effort in progress.

An alternative prover that uses a congruence closure algorithm to establish equality is currently under development. Figure 3.3 shows this new prover in action using web integrated environment for the RESOLVE tool chain [27]. All the VCs are proven in a few seconds by this new automated prover.

![Figure 3.3: Verification of Listing 3.8 Using an Automated Prover](image)

Our sister group at Ohio State has developed an alternative tool chain. Their VC generator produces VCs using the *tabular approach*, where a VC is generated for each state that has a proof obligation [87]. This approach uses the principles outlined in [45]. For a given state, a VC may be generated to establish, for example, the pre-condition for the next operation call, a loop invariant, a progress metric or the post-condition of the current verifying operation. During this process, the VC generator uses the state number as a subscript with a variable name to denote its value at that state. Then it proceeds to simplify the VCs using restructuring rules that are theory independent.

The VCs can either be dispatched to SplitDecision, an in-house prover that uses limited built-
in mathematical theories, or translated into Dafny in order to use the Z3 prover [1][94]. SplitDecision uses specialized decision procedures for commonly used mathematical theories to reduce a VC to either true or false. However, in comparison to the term-rewrite prover outlined above, this prover requires new decision procedures to be written for any new mathematical domains used in the specifications. If a VC cannot be proven, SplitDecision would reduce the VC into a simpler form where proving can be done by hand or using a proof-assistant.

3.3 Previous Work in Formalizing Specifications to Capture Reference Behavior

The work presented in the previous sections highlights how RESOLVE handles automated verification of components that do not have explicit references, whereas the related work by others in Chapter 2 presented different approaches for verifying components with reference behavior. This section presents a summary of the prior work in verifying components with reference behavior using the RESOLVE specification language. All the work summarized in this section was done prior to the development of an automated verification system and the documents provide “principles” that are useful precursors to this research.

The concept Nilpotent_Template described in [49] mathematically models references using mathematical integers. There are two mathematical functions: label that maps a reference location to the object referenced by the location and target that takes a reference location and points to the next reference location. One key feature of Nilpotent_Template is that it constrains the specifications so that the references cannot create a cycle. As a result, this concept is suitable for construction of acyclic data structures such as Lists, Stacks, Queues and Trees.

On the other hand, the concept Linked_Location_Template given in [60][63] models references using a mathematical abstraction Location set. This abstraction avoids connecting references to numbers (which are not sufficiently abstract if arithmetic is not allowed on references) and allows reasoning to be done with reference locations using set operators. This concept also introduces a special location called Void that is never assigned. Type and operation specifications use Void in specifying accessibility and finalization. Linked_Location_Template is a generalized reference concept and can be used to implement programs with reference behavior. While Linked_Location_Template formalizes reference behavior, it makes extensive use of quantifiers,
for example, and is not directly amenable to automated verification. When using a concept such as the Linked_Location_Template, components with reference behavior can avoid reasoning about the heap explicitly, which is a problem in some of the other approaches.

In order to address the cross-boundary reference problem, the RESOLVE specification language uses a swapping paradigm to pass parameters [41]. This ensures that the client code does not retain a reference to the object being passed as a parameter. It also does not require deep copying or making the objects immutable [97]. At any given moment, each object in the heap has exactly one reference, thereby allowing specifications to be supplied only when the program alters a referenced object. This approach avoids the need to explicitly separate the heap and state every modification to the heap, as needed in some of the approaches in Chapter 2.
Chapter 4

Generation of Simplified VCs and Sequents

The Verification Condition (VC) generator produces conditions that are both necessary and sufficient to prove the correctness of an implementation with respect to participating specifications. This is the topic discussed in [42] and has been extended in [100]. The prototype VC generator and its underlying formal system both need substantial improvements to verify the kinds of non-trivial assertions that arise in verifying code involving reference behavior. As the mathematical modeling becomes more involved and non-trivial components are reused to create new components, for scalability, the improvements are necessary. One of the major contributions of this research is the introduction of the idea of using the sequent logic approach in generating VCs. This chapter introduces the techniques for generating simplified verification conditions in a sequent-based system as a prelude to enable proving of non-trivial assertions automatically.

4.1 Reduced Sequents

In the verification pipeline process for the RESOLVE verifying compiler (discussed in Section 3.2 and depicted in Figure 3.2), the VC generator backward sweeps through assertions and programming code to obtain a verification condition of correctness— a single conjuncted confirm assertion such as Confirm \((A \text{ and } B) \text{ and } C\). When this process is done, the VC generator “breaks”
the assertion by separating each conjunct into a VC. However, since each assertion could contain
other logical operators that might affect the assertion to be proved, this decomposition has to be
done carefully to ensure soundness.

The alternative approach developed in this dissertation builds Gentzen-style sequents to
represent VCs [38][39], rather than building a cumulative confirm assertion and breaking at the end
of the process. A sequent is a conditional assertion of the form:

\[ A_1, ..., A_m \vdash C_1, ..., C_n \]

where each \( A_i \) is a conditional assertion called an “antecedent” (given) and each \( C_j \) is a conditional
assertions called a “consequent” (goal) and the operator \( \vdash \) is interpreted as “yields”, “proves” or
“entails”. The antecedent’s commas indicate that the formulas are joined by the \( \land \) operator, while
the consequent’s commas indicate they are joined by the \( \lor \) operator. Therefore, we simply have to
show that given a list of true antecedents, one of the consequents must be true. Note that both the
antecedents and the consequents are sequences and not sets of logical formulas. They are listed in
order and repeated formulas may be added from different specification contexts.

Since our goal is to make the task of the automated prover easy, it is important to simplify the
sequents. To reduce a sequent into simpler logical formulas, we apply a series of logical reduction
rules to both antecedents and consequents until they only contain atomic formulas (i.e., they do
not contain logical operators) [38][39]. Table 4.1 shows the reduction rules for antecedents and
consequents. Note that \( \Gamma \) and \( \Delta \) stand for possible additional antecedents/consequents.

<table>
<thead>
<tr>
<th>Left Rules</th>
<th>Right Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land )</td>
<td>( \land )</td>
</tr>
<tr>
<td>( \Gamma, A, B \vdash \Delta )</td>
<td>( \Gamma \vdash \Delta, A ) ( \Gamma \vdash \Delta, B )</td>
</tr>
<tr>
<td>( \Gamma, A \land B \vdash \Delta )</td>
<td>( \Gamma \vdash \Delta, A \land B )</td>
</tr>
<tr>
<td>( \lor )</td>
<td>( \lor )</td>
</tr>
<tr>
<td>( \Gamma, A \lor B \vdash \Delta )</td>
<td>( \Gamma \vdash \Delta, A \lor B )</td>
</tr>
<tr>
<td>( \Gamma, A \lor B \vdash \Delta )</td>
<td>( \Gamma \vdash \Delta, A \lor B )</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>( \Rightarrow )</td>
</tr>
<tr>
<td>( \Gamma, A \Rightarrow B \vdash \Delta )</td>
<td>( \Gamma, A \vdash \Delta, B )</td>
</tr>
<tr>
<td>( \Gamma, B \vdash \Delta )</td>
<td>( \Gamma, A \vdash \Delta, B )</td>
</tr>
<tr>
<td>( \neg )</td>
<td>( \neg )</td>
</tr>
<tr>
<td>( \Gamma, \neg A \vdash \Delta )</td>
<td>( \Gamma, A \vdash \Delta )</td>
</tr>
<tr>
<td>( \Gamma, \Delta \vdash \Delta )</td>
<td>( \Gamma, \Delta \vdash \neg A )</td>
</tr>
</tbody>
</table>
Notice that the left ∨, left ⇒ and right ∧ rules produce two sequents that must be established to be true. Even though they may share common antecedents or consequents, the prover may establish each of the sequents independently of each other.

The sequent reduction process can be shown using a rooted tree graph where the root of the tree contains the original formula and the leaves are the resulting atomic formulas. Each reduction is documented as an intermediary node that contains a directed edge from the sequent before the reduction. Depending on which rules are applied, additional nodes and edges may be added, but the leaf nodes would always be the same. Consider the following assertion for example:

\[(p \implies r) \lor (q \implies r) \implies ((p \land q) \implies r)\]

The reduction tree generated by the VC generator for this assertion is shown in Figure 4.1. Note the leaf nodes have been colored red to indicate that they are atomic formulas.

![Sample Sequent Reduction Tree Produced by VC Generator](image)

### 4.2 Forming Parsimonious VCs

Generating reduced sequents outlined in the previous section is only the first step in making the task of the automated prover simpler. With the increased complexity of specification and
assertions, especially in the case of shared concepts, and the usage of multiple library components to build other components, the VC generation process might introduce several givens that are not needed to prove a particular VC. An additional given in the sequent potentially increases the search space of the automated prover. Therefore, a mechanism to detect whether or not a given is necessary to prove the current goal is desirable to make the task of the automated proving more efficient. This simplification process must be handled with extreme care, so that no necessary givens are eliminated by mistake and any \textit{false} claims are retained. An example of this latter case to illustrate such a situation is given in Section 4.2.3.

We use the termination VC of \texttt{Reverse\_List} from the previous chapter as an example to see what givens can be eliminated. The listing has been reproduced below:

\textbf{Listing 4.1: Termination VC for} \texttt{Reverse\_List} (Reproduced from Listing 3.7)

\begin{quote}
\textbf{Goal:}\\
\((1 + |\text{Prt\_Btwn}(1, |P.\text{Rem}|, P.\text{Rem})|) \leq |P.\text{Rem}|)\\
\end{quote}

\begin{quote}
\textbf{Given:}\\
1. \((P'''.\text{Prec} = P.\text{Prec})\)\\
2. \((E' = \text{DeString}(\text{Prt\_Btwn}(0, 1, P.\text{Rem})))\)\\
3. \(\text{not}((P.\text{Rem} = \text{Empty\_String}))\)\\
4. \(\text{Entry.Is\_Initial}(E)\)\\
5. \((P.\text{Prec} = \text{Empty\_String})\)\\
6. \((\text{min\_int} \leq 0)\)\\
7. \((1 \leq \text{max\_int})\)\\
8. \((1 \leq \text{Last\_Char\_Num})\)\\
9. \((1 \leq \text{Max\_Char\_Str\_Len})\)
\end{quote}

On a closer look at the givens list, it is easy to see that given \#4 will never be useful. Neither our goal or the rest of the givens contain the symbols \(E\) or the \texttt{Entry.Is\_Initial} predicate. Givens \#6, \#7, \#8 and \#9 all contain a number that does appear in the goal or other givens. However, a closer look at those givens show that they are used to establish a range of possible values for \texttt{min\_int}, \texttt{max\_int}, \texttt{Last\_Char\_Num} and \texttt{Max\_Char\_Str\_Len}. Since the symbols they are establishing the range for does not appear in either of the goals or the givens, it is safe to conclude that they are not needed. If we eliminate givens \#4 and \#6 - \#9, the resulting VC is shown below:
Listing 4.2: Termination VC for Reverse_List with only Relevant Givens

Goal:

\[(1 + |\text{Prt_Btwn}(1, |\text{P}.\text{Rem}|, \text{P}.\text{Rem})|) \leq |\text{P}.\text{Rem}|\]

Given:
1. \((\text{P}''''.\text{Prec} = \text{P}.\text{Prec})\)
2. \((\text{E}' = \text{DeString}(\text{Prt_Btwn}(0, 1, \text{P}.\text{Rem})))\)
3. \(\neg((\text{P}.\text{Rem} = \text{Empty_String}))\)
4. \((\text{P}.\text{Prec} = \text{Empty_String})\)

Furthermore, one could argue that givens #1 and #4 from Listing 4.2 are not useful either. But they still share a common symbol \(p\), and thus may not be eliminated.

There is an additional complication to the simplification process. When conditional statements such as \textbf{If} and \textbf{While} statements are processed, (the ensures clause of) the testing condition must be retained. This is necessary because the condition might generate an expression that creates a contradiction with any of the assumptions about the current code. In other words, it is a branch of the execution that would never be reached or executed. In this case, any of the goals created within the condition branch are automatically satisfied.

In order to have the VC generator perform the elimination of unnecessary givens and generate parsimonious VCs, the underlying proof rules must be updated. The following subsections focus on 3 types of rules for the following kinds of assertive statements: \textbf{Confirm}, \textbf{Assume} and \textbf{Stipulate Assume}. In the following rules,

- \(C\) is the context.

- \(\text{Seq}\) is a sequence calculus predicate of the form: \(\{A_1, \ldots, A_i\} \implies \{C_1, \ldots, C_j\}\), where the commas between antecedents indicate that they are joined by \(\land\) and the commas between consequents indicate they are joined by \(\lor\). Note that the predicate could contain empty antecedents or consequents. Empty antecedents means there are no givens. Empty consequents means false is the goal.

- \(\text{Result Sequent or RS}\) are a collection of \(\text{Seq}\) joined together by the \(\land\) operator.

- \(\text{code}\) refers to a block of either executable statements or assertive code.

- \(\text{Sequent Form(RS, exp)}\) adds \(\exp\) to each sequent in \(\text{RS}\) and applies the reduction rules to obtain the reduced sequent form.
• $\text{CExp}$ is the constraint expression for $t$.
• $\rightarrow$ indicates replacement.

For each of the rules, the code before the application of the rule is shown below the line, while the resultant code is shown above.

### 4.2.1 Confirm Rule

A Confirm statement contains assertions that must be established to be true and that are generated from ensures clauses or other proof rules. The updated Confirm rule is shown below:

Listing 4.3: Confirm Rule

```c
C\ code; Confirm Sequent_Form( Empty_Seq, exp) \land RS;
```

```c
C\ code; Confirm exp; Confirm RS;
```

where Empty_Seq is a sequent that only has true as its goal and has no antecedents.

Suppose the system generated the initial assertive program shown below.

Listing 4.4: Initial Assertive Program

```c
Assume |S| \leq 2 \land |T| \leq 2 \land S = \text{Empty\_String} \land T = <1> \circ <2>;
Confirm (S = \text{Empty\_String or } T = \text{Empty\_String) \land (|S| + |T| = 2) }
```

When applying the Confirm and sequent reduction rules to the assertive program, the VC generator produces the two sequents with empty antecedents:

Listing 4.5: Assertive Program After Applying the Confirm Rule

```c
Assume |S| \leq 2 \land |T| \leq 2 \land S = \text{Empty\_String} \land T = <1> \circ <2>;
\vdash (S = \text{Empty\_String, } T = \text{Empty\_String})
\vdash (|S| + |T| = 2)
```

### 4.2.2 Assume Rule

An Assume statement adds assertions as additional givens to a VC. In order to generate parsimonious VCs, the VC generator must retain only useful givens. The Assume rule has 2 different steps: substitution and assume application. Each of the steps is explained in the following listings.
If the **Assume** statement does not contain the conditions in which the step needs to be applied, then that step is skipped.

**Listing 4.6: Substitution Step**

```c
C\ code; Assume exp3[ y\rightarrow exp1 ]; Confirm ( RS[ y\rightarrow exp1 ] )[ z\rightarrow exp2 ];
```

```c
C\ code;
Assume ( y = exp1\angle x \downarrow ) \land ( z = exp2\angle x \downarrow ) \land exp3\angle y \downarrow ;
Confirm RS\angle y, z \downarrow ;
```

The substitution step can only be used on *replaceable expressions*. A *replaceable expression* in which we define as equalities the form \( a = \text{exp} \) (or \( \text{exp} = a \)), where \( a \) is a mathematical variable and \( \text{exp} \) is a mathematical expression. In the context of the rule shown, \( x, y \) and \( z \) denote variable expressions while \( \text{exp1}, \text{exp2} \) and \( \text{exp3} \) denote mathematical expressions. Applying the substitution step to the assertive program from Listing 4.5 results in the following:

**Listing 4.7: Assertive Program After Applying the Substitution Step**

```plaintext
Assume |S| <= 2 and |<1> o <2>| <= 2 and S = Empty_String;
|¬ (S = Empty_String, <1> o <2> = Empty_String)
|¬ (|S| + |<1> o <2>| = 2)
```

Here, \( T \) has been replaced with \( <1> o <2> \) in both sequents, as well as in the **Assume** statement. Note that even though there is another equality expression present: namely, \( S = \text{Empty\_String} \), both \( S \) and \( \text{Empty\_String} \) are syntactically mathematical variables, the rule as it is currently written does not permit \( S \) to be substituted for \( \text{Empty\_String} \). Once this step is complete, the VC generator proceeds to apply the **Assume Application Step** to finish processing the **Assume** statement.

**Listing 4.8: Assume Application Step**

```c
C\ code;
Confirm Sequent_Form( Sequent_Form(Seq1, exp1), exp3 ) \land
Sequent_Form(Seq2, exp2);
```

```c
C\ code;
Assume exp1\angle x, z \downarrow \land exp2\angle y \downarrow \land exp3\angle z \downarrow ;
Confirm Seq1\angle x \downarrow \land Seq2\angle y \downarrow ;
```

where Seq1 and Seq2 are sequents.

1Since \( \text{Empty\_String} \) is a definitional literal defined in \( \text{String\_Theory} \), a future enhancement to the substitution step should allow mathematical variables to be replaced with definitional literals and VC generator could use this information to ‘tag’ this as a “safely substitutable” equality.
During this step, a mathematical expression (\(exp\)) will only be added to a sequent’s antecedent if the variables in that sequent overlap with those in \(exp\). In the example above, \(exp_1\) and \(exp_3\) are added to \(Seq_1\) and \(exp_2\) is added to \(Seq_2\). Although \(exp_3\) is not constrained to \(x\) directly, \(exp_1\) involves \(x\) and \(z\), therefore we add \(exp_3\) to \(Seq_1\). This step generates *parsimonious* VCs by reducing the number of antecedents in a sequent.\(^2\)

The application of sequent reduction rules illustrated by the Sequent_Form predicate might generate additional sequents. Even though these new sequents may share common antecedents and/or consequents, they are new VCs that must be established to be \(true\).

Applying this step to Listing 4.7 produces the final sequents listed below. Notice that the second consequent in the first sequent cannot be established to be \(true\). A string containing 1 and 2 will never equal to the \(Empty_String\). Therefore, the prover would have to establish that \(S = Empty_String\), which is \(true\). Also notice that the second expression from the original conjunct didn’t contain any relevant symbols, therefore it was not added to either sequent.

Listing 4.9: Final Resulting Sequents

\[
(|S| \leq 2, S = Empty_String) \models (S = Empty_String, <1> \circ <2> = Empty_String)\\
(|S| \leq 2, S = Empty_String) \models (|S| + |<1> \circ <2>| = 2)
\]

### 4.2.3 Stipulate Assume Rule

A *Stipulate* statement is a special kind of *Assume* statement, where elimination of unnecessary givens is not applied. It is important to note that the proof rules should avoid generating a *Stipulate* statement unless it is absolutely necessary to keep its assertions no matter what (e.g., if/loop conditions). An example (partial) assertive program with nested *If* statements is shown below:

Listing 4.10: Partial Assertive Program with Nested-If Statements

```plaintext
If ( I = 0 ) then
  If ( I /= 0 ) then
    K := 1;
    J := 2;
    Confirm K = J;
  end;
end;
```

\(^2\)If the only common variable between \(exp\) and the sequents are definition literals, then anything of note is already provided by the theory and an enhancement similar to the substitution step could be applied to eliminate \(exp\).
First notice that code inside nested \texttt{If} is not reachable. However, when establishing the correctness of a program, all possible paths must be considered. In this case, the \texttt{If} conditions form a contradiction, therefore any generated sequents inside the nested \texttt{If}, such as the one that will result from \texttt{Confirm } K = J, are automatically proven.

The \texttt{Stipulate} \texttt{Assume} rule contains 2 different steps similar to the regular \texttt{Assume} rule: \textit{substitution} and \textit{stipulate application}. Each of the steps is explained in the following listings. If the \texttt{Stipulate} statement does not contain the conditions in which the step needs to be applied, then that step is skipped.

\textbf{Listing 4.11: Stipulate Substitution Step}

\begin{verbatim}
C\ code; \texttt{Stipulate exp3[ y\rightarrow exp1 ]; Confirm ( RS[ y\rightarrow exp1 ] )[ z\rightarrow exp2 ];}
\end{verbatim}

\begin{verbatim}
C\ code;
  \texttt{Stipulate (y = exp1\angle x \angle) \land (z = exp2\angle x \angle) \land exp3\angle y \angle; Confirm RS\angle y, z \angle;}
\end{verbatim}

The substitution step can only be used on \textit{replaceable expressions}. The definition for \textit{replaceable expression} can be found in Section 4.2.2.

\textbf{Listing 4.12: Stipulate Application Step}

\begin{verbatim}
C\ code; \texttt{Confirm Sequent_Form( Sequent_Form(RS, exp1), exp2 );}
\end{verbatim}

\begin{verbatim}
C\ code; \texttt{Stipulate exp1\angle x \angle \land exp2\angle y \angle; Confirm RS\angle x \angle;}
\end{verbatim}

During this step, the stipulated mathematical expressions are always added to each sequent in \texttt{RS}. Recall that each application of \texttt{Sequent_Form} might generate additional sequents.

\subsection{4.3 Other VC Generator Updates}

This dissertation has led to several improvements to the VC generation and proof process to simplify an enable automated verification in the presence of non-trivial specifications. This subsection summarizes other significant changes to the process. The introduction of a mathematical type system ([91]) requires that all the expressions built by the verification condition generator contain appropriate mathematical types. This information is used by the automated prover to deduce which mathematical assertions would be most useful to prove a given VC. A mathematical assertion can only be used if there is a direct relationship between the mathematical types. Therefore,
it is necessary to incorporate the type information for any new expressions formed by VC generator proof rules.

One other routine aspect of non-trivial software that complicates VC generation is the case of nested function calls. When function calls are nested, the pre- and post-conditions of each of the operations must be satisfied. The VC generator must find the corresponding formal specifications for the operation and subsequently substitute in the appropriately modified ensures clause using the inner parameter expressions. When this process is complete, there would be a list of requires clause VCs for each of the operations in the nested function call and the ensures clause for the outermost call with all the formal parameters substituted with the actual arguments. This process is necessary to generate VCs that are provable using the givens in the current context.

All initialized variables in RESOLVE must at all times satisfy its type constraint (if any). Therefore, the assertive program always adds these as assertions it knows to be true. However, in most cases, the type constraint assertions are not helpful in proving a sequent. Adding it simply increases the number of givens the automated prover must consider. At this point, the new verification system includes a flag to avoid adding these type constraints automatically. Further research should be done to provide new constructs (and proof rules) to support adding the type constraints on demand.
Chapter 5

Data Abstractions with

Shared State

One of the contributions of this research is to leverage a language with clean semantics such as RESOLVE to provide data abstractions that hide references wherever possible. This chapter explains that this desirable goal is achievable also for data abstractions that define and use shared variables. The goal of this chapter is to introduce new constructs and reasoning mechanisms for a sharing construct in a reference-hiding data abstraction. The chapter also contains an evaluation of VC generation.

Figure 5.1: Comparing Reasoning Approaches (Reproduced from Chapter 1)
In Chapter 3, we introduced a Globally_Bounded_List_Templaté, where the instantiated objects share the system’s global memory, but with the simplifying assumption that operations on objects have no bearing on global memory space. To motivate and introduce the reasoning proof rules necessary to generate verification conditions for objects that share state, the entire complexity of global sharing is not necessary. So we present the notion of communally bounded objects [2].

Unlike Globally_Bounded_List_Templaté, a communally bounded list concept requires a user supplied maximum size that is used to constraint all the instantiated List objects. It is also different from concepts that are individually bounded, because the maximum capacity constraint is shared across all instantiated objects from the same type rather than each List object being constrained individually. We can think of the communal bounded list’s maximum size as a way to restrict the maximum amount of memory used by all Lists from the same Facility instantiation. The figure below depicts an example communal bounded list instantiation with a maximum capacity of 200 and three List objects that occupy a portion of the allocated storage. Note that the storage space is shown as contiguous for simplicity, but it may not necessarily be the case in the underlying implementation.

Figure 5.2: Example of a Communal List Instantiation

This chapter first presents the Communally_Bounded_List_Templaté concept. It is followed by experimentation and evaluation. We begin with simple client programs to explain and evaluate the updated proof rules used by the VC generator. Lastly, we introduce an Enhancement for searching an element in a List and discuss how to reason about the provided implementation.
5.1 **Communally_Bounded_List_Template**

In order to better explain Communally_Bounded_List_Template, the Concept has been broken down into smaller sections. The complete specification is given in Appendix C.

Listing 5.1: A Communally-Bounded List Concept

```
Shared Concept Communally_Bounded_List_Template{
  type Entry; evaluates Max_Capacity: Integer;
  uses String_Theory, Set_Theory, Integer_Ext_Theory;
  requires 1 <= Max_Capacity which_entails Max_Capacity is_in N;
  ...
end Communally_Bounded_List_Template;
```

The Shared keyword indicates that this concept contains global state variables that is shared across a type instantiation. Similar to Listing 3.1, this Concept has a parameter Entry that represents a generic type and appears throughout the specifications and operation parameters. In addition, this Concept is restricted by the user supplied communal bound, Max_Capacity. The uses line imports String_Theory to specify the mathematical abstraction of List and the behavior of the provided Operations, Set_Theory for using the is_in operator and Integer_Ext_Theory for writing specifications involving mathematical integers. The concept requires that the supplied Max_Capacity to be greater than or equal to 1. Notice that the which_entails clause contains a useful lemma indicating that Max_Capacity is in N. Before we can use this lemma, we must first establish it to be true. The proof rule for which_entails clause is shown below. It needs to be established only once in the life of a Concept and becomes a useful lemma for clients.

Listing 5.2: Which_Entails Rule

```
C\ Assume expl1; Confirm exp2; Confirm RS;
C\ Assertion_Clause expl which_entails exp2; Confirm RS;

where Assertion_Clause is either a requires, ensures, constraint, convention or correspondence clause
```

Applying the proof rule results in the following VC:

---

1Ideally we should import Natural_Number_Theory, since all of our specifications use natural number variables. However, the mathematical type system defined in [91] does not allow us to re-type all instances of Max_Capacity as a natural number.
Listing 5.3: Resulting VC After Applying the Which_Entails Rule

Goal:

(Max_Capacity is in N)

Given:
1. (1 <= Max_Capacity)

The Shared Variables construct shown below allows us to introduce abstractions for global state variables that are shared across all objects generated from the same communal list facility instantiation. This concept introduces a global state variable called Total_Size, where it is initially 0 and constrained to be strictly less than or equal to Max_Capacity.

Listing 5.4: Shared Variables

Shared Variables
Abstract_Var Total_Size: N;

constraint Total_Size <= Max_Capacity;
initialization
ensures Total_Size = 0;

The abstract type List defined by this concept is nearly identical to Listing 3.2, with the exception of the finalization block. The affects clause indicates that the global shared state is being updated and the corresponding specification must state the changes to the shared state. In this case, finalization of a List object affects Total_Size and its ensures clause indicates that Total_Size decreases by the number of elements in #P. When a global state variable isn’t listed in the affects clause, it is assumed to be restored. Notice that Total_Size is not affected by initialization of new Lists. This is because new Lists are initialized to be empty as per the initialization specification.

Listing 5.5: Communal List Specifications

Type Family List is modeled by Cart_Prod
Prec, Rem: Str(Entry);
end;
exemplar P;

initialization
ensures P.Prec = Empty_String and P.Rem = Empty_String;
finalization
affects Total_Size;
ensures Total_Size = #Total_Size - |P.Prec o P.Rem|;
end;
Similarly, the Insert and Remove operations affect Total_Size and the specifications indicate that Total_Size increases or decreases respectively. Other than affecting Total_Size, the requires and ensures clauses for Insert and Remove are identical to those presented in Section 3.1. The new operation Occupied_Size returns the total amount of elements stored across all instantiated List objects. Notice that Occupied_Size simply reports the value of Total_Size and does not make any modifications.

Listing 5.6: Selected Operations for Communal List

```
Operation Insert( clears New_Entry: Entry; updates P: List );
affects Total_Size;
requires Total_Size < Max_Capacity;
ensures P.Prec = #P.Prec and
    P.Rem = <$New_Entry> o #P.Rem and
    Total_Size = #Total_Size + 1;

Operation Remove( replaces Entry_Removed: Entry; updates P: List );
affects Total_Size;
requires P.Rem /= Empty_String;
ensures P.Prec = #P.Prec and
    Entry_Removed = DeString(Prt_Btwn(0, 1, #P.Rem)) and
    P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem) and
    Total_Size = #Total_Size - 1;

Operation Occupied_Size( ) : Integer;
ensures Occupied_Size = ( Total_Size );
```

5.2 Example Client Programs and Evaluation

Up to this point, we have only used the types provided by the associated Concept in Enhancements. In this subsection, a client program (or Facility) that instantiates and uses Communally_Bounded_List_Template is shown in Listing 5.7.

Listing 5.7: Example Client Program #1

```
Facility CBLT_Example_1;
uses Integer_Theory;

Facility List_Fac is Communally_Bounded_List_Template(Integer, 2)
realized by CUVRT_Realiz;

Operation No_Bound_Violation();
    requires List_Fac::Total_Size = 0;
Procedure
    Var L1, L2, L3: List;
    Var I, J, K: Integer;
```
I := 2;
J := 3;
K := 1;

Insert(I, L1);
Insert(J, L2);
Remove(I, L1);
Insert(K, L3);
end No_Bound_Violation;
end CBLT_Example_1;

In order to establish the correctness for this listing, we will need to establish the correctness of all inner declarations in order to verify the correctness of CBLT_Example_1. There are two declarations that we have to deal with: a Facility instantiation (List_Fac) and an Operation declaration (No_Bound_Violation).

The Facility declaration: List_Fac instantiates a type List of Integers with maximum capacity of 2 using the CUVRT_Realiz implementation. In order to establish the correctness of List_Fac, we apply the facility instantiation rule found in Appendix A. The resulting application of the rule generates VCs such as the one listed below that establishes that the requires clause of Communally_Bounded_List_Template. Notice that the maximum capacity of List_Fac is 2, so it has been substituted accordingly. The goal for this is straight forward and any assume assertions has been eliminated by the application of the Assume rule (Section 4.2.2).

Listing 5.8: Facility Instantiation VC for List_Fac

Goal(s):
(1 <= 2)

Given(s):

Along with the facility declaration, we have defined a No_Bound_Violation operation that creates 3 List objects: L1, L2 and L3 and 3 Integer objects: I, J and K. After assigning values to I and J, this operation calls the Insert and Remove operations to add and remove them from the various List objects. Notice that after the first 2 calls to Insert, the Total_Size for List_Fac’s Lists is equal to the maximum capacity, so we cannot insert any more items to any of List_Fac’s Lists. In order to insert K into L3, we remove an element from L1 before we call Insert.

---

2Discussion of concept implementations can be found in Chapter 7.
In order to establish the correctness of No_Bound_Violation, we apply the local operation declaration rule in Appendix A. The VC for establishing the pre-condition of the third Insert call is shown in the listing below.

Listing 5.9: VC for Establishing the Pre-Condition of the Third Call to Insert

\[ \text{Goal}(s): \]
\[ ((1 + (((0 + 1) + 1) - 1)) <= 2) \]

\[ \text{Given}(s): \]

This VC has substituted the values of Total Size from the various states of executing No_Bound_Violation. By applying algebraic simplifications, the resulting goal is \( 2 <= 2 \), which can be established to be \( \text{true} \).

However, what happens if we attempt to call Insert when Total Size is at Max Capacity? The listing depicted below contains an operation Bound_Violation, where it is nearly identical to Listing 5.7, except it omits the call to Remove.

Listing 5.10: Example Client Program #2

```plaintext
Facility CBLT_Example_2;
    uses Integer_Theory;
    
    Facility List_Fac is Communally_Bounded_List_Template(Integer, 2)
        realized by CUVRT_Realiz;
    
    Operation Bound_Violation();
        requires List_Fac::Total_Size = 0;
    
    Procedure
        Var L1, L2, L3: List;
        Var I, J, K: Integer;
        
        I := 2;
        J := 3;
        K := 1;
        
        Insert(I, L1);
        Insert(J, L2);
        Insert(K, L3);
    end Bound_Violation;
end CBLT_Example_2;
```

The call Insert(K, L3) cannot satisfy the pre-condition of Insert, therefore will result in a VC (shown below) that is not provable. We can see that with algebraic simplifications, the resulting goal is \( 3 <= 2 \), which is \( \text{false} \).
Listing 5.11: (Unprovable) VC for Establishing the Pre-Condition of the Third Call to Insert

**Goal(s):**

\(((1 + ((0 + 1) + 1)) <= 2)\)

**Given(s):**

Notice that both No_Bound_Violation and Bound_Violation do not modify List_Fac’s Total_Size, therefore the VC generator will generate a VC that establishes that List_Fac’s Total_Size is equal to its incoming value, which is 0. As an example, the VC generated from Listing 5.7 is shown below.

Listing 5.12: VC for Establishing that List_Fac::Total_Size isn’t Modified

**Goal(s):**

\((((((0 + 1) + 1) - 1) + 1) - (|L1’.Prec| + |Prt_Btwn(1, |(<2> o Empty_String)|, (<2> o Empty_String))|)) - (|L2’.Prec| + |(<3> o Empty_String)|)) - (|L3’.Prec| + |(<1> o Empty_String)|)) = 0)\)

**Given(s):**

1. (L3’.Prec = Empty_String)
2. (L1’.Prec = L1’’.Prec)
3. (I’ = DeString(Prt_Btwn(0, 1, (<2> o Empty_String))))
4. (L2’.Prec = Empty_String)
5. (L1’’.Prec = Empty_String)

The lists L1, L2 and L3 are finalized, therefore the **finalization ensures** clause from Listing 5.5 can be assumed and substituted as shown in the VC.

Although the goal looks complex, we can show that it is provable. As a first step, notice that given #1, #2, #4 and #5 can be used to substitute parts of the goal. This will result in the following simplified VC:

Listing 5.13: Simplifying the VC (Step #1)

**Goal(s):**

\((((((0 + 1) + 1) - 1) + 1) - (|Empty_String| + |Prt_Btwn(1, |(<2> o Empty_String)|, (<2> o Empty_String))|)) - (|Empty_String| + |(<3> o Empty_String)|)) - (|Empty_String| + |(<1> o Empty_String)|)) = 0)\)

**Given(s):**

1. (I’ = DeString(Prt_Btwn(0, 1, (<2> o Empty_String))))
Next, the concatenation \((\circ)\) of any mathematical string with \textit{Empty\_String} can be simplified to the mathematical string itself. This is a corollary in \textit{String\_Theory}. Therefore, if we apply this corollary, our VC will look like the following:

Listing 5.14: Simplifying the VC (Step #2)

\textbf{Goal (s):}

\[
((((((((0 + 1) + 1) - 1) + 1) - (|\text{Empty\_String}| + |\text{Prt\_Btwn}(1, <2>, <2>)|)) - (|\text{Empty\_String}| + |<3>|)) - (|\text{Empty\_String}| + |<1>|)) = 0
\]

\textbf{Given (s):}

1. \((I' = \text{DeString}(\text{Prt\_Btwn}(0, 1, <2>)))\)

At this point, we simplify the \textit{Prt\_Btwn} expression in the goal. This expression is showing the number of elements in \textit{L1}'s \textit{Rem} string. The operator \(|...|\) returns the length of a mathematical string, therefore the expression becomes \textit{Prt\_Btwn}(1, 1, <2>). This \textit{Prt\_Btwn} expression is representing a substring from a start index of 1 to an end index of 1, which is equal to the \textit{Empty\_String} and we can simplify our VC by replacing \textit{Prt\_Btwn}(1, |<2>|, <2>) with \textit{Empty\_String}.\footnote{Recall that \textit{No\_Bound\_Violation} calls \textit{Remove}(I, L1), which removes the only element in \textit{L1}. This means that \textit{L1} is empty.}

Listing 5.15: Simplifying the VC (Step #3)

\textbf{Goal (s):}

\[
((((((0 + 1) + 1) - 1) + 1) - (|\text{Empty\_String}| + |\text{Empty\_String}|)) - (|\text{Empty\_String}| + |<3>|)) - (|\text{Empty\_String}| + |<1>|)) = 0
\]

\textbf{Given (s):}

1. \((I' = \text{DeString}(\text{Prt\_Btwn}(0, 1, <2>)))\)

Lastly, we can simplify all expressions where we use the \(|...|\) operator. The length of \textit{Empty\_String} is 0 and length of a singleton string is 1. This produces the simplified goal where we deal with only mathematical integers. If we apply algebraic simplifications to this new simplified goal, the expression becomes \(0 = 0\), which is \textit{true}. Notice that we do not need the leftover given to prove this VC.
Listing 5.16: Simplifying the VC (Step #4)

**Goal(s):**

\[ ((((((0 + 1) + 1) - 1) + 1) - (0 + 0)) - (0 + 1)) - (0 + 1)) = 0 \]

**Given(s):**

1. \( I' = \text{DeString}(Prt\_Btwn(0, 1, <2>)) \)

The rest of the VCs for Listings 5.7 and 5.10 with the assertive code block generation and proof rule applications can be found in Appendix C.

### 5.3 Enhancement for Searching a List and Evaluation

A common **Enhancement** for container data structures is the ability to search for an element. The enhancement `Searching_Capability` for `Communally_Bounded_List_Template` provides a `Contains` operation that can be used to search for an `Entry` type element in a `List`. Notice that this operation does not affect `Total_Size`, so there is no `affects` clause.

**Listing 5.17: Searching Enhancement**

```plaintext
Enhancement Searching_Capability for Communally_Bounded_List_Template;
Operation Contains(restores E: Entry; restores L: List): Boolean;
  ensures Contains = ( Is_Substring(<E>, L.Prec) or Is_Substring(<E>, L.Rem) );
end Searching_Capability;
```

There is no `requires` clause for `Contains` and the `ensures` clause states that the return value of `Contains` is the result of evaluating whether or not the singleton string `<E>` is a substring of the `Pre` string or the `Rem` string. Similar to the `Reverse_List` specification shown in Listing 3.4, any implementation of `Contains` must adhere to the specifications, but can implement the behavior in any way, thereby allowing for a variety of implementations with different efficiency characteristics. The actual implementation is transparent to the users who use `Contains` and the implementations details are not used when VCs are generated for user code.

The listing below is an implementation of `Searching_Capability`. In addition to the implementation of `Contains`, this **Realization** also has a recursive local operation for searching an element in the `Rem` string. An alternative implementation with a loop-based local operation including the generated VCs can be found in Appendix C.
Realization

Recursive_Searching_Realiz(
  Operation Are_Equal(restores E, F: Entry) : Boolean;
  ensures Are_Equal = ( E = F );
)
for Searching_Capability of Communally_Bounded_List_Template;

Operation Is_Present_In_Rem(restores E: Entry; restores L: List): Boolean;
  requires L.Prec = Empty_String;
  ensures Is_Present_In_Rem = ( Is_Substring(<E>, L.Rem) );

Recursive Procedure
  decreasing |L.Rem|;
  -- notice Total_Size <= Max_Capacity;
  Var Next_Entry: Entry;

  Is_Present_In_Rem := False();
  If ( 1 <= Length_of_Rem(L) ) then
    Remove(Next_Entry, L);
    If ( not Is_Present_In_Rem(E, L) ) then
      Is_Present_In_Rem := Are_Equal(E, Next_Entry);
    else
      Is_Present_In_Rem := True();
    end;
    Insert(Next_Entry, L);
  end;
end Is_Present_In_Rem;

Procedure Contains (restores E: Entry; restores L: List): Boolean;
  Var Temp_Rem_List: List;
  Contains := False();

  -- Store L.Rem in a temporary list
  Swap_Remainders(L, Temp_Rem_List);
  Contains := Is_Present_In_Rem(E, Temp_Rem_List);

  -- If not found, check L.Prec
  If ( not Contains ) then
    Reset(L);
    Contains := Is_Present_In_Rem(E, L);
    Advance_to_End(L);
  end;

  -- Restore the list
  Swap_Remainders(L, Temp_Rem_List);
end Contains;
end Recursive_Searching_Realiz;

In order to establish if the parameter E is equal to an element in L, this realization requires
the user to supply an Are_Equal operation as parameter for determining whether or not two Entry
type elements are equivalent. This allows the user to supply different implementations for each
instantiating **Facility** of Communal_List_Template with different Entry types, as long as they
meet the specifications of Are_Equal.

The implementation for **Contains** is straightforward. We create a temporary list to separate
the **Prec** string from the **Rem** string and subsequently manipulate each list (if necessary) to make
sure all the content is in the **Rem** string. After that we call the local operation **Is_Present_In_Rem**
to check if <E> is a substring of either of the **Rem** strings. When the checks are done, we proceed to
restore **L** to ensure that **L.Prec = #L.Prec** and **L.Rem = #L.Rem**.

**Is_Present_In_Rem** provides a decreasing progress metric that ensures that it termi-
nates when **L**’s **Rem** string is empty. As long as the number of elements in **L.Rem** is greater than
or equal to 1, we remove the next element from the **Rem** string and proceed to check if **E** is in the
sub-list\(^4\). If it is found, **Is_Present_In_Rem** will be **true**, otherwise **Is_Present_In_Rem** will be
updated with the return value of calling **Are_Equal(E, Next_Entry)**. Once we are done checking
the current element, we insert **Next_Entry** back into **L**.

Verifying this implementation will require that each possible execution path meets the
**ensures** clause. As an example, the VC for checking the pre-condition of **Insert** for one of the
paths is shown below. All the other VCs can be found in Appendix C.

**Listing 5.19: Requires Clause of **Insert**

**Goal(s):**

\((1 + (\text{Total\_Size} - 1)) \leq \text{Max\_Capacity})\)

**Given(s):**

1. **Is\_Substring(<E>, Prt\_Btwn(1, |L.Rem|, L.Rem))**
2. \((L''\text{.Prec} = \text{Empty\_String})\)
3. \((\text{Next\_Entry''} = \text{DeString(Prt\_Btwn(0, 1, L.Rem))})\)
4. \((1 \leq |L.Rem|)\)
5. \((1 \leq \text{Max\_Capacity})\)
6. \((\text{Max\_Capacity is in N})\)
7. \((\text{min\_int} \leq \text{Max\_Capacity})\)
8. \((\text{Max\_Capacity} \leq \text{max\_int})\)
9. \((\text{Total\_Size} \leq \text{Max\_Capacity})\)
10. \((\text{min\_int} \leq 0)\)
11. \((1 \leq \text{max\_int})\)

\(^4\text{Notice that if there are multiple matches, the Entry that gets matched will be the last occurrence of E!}\)
5.3.1 Future Improvements

Listing 5.19 contains a lot of givens that are not needed to prove the goal. Given #9 is all we need to establish this to be true, but the parsimonious VC generator is forced to keep givens #4 to #8, #10 and #11. These givens are various constraints about Total_Size and Max_Capacity and how it relates to min_int, max_int, 0 and 1. Ideally, the VC generator should eliminate all of these and only keep given #9 as shown in the listing below.

Listing 5.20: Simplified VC from Listing 5.19

Goal(s):

\((1 + (Total\_Size - 1)) \leq Max\_Capacity)\)

Given(s):

1. \((Total\_Size \leq Max\_Capacity)\)

In order to do this, the language could introduce a new construct using the keyword \textbf{notice} for providing which constraints should be kept. This must be an exact match to what the VC generator produces as a result of applying the various different proof rules. In this case, \textit{Is\_Present\_In\_Rem} should contain \textbf{notice} Total_Size \leq Max\_Capacity.

Furthermore, the VC generator can have a post-processing step for cleaning up givens such as #1 to #4 to substitute and simplify certain givens with equality expressions and/or eliminate givens that are not relevant to the goal. Ideally, the VC generator should produce something like the following listing, but more research needs to be done on this front to ensure that the reasoning system remains sound.

Listing 5.21: Final VC to Show

Goal(s):

\((1 + (Total\_Size - 1)) \leq Max\_Capacity)\)

Given(s):

1. \((Total\_Size \leq Max\_Capacity)\)
Chapter 6

A Concept to Capture Acyclic Reference Behavior

The ultimate goal of this research is to make possible automated reasoning about objects that share a global state. In the last chapter, we introduced the basics of shared state specification using a simple communal List concept. In general, sharing requires that the specification language provide constructs to talk about all objects of the same type even though an Operation may have only one of the objects as its explicit parameter. It is important that the specification constructs preserve clean semantics and allow modular reasoning to be done. A Concept that captures reference behavior exhibits all these problems outlined in Section 1.3 and it is the focus of this chapter.

Although explicit references are to be avoided as much as possible for ease of reasoning, there are data structures and algorithms, such as ones with delayed allocation, that take advantage of explicit references to obtain performance benefits. The goal of this chapter is to present a concept that captures acyclic reference behavior. This concept can be used to implement linked data structures such as lists, stack, queues and can be extended to implement trees.

Ideally, reasoning about components with explicit references should be done the same way as those without. Defining a concept specification for capturing acyclic reference behavior helps achieve this goal. This also allows the concept to have interchangeable implementations for performance benefits. A concept that captures reference behavior could have multiple implementations that deal with memory management and garbage collection differently. This will allow the user to pick the
concept that best reflects its performance needs.

Rather than presenting the formal specification upfront, Section 6.1 motivates the concept informally. The concept specification is thoroughly explained in Sections 6.3 and 6.4, using the background formal notation and terminology introduced in Section 6.2. Section 6.5 presents a brief overview of alternative versions of the concept for building other types of acyclic linked data structures.

## 6.1 An Informal Description

The listing below is a skeleton version of a concept that captures acyclic reference behavior.

**Listing 6.1: A Concept that Captures Acyclic Reference Behavior**

```plaintext
Shared Concept Ultimately_Void_Referencing_Template(type Info);

Shared Variables
  Abstract_Var Ref: Location -> Location;
  Abstract_Var Content: Location -> Info;
end Shared Variables;

Type Family Pos;

Operation Give_New_Loc(updates p: Pos);
Operation Redirect_Ref_at(preserves p: Pos; updates referent: Pos);
Operation Follow_Ref(updates p: Pos);
Operation Swap_Content_of(preserves p: Pos; updates I: Info);
Operation Relocate_to(preserves New_L: Pos; replaces p: Pos);
Operation Are_Colocated(preserves p, q: Pos) : Boolean;
Operation Is_Almost_Inaccessible(preserves p: Pos);
Operation Is_Void(preserves p: Pos) : Boolean;
Operation Set_to_Void(clears p: Pos);
end Ultimately_Void_Referencing_Template;
```

**Ultimately_Void_Referencing_Template (UVRT)** is a concept that captures reference behavior and can be thought as a mathematical system of unique *locations*, which can be used to specify the abstract type *Pos*. Each *Pos* points to some data and has one (or more in a *k*-link version of the UVRT) next location(s), which can be accessed using the two global shared state variables, *Ref* and *Content*. Due to the system’s memory constraints, there are only finitely many locations. Initially, all locations are *free* and are only *taken* when the user allocates and takes ownership of that new location. When a location is no longer *accessible*, the location becomes free again. Notice that this concept is generic, therefore a user of this component will need to pass in the type of
information (Info) that is stored inside each Pos.

There is a special default location called Void that never gets allocated to the user. When a new Pos is created by the system, it is automatically located at Void and can be altered by calling UVRT’s operations. In addition to this, all the next links of a location point to Void initially.

The figure below shows a sample chain of Pos that stores integer values. Here, p and q are two positions. From the figure, it is easy to spot that Ref(p) = q and Ref(q) = Void. Similarly, Content(p) = 5 and Content(q) = 17.

![Sample UVRT Chain](image)

**Figure 6.1: Sample UVRT Chain**

In order to perform meaningful actions, the concept must define operations that alter a position’s data and links. The following is a summary of each the operations in the concept that captures reference behavior using traditional reference descriptions. Most of operations are illustrated in Figure 6.2. The Greek letters indicate data of type Info.

- **Give_New_Loc(p):** The variable p of type Pos initially references Void. Once this operation is called, p allocates a memory location with some initial value and it’s next reference points to Void.

- **Redirect_Ref_at(p, q):** p’s next reference is swapped with q’s reference.

- **Follow_Ref(p):** p now points to the next reference. Any dangling references or unreachable memory addresses are finalized to avoid any memory leak.

- **Swap_Content_of(p, d):** The contents pointed by p is swapped with d.

- **Relocate_to(q, p):** q becomes an “alias” reference of p. Similar to Follow_Ref, any dangling references or unreachable memory addresses are finalized to avoid any memory leak.

- **Are_Colocated(p, q):** If p is an “alias” of q, this operation will return true. Otherwise, it returns false.
• **Is_Almost_Inaccessible(p)**: If p does not have an “alias” and the memory pointed by p cannot be reached by any other reference, the operation returns true. Otherwise, it returns false.

• **IsVoid(p)**: This operation returns true if p is equal to Void. Otherwise, it returns false.

• **Set_to Void(p)**: This operation deallocates the memory referenced by p and finalizes other unreachable memory locations. At the end of the operation, p is now equal to Void.

<table>
<thead>
<tr>
<th>Operation Call</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give_New_Location(p);</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Redirect_Ref_at(p, q);</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Follow_Ref(p);</td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Swap_Content_of(p, d);</td>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Relocate_to(q, p);</td>
<td><img src="image9.png" alt="Diagram" /></td>
<td><img src="image10.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 6.2: Illustration for Selected Operations
Listing 6.2: Simple Example that uses UVRT

**Facility** UVRT_Fac is UltimatelyVoidReferencing_Template(Integer) externally realized by Linking_Realiz;

**Procedure** Insert_Front(updates p: Pos; evaluates i: Integer);

Var q: Pos;

Give_New_Loc(q);  
Swap_Content_of(q, i);  
Redirect_Ref_at(q, p);  
q := p;  
end Insert_Front;

A user can instantiate a system of linked locations using the concept described above. During instantiation, the Info type will be bound to an actual programming type. Suppose that a user created a singly-linked structure using an instantiated version of this concept. Listing 6.2 shows a simple example that uses UVRT’s operations.

<table>
<thead>
<tr>
<th>Operation Call</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give_New_Location(q);</td>
<td>![Before Diagram]</td>
<td>![After Diagram]</td>
</tr>
<tr>
<td>Swap_Content_of(q, i);</td>
<td>![Before Diagram]</td>
<td>![After Diagram]</td>
</tr>
<tr>
<td>Redirect_Ref_at(q, p);</td>
<td>![Before Diagram]</td>
<td>![After Diagram]</td>
</tr>
</tbody>
</table>

Figure 6.3: Illustration of the Effects of Operation Calls in Insert_Front
Listing 6.2 uses an instantiation of UVRT that stores Integers. Insert_Front is an operation that provides a way to insert \( i \) to the front of a chain of Pos. Figure 6.3 illustrates the effects of the operation calls. Notice that the figure assumes that the incoming \( p \) is a Pos storing the integer 5, the incoming \( i \) contains the integer 2 and \( q \) has been initialized to Void.

Similarly, using the operations provided, the linked structure reversal with explicit references (Listing 1.3) presented in Section 1.2 can be written as follows:

Listing 6.3: Reversal using Reference Component

```
-- Definitions that will be helpful to define the invariant and termination
Definition Info_String(p : Pos) : Str = ...;
Definition Distance_toVoid(p : Pos) : N = ...;

Procedure Reverse_List(updates i: Pos);
    Var j: Pos;
    While (not IsVoid(i))
        maintaining Info_String(#i) = Reverse(Info_String(j)) o Info_String(i);
        decreasing Distance_toVoid(i);
        do
            Redirect_Ref_at(i, j);
            i :=: j;
        end;
    end Reverse_List;
```

The figure below illustrates how the reversal occurs for each iteration of the loop. As expected, this figure is almost identical to Figure 1.7.

Figure 6.4: Reversing a singly-linked location
The definition `Info_String` takes a location and constructs a mathematical string from reachable memory addresses’ content and `Distance_to_VOID` returns the distance to `VOID`. The loop invariant states that the string of contents in the incoming location `i` is the same as the reverse of the contents in `j` concatenated with the string of contents in the location `i`.

The `decreasing` clause introduces what is needed to prove that this loop terminates. The definition `Distance_to_VOID` is used to claim that location `i` is moving closer to `VOID` after each iteration of the loop. Once `i` is equal to `VOID`, the loop condition no longer holds and the loop is terminated.

6.2 New Mathematical Theories Needed To Describe Reference Behavior

The new constructs and mathematical definitions presented in the following subsections will be useful in writing the specifications for a component that captures reference behavior (Section 6.3).

6.2.1 Construct: [...]

\[(f : (D : \text{Set}) \to (R : \text{Set}))[S : \mathcal{P}(D)] : \mathcal{P}(R)\]

The construct above allows for a function `f` to be applied repeatedly to the `Powerset` of the domain `D` and returns the `Powerset` of the range `R`. Using `Ref` function from Listing 6.1 and the sample chain in Figure 6.1, `Ref[p] = \{q, VOID\}`. Similarly, `Content[p] = \{5, 17\}`.

6.2.2 Definition: `Fn_Restricted_to`

\[f \upharpoonright S\]

The key idea here is to restrict the function `f` to use a subset of the domain `f, S`. Since the current compiler processes only ASCII characters, an ASCII equivalent version of the mathematical definition `Fn_Restricted_to` is written and used. This definition takes in a function from `D` to `R` and a `Powerset` of `D` and returns the result of `S \to R`. This definition will be useful to restrict the
domain to accessible locations, rather than all the locations provided by a component that captures reference behavior.

6.2.3 Definition: Is Closed with respect to

$$\text{Is\_Closed\_wrt}(U : \text{Set}, FC : \mathcal{P}(U \to U), S : \mathcal{P}(U)) : \mathbb{B} = ( \bigcup_{f \in FC} f[S] \subseteq S)$$

A set $S$ is closed with respect to the Powerset of functions in $FC$, if the range of applying $f$ over the domain $S$ is a subset of $S$. Suppose that $FC$ is a singleton set containing a function $F : S \to S$, then pictorially this definition can be shown as follows:

![Figure 6.5: Is\_Closed\_wrt and ¬ Is\_Closed\_wrt with One Function](image)

The $\text{Is\_Closed\_wrt}$ definition can also be applied to multiple functions. In the following figure, the function $H : S \to S$ is also in $FC$. Now in order for the domain $S$ to satisfy this definition, it must be closed with respect to both $F$ and $H$. If either function fails to satisfy this definition, then $S$ is not closed under this definition.
6.2.4 Definition: Closure for

\[ \text{Closure\_for}(U : \text{Set}, FC : \mathcal{P}(U \to U), G : \mathcal{P}(U)) : \mathcal{P}(U) \]

The Closure\_for definition returns a set that results from applying the functions in the Powerset FC repeated to the set of elements in G; G is a subset of U. The figure below depicts an application of Closure\_for over the functions F and H.

Figure 6.7: Closure\_for
In order for $G$ to satisfy the Closure_for definition, there are multiple conditions that must be met. First, as shown in Figure 6.7, $G \subseteq \text{Closure}_\text{for}(U, FC, G)$. Second, all the functions in $FC$ must be closed with respect to the $\text{Closure}_\text{for}(U, FC, G)$. Lastly, $\forall S : \mathcal{P}(U)$, if $G$ is a subset of $S$ and $S$ is closed with respect to the set $FC$, then $\text{Closure}_\text{for}(U, FC, G) \subseteq S$.

As an example, $\text{Closure}_\text{for}$ can also be used with the chain in Figure 6.1. Our $U$ is the set of all Locations, $FC$ is the singleton set containing the function Ref and $G$ is the set containing $p$ and $q$. The results of applying the $\text{Closure}_\text{for}$ function should be the set: $\{q, \text{Void}\}$.

### 6.2.5 Definition: Is Stable with respect to

$$\text{Is}_\text{Stable}_\text{wrt}(U : \text{Set}, FC : \mathcal{P}(U \rightarrow U), S : \mathcal{P}(U)) : \mathbb{B} = (\bigcup_{f : FC} f[S] = S)$$

A set $S$ is stable with respect to the Powerset of functions in $FC$, if the range of applying $f$ over the domain $S$ is a equal to $S$. Suppose that $FC$ is a set containing a function $F : S \rightarrow S$ and $H : S \rightarrow S$, then pictorially this definition can be shown as follows:

![Figure 6.8: Is_Stable_wrt](image)

In this case, the function application never takes outside of the set $S$ and captures all the elements that are in $S$. 

68
6.2.6 Definition: Terminal Range

\[ \text{Terminal\_Range}(U : \text{Set}, FC : \mathcal{P}(U \rightarrow U), G : \mathcal{P}(U)) : \mathcal{P}(U) \]

The general application of \( \text{Terminal\_Range}(U, \{F, H\}, G) \), where \( U \) is a set, \( G \) is a subset of \( U \) and \( F \) and \( H \) are functions, returns a set of elements that result from applying the functions \( F \) and \( H \) to the limit of each member of the set \( G \). Figure 6.9 provides an illustration of this definition.

![Terminal\_Range](image)

Figure 6.9: Terminal\_Range

This definition will be useful to define that all the references reach a particular reference and can be used to specify that no cycles are formed. Once again using the chain in Figure 6.1, the \( \text{Terminal\_Range}(\text{Locations}, \{\text{Ref}\}, \{p, q\}) \) yields \{Void\}.

6.3 Formal Specification of Communal\_UVR\_Template

Using the new mathematical theories as a base, this section presents a formal specification for a communally bounded referencing concept. Communal\_UVR\_Template (CBUVRT) is a specialized version of a concept that captures reference behavior that is especially suitable for implementing communally bounded and acyclic structures. The Max\_Capacity bound restrict the total number of locations that can be used. In order to better explain CBUVRT specification, the concept has been broken down to smaller sections. The complete specification is given in Appendix G.
Listing 6.4: CBUVRT (Shared State)

**Shared Concept** Communal_UVR_Template(type Info; evaluates Max_Capacity);
uses Function_Theory, Terminal_Range_Op_Ext;
requires Max_Capacity > 0 which entails Max_Capacity : N;

Defines Location: Set;
Defines Void: Location;

**Shared Variables**
Abstract_Var Ref: Location -> Location;
Abstract_Var Content: Location -> Info;

**constraints** Terminal_Range(Location, {Ref}, Location) ⊆ {Void}
which entails Ref(Void) = Void;

**initialization**
ensures Ref[Location] = {Void} and
Info.Is_Initial[Content[Location]] = {True};

end;

Listing 6.4 uses Function_Theory and Terminal_Range_Op_Ext to import the formal function definitions and notations that will be used in the specifications of **Shared Variables**, **Type Family** and CBUVRT’s operations. In the listing, the Location set is an abstraction of the address space and its actual size is defined and constrained by an implementation on the underlying machine. Void is a special Location. A key idea in the concept is the use of two global state variables to capture the **Shared State**: Ref, a function that gives the “next” location for a given location and Content, a function that gives the information value referenced by a given location. The figure below depicts how the global state variables captures the share state.

---

CBUVRT specification has been designed with the goal of enabling automated verification. Specifically, through carefully defined notations and theories, it avoids the use of quantifiers in assertions entirely.
6.3.1 Absence of Cycles

In Listing 6.4, the key constraint is that following the next Ref chain for every location, will ultimately reference (or reach) the Void location. This constraint is the basis for the name for the concept and ensures that there are no cycles. Since this constraint is already a given, when implementing a (communally bounded) list (or stack or tree) using CBUVRT, it becomes a freely established representation invariant that requires no further proof.

In order to formally express this constraint, the specification uses the mathematical definition: Terminal_Range. The constraint requires that the terminal range for repeatedly applying the Ref function over all Locations is simply Void. The which entails clause specifies that the result of applying Ref to Void is simply Void. Notice that this is something that needs to be proved when verifying the correctness of CBUVRT. Once established to be true, it becomes a useful lemma in the automated verification process.

When CBUVRT is instantiated, it ensures initially (only conceptually, of course) that all the locations reference Void. In this assertion, Ref[Location] denotes the set of range values for all Locations are Void and further ensures that the Content of all Locations are all initial values of type Info.

6.3.2 Absence of Memory Leaks

In Listing 6.5, CBUVRT defines a programming type Pos to represent a pointer, mathematically modeled as a Location. Initially, each position takes the value Void. (exemplar is just an example value of type Pos) A second key idea is “accessibility” and it is specified by a variable mathematical definition Accessible_Loc, whose value depends upon the global state variable Ref.

Listing 6.5: CBUVRT (Type Definition)

Type Family Pos is modeled by Location;
  exemplar p;
  initialization
    ensures p = Void;

Def Var Accessible_Loc: P(Location) = (\{Void\} U Closure_for(Location, \{Ref\}, Pos.Val_in[Pos.Receptacles]));

finalization
  affects Ref, Content, Accessible_Loc;
ensures Ref = λ q: Location. (
    { #Ref(q) if q ∈ {Void} ∪ Closure_for( Location, {Ref},
        #Pos.Val_in[#Pos.Receptacles ~ {recp.p}]) } )
    Void otherwise
and Content | Accessible_Loc = #Content | Accessible_Loc
which entails
    if #p ∈ {Void} ∪ Closure_for( Location, {Ref},
        #Pos.Val_in[#Pos.Receptacles ~ {recp.p}] ),
then Ref = #Ref and Accessible_Loc = #Accessible_Loc and
    Content = #Content;

constraint Info.Is_Initial[ Content[Location ~ Accessible_Loc] ⊆ {True}
and Ref[Location ~ Accessible_Loc] ⊆ {Void} and
||Accessible_Loc|| : N and ||Accessible_Loc|| <= Max_Capacity;
end;

The formal definition of Accessible_Loc is based on Closure_for, also defined and elaborated in Terminal_Range_Op_Ext theory. Recall that Closure_for(U, {F, H}, G) returns a set that results from applying the functions F and H repeated to the set of elements in G; G is a subset of U. Here, Accessible_Loc is the set of reachable locations produced by the Closure_for on all programming variables of type Pos (i.e., all void-referencing pointer variables), unionized with Void.

In the definition of Accessible_Loc as well in the specification of operations, the following notations are used. They are a part of the specification language that allow us to make assertions about all objects or a specific object of a certain type or refer to the actual programming variable associated with a name.

- \( T.\text{Receptacles} \) denotes the set of all variables of type \( T \) that have been initialized, but not finalized

- \( \text{recp}.p \) is a specification language construct, it refers to the actual variable that will be associated with \( p \)

- \( \text{Val_in} (\text{recp}.p) \) denotes the mathematical value corresponding to the receptacle \( p \)

The finalization of a CUVRT position variable (or pointer) will have to deal with two scenarios. For all locations in,q ∈ Accessible_Loc that are accessible from the set of all allocated locations minus p (the pointer that is being finalized or removed), then no changes are done to their references, i.e., Ref(q) = Ref(#q). However, if some q is no longer accessible, because of the finalization of p, then q becomes available for allocation. The figure below provides an illustration of this process with the location p being finalized.
In other words, every location is either available for allocation or is accessible, i.e., there are no memory leaks. The specification of CBUVRT demands that the underlying implementation of it do garbage collection. Using the updated Accessible_Loc, the specification also states that that the Content prior to finalizing p is equal to the Content after finalizing p and all the Locations that are only reachable from p.

### 6.4 Formal Specification for CBUVRT Operations

The CBUVRT operation’s formal specifications are given as follows:

Listing 6.6: CBUVRT (Operations)

```plaintext
Operation Give_New_Loc (updates p: Pos);
  affects Accessible_Loc;
  requires p = Void and ||Accessible_Loc|| < Max_Capacity;
  ensures p ∉ #Accessible_Loc;

Operation Occupied_Size() : Integer;
  ensures Occupied_Size = ( ||Accessible_Loc|| );

Operation Redirect_Ref_at (preserves p: Pos, updates referent: Pos);
  affects Ref;
  requires p ∉ Closure_for(Location, {Ref}, {referent});
  ensures Ref = λ q: Location.(
    { #referent if q = p
    { #Ref(q) otherwise }
  and referent = #Ref(p);
```

Figure 6.11: Finalization
Operation Follow_Ref (updates p : Pos);
affects Ref, Accessible_Loc, Content;
requires p ≠ Void;
ensures Ref = λ q: Location. ( 
    { 
        Void if q = #p and #p ∉ Closure_for( Location, {Ref}, 
            #Pos.Val_in[#Pos.Receptacles ∼ {recp.p}] ) 
    } 
    #Ref(q) otherwise
)
and Content | Accessible_Loc = #Content | Accessible_Loc
which_entails
    if #p ∉ Closure_for(Location, {Ref}, 
            #Pos.Val_in[#Pos.Receptacles ∼ {recp.p}] ),
    then Ref = #Ref and Accessible_Loc = #Accessible_Loc and
    Content = #Content;

Operation Swap_Content_of (preserves p: Pos, updates I: Info);
affects Content;
requires p ≠ Void;
ensures Info = #Content(p) and Content = λ q: Location. ( 
    { 
        #I if q = p 
    } 
    #Content(q) otherwise
);

Operation Relocate_to (preserves New_L: Pos; replaces p : Pos);
affects Ref, Accessible_Loc, Content;
ensures p = New_L and Ref = λ q: Location. ( 
    { 
        #Ref(q) if q ∈ {Void} U Closure_for( Location, {#Ref}, 
            #Pos.Val_in[#Pos.Receptacles ∼ {recp.p}] ) 
    } 
    Void otherwise
)
and Content | Accessible_Loc = #Content | Accessible_Loc
which_entails
    if #p ∈ (Void) U Closure_for( Location, {Ref}, 
            #Pos.Val_in[#Pos.Receptacles ∼ {recp.p}] ),
    then Ref = #Ref and Accessible_Loc = #Accessible_Loc and
    Content = #Content;

Operation Are_Colocated (preserves p, q: Pos): Boolean;
enforces Are_Colocated = ( p = q );

Operation Is_Across_Inaccessible (preserves p: Pos): Boolean;
enforces Is_Across_Inaccessible = ( p ∉ {Void} U Closure_for(Location, 
    {Ref}, Pos.Val_in[Pos.Receptacles ∼ {recp.p}] ) );

Operation Is_Void (preserves p: Pos): Boolean;
enforces Is_Void = ( p = Void );

Operation Set_to_Void (clears p: Pos);
affects Ref, Accessible_Loc, Content;
enforces Ref = λ q: Location. ( 
    { 
        #Ref(q) if q ∈ {Void} U Closure_for( Location, {#Ref}, 
            #Pos.Val_in[#Pos.Receptacles ∼ {recp.p}] ) 
    } 
    Void otherwise
)
and Content | Accessible_Loc = #Content | Accessible_Loc
which entails

if \( p \in \{ \text{Void} \} \cup \text{Closure_for( Location, \{Ref\),\
\#Pos. Val_in[\#Pos. Receptacles \sim \{recp.p\} ] ),
then \ Ref = \#Ref and \text{Accessible_Loc} = \#Accessible_Loc and
\text{Content} = \#Content; \)

1. Give_New_Loc: Takes ownership of a location from the system. The location must be unclaimed and there must be at least one free location available. This location is accessible until there are no more references to this location.

2. Occupied_Size: Returns an integer value that indicates the number of locations accessible.

3. Redirect_Ref_at: Redirect the referent’s reference to incoming \( p \)'s successor link. The outgoing \( p \) will now contain the incoming referent’s location.

4. Follow_Ref: This operation requires that \( p \) is not Void. \( p \) now references the incoming \( p \)'s next location. Notice that this operation might have affected accessibility. If there are no more references to the incoming value of \( p \) and/or any locations that were previously only accessible through \( p \), then all those inaccessible locations are finalized and freed.

5. Swap_Content_of: This operation requires that \( p \) is not Void and ensures that the content in \( p \) is swapped with the value in \( I \).

6. Relocate_to: The reference \( p \) is relocated to the location pointed by New_L. Notice that this operation might have affected accessibility. If there are no more references to the incoming value of \( p \) and/or any locations that were previously only accessible through \( p \), then all those inaccessible locations are finalized and freed.

7. Are_Colocated: If both \( p \) and \( q \) reference the same location, this operation will return true. Otherwise, it will return false.

8. Is_Almost_Inaccessible: If the location that \( p \) references does not have any additional references pointing to it, the operation returns true. Otherwise, it will return false.

9. Is_Void: This operation returns true if \( p \) is equal to \( \text{Void} \), otherwise it is false.

10. Set_to_Void: This operation finalizes \( p \) by setting it equal to \( \text{Void} \). Notice that this operation might have affected accessibility. If there are no more references any locations that were
previously only accessible through p, then all those inaccessible locations are finalized and freed.

6.5 Alternative Versions of UVRT

The referencing concept presented in this chapter is only one variation of an acyclic referencing concept. It is possible to slightly modify CBVVRT to create other versions that will be better suited for implementing different data structures. This section briefly introduces two of such variations.

6.5.1 Brief Description of Globally Bounded UVRT

Chapter 3 introduced a Globally_Bounded_List_Template that is globally bounded by the system’s memory and does not contain a user supplied maximum size. Similarly, there is a modified CBUVRT that does not require an user-supplied Max_Capacity bound and can be used to implement these linked-structures. A performance profile [100] that includes specifications for memory allocation and management can be added to ensure the instantiated UVRT does not exceed the system’s memory. Memory management is a topic beyond this dissertation work and will be a topic for future research directions.

6.5.2 Brief Description of \( n \)-link UVRT

The presented CBUVRT and the modified concept from the previous subsection can only be used to implement single-linked components. However, the CBUVRT concept can be extended to allow multiple links per node in order to implement structures like trees. This subsection briefly discusses the changes to CBUVRT to provide a communally bounded \( n \)-link UVRT.

In addition to the actual type information, the concept must also be parameterized by an integer \( k \) to denote the number of links per node. By introducing \( k \), the global state variable \( \text{Ref} \) will now be defined by \( \text{Location} \times [1...k] \rightarrow \text{Location} \) to indicate that the application of \( \text{Ref} \) will depend on an integer from 1 to \( k \). This illustrated in Figure 6.12. The specification must further ensure that the Terminal_Range for all possible links in each location is the Void location.
The definition for accessibility will also need to include an integer from 1 to \( k \) to distinguish the different links. When an operation affects accessibility, there is a need to check every location that is reachable by following the \( \text{Ref} \) links. For example, if a middle node of the tree is finalized, then all of its children should be finalized as well. However, both the finalization and operation ensures should only affect locations that were only previously accessible by following the modified location and not other locations in other links.

A specification of an \texttt{Exploration\_Tree\_Template} is discussed in [72]. The \( k \)-position \texttt{CBUVRT} concepts can be directly used to realize it following the ideas presented in the next chapter.
Chapter 7

VC Generation and Evaluation for Concept Realizations

While the implementations shown in the prior chapters have used data abstractions to hide explicit reference behavior, the problem of implementing lower-level “linked” realizations such as for lists and trees needs to be addressed. Since these concept realizations may involve explicit reference behavior, we will need to explore how to reason about code for an operation affecting objects other than its parameters (also known as frame property).

In order to better evaluate the specification and verification of concept realizations, they have been divided into the different categories shown in Table 7.1. Each of the following sections in this chapter will focus on how to reason about the correctness of an implementation of each category and whether or not there is a need to include reasoning about the frame property. Other categories of sharing implementations will be discussed as future work in Chapter 8.

<table>
<thead>
<tr>
<th>Section</th>
<th>Category Name</th>
<th>Example Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Non-Sharing Implementations</td>
<td>Implementations for bounded data abstractions</td>
</tr>
<tr>
<td>7.2</td>
<td>Interference-Free Sharing Implementations with Independent Correspondence</td>
<td>Communally-bounded implementations</td>
</tr>
<tr>
<td>7.3</td>
<td>Interference-Free Sharing Implementations with Independent Regional Correspondence</td>
<td>Communal UVRT-based implementations</td>
</tr>
<tr>
<td>8</td>
<td>Other sharing implementations</td>
<td>Future Work</td>
</tr>
</tbody>
</table>

Table 7.1: Categories of Concept Realization

78
7.1 Non-Sharing Implementations

This section focuses on concept realizations that do not contain any Shared Variables or Def Vars. This category includes implementations similar to those presented in [42] and [91], which serve as regression testing for the new verification condition generator. Section 7.1.1 introduces a globally bounded Stack concept and presents an implementation using the globally bounded List introduced in Chapter 3. Section 7.1.2 introduces a bounded Stack concept and includes an array implementation. Since there is no sharing involved in this category, we leverage RESOLVE’s clean semantics property and avoid the need to reason about the frame property.

7.1.1 Globally-Bounded Stack

Globally_Bounded_Stack_Template is a generic Concept with a single parameter Entry to represent a generic entry type. Like the globally bounded List, it is also modeled mathematically as an Entry string. The type’s initialization ensures clause indicates that the Stack is initially empty.

Listing 7.1: A Globally-Bounded Stack Concept

```plaintext
Concept Globally_Bounded_Stack_Template(type Entry);
uses String_Theory;

Type Family Stack is modeled by Str(Entry);
exemplar S;
 initialization ensures S = Empty_String;
end;

Operation Push(alters E: Entry; updates S: Stack);
ensures S = <$E> o #S;

Operation Pop(replaces R: Entry; updates S: Stack);
requires not(S = Empty_String);
ensures #S = <R> o S;

Operation Is_Empty(restores S : Stack) : Boolean;
enforces Is_Empty = (S = Empty_String);

Operation Clear(clears S: Stack);
end Globally_Bounded_Stack_Template;
```

This Concept, which exports type Stack, provides the following operations: Push, Pop, Is_Empty and Clear. Push concatenates the incoming value of E with S, a string of entries to form
a new string. The requires clause of Pop indicates that $S$ isn’t empty and ensures that the first element in $S$ is removed and placed in $R$. Is_Empty returns a Boolean to indicate if $S$ is empty and Clear removes all entries from $S$.

The following is a realization of `Globally_Bounded_Stack_Template` using an instantiated facility of *globally bounded* List.

Listing 7.2: A Globally-Bounded List Realization

```plaintext
Realization GBList_Based_Realiz for Globally_Bounded_Stack_Template;
  uses String_Theory;

  Facility GB_List_Fac is Globally_Bounded_List_Template(Entry)
    externally realized by UVRT_Realiz;

  Type Stack = GB_List_Fac::List;
    convention
      S.Prec = Empty_String;
    correspondence
      Conc.S = S.Rem;
  end;

  Procedure Push(alters E: Entry; updates S: Stack);
    Insert(E, S);
  end Push;

  Procedure Pop(replaces R: Entry; updates S: Stack);
    Remove(R, S);
  end Pop;

  Procedure Is_Empty(restores S : Stack) : Boolean;
    Is_Empty := Is_Rem_Empty(S);
  end Depth;

  Procedure Clear(clears S: Stack);
    Clear(S);
  end Clear;

end GBList_Based_Realiz;
```

The type `Stack` is implemented directly using `GB_List_Fac::List`. This means that inside this realization, a `Stack` is simply a `List`, therefore we can call any of the List operations by passing a `Stack`. The type realization block also contains a `convention` (or `representation invariant`) clause, a `correspondence` (or `abstraction function/relation`) clause, an `initialization` block that is used to add additional code for initializing the type and a `finalization` block for adding additional code for finalizing the type. This `Stack` realization does not require any additional code for initialization or finalization; therefore these blocks are empty.
Notice that the convention clause indicates that the \texttt{Prec} string must be empty. This clause holds at the beginning of each \texttt{Operation} defined in the Concept and must be ensured that it holds at the end. This clause must also be proved to hold at the end of \texttt{initialization}. (It may be assumed at the beginning of \texttt{finalization}.) So at the end of each operation, including \texttt{initialization}, but excluding \texttt{finalization}, the VC generator will generate verification conditions to ensure that the type conventions hold.

A correspondence clause indicates how the concrete internal representation is related to the abstract mathematical model of the type it is implementing. In this case, the correspondence clause specifies that the conceptual \texttt{Stack (Conc.S)} is equal to the \texttt{Rem} string of the \texttt{List} used to model the \texttt{Stack}. Figure 7.1 depicts the relationship between the conceptual type’s constraint, the type representation’s convention and type representation’s correspondence. Note that there is a VC for establishing that the correspondence is well defined. This means that the realization level assertions and the convention and correspondence clauses must allow us to show that the type constraint is satisfied.

![Figure 7.1: Relationship Between Type Constraint, Convention and Correspondence](image)

The VCs generated for this realization can be found in Appendix E and the proof rules used to generate these VCs can be found in Appendix A. As examples, VCs for establishing \texttt{initialization ensures} clause of \texttt{globally bounded Stack}, \texttt{type convention} and \texttt{ensures} clause of \texttt{Push} are shown in Listings 7.3, 7.4 and 7.5.

VC in Listing 7.3 has been simplified after substituting the type correspondence and the \texttt{initialization ensures} clause of \texttt{List} into the goal and does not require any additional givens to establish its correctness.
Listing 7.3: VC for Establishing the Initialization Ensures Clause of Stack

**Goal:**

(Empty_String = Empty_String)

**Given:**

The call to Insert in Push does not modify the Prec string, therefore the VC in Listing 7.4 is straightforward.

Listing 7.4: VC for Establishing the Type Convention for Stack Generated by Push

**Goal:**

(S'.Prec = Empty_String)

**Given:**

1. (S’.Prec = Empty_String)
2. (S’.Rem = (<E> o S.Rem))

The following VC is also straightforward after applying the proper substitutions into the conceptual ensures clause for Push.

Listing 7.5: VC for Establishing the Ensures Clause of Push

**Goal:**

((<E> o S.Rem) = (<E> o S.Rem))

**Given:**

1. (S’.Prec = Empty_String)

### 7.1.2 Bounded Stack

The concept shown below is a specification for bounded Stack. Apart from the generic type parameter Entry, the parameter Max_Depth constraints the maximum capacity for each Stack. Although the Max_Depth for Stacks from the same facility instantiation is the same value, this bound is an individual bound that is not shared. Notice that Stack_Template requires that the Max_Depth is at least 1.
Listing 7.6: A Bounded Stack Concept

Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
uses String_Theory, Integer_Ext_Theory;
requires 1 <= Max_Depth;

Type Family Stack is modeled by Str(Entry);
    exemplar S;
    constraint |S| <= Max_Depth;
    initialization
        ensures S = Empty_String;
end;

Operation Push(alters E: Entry; updates S: Stack);
    requires 1 + |S| <= Max_Depth;
    ensures S = <#E> o #S;

Operation Pop(replaces R: Entry; updates S: Stack);
    requires 1 <= |S|;
    ensures #S = <R> o S;

Operation Depth(restores S: Stack): Integer;
    ensures Depth = ( |S| );

Operation Rem_Capacity(restores S: Stack): Integer;
    ensures Rem_Capacity = ( Max_Depth - |S| );

Operation Clear(clears S: Stack);
end Stack_Template;

There are a few differences between Globally_Bound_Stack_Template and Listing 7.6. The type constraint indicates that the Stack depth must be less than or equal to Max_Depth at all times. Push requires that the length of parameter S is strictly less than Max_Depth and Pop’s requires clause simply indicates that the length of the parameter S is greater than or equal to 1. Instead of the Is_Empty operation, this concept provides Depth and Rem_Capacity operations for obtaining the depth of the Stack and its remaining capacity respectively.

One possible implementation of this concept is using an array\(^1\), which is shown in the listing below. This Stack realization is a record that contains an array called Contents and an integer Top that indicates the index position of the top element in the Stack.

\(^1\)In RESOLVE, there is also an abstraction for arrays, but the compiler allows for the more traditional array notations to be used and the notations are automatically converted during the compilation process.
Listing 7.7: An Array Realization

Realization Array_Realiz for Stack_Template;

Type Stack is represented by Record
  Contents: Array 1..Max_Depth of Entry;
  Top: Integer;
end;
convention
  0 <= S.Top <= Max_Depth;
correspondence
  Conc.S = Reverse(Iterated_Concatenation(1, S.Top,
             lambda (i : Z).(S.Contents(i))));
end;

Procedure Push (alters E: Entry; updates S: Stack);
  S.Top := S.Top + 1;
  E :=: S.Contents[S.Top];
end Push;

Procedure Pop (replaces R: Entry; updates S: Stack);
  R :=: S.Contents[S.Top];
  S.Top := S.Top - 1;
end Pop;

Procedure Depth (restores S: Stack): Integer;
  Depth := S.Top;
end Depth;

Procedure Rem_Capacity (restores S: Stack): Integer;
  Rem_Capacity := Max_Depth - S.Top;
end Rem_Capacity;

Procedure Clear (clears S: Stack);
  S.Top := 0;
end Clear;
end Array_Realiz;

The type convention restricts the value of S.Top to be between 0 and Max_Depth to ensure that it is a valid index into the Contents array. In this implementation, the type correspondence indicates that the conceptual Stack is equal to the reverse of the iterated concatenation of elements in S.Contents from 1 to S.Top. This allows elements to be pushed or popped from the Stack without having to shift elements in the array. The VC for establishing the well defined correspondence is shown below. Givens #2 and #3 indicate that S.Top is between 0 and Max_Depth, which allow us to determine that the length of our string of entries is between 0 and Max_Depth and thus prove the correctness of this VC.
Listing 7.8: VC for Establishing the Well Defined Correspondence for Stack

**Goal:**

\(|\text{Reverse(Iterated\_Concatenation(1, S.\text{Top}, \lambda \ (i : \mathbb{Z}).(<S.\text{Contents}([\text{Universal}] i>)))|) \leq \text{Max\_Depth}}|

**Given:**

1. \((1 \leq \text{Max\_Depth})\)
2. \((0 \leq \text{S.\text{Top}})\)
3. \((\text{S.\text{Top}} \leq \text{Max\_Depth})\)

Notice that Push, Pop and Clear modify the value of S.\text{Top} respectively in order to satisfy both the type convention and its corresponding ensures clause from the concept. As an example, Listings 7.9 and 7.10 show the VCs for establishing that the type convention holds at the end of Push and Listing 7.11 is the VC for establishing that Push's ensures clause is satisfied. All other VCs for this realization can be found in Appendix D.

Listing 7.9: VC for Establishing the Type Convention for Stack Generated by Push

**Goal:**

\((0 \leq (\text{S.\text{Top}} + 1))\)

**Given:**

1. \((E' = S.\text{Contents}(\text{S.\text{Top}}))\)
2. \((S.\text{Contents}' = \lambda (j : \mathbb{Z}).(\text{E if } ([\text{Universal}] j = (\text{S.\text{Top}} + 1))\text{S.\text{Contents}([\text{Universal}] j) otherwise}))\)
3. \((0 \leq \text{S.\text{Top}})\)
4. \((\text{S.\text{Top}} \leq \text{Max\_Depth})\)
5. \(((1 + |\text{Reverse(Iterated\_Concatenation(1, S.\text{Top}, \lambda \ (i : \mathbb{Z}).(<S.\text{Contents}([\text{Universal}] i>)))|)\|) \leq \text{Max\_Depth})\)
6. \((1 \leq \text{Max\_Depth})\)

In this VC, we are trying to establish that after increasing S.\text{Top} by 1, it is still greater than our lower bound for the type convention. Given #3 provides the fact that S.\text{Top} must be greater than or equal to 0 and adding 1 to S.\text{Top} should still ensure that this is true, therefore it is possible to prove this VC.
Listing 7.10: VC for Establishing the Type Convention for Stack Generated by Push

**Goal:**

((S.Top + 1) <= Max_Depth)

**Given:**

1. (E’ = S.Contents(S.Top))
2. (S.Contents’ = lambda (j : Z).(E if ([Universal] j = (S.Top + 1)) S.Contents([Universal] j) otherwise))
3. (1 <= Max_Depth)
4. (0 <= S.Top)
5. (S.Top <= Max_Depth)
6. ((1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))) <= Max_Depth)

However, proving that S.Top + 1 <= Max_Depth requires a little more work. Notice that given #6 is the requires clause of Push after substituting in the type correspondence. It indicates that 1 plus the length of the iterated concatenation of all elements from 1 to S.Top in S.Contents is less than or equal to Max_Depth. This allow us to determine that by incrementing the value of S.Top by 1, it is still less than or equal to Max_Depth and establish the correctness of this VC.

Listing 7.11: VC for Establishing the Ensures Clause of Push

**Goal:**

(Reverse(Iterated_Concatenation(1, (S.Top + 1), lambda (i : Z).(<S.Contents’([Universal] i)>))) = (<E> o Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>)))))

**Given:**

1. (E’ = S.Contents(S.Top))
2. (S.Contents’ = lambda (j : Z).(E if ([Universal] j = (S.Top + 1)) S.Contents([Universal] j) otherwise))
3. (0 <= S.Top)
4. (S.Top <= Max_Depth)
5. ((1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))) <= Max_Depth)
6. (1 <= Max_Depth)

The VC shown above must be proven in order to establish that the code used to implement Push satisfies the ensures clause from the concept. Given #2 indicates that S.Contents’ is equal to S.Contents except at index S.Top + 1. This means that the reverse of the iterated concatenation of elements from 1 to S.Top + 1 in S.Contents’ will have E as the first element in
the string followed by the reverse of elements from 1 to $S$.Top in $S$.Contents, thus proving this VC.

Listing 7.12: A Communally-Bounded Stack Concept

```
Shared Concept Communally_Bounded_Stack_Template{
    type Entry; evaluates Max_Capacity: Integer;
    uses String_Theory, Set_Theory, Integer_Ext_Theory;
    requires 1 <= Max_Capacity which entails Max_Capacity is_in N;

    Shared Variables
    Abstract_Var Total_Size: N;

    constraint Total_Size <= Max_Capacity;
    initialization
    ensures Total_Size = 0;
    end;

    Type Family Stack is modeled by Str(Entry);
    exemplar S;

    initialization
    ensures S = Empty_String;
    finalization
    affects Total_Size;
    ensures Total_Size = #Total_Size - #S;
    end;

    Operation Push(alters E: Entry; updates S: Stack);
    affects Total_Size;
    requires 1 + Total_Size <= Max_Capacity;
    ensures S = <#E> o #S and
    Total_Size = #Total_Size + 1;

    Operation Pop(replaces R: Entry; updates S: Stack);
    affects Total_Size;
    requires 1 <= |S|;
    ensures #S = <R> o S and
    Total_Size = #Total_Size - 1;

    Operation Depth(restores S: Stack) : Integer;
    ensures Depth = (#S);

    Operation Occupied_Size(): Integer;
    ensures Occupied_Size = ( Total_Size );

    Operation Clear(clears S: Stack);
    affects Total_Size;
    ensures Total_Size = #Total_Size - #S;

    end Communally_Bounded_Stack_Template;
```
7.2 Interference-Free Sharing Implementations with Independent Correspondence

The previous section outlined how we reason about concept implementations where there is no sharing involved. In this section, we explore and reason about an implementation of a *communally bounded* Stack using the *communally bounded* List concept presented in Chapter 5. Listing 7.12 presents the specifications for Communally_Bounded_Stack_Template. It contains a user supplied communal bound Max_Capacity and includes a Shared Variables construct with a global state variable Total_Size to keep track of the total number of elements stored across all Stacks generated from the same communal stack instantiation. Total_Size is initially 0 and is constrained to be less than or equal to Max_Capacity.

The type specification is similar to the concept presented in Sections 7.1.1 and 7.1.2, except that it adds the specification for finalization. Finalizing a communal stack affects the global state variable Total_Size, and it ensures that the resulting Total_Size is equal to the Total_Size prior to finalizing S minus the number of elements in the stack S prior to finalization.

The operation specifications are very similar to the ones given in the bounded Stack. However, notice that operations Push, Pop and Clears affect Total_Size, while operations Depth and Occupied_Size do not. Push now requires that 1 + Total_Size <= Max_Capacity and additionally ensures that Total_Size increases by 1. Conversely, Pop additionally ensures that Total_Size decreases by 1. The Depth operation returns the number of elements in the stack S, while Occupied_Size returns Total_Size. Lastly, Clears has an ensures clause that states that Total_Size decreases by the number of the elements in the incoming stack, S.

An implementation of this concept using communal list is given in Listing 7.13. Notice that rather than declaring a variable to keep track of the conceptual Total_Size, the implementation directly uses the instantiated facility's (CB_List_Fac) Total_Size. The involves keyword indicates that a correspondence clause affect the provided Shared Variables and/or Def Vars. In this realization, only the Shared Variables' correspondence involves a global shared variable. The type correspondence is an independent correspondence, where changes to any global state variables by code in any of the concept defined operations, do not affect objects other than its parameters. Even though the Shared Variables' correspondence involves other global state variables, the implementing Procedures can indicate there is are changes through the use of affects and internal
Listing 7.13: A Communally-Bounded List Realization

Realization CBLList_Based_Realiz for Communally_Bounded_Stack_Template;
uses String_Theory;

Facility CB_List_Fac is
Communally_Bounded_List_Template(Entry, Max_Capacity)
externally realized by UVRT_List_Realiz;

Shared Variables

correspondence
involves CB_List_Fac::Total_Size;
Conc.Total_Size = CB_List_Fac::Total_Size;
end;

Type Stack = CB_List_Fac::List;

convention
S.Prec = Empty_String;
correspondence
Conc.S = S.Rem;
finalization
affects CB_List_Fac::Total_Size;
end;
end;

Procedure Push(alters E: Entry; updates S: Stack);
affects CB_List_Fac::Total_Size;

Insert(E, S);
end Push;

Procedure Pop(replaces R: Entry; updates S: Stack);
affects CB_List_Fac::Total_Size;

Remove(R, S);
end Pop;

Procedure Depth(restores S: Stack) : Integer;
Depth := Length_of_Rem(S);
end Depth;

Procedure Occupied_Size(): Integer;
Occupied_Size := CB_List_Fac::Occupied_Size();
end Occupied_Size;

Procedure Clear(clears S: Stack);
affects CB_List_Fac::Total_Size;

CB_List_Fac::Clear(S);
end Clear;
end CBLList_Based_Realiz;
ensures clauses (if necessary) to ensure that the conceptual Shared Variables’ changes satisfy its conceptual ensures clause. In this implementation category, our changes to any global state variables can be thought of as interference-free. Both interference-free and independent correspondence properties are important, because those properties makes it possible to take advantage of RESOLVE’s clean semantics property and thereby avoid the need to reason about the frame property.

Similar to the globally bounded List implementation of globally bounded Stack, the communally bounded Stack is implemented directly using CB_List_Fac::List. The type convention ensures that Prec string is always empty and the correspondence indicates that Stack’s contents are stored inside the Rem string. Notice that type finalization affects CB_List_Fac::Total_Size, but there is no need to provide any additional code to finalize a Stack. At the end of the finalization block, the implementing List will be finalized properly by List’s realization, therefore we can assume the finalization ensures clause from Communally_Bounded_List_Template. This combined with convention and correspondence clauses will generate the following VC. The proof rule for type finalization realization can be found in Appendix A.

Listing 7.14: VC for Establishing the Finalization Ensures Clause of Stack

Goal:

((CB_List_Fac::Total_Size - (|Empty_String| + |S.Rem|)) = 
(CB_List_Fac::Total_Size - |S.Rem|))

Given:

The specification for Depth and Occupied_Size state that it does not modify the conceptual Total_Size and this implementation also does not modify CB_List_Fac’s Total_Size. Since it is possible that we temporarily modified either of them, it is important to generate VCs to ensure that at the end of the Procedure, they have been properly restored. VC for establishing that the conceptual Total_Size hasn’t been modified by the code for Depth is shown below.

Listing 7.15: VC: Ensures Clause of Depth (Condition from Non-Affected Shared Variable)

Goal:

(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given:
Notice that since our Shared Variables’ correspondence states that Conc.Total_Size = CB_List_Fac::Total_Size, both the goal for this VC and the one for establishing CB_List_Fac’s Total_Size have been substituted with CB_List_Fac::Total_Size. All of the VCs generated for this realization are straightforward to prove and can be found in Appendix F.

7.3 Interference-Free Sharing Implementations with Independent Regional Correspondence

Sections 7.1 and 7.2 have used (reference-hiding) data abstractions to implement the variants of Stacks. In this section, we use the Communal_UVR_Template presented in Chapter 6 to implement Communal_Stack_Template shown in Listing 7.12.

7.3.1 Communally-Bounded UVRT Implementation

Listing 7.16 instantiates a facility of Communal_UVR_Template and uses it to implement the communal Stack. The Shared Variables’ correspondence indicates that the conceptual Total_Size involves Accessible_Loc and is equal to its cardinality².

Listing 7.16: A Communally-Bounded UVRT Realization

Realization UVRT_Stack_Realiz for Communally_Bounded_Stack_Template;
uses Set_Theory, Set_App_Op_Ext;

Facility UVRT_Fac is Communal_UVR_Template(Entry, Max_Capacity)
externally realized by Communal_Array_Realiz;

Shared Variables
correspondence
involves UVRT_Fac::Accessible_Loc;
Conc.Total_Size = ||UVRT_Fac::Accessible_Loc|| - 1;
end;

-- Refˆ{times}(start)
-- Note: Future syntax is expected to include a suitable notation and
-- this definition will be elided.
Definition Iterated_Apply(
  f : Location -> Location, start : Location, times : Z) : Location;

²Recall that Accessible_Loc includes Void, therefore we need to subtract 1 to obtain the proper total size.
Type Stack is represented by Record

  Top_Pos: UVRT_Fac::Pos;
  Depth: Integer;
end;

convention 0 <= S.Depth and Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void;

correspondence
  involves UVRT_Fac::Ref, UVRT_Fac::Content;
  Conc.S = Iterated_Concatenation(1, S.Depth,
    lambda(i: Z).(<UVRT_Fac::Content( Iterated_Apply( UVRT_Fac::Ref, S.Top_Pos, i-1))>));

finalization
  affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
    UVRT_Fac::Ref, UVRT_Fac::Content;
end;
end;

Procedure Push(alters E: Entry; updates S: Stack);
  affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
    UVRT_Fac::Ref, UVRT_Fac::Content;

  Var New_Pos: Pos;

  Give_New_Loc(New_Pos);
  Swap_Content_of(New_Pos, E);
  Redirect_Ref_at(New_Pos, S.Top_Pos);
  S.Top_Pos :=: New_Pos;
  S.Depth := S.Depth + 1;
end Push;

Procedure Pop(replaces R: Entry; updates S: Stack);
  affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
    UVRT_Fac::Ref, UVRT_Fac::Content;

  Swap_Content_of(S.Top_Pos, R);
  Follow_Ref(S.Top_Pos); -- Let UVRT take care of finalization
  S.Depth := S.Depth - 1;
end Pop;

Procedure Depth(restores S: Stack) : Integer;
  Depth := S.Depth;
end Depth;

Procedure Occupied_Size(): Integer;
  Occupied_Size := UVRT_Fac::Occupied_Size() - 1;
end Occupied_Size;
**Procedure** Clear(clears S: Stack);

    affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
    UVRT_Fac::Ref, UVRT_Fac::Content;

    Set_to Void(S.Top_Pos); -- Let UVRT take care of finalization
    S.Depth := 0;
end Clear;

end UVRT_Stack_Realiz;

Similar to the *communal* List realization, this implementation does not require additional code to initialize the shared state. Initializing UVRT_Fac allows us to assume the concept’s Shared Variables initialization ensures clause as well as the definition of Accessible_Loc. In addition, immediately after the Facility declaration, we can assume that Pos.Receptacles is the empty set. The resulting VC for establishing the Shared Variables initialization is shown below\(^3\). The Closure_for application in the goal simply returns a set containing Void, thus enabling us to establish that \(|{|\text{Void}||} - 1 = 0|\).

Listing 7.17: VC for Establishing the Initialization Ensures Clause of Shared Variables

**Goal:**

\[
(|{|\text{Void}||} - 1) = 0
\]

**Given:**

1. (SqBr(UVRT_Fac::Ref, Location) = {Void})
2. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
3. (UVRT_Fac::Cast_Accessible_Loc = (|{|\text{Void}||} union Closure_for(Location, 
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Pos.Receptacles = { })

A Stack is represented using a record containing Top_Pos that references the top element’s position and a Depth variable that keeps track of the Stack’s current depth. This realization provides a local definition Iterated_Apply\(^4\) function that allows us to repeatedly apply the Ref function a specified number of times. This definition is used in the the type convention to state that applying the Ref function S.Depth times to S.Top_Pos will return Void.

The type correspondence involves UVRT’s Shared Variables: Ref and Content and it indicates that it is the iterated concatenation of all the content in the Locations obtained from following the references starting at Top_Pos. It is important to note here that this type correspondence

\(^3\)UVRT_Fac::Cast_Accessible_Loc is equivalent to UVRT_Fac::Accessible_Loc, but with a mathematical type that allow us to type check this Concept.

\(^4\)Future syntax is expected to include a suitable notation and this definition will be elided.
is different that the ones presented earlier in this chapter. Since it involves global state variables, it no longer fits our definition of independent correspondence. It is possible that changes to either UVRT_Fac::Ref or UVRT_Fac::Content modify other Stacks. However, since no two Stacks share any Location other than Void, it is possible to partition the Accessible_Loc into independent regions. This idea is critical to the simplified proof of this implementation. We further discuss the frame property in Section 7.3.2.

We briefly discuss the implementation of each Operation here. Push creates a new Pos to store the incoming value of E and sets this new Pos as the new Top_Pos for S. The Pop operation swaps out the value stored in S.Top_Pos and places it in R. It then calls Follow_Ref to set S.Top_Pos accordingly. Note that finalization of the original top position is handled by Follow_Ref. Clear sets the Top_Pos to Void. This finalizes all positions that were accessible by following the incoming S’s Top_Pos. These three Operations modify the global shared state as indicated by the affects clauses.

Depth and Occupied_Size do not modify any Shared Variables, therefore there will be VCs generated to ensure that the Shared Variables remain the same. All the VCs generated for this realization can be found in Appendix F.

However, at the moment not all VCs have sufficient information to be provable. Some givens are missing in the VC generation process. In order to better explore why this is the case, lets look at the VC for establishing the finalization ensures clause of Stack shown below.

Listing 7.18: VC for Establishing the Finalization Ensures Clause of Stack

Goal:

\[(||\text{UVRT Fac::Accessible Loc'}|| - 1) = (||\text{UVRT Fac::Accessible Loc}|| - 1) - |
\text{Iterated Concatenation}(1, \text{S.Depth}, \lambda (i : Z).<\text{UVRT Fac::Content}(\text{Iterated Apply(\text{UVRT Fac::Ref, S.Top_Pos,}\n([\text{Universal} i - 1]))})>)\])\]

Given:

1. \((0 \leq \text{S.Depth})\)
2. \((\text{Iterated Apply(\text{UVRT Fac::Ref, S.Top_Pos, S.Depth}) = UVRT Fac::Void})\)

When considering the effects of finalization, the VC generator needs to account for the finalization of all the information being held by the finalized variable’s receptacle. While no other Stack positions are aliased to point to the position that is being finalized, it is possible that the realization has cached the top position in an (unused) global variable. In this case, the locations in
the Stack cannot be reclaimed because it is still accessible. So the VC needs to include information on other position variables (including the case that there are no other holdings). In general, it may also be necessary to reveal such holdings through the mechanisms discussed in [100] for performance profiles. The VC generator should be able to use the information provided to add additional givens that will help us prove this VC. The details of what needs to be added to the list of givens is a future direction that must be pursued.

So far, we have been able to leverage RESOLVE's clean semantics property to ensure that the frame property holds. However, when the type correspondence involves Shared Variables, there is a need to establish that the frame property holds explicitly. This is complex because we have introduced the possibility of aliased references.

Proof of the frame property for the present implementation is straightforward, because it is possible to syntactically establish that there are no calls to the Relocate_to operation. Recall that the operation Relocate_to allows us to relocate a Pos to another Pos, thus creating them aliased references. However, this realization has been carefully implemented to avoid the need to use this operation. If we can syntactically observe that this operation from CUVRT is never called (meaning it is only using a reduced functionality of CUVRT), then it is no longer possible to create aliases and we can leverage RESOLVE's clean semantics property to ensure that the frame property holds. Although this works for this particular class of non-aliasing implementations, the ability to create aliased references is required to perform certain actions (such as iterating over a chain of Pos).

7.3.2 Proving the Frame Property in the Presence of Aliasing

As future work, our VCs will have to show that modifying a Stack does not inadvertently modify other Stacks. This subsection addresses the situation where implementations create aliasing. We present only a solution approach here that needs to be implemented in the future. This solution approach demands much programmer effort and is not desirable. A more general solution is the topic of the next subsection.

Suppose that the compiler automatically generated a frame exemplar variable _Other_S to stand for some other conceptual Stack that is not one of the Stacks passed as parameters5. We will need to establish the frame property of Conc._Other_S = #Conc._Other_S at the end of each operation, including initialization and finalization.

5Note that this is a proposed solution is not supported by the current iteration of the compiler.
In order to assist the automated proof system in establishing this new assertion, for each 
**Procedure** a programmer can specify a local **ensures** clause that helps prove this property. For example, Push’s local **ensures** clause might state something like the following listing.

**Listing 7.19: Push with Local Ensures Clause**

```
Procedure Push(alters E: Entry; updates S: Stack);
  affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_ACCESSIBLE_LOC,
          UVRT_Fac::Ref, UVRT_Fac::Content;
  ensures Fn_Restricted_to(UVRT_Fac::Ref, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) =
                        Fn_Restricted_to(#UVRT_Fac::Ref, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) and
                        Fn_Restricted_to(UVRT_Fac::Content, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) =
                        Fn_Restricted_to(#UVRT_Fac::Content, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos});
  ...
end Push;
```

There are two parts to this local **ensures** clause. First, it is stating that the range values of UVRT_Fac::Ref and #UVRT_Fac::Ref are the same for the domain of locations other than those in the closure of locations reachable from recp.S.Top_Pos. Second, the range values of UVRT_Fac::Content and #UVRT_Fac::Content contain the same values for the domain of locations other than those in the closure of locations reachable from recp.S.Top_Pos. In other words, nothing else other than S changes.

Note that this local **ensures** clause will need to be established to be **true**, before it can be used to prove the frame property. The generated **assertive program** for Push might look like the following:

**Listing 7.20: Potential Assertive Program for Push**

```
Assume <Pre_Push> and Conc.S /= Conc._Other_S;
...
Presume Fn_Restricted_to(UVRT_Fac::Ref, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) =
                        Fn_Restricted_to(#UVRT_Fac::Ref, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) and
                        Fn_Restricted_to(UVRT_Fac::Content, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos}) =
                        Fn_Restricted_to(#UVRT_Fac::Content, Location ∼
                        Closure_for(Location, (#UVRT_Fac::Ref), {recp.S.Top_Pos});
...
Confirm <Post_Push> and Conc._Other_S = #Conc._Other_S;
```
Recall that **Presume** will require us to first **Confirm** the assertion is *true*, before we can **Assume** it. This will need be filled in for all **Operations** and **initialization/finalization** blocks that affect any global state variables. Although this seems like a potential solution to show that the frame property holds, the local **ensures** clause will need to be duplicated multiple times, which makes this solution less than ideal.

### 7.3.3 Future Work: Co-confinement Invariants

For concepts such as **UVRT** that allow aliasing of **Pos** variables by co-locating, there is a need to introduce invariants to *confine* simultaneously parameters and global state space. In the case of **UVRT**, every operation can be confined to modify locations and contents in the global state space that are reachable from its parameters; nothing else can be modified. The implementations of operations may, however, allocate new locations and change their contents (since unallocated new locations can never be a part of other objects). This new type of co-confinement invariant must be proven for each operation in the concept, but becomes a useful lemma for realizations that uses an instantiated facility of this concept. So a facility instantiation of **UVRT** to implement a stack or a list, for example, may co-locate positions, yet with a co-confinement construct, no additional proof may be necessary to show non-parameters are not affected.
Chapter 8

Conclusions and Future Research

This dissertation aims to demonstrate that it is possible to develop specification and verification machinery towards automated verification of object-based components with (and without) reference behavior, without using any specialized logic. The first goal is to avoid reasoning about references explicitly and routinely through clean language mechanisms and reference-hiding data abstractions. This research has introduced shared concepts and realization constructs consistent with RESOLVE’s clean semantics principles to make it possible to build and use data abstractions that avoid the need to reason about references explicitly. Furthermore, when references are unavoidable, this research presents a concept amenable to automation for capturing acyclic reference behavior that is useful to implement and reason about lower-level components. The dissertation contains experimentation with a variety of component implementations. For some implementations, the frame property follows automatically from the clean semantics of the language and for some others, explicit proofs are necessary.

In order to support both components with (and without) reference behavior, this research has developed a proof system and a new prototype verification system for handling shared state. To simplify the resulting VCs, the new implementation is built using a sequent-based system with logical reduction rules and is designed to generate parsimonious VCs, where only relevant givens are retained. Taken together, the VCs generated are a prelude to enable proving of non-trivial assertions automatically.

Several directions for further research remain. There is much work left to improve the verification system, including generation of information necessary to prove a class of VCs that are
currently unprovable. Underlying mathematical type system needs to be revamped in order to provide the capabilities to type and type check new mathematical specifications. In addition, the VC generator can employ other simplification mechanisms to further reduce the givens and goals. Lastly, work needs to be done to the underlying automated prover in order to process the new sequent-based VCs.

While this research focuses on list-like data structures, there is a need to introduce and reason about other acyclic data structures such as trees. This future research can build on the work in this dissertation and the work of [72] that presents a new mathematical theory to specify tree behavior and a Tree concept. Extensions of this new concept should provide users the capabilities of traditional algorithms involving Trees. An n-link UVRT, such as the one described in Section 6.5.2, will have to be specified in order to implement this new Tree concept.

Clearly, it is possible to develop more complex sharing implementations for performance reasons. For example, consider a cactus-like Stack realization shown in the figure below, where the representation makes it possible to copy a stack in (amortized cost) constant time [96]. Although this implementation is also a linked implementation, each node in the Stack also includes a reference count field, which keeps track of the number of references pointing to that node.

![Figure 8.1: A cactus-like Stack Realization (Reproduced from [96])](image)

When copying a Stack, the implementation only has to copy a reference to the top of the Stack and increment the reference count fields accordingly. When a given Stack is modified, there needs to be a clean way to specify and reason about changes to this (and other) Stacks and ensure that the frame property holds for all other non-affected Stacks.

A natural extension to this research is to provide implementations for CUVRT. One possible implementation is to use the Linked_Location_Template presented in Section 3.3 in conjunction with extensions to handle finalization to avoid any dangling references. This implementation will
have to ensure that all allocated Locations reach Void. Another possible implementation is to use a communal array to hold all accessible locations. This implementation will need to have a chain of unused locations for quickly finding a location in the array and finding the next available spot in the communal array to create and insert a new Pos as part of implementing the Give_New_Loc operation.

Lastly, but most importantly, there is a need to figure out how to educate teachers, students and other researchers to reason about components with (and without) reference behavior. The RESOLVE WebIDE [27] has been extensively used by software engineering students at various different institutions to reason about components [28][33]. Most recently, an alternative reasoning tool has been designed to pinpoint student obstacles in learning how to reason about code [29]. The tool should also teach advanced users how to write specifications where sharing is involved and how to ensure that the frame property needs to be proven explicitly. Even though a majority of users will simply use a linked-based structure such as Lists and Stacks, it is important to be able to build new components that address various different functional requirements and performance constraints of large scale systems.
Appendices
Appendix A  Proof Rules

This appendix contains the formal proof system. In the following rules,

- **C** is the context.

- **Seq** is a sequence calculus predicate of the form: \( \{A_1, \ldots, A_i\} \implies \{C_1, \ldots, C_j\} \), where the commas between antecedents indicate that they are joined by \( \land \) and the commas between consequents indicate they are joined by \( \lor \). Note that the predicate could contain empty antecedents or consequents. Empty antecedents means there are no givens. Empty consequents means false is the goal.

- **Result Sequent** or **RS** are a collection of **Seq** joined together by the \( \land \) operator.

- **code** refers to a block of either executable statements or assertive code.

- **decl** refers to a declaration node in the AST.

- **decls** are a collection of **decl**.

- \( \langle \text{decls}, \text{code} \rangle \) refers to a block of **decl** followed by a block of **code**.

- **Invk_Cond(exp)** conjoins all pre-conditions for all the programming functions in **exp**.

- **Sequent_Form(RS, exp)** adds **exp** to each sequent in **RS** and applies the reduction rules to obtain the reduced sequent form.

- **Math(exp)** composes the mathematical expressions for all the programming functions in **exp**.

- **BE** is a Boolean valued programming expression.

- **P_Exp** is the ordinal valued progress metric expression, the system variables \( ?^k P \_\text{Val} \) hold progress metric values.

- \( NQV(\text{RS}, x) \) produces the next new variable name of the form \( x^m \)' such that \( m \) is the smallest value for which \( x^m \) doesn’t occur in **RS**.

- **CExp \preceq t \preceq** is the constraint expression for **T**.

- \( \sim \) indicates replacement.
Declaration Rules

Each of the following rules is applied in the order of where they appear in the abstract syntax tree (AST) generated by the RESOLVE compiler. Unless specified otherwise, the same declaration rule applies to all the different RESOLVE modules. The general declaration rule template is given below:

\[
\begin{align*}
&\text{C} \setminus \text{Correct\_Decl\_Hyp} (\text{decl}); \\
&\text{C} \cup \{\text{decl}\} \setminus \text{Prec\_Code}; <\text{decls}, \text{code}>; \textbf{Confirm RS} ; \\
&\text{C} \setminus \text{decl}; <\text{decls}, \text{code}>; \textbf{Confirm RS}; \\
\end{align*}
\]

where \text{Prec\_Code} could contain generated assertive codes that precede the \text{<decls, code>} block.

Note. \text{Correct\_Decl\_Hyp} is an operator that establishes that \text{decl}'s inner declarations and/or code have been declared correctly.

Suppose a Concept template is specified by:

\[
\begin{align*}
\text{CT} &= \text{Shared Concept CN( type T; evaluates n: U; Definition R: T} \times \text{V} \rightarrow \text{B );} \\
& \quad \text{uses AFac, BTh;} \\
& \quad \text{requires CPC} \not\subset n, R \ \text{which entails CPCExp}; \\
& \quad \text{Definition S: W} \rightarrow \text{B} = (DExp ); \\
& \quad \text{Defines f: W} \rightarrow \text{T}; \\
& \quad \text{Constraint DC} \not\subset f_1, n, R \ \text{which entails DCExp}; \\
& \quad \text{Shared Variables} \\
& \quad \text{Abstract\_Var gv1: MX1;} \\
& \quad \text{Abstract\_Var gv2: MX2;} \\
& \quad \text{constraint VC} \not\subset gv1, gv2, f, n, R, S \ \text{which entails VCExp}; \\
& \quad \text{initialization} \\
& \quad \quad \text{ensures GIC} \not\subset gv1, gv2, f, n, R, S \not\subset; \\
& \quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{Type Family TF is modeled by MTE;} \\
& \quad \text{exemplar x;} \\
& \quad \text{constraint TC} \not\subset x, gv1, gv2, f, n, R, S \ \text{which entails TCExp}; \\
& \quad \text{initialization} \\
& \quad \quad \text{affects gv1, gv2;} \\
& \quad \quad \text{ensures IC} \not\subset x, gv1, #gv1, gv2, #gv2, f, n, R, S \not\subset; \\
& \quad \quad \text{finalization} \\
& \quad \quad \text{affects gv1, gv2;} \\
& \quad \quad \text{ensures FC} \not\subset #x, gv1, #gv1, gv2, #gv2, f, n, R, S \not\subset; \\
& \quad \text{end TF;}
\end{align*}
\]
Operation \( P(\text{updates } x: \text{TF}; \text{restores } y: \text{T1} ); \)
\hspace{1em} a\text{ffects } \text{gv1};
\hspace{1em} \text{requires } \text{Pre} \not\text{\Delta} x, y, \text{gv1}, \#\text{gv1}, \text{gv2}, \text{f}, n, R, S \not\text{\Delta} \text{ which_entails } \text{PrePExp};
\hspace{1em} \text{ensures } \text{Post} \not\text{\Delta} x, \#x, y, \text{gv1}, \#\text{gv1}, \text{gv2}, \text{f}, n, R, S \not\text{\Delta} ;

Operation \( F(\text{restores } x: \text{TF}; \text{preserves } y: \text{T1}; \text{evaluates } z: \text{T2} ) : \text{T3}; \)
\hspace{1em} \text{requires } \text{Pre} \not\text{\Delta} x, y, z, \text{gv1}, \#\text{gv1}, \text{gv2}, \text{f}, n, R, S \not\text{\Delta} \text{ which_entails } \text{PreFExp};
\hspace{1em} \text{ensures } F = \text{Post} \not\text{\Delta} x, y, z, \text{gv1}, \#\text{gv1}, \text{gv2}, \text{f}, n, R, S \not\text{\Delta} ;
end \text{CN};

\text{Note.} \text{ A Shared Concept is a Concept that has Shared Variables. uses lists cannot have facilities of Shared Concepts.}

A.1 Concept Declaration Rule

\text{C\backslash Correct\_Concept\_Hyp(CT);}\hspace{1em} C \cup \{CT\}\textless \textless <\text{decls, code}>; \text{Confirm RS};
\text{C\backslash CT; <\text{decls, code}>; \text{Confirm RS};}

\text{Note.} \text{Correct\_Concept\_Hyp is an operator that establishes that all declarations in CT are correct. It also takes care of any which_entails clauses for CN's requires or constraint clauses.}

A.2 Which_Entails Declaration Rule

\text{There are two forms for which_entails declaration rule: one for type assertions and one for mathematical expressions. An Assertion Clause could be either a requires, ensures, constraint, convention, correspondence, decreasing or maintaining clause.}

A.2.1 Type Assertions:

\text{C\backslash Assume expl1; \text{Confirm \text{Sequent\_Form(RS}, x \in T);}\
\text{C\backslash Assertion\_Clause expl1 which_entails x : T; \text{Confirm RS};}

A.2.2 Mathematical Expressions:

\text{C\backslash Assume expl1; \text{Confirm \text{Sequent\_Form(RS}, exp2);}\
\text{C\backslash Assertion\_Clause expl1 which_entails exp2; \text{Confirm RS};}
A.3 Definition Declaration Rule

\[ C \cup \{ \text{Definition } S \ldots \} \setminus \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]
\[ C \setminus \text{Definition } S : W \rightarrow B = \{ \text{DExp} \} ; \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]

A.4 Shared Variables Declaration Rule

\[ C \setminus \text{Entailment}_\text{Hyp}(\text{SVT}); \]
\[ C \cup \{ \text{SVT} \} \setminus \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]
\[ C \setminus \text{SVT}; \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]

where SVT is:

\begin{verbatim}
Shared Variables
  Abstract_Var gvl: MX1;
  ...
end;
\end{verbatim}

Note. Entailment_Hyp is an operator that establishes any which_entails clauses inside a declaration.

A.5 Type Family Declaration Rule

\[ C \setminus \text{Entailment}_\text{Hyp}(\text{TT}); \]
\[ C \cup \{ \text{TT} \} \setminus \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]
\[ C \setminus \text{TT}; \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]

where TT is:

\begin{verbatim}
Type Family TF is modeled by MTE;
  ...
end TF;
\end{verbatim}

A.6 Operation Declaration Rule

\[ C \setminus \text{Entailment}_\text{Hyp}(\text{PT}); \]
\[ C \cup \{ \text{PT} \} \setminus \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]
\[ C \setminus \text{PT}; \langle \text{decls, code} \rangle ; \text{Confirm } RS; \]

where PT is:

\begin{verbatim}
Operation P( updates x: TF; restores y: T1 );
  affects gvl;
  ...
\end{verbatim}
A.7 Function Declaration Rule

\[ C \setminus \text{Entailment_Hyp}(FT); \]
\[ C \cup \{ FT \} \setminus \text{<decls, code>}; \text{Confirm} \ RS; \]

\[ C \setminus \text{Operation} \ F \ldots; \text{<decls, code>}; \text{Confirm} \ RS; \]

where \( FT \) is:

\[ \text{Operation} \ F( \text{restores} \ x: \ TF; \text{preserves} \ y: \ T1; \text{evaluates} \ z: \ T2 ) : \ T3; \]

\ldots

Suppose a Concept implementation is specified by:

\[ \text{CRT} = \text{Realization} \ CRN( \text{evaluates} \ rn: \ RU); \text{Definition} \ RR: \ RT \times RV \rightarrow B; \]
\[ \quad \text{Realization} \ F_{\text{Realiz}}( \text{evaluates} \ e: \ FRU ); \]
\[ \quad \text{Operation} \ RP( \text{updates} \ rx: \ RT2 ); \]
\[ \quad \text{requires} \ preRP\angle rx, \ rn \ \backslash; \]
\[ \quad \text{ensures} \ postRP\angle rx, \ #rx, \ rn \ \backslash; \text{ for} \ CN; \]
\[ \text{uses} \ RA_{\text{Fac}}, \ RB_{\text{Th}}, \ G_{\text{Ty}}, \ R_{\text{C}}; \]
\[ \text{requires} \ RPC\angle \ rn, \ RR \ \backslash; \]

\[ \text{Definition} \ RS: \ RW \rightarrow B = ( RD_{\text{Exp}} ); \]
\[ \text{Definition} \ f: \ W \rightarrow T = ( F_{\text{Exp}} ); \]

\[ \text{Constraint} \ RDC\angle \ rn, \ RR \ \backslash; \]

\[ \text{Facility} \ FDN \ is \ R_{\text{C}}( f_{\text{exp}}\angle rx, \ rn \ \backslash, \ G_{\text{Ty}}, \ RR, \ RS ) \]
\[ \text{realized by} \ F_{\text{Realiz}}( f_{\text{exp}}\angle rx, \ rn \ \backslash, \ RR, \ RS ); \]

\[ \text{Shared Variables} \]
\[ \quad \text{Var} \ rgv1: \ RX1; \]
\[ \quad \text{Var} \ rgv2: \ RX2; \]

\[ \quad \text{convention} \ SS_{\text{RC}}\angle rgv1, \ rgv2, \ f, \ rn, \ RR, \ RS \ \backslash; \]
\[ \quad \text{correspondence} \ SS_{\text{Cor}_{\text{Exp}}}\angle gv1, \ gv2, \ rgv1, \ rgv2, \ rn, \ RR, \ RS \ \backslash; \]
\[ \quad \text{initialization} \ \langle SS_{\text{I}_{\text{decls}}}, \ SS_{\text{I}_{\text{code}}} \rangle; \text{end}; \]
\[ \text{end}; \]

\[ \text{Type} \ TF = RT; \]
\[ \quad \text{convention} \ RC\angle x, \ rgv1, \ rgv2, \ f, \ rn, \ RR, \ RS \ \backslash; \]
\[ \quad \text{correspondence} \ Cor_{\text{Exp}}\angle x, \ gv1, \ gv2, \ rgv1, \ rgv2, \ f, \ n, \ rn, \ R, \ RR, \ S, \ RS \ \backslash; \]
\[ \quad \text{initialization} \ \langle Type_{\text{I}_{\text{decls}}}, Type_{\text{I}_{\text{code}}} \rangle; \text{end}; \]
\[ \text{finalization} \ \langle Type_{\text{F}_{\text{decls}}}, Type_{\text{F}_{\text{code}}} \rangle; \text{end}; \]
\[ \text{end} \ TF; \]

\[ \text{Procedure} \ P( \text{updates} \ x: \ TF; \text{evaluates} \ y: \ T1 ); \]
\[ \quad \langle P_{\text{decls}}, P_{\text{code}} \rangle; \]
\[ \text{end} \ P; \]

\[ \text{Procedure} \ F( \text{restores} \ x: \ TF; \text{preserves} \ y: \ T1; \text{evaluates} \ z: \ T2 ) : \ T3; \]
\[ \quad \langle F_{\text{decls}}, F_{\text{code}} \rangle; \]
\[ \text{end} \ F; \]
Operation Local_Op( updates x: TF )
affects gv1;
requires Pre∠ x, gv1, gv2, f, n, rn, R, RR \);
ensures Post∠ x, gv1, #gv1, #gv2, f, n, rn, R, RR \);
Procedure
   <LocalOp_decls, LocalOp_code>;
end Local_Op;

end CRN;

A.8 Concept Realization Declaration Rule

C U {CT}\ Correct_Concept_Realiz_Hyp(CRT);
C U {CT} U {CRT}\ <decls, code>; Confirm RS;

C U {CT}\ CRT; <decls, code>; Confirm RS;

Note. Correct_Concept_Realiz_Hyp is an operator that establishes that all declarations and code in CRT are correct. It also takes care of any which_entails clauses for CRN's requires or constraint clauses. The rule for dealing with facility instantiations is given in Section A.20

A.9 Shared Variables Realization Declaration Rule

C U {SVT}\ Correct_SS_Realiz_Hyp(SVRT);
C U {SVT} U {SVRT}\ <decls, code>; Confirm RS;

C U {SVT}\ SVRT; <decls, code>; Confirm RS;

where SVRT is:
    Shared Variables
       Var rgv1: RX1;
...
end;

in CRT.

and where SVT is the Shared Variables declaration in CT.

Note. Correct_SS_Realiz_Hyp is an operator that establishes the shared state realization SS has a well defined correspondence (Well_Def_SS_Corr_Hyp) and initialization block (SS_Init_Hyp). Currently, we do not deal with any finalization blocks. Finalization will affect some external shared variable g that is shared among all objects instantiated.
A.9.1 Well Defined Correspondence Rule

Well_Def_SS_Corr_Hyp is

Assume CPC \land RPC \land DC \land RDC \land SS_RC;
Assume SS_Cor_Exp;
Confirm VC;

Note. Well_Def_SS_Corr_Hyp establishes that the shared variables’ correspondence is well defined.

A.9.2 Shared Variable Initialization Rule

SS_Init_Hyp is

Assume CPC \land RPC \land DC \land RDC;
RX1.Var_Init_Exp(rgv1);
RX2.Var_Init_Exp(rgv2);
\preceq SS_I_decls, SS_I_code
\succeq;
Confirm SS_RC;
Assume SS_Cor_Exp;
Confirm GIC;

Note. SS_Init_Hyp establishes that the shared variables’ convention and initialization clauses are satisfied.

A.10 Variable Declaration Rule for Generic Types

\begin{align*}
\text{C} \cup \{x : T\} & \text{ \Assume T.Is_Initial}(x); \langle \text{decls, code}\rangle; \text{Confirm RS}; \\
\text{C} \setminus \text{Var} x : T; \langle \text{decls, code}\rangle; \text{Confirm RS};
\end{align*}

A.11 Variable Declaration and Finalization Rule for Known Types

\begin{align*}
\text{C} \cup \{\text{SVRT}\} \cup \{y : T\} & \text{ \Assume T.Var_Init_Exp}(y); \\
& \langle \text{decls, code}\rangle; \\
& \Assume T.Var_Final_Exp(y); \\
& \text{Confirm RS} \angle \text{gv} \rangle;
\end{align*}

\begin{align*}
\text{C} \cup \{\text{SVRT}\} & \text{ \Var y : T; } \langle \text{decls, code}\rangle; \text{Confirm RS} \angle \text{gv} \rangle;
\end{align*}

where ST is:

\begin{verbatim}
Shared Variables
    Abstract_Var gv: MX;
    ...
end;
\end{verbatim}
Note. Var_Init_Exp applies the appropriate substitutions to T’s initialization ensures clause using its exemplar. It is possible that Var_Init_Exp might produce a replaceable expression, however this rule will let the Assume rule take care of it. Similarly, Var_Final_Exp applies the appropriate substitutions to T’s finalization ensures clause and might produce a replaceable expression.

A.12 Type Representation Declaration Rule

\[
\begin{align*}
\text{C} \cup \{\text{TT}\} \setminus \text{Correct_Type_Realiz_Hyp}(\text{TRT}); \\
\text{C} \cup \{\text{TT}\} \cup \{\text{TRT}\} \setminus \{\text{decls, code}\}; \text{Confirm RS}; \\
\text{C} \cup \{\text{TT}\} \setminus \text{TRT}; \{\text{decls, code}\}; \text{Confirm RS};
\end{align*}
\]

where TRT is:

\[
\begin{align*}
\text{Type} \ \text{TF} & = \text{RT}; \\
\ldots \\
\text{end} \ \text{TF};
\end{align*}
\]

in CRT.

and where TT is the Type Family declaration in CT.

Note. Correct_Type_Realiz_Hyp is an operator that establishes the type representation TF has a well defined correspondence (Well_Def_Corr_Hyp), initialization block (T_Init_Hyp) and finalization block (T_Final_Hyp).

A.12.1 Well Defined Correspondence Rule

Well_Def_Corr_Hyp is

\[
\begin{align*}
\text{Assume CPC} \land \text{RPC} \land \text{DC} \land \text{RDC} \land \text{SS_RC} \land \text{RC}; \\
\text{Assume SS_Corr_Exp} \land \text{Cor_Exp}; \\
\text{Confirm TC};
\end{align*}
\]

Note. Well_Def_Corr_Hyp establishes that the type representation’s correspondence is well defined.

A.12.2 Type Initialization Rule

T_Init_Hyp is

\[
\begin{align*}
\text{Assume CPC} \land \text{RPC} \land \text{DC} \land \text{RDC} \land \text{SS_RC} \land \text{SS_Corr_Exp}; \\
\text{Remember}; \\
\text{Assume RT.Var_Init_Exp(x);} \\
\{\text{Type_I_decls, Type_I_code}\}; \\
\text{Confirm SS_RC} \land \text{RC}; \\
\text{Assume SS_Corr_Exp} \land \text{Cor_Exp}; \\
\text{Confirm IC};
\end{align*}
\]
Note. \( T_{\text{Init-Hyp}} \) establishes that the shared variables’ convention, type realization’s convention, and type realization’s initialization clauses are satisfied.

### A.12.3 Type Finalization Rule

\[ T_{\text{Final-Hyp}} \]

\[
\begin{align*}
& \text{Assume} \quad \text{CPC} \land \text{RPC} \land \text{DC} \land \text{RDC} \land \text{SS\_RC} \land \text{RC} \land \text{SS\_Corr\_Exp}; \\
& \text{Remember}; \\
& <\text{Type\_F\_decls}, \text{Type\_F\_code}>; \\
& \text{Assume} \quad \text{RT\_Var\_Final\_Exp}(x); \\
& \text{Confirm} \quad \text{SS\_RC}; \\
& \text{Assume} \quad \text{SS\_Corr\_Exp}; \\
& \text{Confirm} \quad \text{FC};
\end{align*}
\]

Note. \( T_{\text{Final-Hyp}} \) establishes that the shared variables’ convention and type realization’s finalization clauses are satisfied.

### A.13 Concept Procedure Declaration Rule

\[
\begin{align*}
& C \cup \{\text{COP}\} \setminus \text{Correct\_Op\_Hyp}(\text{Procedure} \ P \ldots ); \\
& C \cup \{\text{COP}\} \setminus <\text{decls}, \text{code}>; \quad \text{Confirm} \quad \text{RS}; \\
& C \cup \{\text{COP}\} \setminus \text{Procedure} \ P(...); \quad <\text{P\_decls}, \text{P\_code}>; \quad \text{end} \ P; \\
& \quad <\text{decls}, \text{code}>; \quad \text{Confirm} \quad \text{RS};
\end{align*}
\]

where \( \text{Correct\_Op\_Hyp} \) is:

\[
\begin{align*}
& \text{Assume} \quad \text{CPC} \land \text{DC} \land \text{VC} \land \text{RPC} \land \text{RDC} \land \text{SS\_RC} \land \text{TC} \land \text{RC} \land \text{SS\_Corr\_Exp} \land \text{Corr\_Exp} \land \text{Pre} \land \text{PrePExp} \land \text{T1.Constraint}(y); \\
& \text{Remember}; \\
& <\text{P\_decls}, \text{P\_code}>; \\
& \text{Confirm} \quad \text{SS\_RC} \land \text{RC}; \\
& \text{Assume} \quad \text{SS\_Corr\_Exp} \land \text{Cor\_Exp}; \\
& \text{Confirm} \quad \text{Post} \land \text{gv2} = \#\text{gv2};
\end{align*}
\]

and where COP is the Operation \( P \) declaration in CT.

Note. \( \text{Correct\_Op\_Hyp} \) operator applies the appropriate formal-to-actual substitutions to the different assertion expressions. \( \text{T1.Constraint}(y) \) returns the constraint for \( y \) with the appropriate formal-to-actual substitution. In this case, our template only affects \( \text{gv1} \), therefore the compiler will generate an expression to ensure that \( \text{gv2} \) is restored.

However, the \( \text{Correct\_Op\_Hyp} \) defined above is specific to \( P \). In order to handle the different parameter modes, the general procedure declaration rule is defined below using the following Operation template:
\[ \text{COP} = \text{Operation} \ P(\text{updates} \ t: T1; \text{evaluates} \ u: T2; \text{replaces} \ v: T3; \text{restores} \ w: T4; \text{preserves} \ x: T5; \text{alters} \ y: T6; \text{clears} \ z: T7); \]
\[ \text{requires} \ \text{Pre} \land t, u, w, x, y, z \land; \]
\[ \text{ensures} \ \text{Post} \land t, \#t, u, v, w, x, \#y, \#z \land; \]

A.13.1 General Procedure Declaration Rule

\[
\begin{align*}
C \cup \{\text{COP}\} & \\lambda \text{Assume} \ \text{Pre} \land T1.\text{Constraint}(t) \land T2.\text{Constraint}(u) \land T3.\text{Is_Init}(v) \land T4.\text{Constraint}(w) \land T5.\text{Constraint}(x) \land T6.\text{Constraint}(y) \land T7.\text{Constraint}(z); \\
& \\lambda \text{Remember}; \\
& \quad \langle P_{\text{decls}}, P_{\text{code}} \rangle; \\
& \quad \text{Confirm} \ \text{Post} \land w = \#w \land T7.\text{Is_Initial}(z); \\
\end{align*}
\]
\[
\begin{align*}
C \cup \{\text{COP}\} & \\lambda \langle \text{decls}, \text{code} \rangle; \ \text{Confirm} \ \text{RS}; \\
\end{align*}
\]

\[
\begin{align*}
C \cup \{\text{COP}\} & \\lambda \text{Procedure} \ P(...); \ \langle P_{\text{decls}}, P_{\text{code}} \rangle; \ \langle \text{decls}, \text{code} \rangle; \ \text{Confirm} \ \text{RS}; \\
\end{align*}
\]

A.14 Concept Function Procedure Declaration Rule

\[
\begin{align*}
C \cup \{\text{CFP}\} & \\lambda \text{Correct_Funct_Hyp(} \ \text{Procedure} \ F \ \ldots \ )); \\
C \cup \{\text{CFP}\} & \\lambda \langle \text{decls}, \text{code} \rangle; \ \text{Confirm} \ \text{RS}; \\
\end{align*}
\]
\[
\begin{align*}
C \cup \{\text{CFP}\} & \\lambda \text{Procedure} \ F(...); \ \langle F_{\text{decls}}, F_{\text{code}} \rangle; \ \langle \text{decls}, \text{code} \rangle; \ \text{Confirm} \ \text{RS}; \\
\end{align*}
\]

where Correct_Funct_Hyp is:
\[
\begin{align*}
\text{Assume} & \ \text{CPC} \land DC \land VC \land RPC \land RDC \land SS_{\text{RC}} \land TC \land RC \land SS_{\text{Corr_Exp}} \land Corr_{\text{Exp}} \land \text{Pre} \land PreFExp \land T1.\text{Constraint}(y) \land T2.\text{Constraint}(z); \\
& \ \text{Remember}; \\
& \quad \langle F_{\text{decls}}, F_{\text{code}} \rangle; \\
& \quad \text{Confirm} \ \text{SS_{RC}} \land RC; \\
& \quad \text{Assume} \ SS_{\text{Corr_Exp}} \land Corr_{\text{Exp}}; \\
& \quad \text{Confirm} \ F = \text{Post} \land x = \#x \land gv1 = \#gv1 \land gv2 = \#gv2; \\
\end{align*}
\]

and where CFP is the Operation F declaration in CT.

\textit{Note.} Correct_Funct_Hyp operator applies the appropriate formal-to-actual substitutions to the different assertion expressions. Functions cannot modify global state variables (e.g. no side effects), therefore the compiler will generate expressions to ensure that \(gv1\) and \(gv2\) are restored.
A.15 Concept Local Operation Rule

\[ C \setminus \text{Correct\_Local\_Op\_Hyp}(\text{Local\_Op}); \]
\[ C \cup \{ \text{Operation Local\_Op ...}\} \setminus \langle \text{decls, code}; \text{Confirm RS}; \]

\[ C \setminus \text{Operation Local\_Op(...)}; \langle \text{LocalOp\_decls, LocalOp\_code}; \text{end Local\_Op}; \]
\[ \langle \text{decls, code}; \text{Confirm RS}; \]

where Correct\_Local\_Op\_Hyp is:

\[ \text{Assume CPC} \land DC \land VC \land RPC \land RDC \land SS\_RC \land TC \land RC \land \]
\[ SS\_Corr\_Exp \land Corr\_Exp \land T1.\text{Constraint}(y) \land Pre; \]
\[ \text{Remember}; \]
\[ \langle \text{LocalOp\_decls, LocalOp\_code}; \]
\[ \text{Confirm SS\_RC} \land RC; \]
\[ \text{Assume SS\_Corr\_Exp} \land Cor\_Exp; \]
\[ \text{Confirm Post} \land x = \#x \land gv2 = \#gv2; \]

Note. Correct\_Local\_Op\_Hyp operator applies the appropriate formal-to-actual substitutions to the different assertion expressions. \text{T1.Constraint}(y) returns the constraint for y with the appropriate formal-to-actual substitution. In this case, our template only affects gv1, therefore the compiler will generate an expression to ensure that gv2 is restored.

However, the Correct\_Local\_Op\_Hyp defined above is specific to Local\_Op. If a local operation has other parameter modes, a rule similar to the one in Section A.13.1 is applied. If a local operation is recursive, the recursive procedure declaration rule is defined below using the following template:

\[ \text{Operation Q( updates t: T1 );} \]
\[ \text{requires Pre} \triangleq t \land \]
\[ \text{ensures Post} \triangleq t, \#t \land \]
\[ \text{Recursive Procedure} \]
\[ \text{decreasing} \; \text{dec}\_exp; \]
\[ \langle \text{Q\_decls, Q\_code}; \]
\[ \text{end Q}; \]

A.15.1 Recursive Procedure Declaration Rule

\[ C \setminus \text{Assume Pre} \land T1.\text{Constraint}(t) \land P\_Val = \text{dec}\_exp; \]
\[ \text{Remember}; \]
\[ \langle \text{Q\_decls, Q\_code}; \]
\[ \text{Confirm Post}; \]
\[ C \cup \{ \text{Operation Q ...}\} \setminus \langle \text{decls, code}; \text{Confirm RS}; \]

\[ C \setminus \text{Operation Q(...)}; \langle \text{Q\_decls, Q\_code}; \text{end Q}; \langle \text{decls, code}; \text{Confirm RS}; \]
Suppose an Enhancement template is specified by:

\[ \text{ET} = \text{Enhancement EN for CN; requires EPC which entails EPCExp; } \]

\[ \text{Operation EP} \left( \text{updates} \ x: \text{TF } \right); \]
\[ \text{affects } \text{gv1; } \]
\[ \text{requires } \text{Pre} \angle x, \text{gv1}, \text{gv2}, f, n, R, S \angle \text{which entails } \text{PreEPExp; } \]
\[ \text{ensures } \text{Post} \angle x, \#x, \text{gv1}, \#\text{gv1}, \text{gv2}, f, n, R, S \angle; \]

end EN;

A.16 Enhancement Declaration Rule

\[ \text{C } \cup \{ \text{CT} \} \setminus \text{Correct Enhancement Hyp(ET); } \]
\[ \text{C } \cup \{ \text{CT} \} \cup \{ \text{ET} \} \setminus \{ \text{decls, code} \}; \text{ Confirm RS; } \]

\[ \text{C } \cup \{ \text{CT} \} \setminus \{ \text{ET} \}; \{ \text{decls, code} \}; \text{ Confirm RS; } \]

Note. Correct Enhancement Hyp is an operator that establishes that all declarations and code in ET are correct. It also takes care of any which entails clauses for EN's requires or constraint clauses.

A.17 Enhancement Operation Declaration Rule

\[ \text{C } \setminus \text{Entailment Hyp(EPT); } \]
\[ \text{C } \cup \{ \text{EPT} \} \setminus \{ \text{decls, code} \}; \text{ Confirm RS; } \]

\[ \text{C } \setminus \{ \text{EPT} \}; \{ \text{decls, code} \}; \text{ Confirm RS; } \]

where EPT is:

\[ \text{Operation EP} \left( \text{updates} \ x: \text{TF } \right); \]
\[ \text{affects } \text{gv1; } \]
\[ ... \]
Suppose an Enhancement implementation is specified by:

\[
\text{ERT} = \text{Realization ERN( evaluates ern: ERU;} \\
\text{Operation ERP ( updates erx: ERT1 );} \\
\text{requires preERP\ angle\ ern, ern \ angle;} \\
\text{ensures postERP\ angle\ erx, #erx, ern \ angle; ) for EN of CN;} \\
\text{requires ERPC which_entails ERPCExp;}
\]

\[
\text{Procedure EP( updates x: TF );} \\
\text{<EP_decls, EP_code>;} \\
\text{end EP;}
\]

\text{end ERT;}

\text{A.18 Enhancement Realization Declaration Rule}

\[
\text{C} \cup \{\text{CT, ET}\} \setminus \text{Correct\_Enhancement\_Realiz\_Hyp(ERT);} \\
\text{C} \cup \{\text{CT, ET}\} \cup \{\text{ERT}\} \setminus \langle\text{decls, code}\rangle; \text{Confirm RS;}
\]

\[
\text{C} \cup \{\text{CT, ET}\} \setminus \text{ERT;} \langle\text{decls, code}\rangle; \text{Confirm RS;}
\]

\text{Note.} \text{Correct\_Enhancement\_Realiz\_Hyp} \text{is an operator that establishes that all declarations and code in ERT are correct. It also takes care of any which\_entails clauses for ERN's requires or constraint clauses.}

\text{A.19 Enhancement Procedure Declaration Rule}

\[
\text{C} \cup \{\text{EOP}\} \setminus \text{Correct\_Op\_Hyp( Procedure EP ... );} \\
\text{C} \cup \{\text{EOP}\} \setminus \langle\text{decls, code}\rangle; \text{Confirm RS;}
\]

\[
\text{C} \cup \{\text{EOP}\} \setminus \text{Procedure P(...);} \langle\text{EP_decls, EP_code}\rangle; \text{end P;} \\
\text{<decls, code>;} \text{Confirm RS;}
\]

where \text{Correct\_Op\_Hyp} \text{is}: 
\text{Assume CPC \land DC \land VC \land TC \land Pre \land PreEPExp;} \\
\text{Remember;} \\
\langle\text{EP_decls, EP_code}\rangle; \\
\text{Confirm Post \land gv2 = #gv2;}

and where \text{EOP} \text{is the Operation EP declaration in ET.}

\text{Note.} \text{Correct\_Op\_Hyp} \text{operator applies the appropriate formal-to-actual substitutions to the different assertion expressions. In this case, our template only affects gv1, therefore the compiler will generate an expression to ensure that gv2 is restored. If EP has other parameter modes, a rule similar to the one in Listing A.13.1 is applied.}
The following rule will use the Facility instantiation template given below.

\[ F_{Instn} = \text{Facility } FN \text{ is } CN( IT, n_{exp}, IR \) realized by RN( rn_{exp}, IRR, ICR, IRP ) enhanced by EN realized by ERN( ern_{exp}, IERP ) \]

where IRP is an operation with specifications:

- **Operation IRP( updates irx: RT2 );**
  - requires preIRP\( \downarrow \) rn_{exp}, irx \( \downarrow \);
  - ensures postIRP\( \downarrow \) rn_{exp}, #irx, irx \( \downarrow \);

- **Operation IERP( updates ierx: RT3 );**
  - requires preIERP\( \downarrow \) ern_{exp}, ierx \( \downarrow \);
  - ensures postIERP\( \downarrow \) ern_{exp}, #ierx, ierx \( \downarrow \);

### A.20 Facility Instantiation Rule

\[ C \cup \{CT, CRT, ET, ERT\} \setminus \text{Fac\_Instantiation\_Hyp}; \]
\[ C \cup \{CT, CRT, ET, ERT\} \cup \{F_{I\_Spec}\} \setminus \text{Assume } I_{Exp}; \text{ code; Confirm } RP; \]
\[ C \cup \{CT, CRT, ET, ERT\} \setminus \text{F\_Instn}; \text{ code; Confirm } RP; \]

where

- **F\_I\_Spec** is Facility Instantiation Specification
- **I\_Exp** is GIC[ S\( \rightarrow \)DExp, f\( \rightarrow \)F\_Exp[ rn\( \rightarrow \)rn_{exp}, RR\( \rightarrow \)IRR ],
  n\( \rightarrow \)n_{exp}, R\( \rightarrow \)IR, T\( \rightarrow \)IT ];

- **Fac\_Instantiation\_Hyp** is
  \((\text{RPC}[ rn\rightarrow rn_{exp}, RR\rightarrow IRR ] \land \text{CPC})( n\rightarrow n_{exp}, R\rightarrow IR ) \land
  (\text{preRP}[ rn\rightarrow rn_{exp}, rx\rightarrow irx ] \implies \text{preIRP} ) \land
  (\text{postIRP} \implies \text{postRP}[ rn\rightarrow rn_{exp}, #rx\rightarrow #irx, rx\rightarrow irx ] ) \land
  (\text{preERP}[ ern\rightarrow ern_{exp}, exr\rightarrow ierx ] \implies \text{preIERP} ) \land
  (\text{postIERP} \implies \text{postERP}[ ern\rightarrow ern_{exp}, #erx\rightarrow #ierx, erx\rightarrow ierx ] ));

Note. CT, CRT, GIC, S, DExp, f, F\_Exp, rn, rn_{exp}, RR, n, n_{exp}, R, preRP, rx, postRP and T are defined in Concept template and it’s implementing template. ET, ERT, ern, ern_{exp}, preERP, erx, postERP are defined in Enhancement template and it’s implementing template.
Suppose a Facility template is specified by:

\[
F = \text{Facility FN} \\
\ldots \\
\text{end FN;}
\]

### A.21 Facility Declaration Rule

\[
\text{C\ \text{Correct\_Facility\_Hyp}(F);} \\
\text{C \cup \{F\}\ \langle\text{decls, code}\rangle; \text{Confirm RS};}
\]

\[
\text{C\ \text{F}; \langle\text{decls, code}\rangle; \text{Confirm RS};}
\]

*Note.* Any declarations inside F will use rules similar to those defined previously.

#### Statement Rules

Each of the following rules are applied in the reverse order of where they appear in the abstract syntax tree (AST) generated by the RESOLVE compiler. Unless it is otherwise specified, the same declaration rule applies to all the different RESOLVE modules. The general statements rule template is given below:

\[
\text{C\ \text{code; Correct\_Stmt\_Hyp(statement)}; \text{Confirm RS};}
\]

\[
\text{C\ \text{code; statement; Confirm RS};}
\]

*Note.* Correct\_Stmt\_Hyp is an operator that establishes that the statement is correct.

### A.22 Assume Rule

The assume rule has 2 different steps: *substitution* and *assume application*. Each of the steps are explained in the following subsections. If the Assume statement does not contain the conditions in which the step needs to be applied, then that step is skipped.

#### A.22.1 Substitution Step

\[
\text{C\ \text{code; Assume exp3[ y\leadsto exp1 ]; Confirm ( RS[ y\leadsto exp1 ] )[ z\leadsto exp2 ]);}
\]

\[
\text{C\ \text{code;}
\quad \text{Assume (y = exp1\leadsto x \&\& (z = exp2\leadsto x \&\& exp3\leadsto y \&\& Confirm RS\leadsto y, z \&\& } }
\]

116
Note. The substitution step can only be used on replaceable expressions. A replaceable expression is an equality expression of the form $a = \text{exp}$ (or $\text{exp} = a$), where $a$ is a mathematical variable and $\text{exp}$ is a mathematical expression. In this rule, $x$, $y$ and $z$ are variable expressions and $\text{exp}_1$, $\text{exp}_2$ and $\text{exp}_3$ are mathematical expressions.

### A.22.2 Assume Application Step

```c
Confirm Sequent_Form(Seq1, \text{exp}_1) \land Sequent_Form(Seq2, \text{exp}_2);
```

Note. During this step, a mathematical expression ($\text{exp}$) will only be added to a sequent’s antecedent if the variables in that sequent overlap with those in $\text{exp}$. In the example above, $\text{exp}_1$ and $\text{exp}_3$ is added to $\text{Seq}_1$ and $\text{exp}_2$ is added to $\text{Seq}_2$. Although $\text{exp}_3$ is not constrained to $x$ directly, $\text{exp}_1$ involves $x$ and $z$, therefore we add $\text{exp}_3$ to $\text{Seq}_1$. Note that this step generates parsimonious VCs where we reduce the number of antecedents in a sequent.

### A.23 Stipulate Assume Rule

This rule is a special version of the assume rule and has 2 different steps: substitution and stipulate application. Each of the steps are explained in the following subsections. If the Stipulate statement does not contain the conditions in which the step needs to be applied, then that step is skipped.

#### A.23.1 Substitution Step

```c
Confirm \text{exp}_3[ y \rightsquigarrow \text{exp}_1 ]; ( RS[ y \rightsquigarrow \text{exp}_1 ] )[ z \rightsquigarrow \text{exp}_2 ];
```

Note. The substitution step can only be used on replaceable expressions. The definition for replaceable expression can be found in Section A.22.1.
A.23.2  Stipulate Application Step

\begin{verbatim}
C\ code; Confirm Sequent_Form( Sequent_Form(RS, exp1), exp2 );
\end{verbatim}

\begin{verbatim}
C\ code; Stipulate exp1 \angle x \angle \land exp2 \angle y \angle; Confirm RS \angle x \angle;
\end{verbatim}

Note. During this step, we always add the stipulated mathematical expressions to each sequent in RS.

A.24  Confirm Rule

\begin{verbatim}
C\ code; Confirm Sequent_Form( Empty_Seq, exp) \land RS;
\end{verbatim}

\begin{verbatim}
C\ code; Confirm exp; Confirm RS;
\end{verbatim}

where Empty_Seq is a sequent that only has true as its goal and has no antecedents.

A.25  Presume Rule

\begin{verbatim}
C\ code; Confirm exp; Assume exp; Confirm RS;
\end{verbatim}

\begin{verbatim}
C\ code; Presume exp; Confirm RS;
\end{verbatim}

A.26  Swap Rule

\begin{verbatim}
C\ code; Confirm RS[ x \rightarrow y, y \rightarrow x ];
\end{verbatim}

\begin{verbatim}
C\ code; x :=: y; Confirm RS;
\end{verbatim}

A.27  If/Else Rule

\begin{verbatim}
C\ code; Confirm Invk_Cond(BE); Stipulate Math(BE); code1; Confirm RS;
\end{verbatim}

\begin{verbatim}
C\ code; Stipulate \neg Math(BE); code2; Confirm RS;
\end{verbatim}

\begin{verbatim}
C\ code; If BE then code1 else code2 end_if; Confirm RS;
\end{verbatim}

A.28  If Rule

\begin{verbatim}
C\ code; If BE then code1 else end_if; Confirm RS;
\end{verbatim}

\begin{verbatim}
C\ code; If BE then code1 end_if; Confirm RS;
\end{verbatim}
A.29 While Rule

C\ code; Confirm Inv; Change Vlist; Assume Inv ∧ NQV(RS, P_Val) = P_Exp;
If BE then body; Confirm Inv ∧ P_Exp ≤ 1 + NQV(RS, P_Val);
else Confirm RS end_if;
Confirm true;

C\ code;
While BE
changing VList;
maintaining Inv;
decreasing P_Exp;
do
  body
end;
Confirm RS;

A.30 Change Rule

C ∪ \{NQV(RS, x) \ : \ T\}\ code; Confirm RS[ x⇝NQV(RS, x) ];

C\ code; Change x; Confirm RS;

Note. The context indicates that x is of type T.

A.31 Remember Rule

C\ code; Confirm RS[ #s⇝s, #t⇝t ];

C\ code; Remember; Confirm RS/ s, #s, t, #t, u, v, ... \);

The following rules will use the example function template for Q given below.

FOD = Operation F( evaluates x: T1; restores y: T2; preserves z: T3 ) : T4;
  requires Pre≤ x, y, z, gv ↓;
  ensures F = f≤ x, y, z, gv ↓;

A.32 Function Call/Expression Reassignment Rule

C ∪ {FOD}\ code; Confirm Invk_Cond( F(exp, b, c) ); Confirm RS[ a⇝F ];

C ∪ {FOD}\ code; a := F( exp, b, c ); Confirm RS;
The following rule will use the example operation template for \( P \) given below.

\[
\text{ODP} = \text{Operation } P(\text{updates } t, \text{te}: T_1; \text{evaluates } u: T_2; \text{replaces } v, \text{ve}: T_3; \text{restores } w: T_4; \text{preserves } x: T_5; \text{alters } y: T_6; \text{clears } z: T_7 );
\]

- affects \( gv, gve \);
- requires \( \text{Pre} \prec t, \text{te}, u, w, x, y, z, gv, gve \); 
- ensures \( \text{Implicit Post} \prec t, #t, #\text{te}, u, v, w, x, #y, #z, \text{gv}, #\text{gv}, #\text{gve} \); 

\[
\begin{align*}
\text{te} &= \text{T1expr} \prec t, #t, #\text{te}, u, v, w, x, #y, #z, \text{gv}, #\text{gv}, #\text{gve} \prec \wedge \\
\text{ve} &= \text{T3expr} \prec t, #t, #\text{te}, u, v, w, x, #y, #z, \text{gv}, #\text{gv}, #\text{gve} \prec \wedge \\
\text{gve} &= \text{Gexpr} \prec t, #t, #\text{te}, u, v, w, x, #y, #z, \text{gv}, #\text{gv}, #\text{gve} \prec ;
\end{align*}
\]

A.33 Operation Invocation Rule

\[
\begin{align*}
\text{C} \cup \{\text{ODP}\} \text{ code; Confirm Invk_Cond(exp) } &\wedge \text{Pre[ Pre Subs ];} \\
\text{Assume Implicit Post[ Post Subs ] } &\wedge \\
\text{T6. Constraint}(g) &\wedge \text{T7. Is Initial}(\text{NQV(RS, h)}); \\
\text{Confirm RS[ } &l\rightarrow \text{NQV(RS, a), } b\rightarrow \text{T1expr[ Post Subs ], } c\rightarrow \text{NQV(RS, c), } \\
&d\rightarrow \text{T3expr[ Post Subs ], } g\rightarrow \text{NQV(RS, g), } h\rightarrow \text{NQV(RS, h),} \\
&\text{gve}\rightarrow \text{Gexpr[ Post Subs ] ];}
\end{align*}
\]

\[
\begin{align*}
\text{C} \cup \{\text{ODP}\} \text{ code; } P( a, b, \text{exp}, c, d, e, f, g, h ); \\
\text{Confirm RS} \prec a, b, c, d, e, f, g, h, i, \ldots \prec ;
\end{align*}
\]

This rule uses the following substitutions:

\[
\begin{align*}
\text{Pre Subs} &= [ t\rightarrow a, \text{te}\rightarrow b, u\rightarrow \text{Math(exp)}, w\rightarrow e, x\rightarrow f, y\rightarrow g, z\rightarrow h ] \\
\text{Post Subs} &= [ t\rightarrow \text{NQV(RS, a)}, #t\rightarrow a, #\text{te}\rightarrow b, u\rightarrow \text{Math(exp)}, v\rightarrow \text{NQV(RS, c)}, w\rightarrow e, \\
x\rightarrow f, #y\rightarrow g, #z\rightarrow h, \text{gv}\rightarrow \text{NQV(RS, gv)}, #\text{gv}\rightarrow \text{gv}, #\text{gve}\rightarrow \text{gve} ];
\end{align*}
\]
Appendix B  Globally Bounded List Collection

B.1  Concept

Concept  Globally_Bounded_List_Template(type Entry);
    uses  String_Theory;

    Type Family  List is modeled by Cart_Prod
        Prec, Rem: Str(Entry);
    end;
    exemplar  P;
    initialization
        ensures  P.Prec = Empty_String and P.Rem = Empty_String;
    end;

    Operation  Advance( updates  P: List );
        requires  not(P.Rem = Empty_String);
        ensures  P.Prec = #P.Prec o Prt_Btwn(0, 1, #P.Rem) and
                  P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);

    Operation  Reset( updates  P: List );
        ensures  P.Prec = Empty_String and P.Rem = #P.Prec o #P.Rem;

    Operation  Is_Rem_Empty( restores  P: List ): Boolean;
        ensures  Is_Rem_Empty = ( P.Rem = Empty_String );

    Operation  Insert( alters  New_Entry: Entry; updates  P: List );
        ensures  P.Prec = #P.Prec and P.Rem = <#New_Entry> o #P.Rem;

    Operation  Remove( replaces  Entry_Removed: Entry; updates  P: List );
        requires  not(P.Rem = Empty_String);
        ensures  P.Prec = #P.Prec and
                  Entry_Removed = DeString(Prt_Btwn(0, 1, #P.Rem)) and
                  P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);

    Operation  Advance_to_End( updates  P: List );
        ensures  P.Prec = #P.Prec o #P.Rem and
                  P.Rem = Empty_String;

    Operation  Swap_Remainders( updates  P, Q: List );
        ensures  P.Prec = #P.Prec and Q.Prec = #Q.Prec and
                  P.Rem = #Q.Rem and Q.Rem = #P.Rem;

    Operation  Is_Prec_Empty( restores  P: List ): Boolean;
        ensures  Is_Prec_Empty = ( P.Prec = Empty_String );

    Operation  Clear( clears  P: List );

end Globally_Bounded_List_Template;
B.2 Enhancements

B.2.1 List Reversal

Enhancement Reversal_Capability for Globally_Bounded_List_Template;
  Operation Reverse_List(updates L: List);
  requires L.Prec = Empty_String;
  ensures L.Prec = Reverse(#L.Rem) and L.Rem = Empty_String;
end Reversal_Capability;

B.2.2 List Search

Enhancement Searching_Capability for Globally_Bounded_List_Template;
  Operation Contains(restores E: Entry; restores L: List): Boolean;
  ensures Contains = ( Is_Substring(<E>, L.Prec) or
                        Is_Substring(<E>, L.Rem) );
end Searching_Capability;

B.3 Enhancement Realizations

B.3.1 Iterative List Reversal

Realization Iterative_Reversal_Realiz for Reversal_Capability
  of Globally_Bounded_List_Template;

Procedure Reverse_List(updates L: List);
  Var Temp_List: List;
  Var Next_Entry: Entry;
  While ( not Is_Rem_Empty(L) )
    maintaining Temp_List.Prec = Empty_String and
    Reverse(Temp_List.Rem) o L.Rem = #L.Rem;
    decreasing |L.Rem|;
    do
      Remove(Next_Entry, L);
      Insert(Next_Entry, Temp_List);
    end;
    Advance_to_End(Temp_List);
  L :=: Temp_List;
end Reverse_List;

end Iterative_Reversal_Realiz;
B.3.2 Iterative List Reversal VCs

VCs for Iterative_Reversal_Realiz.rb generated Sun Apr 15 14:49:23 EDT 2018

=================================
VC (s): =================================

VC 0_1
Base Case of the Invariant of While Statement:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):
(Empty_String = Empty_String)

Given(s):

VC 0_2
Base Case of the Invariant of While Statement:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):
((Reverse(Empty_String) o L.Rem) = L.Rem)

Given(s):
1. (L.Prec = Empty_String)

VC 0_3
Requires Clause of Remove [After Logical Reduction(s)]:
Iterative_Reversal_Realiz.rb(13:3)

Goal(s):
(L''.Rem = Empty_String)

Given(s):
1. (L''.Rem = Empty_String)
2. ((Reverse(Temp_List''''.Rem) o L''.Rem) = L.Rem)
3. (Temp_List''''.Prec = Empty_String)
4. (L.Prec = Empty_String)

VC 0_4
Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):
Given(s):
1. (Temp_List''.Prec = Empty_String)
2. (Temp_List''.Rem = (<DeString(Prt_Btwn(0, 1, L''.Rem))> o Temp_List'''.Rem))
3. (L'.Prec = L''.Prec)
4. (L'.Rem = Prt_Btwn(1, |L''.Rem|, L''.Rem))
5. ((Reverse(Temp_List'''.Rem) o L''.Rem) = L.Rem)
6. (L.Prec = Empty_String)

VC 0_5

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):

((Reverse((<DeString(Prt_Btwn(0, 1, L''.Rem))> o Temp_List'''.Rem)) o Prt_Btwn(1, |L''.Rem|, L''.Rem)) = (Reverse(Temp_List'''.Rem) o L''.Rem)) or (L''.Rem = Empty_String)

Given(s):
1. (Temp_List''.Prec = Empty_String)
2. (L'.Prec = L''.Prec)

VC 0_6

Termination of While Statement [After Logical Reduction(s)]:
Iterative_Reversal_Realiz.rb(ll:3)

Goal(s):

((1 + |Prt_Btwn(1, |L''.Rem|, L''.Rem)|) <= |L''.Rem|) or (L''.Rem = Empty_String)

Given(s):
1. (L'.Prec = L''.Prec)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L''.Rem)))
3. ((Reverse(Temp_List'''.Rem) o L''.Rem) = L.Rem)
4. (Temp_List'''.Prec = Empty_String)
5. (L.Prec = Empty_String)

VC 1_1

Base Case of the Invariant of While Statement:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):

(Empty_String = Empty_String)
Given(s):

VC 1_2

Base Case of the Invariant of While Statement:
Iterative_Reversal_Realiz.rb(9:15)

Goal(s):

((Reverse(Empty_String) o L.Rem) = L.Rem)

Given(s):

1. (L.Prec = Empty_String)

VC 1_3

Ensures Clause of Reverse_List: Iterative_Reversal_Realiz.rb(4:14)

Goal(s):

((Empty_String o Temp_List''.Rem) = Reverse((Reverse(Temp_list’’.Rem) o L’.Rem)))

Given(s):

1. (L’.Rem = Empty_String)

VC 1_4

Ensures Clause of Reverse_List: Iterative_Reversal_Realiz.rb(4:14)

Goal(s):

(Empty_String = Empty_String)

Given(s):

1. (L’.Rem = Empty_String)
2. ((Reverse(Temp_List’’.Rem) o L’.Rem) = L.Rem)
3. (Temp_List’’.Prec = Empty_String)
4. (L.Prec = Empty_String)
B.3.3 Recursive List Reversal

**Realization** Recursive_Reversal_Realiz for Reversal_Capability of Globally_Bounded_List_Template;

**Recursive Procedure** Reverse_List(updates L: List);

decreasing |L.Rem|;

Var E: Entry;

If (not Is_Rem_Empty(L) ) then
    Remove(E, L);
    Reverse_List(L);
    Insert(E, L);
    Advance(L);
end;
end Reverse_List;

end Recursive_Reversal_Realiz;

B.3.4 Recursive List Reversal VCs

VCs for Recursive_Reversal_Realiz.rb generated Sun Apr 15 14:49:36 EDT 2018

=================================
VC(s): =================================

VC 0_1
Requires Clause of Remove [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(10:12)

Goal(s):
(L.Rem = Empty_String)

Given(s):
1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

VC 0_2
Termination of Recursive Call [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(11:12)

Goal(s):
((1 + |Prt_Btwn(1, |L.Rem|, L.Rem)|) <= |L.Rem|) or
(L.Rem = Empty_String)

Given(s):
1. (L''''.Prec = Empty_String)
2. \((E'' = \text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem})))\)

\textbf{VC 0_3}

Requires Clause of Reverse_List [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(11:12)

\textbf{Goal(s)}:
\[(L''''\text{.Prec} = \text{Empty\_String}) \text{ or } (\text{L.Rem} = \text{Empty\_String})\]

\textbf{Given(s)}:
1. \((L''''\text{.Prec} = \text{Empty\_String})\)
2. \((E'' = \text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem})))\)
3. \((L''''\text{.Rem} = \text{Prt_Btwn}(1, |\text{L.Rem}|, \text{L.Rem}))\)

\textbf{VC 0_4}

Requires Clause of Advance [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(13:12)

\textbf{Goal(s)}:
\((\text{L.Rem} = \text{Empty\_String})\)

\textbf{Given(s)}:
1. \(\langle\text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem}))\rangle \text{ o Empty\_String} = \text{Empty\_String}\)
2. \((L''\text{.Prec} = \text{Reverse}(\text{Prt_Btwn}(1, |\text{L.Rem}|, \text{L.Rem})))\)
3. \((L''''\text{.Prec} = \text{Empty\_String})\)

\textbf{VC 0_5}

Ensures Clause of Reverse_List [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(4:24)

\textbf{Goal(s)}:
\(((\text{L.''\text{.Prec o Prt_Btwn}(0, 1, \langle\text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem}))\rangle \text{ o Empty\_String}) = \text{Reverse}(\text{L.Rem})) \text{ or } (\text{L.Rem} = \text{Empty\_String}))\)

\textbf{Given(s)}:
1. \((\text{L'.Rem} = \text{Prt_Btwn}(1, |\langle\text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem}))\rangle \text{ o Empty\_String})|,\langle\text{DeString}(\text{Prt_Btwn}(0, 1, \text{L.Rem}))\rangle \text{ o Empty\_String}))\)
2. \((L''\text{.Prec} = \text{Reverse}(\text{Prt_Btwn}(1, |\text{L.Rem}|, \text{L.Rem})))\)
3. \((L''''\text{.Prec} = \text{Empty\_String})\)

\textbf{VC 0_6}

Ensures Clause of Reverse_List [After Logical Reduction(s)]:
Recursive_Reversal_Realiz.rb(4:24)
Goal(s):

(Prt_Btwn(1, |(<DeString(Prt_Btwn(0, 1, L.Rem))> o Empty_String)),
   (<DeString(Prt_Btwn(0, 1, L.Rem))> o Empty_String)) = Empty_String) or
(L.Rem = Empty_String)

Given(s):
1. (L'.Prec = (L''.Prec o Prt_Btwn(0, 1, (<DeString(Prt_Btwn(0, 1, L.Rem))> o
   Empty_String))))
2. (L''.Prec = Reverse(Prt_Btwn(1, |L.Rem|, L.Rem)))
3. (L''''.Prec = Empty_String)

VC 1_1

Ensures Clause of Reverse_List: Recursive_Reversal_Realiz.rb(4:24)

Goal(s):

(Empty_String = Reverse(L.Rem))

Given(s):
1. (L.Rem = Empty_String)

VC 1_2

Ensures Clause of Reverse_List: Recursive_Reversal_Realiz.rb(4:24)

Goal(s):

(Empty_String = Empty_String)

Given(s):
1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)
B.3.5 Iterative List Search

Realization  Iterative_Searching_Realiz(
    Operation  Compare_Entry(restores E, F: Entry) : Boolean;
        ensures  Compare_Entry = ( E = F );
    for  Searching_Capability of  Globally_Bounded_List_Template;

    Operation  Is_Present_In_Rem (restores E: Entry; restores L: List): Boolean;
        requires  L.Prec = Empty_String;
        ensures  Is_Present_In_Rem = ( Is_Substring(<E>, L.Rem) );
    Procedure
        Var  Next_Entry: Entry;
        Is_Present_In_Rem := False();
        While  ( not Is_Rem_Empty(L) )
            maintaining  L.Prec o L.Rem = #L.Rem and E = #E and
                Is_Present_In_Rem = Is_Substring(<E>, L.Prec);
            decreasing  |L.Rem|;
            do
                Remove(Next_Entry, L);
                If  ( Compare_Entry(Next_Entry, E) ) then
                    Is_Present_In_Rem := True();
                    Insert(Next_Entry, L);
                    Advance_to_End(L);
                else
                    Insert(Next_Entry, L);
                    Advance(L);
                end;
            end;
        Reset(L);
    end  Is_Present_In_Rem;

    Procedure  Contains (restores E: Entry; restores L: List): Boolean;
        Var  Temp_Rem_List: List;
        Contains := False();
          -- Store L.Rem in a temporary list
        Swap_Remainders(L, Temp_Rem_List);
        Contains := Is_Present_In_Rem(E, Temp_Rem_List);
          -- If not found, check L.Prec
        If  ( not Contains ) then
            Reset(L);
            Contains := Is_Present_In_Rem(E, L);
            Advance_to_End(L);
        end;
          -- Restore the list
        Swap_Remainders(L, Temp_Rem_List);
    end  Contains;

end  Iterative_Searching_Realiz;
B.3.6 Iterative List Search VCs

VCs for Iterative_Searching_Realiz.rb generated Sun Apr 15 15:28:19 EDT 2018

=================================
VC(s): =================================
VC 0_1
Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
((L.Prec o L.Rem) = L.Rem)

Given(s):
1. (L.Prec = Empty_String)

VC 0_2
Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
(E = E)

Given(s):

VC 0_3
Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
(false = Is_Substring(<E>, L.Prec))

Given(s):
1. (L.Prec = Empty_String)

VC 0_4
Requires Clause of Remove [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(18:12)

Goal(s):
(L'''''.Rem = Empty_String)

Given(s):
1. (L'''.Rem = Empty_String)
2. ((L'''''.Prec o L'''''.Rem) = L.Rem)
3. (false = Is_Substring(<E>, L'''''.Prec))
4. (E' = E)
5. (L.Prec = Empty_String)

VC 0_5

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

((L'''.Prec o (<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))) o L''.Rem) = (L'''''.Prec o L'''''.Rem)) or (L'''''.Rem = Empty_String)

Given(s):

1. (L''.Rem = Empty_String)
2. (L'''.Prec = L'''''.Prec)
3. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
4. (L'''.Prec = L'''''.Prec)
5. (false = Is_Substring(<E>, L'''''.Prec))
6. (L.Prec = Empty_String)

VC 0_6

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(E = E) or (L'''''.Rem = Empty_String)

Given(s):

1. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
2. (L'''.Prec = L'''''.Prec)
3. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
4. (L'''.Prec = L'''''.Prec)
5. (false = Is_Substring(<E>, L'''''.Prec))
6. (L.Prec = Empty_String)

VC 0_7

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(true = Is_Substring(<E>, (L'''''.Prec o (<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem)))) or
(L''.Rem = Empty_String)

**Given**(s):

1. (L''.Prec = L''.Prec)
2. (DeString(Prt_Btwn(0, 1, L''.Rem)) = E)
3. (L''.Prec = L''.Prec)
4. ((L''.Prec o L''.Rem) = L.Rem)
5. (false = Is_Substring(<E>, L''.Prec))
6. (L.Prec = Empty_String)

**VC** 0_8

Termination of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(16:12)

**Goal**(s):

\(((1 + |L''.Rem|) <= |L''.Rem|) or (L''.Rem = Empty_String)\)

**Given**(s):

1. (L''.Prec = (L''.Prec o (DeString(Prt_Btwn(0, 1, L''.Rem)) o Prt_Btwn(1, |L''.Rem|, L''.Rem))))
2. (L''.Rem = Empty_String)
3. (L''.Prec = L''.Prec)
4. (DeString(Prt_Btwn(0, 1, L''.Rem)) = E)
5. (L''.Prec = L''.Prec)
6. ((L''.Prec o L''.Rem) = L.Rem)
7. (false = Is_Substring(<E>, L''.Prec))
8. (L.Prec = Empty_String)

**VC** 1_1

Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

**Goal**(s):

\(((L.Prec o L.Rem) = L.Rem)\)

**Given**(s):

1. (L.Prec = Empty_String)

**VC** 1_2

Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

**Goal**(s):

(E = E)
Given(s):

VC 1_3

Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(false = Is_Substring(<E>, L.Prec))

Given(s):

1. (L.Prec = Empty_String)

VC 1_4

Requires Clause of Remove [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(18:12)

Goal(s):

(L'''''.Rem = Empty_String)

Given(s):

1. (L'''''.Rem = Empty_String)
2. ((L'''''.Prec o L'''''.Rem) = L.Rem)
3. (false = Is_Substring(<E>, L'''''.Prec))
4. (E' = E)
5. (L.Prec = Empty_String)

VC 1_5

Requires Clause of Advance [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(25:16)

Goal(s):

(L'''''.Rem = Empty_String)

Given(s):

1. ((DeString(Prt_Btwn(0, 1, L'''''.Rem)) o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem)) = Empty_String)
2. (L'''''.Prec = L'''''.Prec)
3. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) /= E)
4. (L'''''.Prec = L''''''.Prec)
5. ((L''''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (L.Prec = Empty_String)

VC 1_6
Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

\((((L^{''}.\text{Prec} \circ \text{Prt\_Btwn}(0, 1, \langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}))) \circ \text{Prt\_Btwn}(1, |\langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}||, (\langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem})) = (L^{''}.\text{Prec} \circ L^{'''}).\text{Rem})\) or
\((L^{'''}).\text{Rem} = \text{Empty\_String})

Given(s):

1. \((L^{''}.\text{Prec} = L^{'''}).\text{Prec})
2. \((\text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\) /= E\))
3. \((L^{''}.\text{Prec} = L^{'''}).\text{Prec})
4. \((\text{false} = \text{Is\_Substring}(E, L^{'''}).\text{Prec})\))
5. \((L.\text{Prec} = \text{Empty\_String})\)

VC 1_7

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

\((E = E)\) or \((L^{'''}).\text{Rem} = \text{Empty\_String})

Given(s):

1. \((\text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\) /= E\))
2. \((L^{''}.\text{Prec} = L^{'''}).\text{Prec})
3. \((L^{''}.\text{Rem} = \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}))
4. \((\langle L^{''}.\text{Prec} \circ L^{'''}).\text{Rem} \rangle = L.\text{Rem})
5. \((\text{false} = \text{Is\_Substring}(E, L^{'''}).\text{Prec})\))
6. \((L.\text{Prec} = \text{Empty\_String})\)

VC 1_8

Inductive Case of Invariant of While Statement [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):

\((\text{Is\_Substring}(E, L^{'''}).\text{Prec}) = \text{Is\_Substring}(E, (L^{''}.\text{Prec} \circ \text{Prt\_Btwn}(0, 1, \langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}))) \) or
\((L^{'''}).\text{Rem} = \text{Empty\_String})

Given(s):

1. \((L^{''}.\text{Rem} = \text{Prt\_Btwn}(1, |\langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}||, (\langle \text{DeString}(\text{Prt\_Btwn}(0, 1, L^{'''}).\text{Rem})\rangle \circ \text{Prt\_Btwn}(1, |L^{'''}\text{Rem}|, L^{'''}).\text{Rem}))\))
1. \( L'\).Rem) > o Prt_Btwn(1, |L'\'.Rem|, L'\'.Rem)))
2. (L'\'.Prec = L'\'.Prec)
3. (DeString(Prt_Btwn(0, 1, L'\'.Rem)) /= E)
4. (L'\'.Prec = L'\'.Prec)
5. ((L'\'.Prec o L'\'.Rem) = L.Rem)
6. (L.Prec = Empty_String)

**VC 1_9**

Termination of While Statement [After Logical Reduction(s)]:
Iterative.Searching.Realiz.rb(16:12)

**Goal(s):**

\[\((1 + |Prt_Btwn(1, \langle DeString(Prt_Btwn(0, 1, L'\'.Rem)\rangle o Prt_Btwn(1, |L'\'.Rem|, L'\'.Rem))\rangle o Prt_Btwn(1, |L'\'.Rem|, L'\'.Rem))))| \leq |L'\'.Rem|\) or (L'\'.Rem = Empty_String)\]

**Given(s):**

1. (L'\'.Prec = (L'\'.Prec o Prt_Btwn(0, 1, \langle DeString(Prt_Btwn(0, 1, L'\'.Rem)\rangle o Prt_Btwn(1, |L'\'.Rem|, L'\'.Rem))))
2. (L'\'.Prec = L'\'.Prec)
3. (DeString(Prt_Btwn(0, 1, L'\'.Rem)) /= E)
4. (L'\'.Prec = L'\'.Prec)
5. ((L'\'.Prec o L'\'.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'\'.Prec))
7. (L.Prec = Empty_String)

**VC 2_1**

Base Case of the Invariant of While Statement:
Iterative.Searching.Realiz.rb(14:24)

**Goal(s):**

\((L.Prec o L.Rem) = L.Rem)\]

**Given(s):**

1. (L.Prec = Empty_String)

**VC 2_2**

Base Case of the Invariant of While Statement:
Iterative.Searching.Realiz.rb(14:24)

**Goal(s):**

\((E = E)\]

**Given(s):**
Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
(false = Is_Substring(<E>, L.Prec))

Given(s):
1. (L.Prec = Empty_String)

VC 2_4

Ensures Clause of Is_Present_In_Rem: Iterative_Searching_Realiz.rb(6:14)

Goal(s):
(Is_Substring(<E>, L''.Prec) = Is_Substring(<E>, (L''.Prec o L''.Rem)))

Given(s):
1. (L''.Rem = Empty_String)
2. ((L''.Prec o L''.Rem) = L.Rem)
3. (L.Prec = Empty_String)

VC 2_5

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Iterative_Searching_Realiz.rb(6:42)

Goal(s):
(E = E)

Given(s):
1. (L''.Rem = Empty_String)
2. ((L''.Prec o L''.Rem) = L.Rem)
3. (false = Is_Substring(<E>, L''.Prec))
4. (L.Prec = Empty_String)

VC 2_6

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Iterative_Searching_Realiz.rb(6:61)

Goal(s):
(L''.Prec = L.Prec)

Given(s):
1. (L''.Prec = Empty_String)
2. \((L'.Rem = (L''.Prec o L''.Rem))\)
3. \((L''.Rem = Empty_String)\)
4. \(((L''.Prec o L''.Rem) = L.Rem)\)
5. \((false = Is_Substring(<E>, L''.Prec))\)
6. \((E' = E)\)
7. \((L.Prec = Empty_String)\)

**VC 2_7**

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):

Iterative_Searching_Realiz.rb(6:61)

**Goal(s):**

\((L''.Prec o L''.Rem) = (L''.Prec o L''.Rem)\)

**Given(s):**

1. \((L''.Rem = Empty_String)\)
2. \((false = Is_Substring(<E>, L''.Prec))\)
3. \((E' = E)\)
4. \((L.Prec = Empty_String)\)

**VC 3_1**

Requires Clause of Is_Present_In_Rem: Iterative_Searching_Realiz.rb(38:20)

**Goal(s):**

\((Temp_Rem_List''.Prec = Empty_String)\)

**Given(s):**

1. \((Temp_Rem_List''.Prec = Temp_Rem_List.Prec)\)
2. \((L''''.Rem = Temp_Rem_List.Rem)\)
3. \((Temp_Rem_List''.Rem = L.Rem)\)
4. \((L''''.Prec = L.Prec)\)
5. \((Temp_Rem_List.Prec = Empty_String)\)
6. \((Temp_Rem_List.Rem = Empty_String)\)

**VC 3_2**

Requires Clause of Is_Present_In_Rem [After Logical Reduction(s)]:

Iterative_Searching_Realiz.rb(43:24)

**Goal(s):**

\((L''''.Prec = Empty_String)\) or
\(Is_Substring(<E>, Temp_Rem_List''.Rem)\)

**Given(s):**

1. \((L''''.Prec = Empty_String)\)
2. \((L''''.Rem = (L''''.Prec o L''''.Rem))\)
3. \((L''''.Prec = L.Prec)\)
4. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
5. (L''''.Rem = Temp_Rem_List.Rem)
6. (Temp_Rem_List''.Rem = L.Rem)
7. (Temp_Rem_List.Prec = Empty_String)
8. (Temp_Rem_List.Rem = Empty_String)

VC 3_3

Ensures Clause of Contains [After Logical Reduction(s)]:
Iterative_Searching_Realiz.rb(32:14)

Goal(s):
(Is_Substring(<E>, (L''''.Prec o L''''.Rem)) = (Is_Substring(<E>, L'.Prec) or Is_Substring(<E>, L'.Rem)))
or
Is_Substring(<E>, Temp_Rem_List''.Rem)

Given(s):
1. (L'.Prec = (L''''.Prec o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List''.Rem = L''.Rem)
4. (Temp_Rem_List'.Rem = Temp_Rem_List''.Prec)
5. (L''.Rem = Empty_String)
6. (L'''.Prec = Empty_String)
7. (L'''.Prec = L.Prec)
8. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
9. (L'''.Rem = Temp_Rem_List.Rem)
10. (Temp_Rem_List''.Rem = L.Rem)
11. (Temp_Rem_List.Prec = Empty_String)
12. (Temp_Rem_List.Rem = Empty_String)

VC 3_4

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:33)

Goal(s):
(E = E) or
Is_Substring(<E>, Temp_Rem_List''.Rem)

Given(s):
1. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
2. (L'''.Rem = Temp_Rem_List.Rem)
3. (Temp_Rem_List''.Rem = L.Rem)
4. (L'''.Prec = L.Prec)
5. (Temp_Rem_List'.Prec = Empty_String)
6. (Temp_Rem_List.Rem = Empty_String)

VC 3_5

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:52)
Goal(s):

(L'.Prec = L.Prec) or
Is_Substring(<E>, Temp_Rem_List''.Rem)

Given(s):

1. (L'.Prec = (L'''.Prec o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List''.Rem = L''.Rem)
4. (Temp_Rem_List’’.Prec = Temp_Rem_List’’.Prec)
5. (L''.Rem = Empty_String)
6. (L''''.Prec = Empty_String)
7. (L'''.Prec = L.Prec)
8. (Temp_Rem_List’’.Prec = Temp_Rem_List.Prec)
9. (L'''.Rem = Temp_Rem_List.Rem)
10. (Temp_Rem_List’’.Rem = L.Rem)
11. (Temp_Rem_List.Prec = Empty_String)
12. (Temp_Rem_List.Rem = Empty_String)

VC 3_6

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:52)

Goal(s):

(L'.Rem = L.Rem) or
Is_Substring(<E>, Temp_Rem_List’’.Rem)

Given(s):

1. (L'.Prec = (L'''.Prec o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List’’.Rem = L''.Rem)
4. (Temp_Rem_List’’.Prec = Temp_Rem_List’’.Prec)
5. (L''.Rem = Empty_String)
6. (L''''.Prec = Empty_String)
7. (L'''.Prec = L.Prec)
8. (Temp_Rem_List’’.Prec = Temp_Rem_List.Prec)
9. (L'''.Rem = Temp_Rem_List.Rem)
10. (Temp_Rem_List’’.Rem = L.Rem)
11. (Temp_Rem_List.Prec = Empty_String)
12. (Temp_Rem_List.Rem = Empty_String)

VC 4_1

Requires Clause of Is_Present_In_Rem: Iterative_Searching_Realiz.rb(38:20)

Goal(s):

(Temp_Rem_List’’.Prec = Empty_String)

Given(s):
1. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
2. (L''.Rem = Temp_Rem_List.Rem)
3. (Temp_Rem_List''.Rem = L.Rem)
4. (L''.Prec = L.Prec)
5. (Temp_Rem_List.Prec = Empty_String)
6. (Temp_Rem_List.Rem = Empty_String)

VC 4.2

Ensures Clause of Contains: Iterative_Searching_Realiz.rb(32:14)

Goal(s):
(Is_Substring(<E>, Temp_Rem_List''.Rem) = (Is_Substring(<E>, L'.Prec) or
   Is_Substring(<E>, L'.Rem)))

Given(s):
1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List''.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
8. (Temp_Rem_List'.Rem = Temp_Rem_List.Rem)
9. (Temp_Rem_List''.Rem = L.Rem)
10. (Temp_Rem_List.Rem = Empty_String)
11. (Temp_Rem_List.Rem = Empty_String)

VC 4.3

Ensures Clause of Contains (Condition from "RESTORES" parameter mode): Iterative_Searching_Realiz.rb(32:33)

Goal(s):
(E = E)

Given(s):

1. Is_Substring(<E>, Temp_Rem_List''.Rem)
2. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
3. (L''.Rem = Temp_Rem_List.Rem)
4. (Temp_Rem_List''.Rem = L.Rem)
5. (L''.Prec = L.Prec)
6. (Temp_Rem_List.Prec = Empty_String)
7. (Temp_Rem_List.Rem = Empty_String)

VC 4.4

Ensures Clause of Contains (Condition from "RESTORES" parameter mode): Iterative_Searching_Realiz.rb(32:52)
Goal(s):

(L'.Prec = L.Prec)

Given(s):

1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
8. (L''.Rem = Temp_Rem_List.Rem)
9. (Temp_Rem_List''.Rem = L.Rem)
10. (Temp_Rem_List.Prec = Empty_String)
11. (Temp_Rem_List.Rem = Empty_String)

VC 4_5

Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Iterative_Searching_Realiz.rb(32:52)

Goal(s):

(L'.Rem = L.Rem)

Given(s):

1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Temp_Rem_List.Prec)
8. (L''.Rem = Temp_Rem_List.Rem)
9. (Temp_Rem_List''.Rem = L.Rem)
10. (Temp_Rem_List.Prec = Empty_String)
11. (Temp_Rem_List.Rem = Empty_String)
B.3.7 Recursive List Search

**Realization** Recursive_searching_realiz(
    
    **Operation** Compare_Entry(restores E, F: Entry) : Boolean;
    
    **ensures** Compare_Entry = ( E = F );
) **for** Searching_Capability **of** Globally_Bounded_List_Template;

**Operation** Is_Present_In_Rem (restores E: Entry; restores L: List): Boolean;

**requires** L.Prec = Empty_String;

**ensures** Is_Present_In_Rem = ( Is_Substring(<E>, L.Rem) );

**Recursive Procedure**

    decreasing |L.Rem|;

    **Var** Next_Entry: Entry;

    Is_Present_In_Rem := False();

    **If** ( not Is_Rem_Empty(L) ) **then**

    Remove(Next_Entry, L);

    **If** ( not Is_Present_In_Rem(E, L) ) **then**

    Is_Present_In_Rem := Compare_Entry(E, Next_Entry);

    **else**

    Is_Present_In_Rem := True();

    **end**;

    Insert(Next_Entry, L);

    **end**; 

end Is_Present_In_Rem;

**Procedure** Contains (restores E: Entry; restores L: List): Boolean;

    **Var** Temp_Rem_List: List;

    Contains := False();

    **-- Store L.Rem in a temporary list**

    Swap_Remainders(L, Temp_Rem_List);

    Contains := Is_Present_In_Rem(E, Temp_Rem_List);

    **-- If not found, check L.Prec**

    **If** ( not Contains ) **then**

    Reset(L);

    Contains := Is_Present_In_Rem(E, L);

    Advance_to_End(L);

    **end**;

    **-- Restore the list**

    Swap_Remainders(L, Temp_Rem_List);

end Contains;

end Recursive_searching_realiz;
B.3.8 Recursive List Search VCs

VCs for Recursive_Searching_Realiz.rb generated Sun Apr 15 14:50:03 EDT 2018

=================================
VC(s): ==================================

VC 0_1
Requires Clause of Remove [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(16:12)

Goal(s):
(L.Rem = Empty_String)

Given(s):
1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

VC 0_2
Termination of Recursive Call [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(18:21)

Goal(s):
((1 + |Prt_Btwn(1, |L.Rem|, L.Rem)|) <= |L.Rem|) or
(L.Rem = Empty_String)

Given(s):
1. (L''.Prec = Empty_String)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))

VC 0_3
Requires Clause of Is_Present_In_Rem [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(18:21)

Goal(s):
(L''.Prec = Empty_String) or
(L.Rem = Empty_String)

Given(s):
1. (L''.Prec = Empty_String)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
3. (L''.Rem = Prt_Btwn(1, |L.Rem|, L.Rem))

VC 0_4
Ensures Clause of Is_Present_In_Rem [After Logical Reduction(s)]:

143
Recursive_Searching_Realiz.rb(6:14)

Goal(s):

((E = DeString(Prt_Btwn(0, 1, L.Rem))) = Is_Substring(<E>,
  (DeString(Prt_Btwn(0, 1, L.Rem)) o Prt_Btwn(1, |L.Rem|, L.Rem)))) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem)) or
(L.Rem = Empty_String)

Given(s):

1. (L’.Prec = L’’’.Prec)
2. (L’’’.Prec = Empty_String)

VC 0_5

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:42)

Goal(s):

(E = E) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem)) or
(L.Rem = Empty_String)

Given(s):

1. (L’’’.Prec = Empty_String)
2. (Next_Entry’’ = DeString(Prt_Btwn(0, 1, L.Rem)))

VC 0_6

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

(L’.Prec = Empty_String) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem)) or
(L.Rem = Empty_String)

Given(s):

1. (L’.Prec = L’’’.Prec)
2. (L’.Rem = (DeString(Prt_Btwn(0, 1, L.Rem)) o Prt_Btwn(1, |L.Rem|, L.Rem)))
3. (L’’’.Prec = Empty_String)

VC 0_7

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

((DeString(Prt_Btwn(0, 1, L.Rem)) o Prt_Btwn(1, |L.Rem|, L.Rem)) = L.Rem) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem)) or 
(L.Rem = Empty_String)

Given(s):
1. (L’.Prec = L’’.Prec)
2. (L’’.Prec = Empty_String)

VC 1_1

Requires Clause of Remove [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(16:12)

Goal(s):
(L.Rem = Empty_String)

Given(s):
1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

VC 1_2

Ensures Clause of Is_Present_In_Rem [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(6:14)

Goal(s):
(true = Is_Substring(<E>, {DeString(Prt_Btwn(0, 1, L.Rem))} o Prt_Btwn(1, 
|L.Rem|, L.Rem))) or 
(L.Rem = Empty_String)

Given(s):
1. (L’.Prec = L’’.Prec)
2. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
3. (L’’.Prec = Empty_String)

VC 1_3

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode) 
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:42)

Goal(s):
(E = E) or 
(L.Rem = Empty_String)

Given(s):
1. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
2. (L’’.Prec = Empty_String)
3. (Next_Entry’’ = DeString(Prt_Btwn(0, 1, L.Rem)))
VC 1_4

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

(L'.Prec = Empty_String) or
(L.Rem = Empty_String)

Given(s):

1. (L'.Prec = L''.Prec)
2. (L.Rem = (<DeString(Prt_Btwn(0, 1, L.Rem))> o Prt_Btwn(1, |L.Rem|, L.Rem)))
3. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
4. (L''.Prec = Empty_String)

VC 1_5

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

((<DeString(Prt_Btwn(0, 1, L.Rem))> o Prt_Btwn(1, |L.Rem|, L.Rem)) = L.Rem) or
(L.Rem = Empty_String)

Given(s):

1. (L'.Prec = L''.Prec)
2. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
3. (L''.Prec = Empty_String)

VC 2_1

Ensures Clause of Is_Present_In_Rem: Recursive_Searching_Realiz.rb(6:14)

Goal(s):

(false = Is_Substring(<E>, Empty_String))

Given(s):

1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

VC 2_2

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode): Recursive_Searching_Realiz.rb(6:42)

Goal(s):

(E = E)
**Given (s):**

1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

**VC 2.3**

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(6:61)

**Goal (s):**

(Empty_String = Empty_String)

**Given (s):**

1. (L.Rem = Empty_String)

**VC 2.4**

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(6:61)

**Goal (s):**

(Empty_String = L.Rem)

**Given (s):**

1. (L.Rem = Empty_String)
2. (L.Prec = Empty_String)

**VC 3.1**

Requires Clause of Is_Present_In_Rem: Recursive_Searching_Realiz.rb(35:20)

**Goal (s):**

(Temp.Rem.List'''.Prec = Empty_String)

**Given (s):**

1. (Temp.Rem.List'''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
4. (L''''.Prec = L.Prec)

**VC 3.2**

Requires Clause of Is_Present_In_Rem [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(40:24)

**Goal (s):**

(Empty_String = Empty_String) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

**Given**

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L''''.Prec = L.Prec)

**VC 3_3**

Ensures Clause of Contains [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(29:14)

**Goal**

(Is_Substring(<E>, (L''''.Prec o L''''.Rem)) = (Is_Substring(<E>, L'.Prec) or Is_Substring(<E>, L'.Rem))) or Is_Substring(<E>, Temp_Rem_List'''.Rem)

**Given**

1. (L'.Prec = (Empty_String o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'.Rem = Empty_String)
4. (Temp_Rem_List'.Prec = Temp_Rem_List'''.Prec)
5. (L''''.Prec = L.Prec)
6. (Temp_Rem_List'''.Prec = Empty_String)
7. (L''''.Rem = Empty_String)
8. (Temp_Rem_List'''.Rem = L.Rem)

**VC 3_4**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(29:33)

**Goal**

(E = E) or Is_Substring(<E>, Temp_Rem_List'''.Rem)

**Given**

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L''''.Prec = L.Prec)

**VC 3_5**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(29:52)

**Goal**
\( L'.\text{Prec} = L.\text{Prec} \) or
Is_Substring\(<E>, Temp\_Rem\_List''.Rem\)

**Given**\( (s) \):

1. \( (L'.\text{Prec} = (\text{Empty\_String} \circ (L''''.\text{Prec} \circ L''''.\text{Rem}))) \)
2. \( (L'.\text{Rem} = \text{Temp\_Rem\_List''''.Rem}) \)
3. \( (\text{Temp\_Rem\_List'}\text{Rem} = \text{Empty\_String}) \)
4. \( (\text{Temp\_Rem\_List'}\text{Prec} = \text{Temp\_Rem\_List''''.Prec}) \)
5. \( (L''''.\text{Prec} = L.\text{Prec}) \)
6. \( (\text{Temp\_Rem\_List''}.\text{Prec} = \text{Empty\_String}) \)
7. \( (L''''.\text{Rem} = \text{Empty\_String}) \)
8. \( (\text{Temp\_Rem\_List''}.\text{Rem} = L.\text{Rem}) \)

**VC 3_6**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive\_Searching\_Realiz.rb(29:52)

**Goal**\( (s) \):

\( (L'.\text{Rem} = L.\text{Rem}) \) or
Is_Substring\(<E>, Temp\_Rem\_List'''.Rem\)

**Given**\( (s) \):

1. \( (L'.\text{Prec} = (\text{Empty\_String} \circ (L''''.\text{Prec} \circ L''''.\text{Rem}))) \)
2. \( (L'.\text{Rem} = \text{Temp\_Rem\_List''''.Rem}) \)
3. \( (\text{Temp\_Rem\_List'}\text{Rem} = \text{Empty\_String}) \)
4. \( (\text{Temp\_Rem\_List'}\text{Prec} = \text{Temp\_Rem\_List''''.Prec}) \)
5. \( (L''''.\text{Prec} = L.\text{Prec}) \)
6. \( (\text{Temp\_Rem\_List''}.\text{Prec} = \text{Empty\_String}) \)
7. \( (L''''.\text{Rem} = \text{Empty\_String}) \)
8. \( (\text{Temp\_Rem\_List''}.\text{Rem} = L.\text{Rem}) \)

**VC 4_1**

Requires Clause of Is\_Present\_In\_Rem: Recursive\_Searching\_Realiz.rb(35:20)

**Goal**\( (s) \):

\( (\text{Temp\_Rem\_List''}.\text{Prec} = \text{Empty\_String}) \)

**Given**\( (s) \):

1. \( (\text{Temp\_Rem\_List''}.\text{Prec} = \text{Empty\_String}) \)
2. \( (L''.\text{Rem} = \text{Empty\_String}) \)
3. \( (\text{Temp\_Rem\_List''}.\text{Rem} = L.\text{Rem}) \)
4. \( (L''.\text{Prec} = L.\text{Prec}) \)

**VC 4_2**

Ensures Clause of Contains: Recursive\_Searching\_Realiz.rb(29:14)

**Goal**\( (s) \):

149
(Is_Substring(<E>, Temp_Rem_List'''.Rem) = (Is_Substring(<E>, L'.Prec) or
   Is_Substring(<E>, L'.Rem)))

**Given** (s):

1. (L'.Prec = L'''.Prec)
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'''.Rem = L''''.Rem)
4. (Temp_Rem_List'''.Prec = Temp_Rem_List'''.Prec)
5. Is_Substring(<E>, Temp_Rem_List'''.Rem)
6. (L'''.Prec = L.Prec)
7. (Temp_Rem_List'''.Prec = Empty_String)
8. (L'''.Rem = Empty_String)
9. (Temp_Rem_List'''.Rem = L.Rem)

**VC 4_3**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(29:33)

**Goal** (s):

(E = E)

**Given** (s):

1. Is_Substring(<E>, Temp_Rem_List'''.Rem)
2. (Temp_Rem_List'''.Prec = Empty_String)
3. (L'''.Rem = Empty_String)
4. (Temp_Rem_List'''.Rem = L.Rem)
5. (L'''.Prec = L.Prec)

**VC 4_4**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(29:52)

**Goal** (s):

(L'.Prec = L.Prec)

**Given** (s):

1. (L'.Prec = L'''.Prec)
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'''.Rem = L''''.Rem)
4. (Temp_Rem_List'''.Prec = Temp_Rem_List'''.Prec)
5. Is_Substring(<E>, Temp_Rem_List'''.Rem)
6. (L'''.Prec = L.Prec)
7. (Temp_Rem_List'''.Prec = Empty_String)
8. (L'''.Rem = Empty_String)
9. (Temp_Rem_List'''.Rem = L.Rem)

**VC 4_5**
Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(29:52)

Goal(s):

(L'.Rem = L.Rem)

Given(s):

1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List’.Prec = Temp_Rem_List’’.Prec)
5. Is_Substring(<E>, Temp_Rem_List’’’.Rem)
6. (L’’.Prec = L.Prec)
7. (Temp_Rem_List’’.Prec = Empty_String)
8. (L’’.Rem = Empty_String)
9. (Temp_Rem_List’’’.Rem = L.Rem)
Appendix C  Communally Bounded List Collection

C.1 Concept

Shared Concept  Communally_Bounded_List_Template(
    type Entry; evaluates Max_Capacity: Integer );
uses String_Theory, Set_Theory, Integer_Ext_Theory;
requires 1 <= Max_Capacity which entails Max_Capacity is_in N;

Shared Variables
    Abstract_Var Total_Size: N;
    constraint Total_Size <= Max_Capacity;
    initialization
        ensures Total_Size = 0;
end;

Type Family List is modeled by Cart_Prod
    Prec, Rem: Str(Entry);
end;
exemplar P;

    initialization
        ensures P.Prec = Empty_String and P.Rem = Empty_String;
    finalization
        affects Total_Size;
        ensures Total_Size = #Total_Size - ( |#P.Prec| + |#P.Rem| );
end;

Operation Advance(updates P: List);
    requires 1 <= |P.Rem|;
    ensures P.Prec = #P.Prec o Prt_Btwn(0, 1, #P.Rem) and
            P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);

Operation Reset(updates P: List);
    ensures P.Prec = Empty_String and
            P.Rem = #P.Prec o #P.Rem;

Operation Length_of_Rem(restores P: List): Integer;
    ensures Length_of_Rem = ( |P.Rem| );

Operation Insert(alters New_Entry: Entry; updates P: List);
    affects Total_Size;
    requires 1 + Total_Size <= Max_Capacity;
    ensures P.Prec = #P.Prec and
            P.Rem = <#New_Entry> o #P.Rem and
            Total_Size = #Total_Size + 1;

Operation Occupied_Size() : Integer;
    ensures Occupied_Size = ( Total_Size );

Operation Remove(replaces Entry_Removed: Entry; updates P: List);
    affects Total_Size;
\textbf{Operation} Advance_to_End(\texttt{updates P: List});
\textbf{ensures} P.Prec = #P.Prec o #P.Rem and P.Rem = Empty_String;

\textbf{Operation} Swap_Remainders(\texttt{updates P, Q: List});
\textbf{ensures} P.Prec = #P.Prec and Q.Prec = #Q.Prec and P.Rem = #Q.Rem and Q.Rem = #P.Rem;

\textbf{Operation} Length_of_Prec(\texttt{restores P: List}): Integer;
\textbf{ensures} Length_of_Prec = |P.Prec|;

\textbf{Operation} Clear(\texttt{clears P: List});
\textbf{affects} Total_Size;
\textbf{ensures} Total_Size = #Total_Size - ( |#P.Prec| + |#P.Rem| );

\textit{end} Communally_Bounded_List_Template;

\begin{description}
\item[C.1.1 Concept VCs]

VCs for Communally_Bounded_List_Template.co generated Tue Apr 03 16:00:18 EDT 2018

\begin{verbatim}
------------ VC(s): -----------------
VC 0_1
Which_Entails Expression Located at Communally_Bounded_List_Template.co(3:10):
    Communally_Bounded_List_Template.co(3:42)

Goal(s):
(\text{Max\_Capacity\ is\ in\ N})

Given(s):
1. (1 <= Max\_Capacity)
\end{verbatim}

\item[C.2 Enhancements]

\textbf{Enhancement} Searching\_Capability\ for Communally_Bounded_List_Template;
\textbf{Operation} Contains(\texttt{restores E: Entry; restores L: List}): Boolean;
\textbf{ensures} Contains = ( Is\_Substring(<E>, L.Prec) or Is\_Substring(<E>, L.Rem) );
\textit{end} Searching\_Capability;

\end{description}
C.3 Enhancement Realizations

C.3.1 Iterative List Search

Revised

Realization Iterative_Searching_Realiz

Operation Are_Equal(restores E, F: Entry) : Boolean;
ensures Are_Equal = ( E = F );
) for Searching_Capability of Communally_Bounded_List_Template;

Operation Is_Present_In_Rem (restores E: Entry; restores L: List): Boolean;
requires L.Prec = Empty_String;
ensures Is_Present_In_Rem = ( Is_Substring(<E>, L.Rem) );

Procedure
Var Next_Entry: Entry;

Is_Present_In_Rem := False();
While ( 1 <= Length_of_Rem(L) )
  maintaining L.Prec o L.Rem = #L.Rem and E = #E and
  Is_Present_In_Rem = Is_Substring(<E>, L.Prec);
  decreasing |L.Rem|;
do
  Remove(Next_Entry, L);
  If ( Are_Equal(Next_Entry, E) ) then
    Is_Present_In_Rem := True();
    Insert(Next_Entry, L);
    Advance_to_End(L);
  else
    Insert(Next_Entry, L);
    Advance(L);
  end;
end;
Reset(L);
end Is_Present_In_Rem;

Procedure Contains (restores E: Entry; restores L: List): Boolean;
Var Temp_Rem_List: List;
Contains := False();

-- Store L.Rem in a temporary list
Swap_Remainders(L, Temp_Rem_List);
Contains := Is_Present_In_Rem(E, Temp_Rem_List);

-- If not found, check L.Prec
If ( not Contains ) then
  Reset(L);
  Contains := Is_Present_In_Rem(E, L);
  Advance_to_End(L);
end;

-- Restore the list
Swap_Remainders(L, Temp_Rem_List);
end Contains;

end Iterative_Searching_Realiz;
C.3.2 Iterative List Search VCs

VCs for Iterative_Searching_Realiz.rb generated Sat Aug 18 11:49:38 EDT 2018

=================================
VC(s):  =================================

VC 0_1
Base Case of the Invariant of While Statement:
IterativeSearchParams.rb(14:24)

Goal(s):

((Empty_String o L.Rem) = L.Rem)

Given(s):

VC 0_2
Base Case of the Invariant of While Statement:
IterativeSearchParams.rb(14:24)

Goal(s):

(E = E)

Given(s):

VC 0_3
Base Case of the Invariant of While Statement:
IterativeSearchParams.rb(14:24)

Goal(s):

(false = Is_Substring(<E>, Empty_String))

Given(s):

VC 0_4
Requires Clause of Remove: IterativeSearchParams.rb(18:12)

Goal(s):

(1 <= |L''.Rem|)

Given(s):

1. (1 <= |L''.Rem|)
2. ((L''.Prec o L''.Rem) = L.Rem)
3. (false = Is_Substring(<E>, L′′′′.Prec))
4. (E′ = E)
5. (L.Prec = Empty_String)

VC 0_5

Requires Clause of Insert: Iterative_Searching_Realiz.rb(21:16)

Goal(s):

((1 + (Total_Size - 1)) <= Max_Capacity)

Given(s):

1. (DeString(Prt_Btwn(0, 1, L′′′′.Rem)) = E)
2. (L′′′′.Prec = L′′′′.Prec)
3. (L′′′′.Rem = Prt_Btwn(1, |L′′′′.Rem|, L′′′′.Rem))
4. (1 <= |L′′′′.Rem|)
5. ((L′′′′.Prec o L′′′′.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L′′′′.Prec))
7. (1 <= Max_Capacity)
8. (Max_Capacity is_in N)
9. (min_int <= Max_Capacity)
10. (Max_Capacity <= max_int)
11. (Total_Size <= Max_Capacity)
12. (L.Prec = Empty_String)
13. (min_int <= 0)
14. (1 <= max_int)

VC 0_6

Inductive Case of Invariant of While Statement: Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(((L′′′.Prec o (DeString(Prt_Btwn(0, 1, L′′′.Rem))) o Prt_Btwn(1, |L′′′.Rem|, L′′′.Rem))) o Empty_String) = (L′′′.Prec o L′′′.Rem))

Given(s):

1. (L′′′.Prec = L′′′.Prec)
2. (DeString(Prt_Btwn(0, 1, L′′′.Rem)) = E)
3. (L′′′.Prec = L′′′.Prec)
4. (1 <= |L′′′.Rem|)
5. (false = Is_Substring(<E>, L′′′.Prec))

VC 0_7

Inductive Case of Invariant of While Statement: Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(E = E)
Given(s):
1. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
2. (L'''''.Prec = L'''''.Prec)
3. (L'''''.Rem = Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))
4. (1 <= |L'''''.Rem|)
5. ((L'''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (L.Prec = Empty_String)

VC 0_8

Inductive Case of Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
(true = Is_Substring(<E>, (L'''.Prec o ((DeString(Prt_Btwn(0, 1, L'''''.Rem))>
o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))))))

Given(s):
1. (L'''.Prec = L'''''.Prec)
2. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
3. (L'''''.Prec = L'''''.Prec)
4. (1 <= |L'''''.Rem|)
5. ((L'''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (L.Prec = Empty_String)

VC 0_9

Termination of While Statement: Iterative_Searching_Realiz.rb(16:12)

Goal(s):
((1 + |Empty_String|) <= |L'''''.Rem|)

Given(s):
1. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) = E)
2. (L'''''.Prec = L'''''.Prec)
3. (L'''''.Rem = Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))
4. (1 <= |L'''''.Rem|)
5. ((L'''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (L.Prec = Empty_String)

VC 1_1

Base Case of the Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
((Empty_String o L.Rem) = L.Rem)

**Given**:

**VC 1_2**
Base Case of the Invariant of While Statement:
Iterative Searching Realiz.rb(14:24)

**Goal**:
(E = E)

**Given**:

**VC 1_3**
Base Case of the Invariant of While Statement:
Iterative Searching Realiz.rb(14:24)

**Goal**:
(false = Is_Substring(<E>, Empty_String))

**Given**:

**VC 1_4**
Requires Clause of Remove: Iterative Searching Realiz.rb(18:12)

**Goal**:
(1 <= |L'''''.Rem|)

**Given**:
1. (1 <= |L'''''.Rem|)
2. ((L'''''.Prec o L'''''.Rem) = L.Rem)
3. (false = Is_Substring(<E>, L'''''.Prec))
4. (E' = E)
5. (L.Prec = Empty_String)

**VC 1_5**
Requires Clause of Insert: Iterative Searching Realiz.rb(24:16)

**Goal**:

((1 + (Total_Size - 1)) <= Max_Capacity)

**Given**:

158
1. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) /= E)
2. (L'''''.Prec = L'''''.Prec)
3. (L'''''.Rem = Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))
4. (1 <= |L'''''.Rem|)
5. ((L'''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (1 <= Max_Capacity)
8. (Max_Capacity is_in N)
9. (min_int <= Max_Capacity)
10. (Max_Capacity <= max_int)
11. (Total_Size <= Max_Capacity)
12. (L.Prec = Empty_String)
13. (min_int <= 0)
14. (l <= max_int)

VC 1_6

Requires Clause of Advance: Iterative_Searching_Realiz.rb(25:16)

Goal(s):
(1 <= |(<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))))

Given(s):
1. (L'''''.Prec = L'''''.Prec)
2. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) /= E)
3. (L'''''.Rem = Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))
4. (1 <= |L'''''.Rem|)
5. ((L'''''.Prec o L'''''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''''.Prec))
7. (L.Prec = Empty_String)

VC 1_7

Inductive Case of Invariant of While Statement:
Iterative_Searching_Realiz.rb(14:24)

Goal(s):
(((L'''''.Prec o Prt_Btwn(0, 1, (<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |(<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem)))) o Prt_Btwn(1, |(<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))|, (<DeString(Prt_Btwn(0, 1, L'''''.Rem))> o Prt_Btwn(1, |L'''''.Rem|, L'''''.Rem))|) = (L'''''.Prec o L'''''.Rem))

Given(s):
1. (L'''''.Prec = L'''''.Prec)
2. (DeString(Prt_Btwn(0, 1, L'''''.Rem)) /= E)
3. (L'''''.Prec = L'''''.Prec)
4. (1 <= |L'''''.Rem|)
5. (false = Is_Substring(<E>, L'''''.Prec))

159
Inductive Case of Invariant of While Statement:
  Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(E = E)

Given(s):

1. (DeString(Prt_Btwn(0, 1, L''''.Rem)) /= E)
2. (L'''.Prec = L'''.Prec)
3. (L'''.Rem = Prt_Btwn(1, |L'''.Rem|, L'''.Rem))
4. (1 <= |L'''.Rem|)
5. ((L'''.Prec o L'''.Rem) = L.Rem)
6. (false = Is_Substring(<E>, L'''.Prec))
7. (L.Prec = Empty_String)

Inductive Case of Invariant of While Statement:
  Iterative_Searching_Realiz.rb(14:24)

Goal(s):

(Is_Substring(<E>, L'''.Prec) = Is_Substring(<E>, (L'''.Prec o Prt_Btwn(0, 1, 
  (DeString(Prt_Btwn(0, 1, L'''.Rem)) o Prt_Btwn(1, |L'''.Rem|, L'''.Rem)))))

Given(s):

1. (L''.Rem = Prt_Btwn(1, |(<DeString(Prt_Btwn(0, 1, L'''.Rem))> o Prt_Btwn(1, |L'''.Rem|, L'''.Rem))|, (<DeString(Prt_Btwn(0, 1, L'''.Rem))> o Prt_Btwn(1, |L'''.Rem|, L'''.Rem)))) o Prt_Btwn(1, |L'''.Rem|, L'''.Rem)))
2. (L'''.Prec = L'''.Prec)
3. (DeString(Prt_Btwn(0, 1, L'''.Rem)) /= E)
4. (L'''.Prec = L'''.Prec)
5. (1 <= |L'''.Rem|)
6. ((L'''.Prec o L'''.Rem) = L.Rem)
7. (L.Prec = Empty_String)

Termination of While Statement: Iterative_Searching_Realiz.rb(16:12)

Goal(s):

((1 + |Prt_Btwn(1, |(<DeString(Prt_Btwn(0, 1, L'''.Rem))> o Prt_Btwn(1, 
  |L'''.Rem|, L'''.Rem))|, (<DeString(Prt_Btwn(0, 1, L'''.Rem))> o Prt_Btwn(1, |L'''.Rem|, L'''.Rem))))|) <= |L'''.Rem|)

Given(s):
1. \((L''\text{.Prec} = (L'''\text{.Prec} \circ \text{Prt\_Btwn}(0, 1, (<\text{DeString}(\text{Prt\_Btwn}(0, 1, L''\text{.Rem})))) \circ \text{Prt\_Btwn}(1, |L''\text{.Rem}|, L''\text{.Rem}))))\)
2. \((L''\text{.Prec} = L'''\text{.Prec})\)
3. \((\text{DeString}(\text{Prt\_Btwn}(0, 1, L''\text{.Rem})) /= E)\)
4. \((L''\text{.Prec} = L''\text{.Prec})\)
5. \((1 <= |L''\text{.Rem}|)\)
6. \(((L''\text{.Prec} \circ L''\text{.Rem}) = L\text{.Rem})\)
7. \((\text{false} = \text{Is\_Substring}(E, L''''\text{.Prec}))\)
8. \((L\text{.Prec} = \text{Empty\_String})\)

VC 2_1

Base Case of the Invariant of While Statement:
Iterative\_Searching\_Realiz.rb(14:24)

Goal(s):
\(((\text{Empty\_String} \circ L\text{.Rem}) = L\text{.Rem})\)

Given(s):

VC 2_2

Base Case of the Invariant of While Statement:
Iterative\_Searching\_Realiz.rb(14:24)

Goal(s):
\((E = E)\)

Given(s):

VC 2_3

Base Case of the Invariant of While Statement:
Iterative\_Searching\_Realiz.rb(14:24)

Goal(s):
\((\text{false} = \text{Is\_Substring}(E, \text{Empty\_String}))\)

Given(s):

VC 2_4

Ensures Clause of Is\_Present\_In\_Rem [After Logical Reduction(s)]:
Iterative\_Searching\_Realiz.rb(6:14)

Goal(s):
\((\text{Is\_Substring}(E, L''\text{.Prec}) = \text{Is\_Substring}(E, (L''\text{.Prec} \circ L''\text{.Rem})))\) or
\((1 <= |L''\text{.Rem}|)\)
Given(s):
1. ((L''.Prec o L''.Rem) = L.Rem)
2. (L.Prec = Empty_String)

VC 2_5
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Iterative Searching Realiz.rb(6:42)

Goal(s):
(E = E) or
(1 <= |L''.Rem|)

Given(s):
1. ((L''.Prec o L''.Rem) = L.Rem)
2. (false = Is_Substring(<E>, L''.Prec))
3. (L.Prec = Empty_String)

VC 2_6
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Iterative Searching Realiz.rb(6:61)

Goal(s):
(Empty_String = Empty_String) or
(1 <= |L''.Rem|)

Given(s):
1. ((L''.Prec o L''.Rem) = L.Rem)
2. (false = Is_Substring(<E>, L''.Prec))
3. (E' = E)

VC 2_7
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Iterative Searching Realiz.rb(6:61)

Goal(s):
(((L''.Prec o L''.Rem) = (L''.Prec o L''.Rem)) or
(1 <= |L''.Rem|))

Given(s):
1. (false = Is_Substring(<E>, L''.Prec))
2. (E' = E)

VC 2_8
Ensures Clause of Is_Present_In_Rem (Condition from Non-Affected Shared Variable) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(6:14)

Goal(s):

(Total_Size = Total_Size) or
(1 <= |L''.Rem|)

Given(s):

1. ((L''.Prec o L''.Rem) = L.Rem)
2. (false = Is_Substring(<E>, L''.Prec))
3. (E' = E)
4. (Total_Size <= Max_Capacity)
5. (L.Prec = Empty_String)
6. (1 <= Max_Capacity)
7. (Max_Capacity is_in N)
8. (min_int <= Max_Capacity)
9. (Max_Capacity <= max_int)
10. (min_int <= 0)
11. (1 <= max_int)

VC 3_1

Requires Clause of Is_Present_In_Rem: Iterative_Searching_Realiz.rb(38:20)

Goal(s):

(Temp_Rem_List''.Prec = Empty_String)

Given(s):

1. (Temp_Rem_List''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List''.Rem = L.Rem)
4. (L'''''.Prec = L.Prec)

VC 3_2

Requires Clause of Is_Present_In_Rem [After Logical Reduction(s)]:

Iterative_Searching_Realiz.rb(43:24)

Goal(s):

(Empty_String = Empty_String) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L'''''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L'''''.Prec = L.Prec)

VC 3_3
Ensures Clause of Contains [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:14)

Goal(s):

(Is_Substring(<E>, (L'''.Prec o L''''.Rem)) = (Is_Substring(<E>, L'.Prec) or Is_Substring(<E>, L'.Rem))) or Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (L'.Prec = (Empty_String o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List''.Rem = Empty_String)
4. (Temp_Rem_List'.Prec = Temp_Rem_List'''.Prec)
5. (L'''.Prec = L.Prec)
6. (Temp_Rem_List'''.Prec = Empty_String)
7. (L'''.Rem = Empty_String)
8. (Temp_Rem_List''.Rem = L.Rem)

VC 3_4

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:33)

Goal(s):

(E = E) or Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L'''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L'''.Prec = L.Prec)

VC 3_5

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:52)

Goal(s):

(L'.Prec = L.Prec) or Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (L'.Prec = (Empty_String o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'.Rem = Empty_String)
4. (Temp_Rem_List''.Prec = Temp_Rem_List'''.Prec)
5. (L'''.Prec = L.Prec)
6. (Temp_Rem_List′′.Prec = Empty_String)
7. (L′′′.Rem = Empty_String)
8. (Temp_Rem_List′′.Rem = L.Rem)

VC 3_6

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:52)

Goal(s):

(L′.Rem = L.Rem) or Is_Substring(<E>, Temp_Rem_List′′.Rem)

Given(s):

1. (L′.Prec = (Empty_String o (L′′′.Prec o L′′′.Rem)))
2. (L′.Rem = Temp_Rem_List′′.Rem)
3. (Temp_Rem_List′.Rem = Empty_String)
4. (Temp_Rem_List′′.Prec = Temp_Rem_List′′.Prec)
5. (L′′′.Prec = L.Prec)
6. (Temp_Rem_List′′.Prec = Empty_String)
7. (L′′′.Rem = Empty_String)
8. (Temp_Rem_List′′.Rem = L.Rem)

VC 3_7

Ensures Clause of Contains (Condition from Non-Affected Shared Variable) [After Logical Reduction(s)]: Iterative_Searching_Realiz.rb(32:14)

Goal(s):

((Total_Size - (|Temp_Rem_List′.Prec| + |Temp_Rem_List′.Rem|)) = Total_Size) or Is_Substring(<E>, Temp_Rem_List′′.Rem)

Given(s):

1. (Temp_Rem_List′.Prec = Temp_Rem_List′′.Prec)
2. (L′.Rem = Temp_Rem_List′′.Rem)
3. (Temp_Rem_List′.Rem = Empty_String)
4. (L′.Prec = (Empty_String o (L′′′.Prec o L′′′.Rem)))
5. (L′′′.Prec = L.Prec)
6. (Temp_Rem_List′′.Prec = Empty_String)
7. (L′′′.Rem = Empty_String)
8. (Temp_Rem_List′′.Rem = L.Rem)
9. (Total_Size <= Max_Capacity)
10. (1 <= Max_Capacity)
11. (Max_Capacity is in N)
12. (min_int <= Max_Capacity)
13. (Max_Capacity <= max_int)
14. (min_int <= 0)
15. (1 <= max_int)

VC 4_1
Requires Clause of Is_Present_In_Rem: Iterative_Searching_Realiz.rb(38:20)

Goal(s):

(Temp_Rem_List'''.Prec = Empty_String)

Given(s):

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L''.Prec = L.Prec)

VC 4_2

Ensures Clause of Contains: Iterative_Searching_Realiz.rb(32:14)

Goal(s):

(Is_Substring(<E>, Temp_Rem_List'''.Rem) = (Is_Substring(<E>, L'.Prec) or Is_Substring(<E>, L'.Rem)))

Given(s):

1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. (L''.Prec = L.Prec)
6. (Temp_Rem_List'''.Prec = Empty_String)
7. (L''.Rem = Empty_String)
8. (Temp_Rem_List'''.Rem = L.Rem)

VC 4_3

Ensures Clause of Contains (Condition from "RESTORES" parameter mode):

Iterative_Searching_Realiz.rb(32:33)

Goal(s):

(E = E)

Given(s):

1. Is_Substring(<E>, Temp_Rem_List'''.Rem)
2. (Temp_Rem_List'''.Prec = Empty_String)
3. (L''.Rem = Empty_String)
4. (Temp_Rem_List'''.Rem = L.Rem)
5. (L''.Prec = L.Prec)

VC 4_4

Ensures Clause of Contains (Condition from "RESTORES" parameter mode):

Iterative_Searching_Realiz.rb(32:52)
Goal(s):
(L'.Prec = L.Prec)

Given(s):
1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Empty_String)
8. (L''.Rem = Empty_String)
9. (Temp_Rem_List''.Rem = L.Rem)

VC 4_5
Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Iterative_Searching_Realiz.rb(32:52)

Goal(s):
(L'.Rem = L.Rem)

Given(s):
1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List’.Prec = Temp_Rem_List’’.Prec)
5. Is_Substring(<E>, Temp_Rem_List’’.Rem)
6. (L’’.Prec = L.Prec)
7. (Temp_Rem_List’’.Prec = Empty_String)
8. (L’’.Rem = Empty_String)
9. (Temp_Rem_List’’.Rem = L.Rem)

VC 4_6
Ensures Clause of Contains (Condition from Non-Affected Shared Variable):
Iterative_Searching_Realiz.rb(32:14)

Goal(s):
((Total_Size - (|Temp_Rem_List’.Prec| + |Temp_Rem_List’’.Rem|)) = Total_Size)

Given(s):
1. (Temp_Rem_List’.Prec = Temp_Rem_List’’.Prec)
2. (L’.Rem = Temp_Rem_List’’.Rem)
3. (Temp_Rem_List’.Rem = L’’.Rem)
4. (L’.Prec = L’’.Prec)
5. Is_Substring(<E>, Temp_Rem_List’’.Rem)
6. (L’’.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Empty_String)
8. (L''.Rem = Empty_String)
9. (Temp_Rem_List''.Rem = L.Rem)
10. (Total_Size <= Max_Capacity)
11. (1 <= Max_Capacity)
12. (Max_Capacity is in N)
13. (min_int <= Max_Capacity)
14. (Max_Capacity <= max_int)
15. (min_int <= 0)
16. (1 <= max_int)

C.3.3 Recursive List Search

Realization Recursive_Searching_Realiz(
  Operation Are_Equal(restores E, F: Entry) : Boolean;
  ensures Are_Equal = ( E = F );
) for Searching_Capability of Communally_Bounded_List_Template;

Operation Is_Present_In_Rem (restores E: Entry; restores L: List): Boolean;
  requires L.Prec = Empty_String;
  ensures Is_Present_In_Rem = ( Is_Substring(<E>, L.Rem) );
Recursive Procedure
  decreasing |L.Rem|;
  -- notice Total_Size <= Max_Capacity;
  Var Next_Entry: Entry;
  Is_Present_In_Rem := False();
  If ( 1 <= Length_of_Rem(L) ) then
    Remove(Next_Entry, L);
    If ( not Is_Present_In_Rem(E, L) ) then
      Is_Present_In_Rem := Are_Equal(E, Next_Entry);
    else
      Is_Present_In_Rem := True();
    end;
    Insert(Next_Entry, L);
  end;
end Is_Present_In_Rem;

Procedure Contains (restores E: Entry; restores L: List): Boolean;
  Var Temp_Rem_List: List;
  Contains := False();
  -- Store L.Rem in a temporary list
  Swap_Remainders(L, Temp_Rem_List);
  Contains := Is_Present_In_Rem(E, Temp_Rem_List);
  -- If not found, check L.Prec
  If ( not Contains ) then
    Reset(L);
    Contains := Is_Present_In_Rem(E, L);
end Contains;
Advance_to_End(L);
end;

-- Restore the list
Swap_Remainders(L, Temp_Rem_List);
end  Contains;
end Recursive_Searching_Realiz;

C.3.4 Recursive List Search VCs

VC 0_1
Requires Clause of Remove: Recursive_Searching_Realiz.rb(17:12)

Goal(s):
(1 <= |L.Rem|)

Given(s):
1. (1 <= |L.Rem|)
2. (L.Prec = Empty_String)

VC 0_2
Termination of Recursive Call: Recursive_Searching_Realiz.rb(19:21)

Goal(s):
((1 + |Prt_Btwn(1, |L.Rem|, L.Rem)|) <= |L.Rem|)

Given(s):
1. (L''.Prec = Empty_String)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
3. (1 <= |L.Rem|)

VC 0_3
Requires Clause of Is_Present_In_Rem: Recursive_Searching_Realiz.rb(19:21)

Goal(s):
(L''.Prec = Empty_String)

Given(s):
1. (L''.Prec = Empty_String)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
3. \( (L'.Rem = \text{Prt}_\text{Btwn}(1, |L.Rem|, L.Rem)) \)
4. \( (1 <= |L.Rem|) \)

**VC 0_4**

Requires Clause of Insert [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(25:12)

**Goal(s):**

\((1 + (\text{Total} - 1)) <= \text{Max} - \text{Capacity}) \text{ or } \text{Is} - \text{Substring(<E>, \text{Prt}_\text{Btwn}(1, |L.Rem|, L.Rem))} \)

**Given(s):**

1. \((L''.Prec = \text{Empty} - \text{String})\)
2. \((\text{Next} - \text{Entry''} = \text{De} - \text{String}(\text{Prt}_\text{Btwn}(0, 1, L.Rem)))\)
3. \((1 <= |L.Rem|)\)
4. \((1 <= \text{Max} - \text{Capacity})\)
5. \((\text{Max} - \text{Capacity} \text{ is in N})\)
6. \((\text{min} - \text{int} <= \text{Max} - \text{Capacity})\)
7. \((\text{Max} - \text{Capacity} <= \text{max} - \text{int})\)
8. \((\text{Total} - \text{Size} <= \text{Max} - \text{Capacity})\)
9. \((\text{min} - \text{int} <= 0)\)
10. \((1 <= \text{max} - \text{int})\)

**VC 0_5**

Ensures Clause of Is_Present_In_Rem [After Logical Reduction(s)]:
Recursive_Searching_Realiz.rb(6:14)

**Goal(s):**

\((E = \text{De} - \text{String}(\text{Prt}_\text{Btwn}(0, 1, L.Rem))) = \text{Is} - \text{Substring(<E>, <\text{De} - \text{String}(\text{Prt}_\text{Btwn}(0, 1, L.Rem)> \text{ o Prt}_\text{Btwn}(1, |L.Rem|, L.Rem)))) \text{ or } \text{Is} - \text{Substring(<E>, \text{Prt}_\text{Btwn}(1, |L.Rem|, L.Rem))} \)

**Given(s):**

1. \((L'.Prec = L''.Prec)\)
2. \((L''.Prec = \text{Empty} - \text{String})\)
3. \((1 <= |L.Rem|)\)

**VC 0_6**

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:42)

**Goal(s):**

\((E = E) \text{ or } \text{Is} - \text{Substring(<E>, \text{Prt}_\text{Btwn}(1, |L.Rem|, L.Rem))} \)

**Given(s):**
1. (L''.Prec = Empty_String)
2. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
3. (1 <= |L.Rem|)

VC 0_7

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

(L’.Prec = Empty_String) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))

Given(s):

1. (L’.Prec = L’’.Prec)
2. (L’.Rem = ((<DeString(Prt_Btwn(0, 1, L.Rem))> o Prt_Btwn(1, |L.Rem|, L.Rem)))
3. (L’’.Prec = Empty_String)
4. (1 <= |L.Rem|)

VC 0_8

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode)
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):

((<DeString(Prt_Btwn(0, 1, L.Rem))> o Prt_Btwn(1, |L.Rem|, L.Rem)) = L.Rem) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))

Given(s):

1. (L’.Prec = L’’.Prec)
2. (L’’.Prec = Empty_String)
3. (1 <= |L.Rem|)

VC 0_9

Ensures Clause of Is_Present_In_Rem (Condition from Non-Affected Shared
Variable) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:14)

Goal(s):

(((Total_Size - 1) + 1) = Total_Size) or
Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))

Given(s):

1. (L’’.Prec = Empty_String)
2. (Next_Entry’’ = DeString(Prt_Btwn(0, 1, L.Rem)))
3. (1 <= |L.Rem|)
4. (Total_Size <= Max_Capacity)
5. (1 <= Max_Capacity)
6. (Max_Capacity is_in N)
7. (\text{min\_int} \leq \text{Max\_Capacity})
8. (\text{Max\_Capacity} \leq \text{max\_int})
9. (\text{min\_int} \leq 0)
10. (1 \leq \text{max\_int})

\text{VC 1\_1}

Requires Clause of Remove: Recursive\_Searching\_Realiz.rb(17:12)

\text{Goal(s):}
{(1 \leq |L.\text{Rem}|)}

\text{Given(s):}
1. (1 \leq |L.\text{Rem}|)
2. (L.\text{Prec} = \text{Empty\_String})

\text{VC 1\_2}

Requires Clause of Insert: Recursive\_Searching\_Realiz.rb(25:12)

\text{Goal(s):}
{(1 + (\text{Total\_Size} - 1)) \leq \text{Max\_Capacity})

\text{Given(s):}
1. \text{Is\_Substring(<E>, Prt\_Btwn(1, |L.\text{Rem}|, L.\text{Rem}))}
2. (L''.'\text{Prec} = \text{Empty\_String})
3. (\text{Next\_Entry''} = \text{DeString(Prt\_Btwn(0, 1, L.\text{Rem}))})
4. (1 \leq |L.\text{Rem}|)
5. (1 \leq \text{Max\_Capacity})
6. (\text{Max\_Capacity} \text{ is\_in N})
7. (\text{min\_int} \leq \text{Max\_Capacity})
8. (\text{Max\_Capacity} \leq \text{max\_int})
9. (\text{Total\_Size} \leq \text{Max\_Capacity})
10. (\text{min\_int} \leq 0)
11. (1 \leq \text{max\_int})

\text{VC 1\_3}

Ensures Clause of Is\_Present\_In\_Rem: Recursive\_Searching\_Realiz.rb(6:14)

\text{Goal(s):}
(true = \text{Is\_Substring(<E>, (<\text{DeString(Prt\_Btwn(0, 1, L.\text{Rem}))} > \circ \text{Prt\_Btwn(1, |L.\text{Rem}|, L.\text{Rem}))})})

\text{Given(s):}
1. (L'.\text{Prec} = L''.\text{Prec})
2. \text{Is\_Substring(<E>, Prt\_Btwn(1, |L.\text{Rem}|, L.\text{Rem}))}
3. (L''.\text{Prec} = \text{Empty\_String})
4. (1 \leq |L.\text{Rem}|)
VC 1_4

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(6:42)

Goal(s):
(E = E)

Given(s):
1. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
2. (L''.Prec = Empty_String)
3. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
4. (1 <= |L.Rem|)

VC 1_5

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(6:61)

Goal(s):
(L'.Prec = Empty_String)

Given(s):
1. (L'.Prec = L''.Prec)
2. (L'.Rem = (DeString(Prt_Btwn(0, 1, L.Rem))) o Prt_Btwn(1, |L.Rem|, L.Rem))
3. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
4. (L''.Prec = Empty_String)
5. (1 <= |L.Rem|)

VC 1_6

Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(6:61)

Goal(s):
((DeString(Prt_Btwn(0, 1, L.Rem))) o Prt_Btwn(1, |L.Rem|, L.Rem)) = L.Rem)

Given(s):
1. (L'.Prec = L''.Prec)
2. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
3. (L''.Prec = Empty_String)
4. (1 <= |L.Rem|)

VC 1_7

Ensures Clause of Is_Present_In_Rem (Condition from Non-Affected Shared Variable):
Recursive_Searching_Realiz.rb(6:14)
Goal(s):
((Total_Size - 1) + 1) = Total_Size

Given(s):
1. Is_Substring(<E>, Prt_Btwn(1, |L.Rem|, L.Rem))
2. (L''.Prec = Empty_String)
3. (Next_Entry'' = DeString(Prt_Btwn(0, 1, L.Rem)))
4. (1 <= |L.Rem|)
5. (Total_Size <= Max_Capacity)
6. (1 <= Max_Capacity)
7. (Max_Capacity is_in N)
8. (min_int <= Max_Capacity)
9. (Max_Capacity <= max_int)
10. (min_int <= 0)
11. (1 <= max_int)

VC 2_1
Ensures Clause of Is_Present_In_Rem [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:14)

Goal(s):
(false = Is_Substring(<E>, L.Rem)) or (1 <= |L.Rem|)

Given(s):
1. (L.Prec = Empty_String)

VC 2_2
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:42)

Goal(s):
(E = E) or (1 <= |L.Rem|)

Given(s):
1. (L.Prec = Empty_String)

VC 2_3
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):
(Empty_String = Empty_String) or (1 <= |L.Rem|)
Given(s):

VC 2_4
Ensures Clause of Is_Present_In_Rem (Condition from "RESTORES" parameter mode) 
[After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:61)

Goal(s):
(L.Rem = L.Rem) or
(1 <= |L.Rem|)

Given(s):
1. (L.Prec = Empty_String)

VC 2_5
Ensures Clause of Is_Present_In_Rem (Condition from Non-Affected Shared 
Variable) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(6:14)

Goal(s):
(Total_Size = Total_Size) or
(1 <= |L.Rem|)

Given(s):
1. (Total_Size <= Max_Capacity)
2. (L.Prec = Empty_String)
3. (1 <= Max_Capacity)
4. (Max_Capacity is in N)
5. (min_int <= Max_Capacity)
6. (Max_Capacity <= max_int)
7. (min_int <= 0)
8. (1 <= max_int)

VC 3_1
Requires Clause of Is_Present_In_Rem: Recursive_Searching_Realiz.rb(36:20)

Goal(s):
(Temp_Rem_List''.Prec = Empty_String)

Given(s):
1. (Temp_Rem_List''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List''.Rem = L.Rem)
4. (L''''Prec = L.Prec)

VC 3_2
Requires Clause of Is_Present_In_Rem [After Logical Reduction(s)]:
  Recursive_Searching_Realiz.rb(41:24)

Goal(s):

(Empty_String = Empty_String) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L''''.Prec = L.Prec)
VC 3_3

Ensures Clause of Contains [After Logical Reduction(s)]:
  Recursive_Searching_Realiz.rb(30:14)

Goal(s):

(Is_Substring(<E>, (L''''.Prec o L''''.Rem)) = (Is_Substring(<E>, L'.Prec) or
  Is_Substring(<E>, L'.Rem))) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (L'.Prec = (Empty_String o (L''''.Prec o L''''.Rem)))
2. (L'.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List'.Rem = Empty_String)
4. (Temp_Rem_List'.Prec = Temp_Rem_List'''.Prec)
5. (L''''.Prec = L.Prec)
6. (Temp_Rem_List'''.Prec = Empty_String)
7. (L'''.Rem = Empty_String)
8. (Temp_Rem_List'''.Rem = L.Rem)
VC 3_4

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After
  Logical Reduction(s)]: Recursive_Searching_Realiz.rb(30:33)

Goal(s):

(E = E) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (Temp_Rem_List'''.Prec = Empty_String)
2. (L''''.Rem = Empty_String)
3. (Temp_Rem_List'''.Rem = L.Rem)
4. (L''''.Prec = L.Prec)
VC 3_5

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(30:52)

Goal(s):

(L'.Prec = L.Prec) or
Is_Substring(<E>, Temp_Rem_List'''.Rem)

Given(s):

1. (L’.Prec = (Empty_String o (L'''.Prec o L'''.Rem)))
2. (L’.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List’’.Rem = Empty_String)
4. (Temp_Rem_List’’.Prec = Temp_Rem_List’’.Prec)
5. (L’''.Prec = L.Prec)
6. (Temp_Rem_List’’’.Prec = Empty_String)
7. (L’’'.Rem = Empty_String)
8. (Temp_Rem_List’’’.Rem = L.Rem)

VC 3_6

Ensures Clause of Contains (Condition from "RESTORES" parameter mode) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(30:52)

Goal(s):

(L’.Rem = L.Rem) or
Is_Substring(<E>, Temp_Rem_List’’’.Rem)

Given(s):

1. (L’.Prec = (Empty_String o (L’’’.Prec o L’’’.Rem)))
2. (L’.Rem = Temp_Rem_List’’’.Rem)
3. (Temp_Rem_List’’.Rem = Empty_String)
4. (Temp_Rem_List’’.Prec = Temp_Rem_List’’.Prec)
5. (L’’’.Prec = L.Prec)
6. (Temp_Rem_List’’’.Prec = Empty_String)
7. (L’’’.Rem = Empty_String)
8. (Temp_Rem_List’’’.Rem = L.Rem)

VC 3_7

Ensures Clause of Contains (Condition from Non-Affected Shared Variable) [After Logical Reduction(s)]: Recursive_Searching_Realiz.rb(30:14)

Goal(s):

((Total_Size - (|Temp_Rem_List’’.Prec| + |Temp_Rem_List’’.Rem|)) = Total_Size) or
Is_Substring(<E>, Temp_Rem_List’’’.Rem)

Given(s):

1. (Temp_Rem_List’’.Prec = Temp_Rem_List’’’.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = Empty_String)
4. (L'.Prec = (Empty_String o (L'''.Prec o L''''.Rem)))
5. (L''''.Prec = L.Prec)
6. (Temp_Rem_List''.Prec = Empty_String)
7. (L'''.Rem = Empty_String)
8. (Temp_Rem_List'''.Rem = L.Rem)
9. (Total_Size <= Max_Capacity)
10. (1 <= Max_Capacity)
11. (Max_Capacity is in N)
12. (min_int <= Max_Capacity)
13. (Max_Capacity <= max_int)
14. (min_int <= 0)
15. (1 <= max_int)

**VC 4_1**

Requires Clause of Is_Present_In_Rem: Recursive_Searching_Realiz.rb(36:20)

**Goal(s):**

(Temp_Rem_List''.Prec = Empty_String)

**Given(s):**

1. (Temp_Rem_List''.Prec = Empty_String)
2. (L''.Rem = Empty_String)
3. (Temp_Rem_List''.Rem = L.Rem)
4. (L''.Prec = L.Prec)

**VC 4_2**

Ensures Clause of Contains: Recursive_Searching_Realiz.rb(30:14)

**Goal(s):**

(Is_Substring(<E>, Temp_Rem_List''.Rem) = (Is_Substring(<E>, L''.Prec) or Is_Substring(<E>, L''.Rem)))

**Given(s):**

1. (L''.Prec = L'''.Prec)
2. (L''.Rem = Temp_Rem_List'''.Rem)
3. (Temp_Rem_List''''.Rem = L'''.Rem)
4. (Temp_Rem_List''.Prec = Temp_Rem_List'''.Prec)
5. Is_Substring(<E>, Temp_Rem_List'''.Rem)
6. (L'''.Prec = L.Prec)
7. (Temp_Rem_List'''.Prec = Empty_String)
8. (L'''.Rem = Empty_String)
9. (Temp_Rem_List'''.Rem = L.Rem)

**VC 4_3**

Ensures Clause of Contains (Condition from "RESTORES" parameter mode): Recursive_Searching_Realiz.rb(30:33)
Goal(s):
(E = E)

Given(s):
1. Is_Substring(<E>, Temp_Rem_List''.Rem)
2. (Temp_Rem_List''.Prec = Empty_String)
3. (L''.Rem = Empty_String)
4. (Temp_Rem_List''.Rem = L.Rem)
5. (L''.Prec = L.Prec)

VC 4_4
Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(30:52)

Goal(s):
(L''.Prec = L.Prec)

Given(s):
1. (L''.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Empty_String)
8. (L''.Rem = Empty_String)
9. (Temp_Rem_List''.Rem = L.Rem)

VC 4_5
Ensures Clause of Contains (Condition from "RESTORES" parameter mode):
Recursive_Searching_Realiz.rb(30:52)

Goal(s):
(L'.Rem = L.Rem)

Given(s):
1. (L'.Prec = L''.Prec)
2. (L'.Rem = Temp_Rem_List''.Rem)
3. (Temp_Rem_List'.Rem = L''.Rem)
4. (Temp_Rem_List'.Prec = Temp_Rem_List''.Prec)
5. Is_Substring(<E>, Temp_Rem_List''.Rem)
6. (L''.Prec = L.Prec)
7. (Temp_Rem_List''.Prec = Empty_String)
8. (L''.Rem = Empty_String)
9. (Temp_Rem_List''.Rem = L.Rem)
VC 4_6

Ensures Clause of Contains (Condition from Non-Affected Shared Variable):
Recursive_Searching_Realiz.rb(30:14)

Goal(s):

\((\text{Total Size} - (|\text{Temp Rem List}'.Prec| + |\text{Temp Rem List}'.Rem|)) = \text{Total Size})\)

Given(s):

1. (\text{Temp Rem List}'.Prec = \text{Temp Rem List}''.Prec)
2. (\text{L}'.Rem = \text{Temp Rem List}''.Rem)
3. (\text{Temp Rem List}'.Rem = \text{L}''.Rem)
4. (\text{L}'.Prec = \text{L}''.Prec)
5. Is_Substring(<E>, \text{Temp Rem List}''.Rem)
6. (\text{L}''.Prec = \text{L}.Prec)
7. (\text{Temp Rem List}''.Prec = \text{Empty String})
8. (\text{L}''.Rem = \text{Empty String})
9. (\text{Temp Rem List}''.Rem = \text{L}.Rem)
10. (\text{Total Size} <= \text{Max Capacity})
11. (1 <= \text{Max Capacity})
12. (\text{Max Capacity} is in \text{N})
13. (\text{min int} <= \text{Max Capacity})
14. (\text{Max Capacity} <= \text{max int})
15. (\text{min int} <= 0)
16. (1 <= \text{max int})

C.4 Facilities

C.4.1 Example Client Program #1

Facility CBLT_Example_1;
uses Integer_Theory;

Facility List_Fac is Communally_Bounded_List_Template(Integer, 2)
realized by CUVRT_Realiz;

Operation No_Bound_Violation();
requires List_Fac::Total_Size = 0;
Procedure
Var L1, L2, L3: List;
Var I, J, K: Integer;

I := 2;
J := 3;
K := 1;

Insert(I, L1);
Insert(J, L2);
Remove(I, L1);
Insert(K, L3);
end No_Bound_Violation;
end CBLT_Example_1;
C.4.2 Example Client Program #1 VCs

VCs for CBLT_Example_1.fa generated Sat Aug 18 12:46:04 EDT 2018

=================================
VC (s): =================================

VC 0_1
Requires Clause for Communally_Bounded_List_Template in Facility Instantiation
  Rule: CBLT_Example_1.fa(4:22)

Goal(s):
(1 <= 2)
Given(s):

VC 1_1
Requires Clause of Insert: CBLT_Example_1.fa(17:2)

Goal(s):
((1 + 0) <= 2)
Given(s):

VC 1_2
Requires Clause of Insert: CBLT_Example_1.fa(18:2)

Goal(s):
((1 + (0 + 1)) <= 2)
Given(s):

VC 1_3
Requires Clause of Remove: CBLT_Example_1.fa(19:2)

Goal(s):
(1 <= |(<2> o Empty_String)|)
Given(s):
1. (L1’’.Prec = Empty_String)

VC 1_4
Requires Clause of Insert: CBLT_Example_1.fa(20:2)

Goal(s):

\(((1 + ((0 + 1) + 1) - 1)) \leq 2\)

Given(s):

VC 1_5

Ensures Clause of No_Bound_Violation (Condition from Non-Affected Shared Variable): CBLT_Example_1.fa(7:11)

Goal(s):

\(((1 + ((0 + 1) + 1) - 1) + 1) - (\|L1'.Prec\| + \|\text{Prt\_Btwn}(1, (<2> \circ \text{Empty\_String})\|) - (\|L2'.Prec\| + \|(<3> \circ \text{Empty\_String})\|) - (\|L3'.Prec\| + \|(<1> \circ \text{Empty\_String})\|)) = 0\)

Given(s):

1. (L3'.Prec = \text{Empty\_String})
2. (L1'.Prec = L1'''.Prec)
3. (I' = \text{DeString} (\text{Prt\_Btwn}(0, 1, (<2> \circ \text{Empty\_String}))))
4. (L2'.Prec = \text{Empty\_String})
5. (L1'''.Prec = \text{Empty\_String})

C.4.3 Example Client Program #2

Facility CBLT_Example_2;
uses Integer_Theory;

Facility List_Fac is Communally_Bounded_List_Template(Integer, 2)
realized by CUVRT_Realiz;

Operation Bound_Violation();
requires List_Fac::Total_Size = 0;
Procedure
Var L1, L2, L3: List;
Var I, J, K: Integer;
I := 2;
J := 3;
K := 1;
Insert(I, L1);
Insert(J, L2);
Insert(K, L3);
end Bound_Violation;

end CBLT_Example_2;
C.4.4 Example Client Program #2 VCs

VCs for CBLT_Example_2.fa generated Sat Aug 18 12:46:15 EDT 2018

=================================
VC(s):  
VC 0_1
Requires Clause for Communally_Bounded_List_Template in Facility Instantiation
Rule: CBLT_Example_2.fa(4:22)

Goal(s):
(1 <= 2)

Given(s):

VC 1_1
Requires Clause of Insert: CBLT_Example_2.fa(17:2)

Goal(s):
((1 + 0) <= 2)

Given(s):

VC 1_2
Requires Clause of Insert: CBLT_Example_2.fa(18:2)

Goal(s):
((1 + (0 + 1)) <= 2)

Given(s):

VC 1_3
Requires Clause of Insert: CBLT_Example_2.fa(19:2)

Goal(s):
((1 + ((0 + 1) + 1)) <= 2)

Given(s):

VC 1_4
Ensures Clause of Bound_Violation (Condition from Non-Affected Shared
Variable): CBLT_Example_2.fa(7:11)

Goal(s):

\[
(((0 + 1) + 1) + 1) - (|L1'.Prec| + |(<2> o Empty_String)|)) - (|L2'.Prec| + |(<3> o Empty_String)|)) - (|L3'.Prec| + |(<1> o Empty_String)|)) = 0
\]

Given(s):

1. (L3'.Prec = Empty_String)
2. (L2'.Prec = Empty_String)
3. (L1'.Prec = Empty_String)
Appendix D  Bounded Stack Collection

D.1 Concept

Concept  Stack_Template(type Entry; evaluates Max_Depth: Integer);
      uses String_Theory, Integer_Ext_Theory;
      requires 1 <= Max_Depth;

Type Family  Stack is modeled by Str(Entry);
      exemplar S;
      constraint |S| <= Max_Depth;
      initialization ensures S = Empty_String;
end;

Operation  Push(alters E: Entry; updates S: Stack);
      requires 1 + |S| <= Max_Depth;
      ensures S = <#E> o #S;

Operation  Pop(replaces R: Entry; updates S: Stack);
      requires 1 <= |S|;
      ensures #S = <R> o S;

Operation  Depth(restores S: Stack): Integer;
      ensures Depth = (|S|);

Operation  Rem_Capacity(restores S: Stack): Integer;
      ensures Rem_Capacity = (Max_Depth - |S|);

Operation  Clear(clears S: Stack);
end  Stack_Template;

D.2 Concept Realizations

D.2.1 Array Realization

Realization  Array_Realiz  for  Stack_Template;

Type  Stack is represented by Record
      Contents: Array 1..Max_Depth of Entry;
      Top: Integer;
end;
convention
      0 <= S.Top <= Max_Depth;
correspondence
      Conc.S = Reverse(Iterated_Concatenation(1, S.Top, lambda(i : Z).(<S.Contents(i)>)));
end;

Procedure  Push(alters E: Entry; updates S: Stack);
      S.Top := S.Top + 1;
E := S.Contents[S.Top];
end Push;

Procedure Pop(replaces R: Entry; updates S: Stack);
  R := S.Contents[S.Top];
  S.Top := S.Top - 1;
end Pop;

Procedure Depth(restores S: Stack): Integer;
  Depth := S.Top;
end Depth;

Procedure Rem_Capacity(restores S: Stack): Integer;
  Rem_Capacity := Max_Depth - S.Top;
end Rem_Capacity;

Procedure Clear(clears S: Stack);
  S.Top := 0;
end Clear;
end Array_Realiz;

D.2.2 Array Realization VCs

VCs for Array_Realiz.rb generated Tue Jul 17 15:09:09 EDT 2018

=================================================================== VC(s): ===============

VC 0_1
Requires Clause for Static_Array_Template in Facility Instantiation Rule:
  Array_Realiz.rb(5:22)

Goal(s):
(1 <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)

VC 1_1
Well Defined Correspondence for Stack: Array_Realiz.rb(4:9)

Goal(s):
(|Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>)))| <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)
2. (0 <= S.Top)
3. \((S.\text{Top} \leq \text{Max\_Depth})\)

**VC 2\_1**

Convention for Stack Generated by Initialization Rule (Concept Type Realization): Array\_Realiz.rb(4:4)

**Goal** (s):

\((0 \leq 0)\)

**Given** (s):

**VC 2\_2**

Convention for Stack Generated by Initialization Rule (Concept Type Realization): Array\_Realiz.rb(4:4)

**Goal** (s):

\((0 \leq \text{Max\_Depth})\)

**Given** (s):

1. \((1 \leq \text{Max\_Depth})\)

**VC 2\_3**

Initialization Ensures Clause of Stack: Stack\_Template.co(8:23)

**Goal** (s):

\((\text{Reverse}(\text{Iterated\_Concatenation}(1, 0, \lambda (i : \mathbb{Z}).(<S.\text{Contents}(\text{Universal}i)>))) = \text{Empty\_String})\)

**Given** (s):

1. \(\text{Array\_Is\_Initial\_in\_Range}(S.\text{Contents}, \text{Lower\_Bound}, \text{Upper\_Bound})\)

**VC 3\_1**

Requires Clause of Sum: Array\_Realiz.rb(16:17)

**Goal** (s):

\((\text{min\_int} \leq (S.\text{Top} + 1))\)

**Given** (s):

1. \((\text{min\_int} \leq 0)\)
2. \((0 \leq S.\text{Top})\)
3. \((S.\text{Top} \leq \text{Max\_Depth})\)
4. \(((1 + |\text{Reverse}(\text{Iterated\_Concatenation}(1, S.\text{Top}, \lambda (i : \mathbb{Z}).(<S.\text{Contents}(\text{Universal}i)>)))|) \leq \text{Max\_Depth})\)
5. (1 <= Max_Depth)

VC 3_2
Requires Clause of Sum: Array_Realiz.rb(16:17)

Goal(s):

((S.Top + 1) <= max_int)

Given(s):

1. (1 <= max_int)
2. (0 <= S.Top)
3. (S.Top <= Max_Depth)
4. ((1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))) <= Max_Depth)
5. (1 <= Max_Depth)

VC 3_3
Requires Clause of Swap_Entry: Array_Realiz.rb(17:8)

Goal(s):

(1 <= (S.Top + 1))

Given(s):

1. (0 <= S.Top)
2. (S.Top <= Max_Depth)
3. ((1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))) <= Max_Depth)
4. (1 <= Max_Depth)

VC 3_4
Requires Clause of Swap_Entry: Array_Realiz.rb(17:8)

Goal(s):

((S.Top + 1) <= Max_Depth)

Given(s):

1. (1 <= Max_Depth)
2. (0 <= S.Top)
3. (S.Top <= Max_Depth)
4. ((1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))) <= Max_Depth)

VC 3_5
Type Convention for Stack Generated by Push: Array_Realiz.rb(15:14)
Goal(s):

(0 <= (S.Top + 1))

Given(s):

1. (E' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z).(E if ([Universal] j = (S.Top + 1))
   S.Contents([Universal] j) otherwise))
3. (0 <= S.Top)
4. (S.Top <= Max_Depth)
5. (1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).
   ([Universal] i)->))|) <= Max_Depth)
6. (1 <= Max_Depth)

VC 3_6

Type Convention for Stack Generated by Push: Array_Realiz.rb(15:14)

Goal(s):

((S.Top + 1) <= Max_Depth)

Given(s):

1. (E' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z).(E if ([Universal] j = (S.Top + 1))
   S.Contents([Universal] j) otherwise))
3. (1 <= Max_Depth)
4. (0 <= S.Top)
5. (S.Top <= Max_Depth)
6. (1 + |Reverse(Iterated_Concatenation(1, S.Top, lambda (i :
   Z).([Universal] i)->))|) <= Max_Depth)

VC 3_7

Ensures Clause of Push: Array_Realiz.rb(15:14)

Goal(s):

(Reverse(Iterated_Concatenation(1, (S.Top + 1), lambda (i :
   Z).([Universal] i)->))) = (<E> o
Reverse(Iterated_Concatenation(1, S.Top, lambda (i :
   Z).([Universal] i)->))))

Given(s):

1. (E' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z).(E if ([Universal] j = (S.Top + 1))
   S.Contents([Universal] j) otherwise))
3. (0 <= S.Top)
4. (S.Top <= Max_Depth)
5. \((1 + \lvert\text{Reverse(Iterated.Concatenation}(1, \text{S.Top}, \lambda (i : Z).(<\text{S.Contents}([\text{Universal}] i)>)))\rvert) \leq \text{Max_Depth})
6. \((1 \leq \text{Max_Depth})

**VC 4_1**

Requires Clause of Swap_Entry: Array_Realiz.rb(21:8)

**Goal**\((s)\):

\((1 \leq \text{S.Top})

**Given**\((s)\):

1. \((0 \leq \text{S.Top})
2. \((\text{S.Top} \leq \text{Max_Depth})
3. \((1 \leq \lvert\text{Reverse(Iterated.Concatenation}(1, \text{S.Top}, \lambda (i : Z).(<\text{S.Contents}([\text{Universal}] i)>)))\rvert)\)
4. \((1 \leq \text{Max_Depth})

**VC 4_2**

Requires Clause of Swap_Entry: Array_Realiz.rb(21:8)

**Goal**\((s)\):

\((\text{S.Top} \leq \text{Max_Depth})

**Given**\((s)\):

1. \((1 \leq \text{Max_Depth})
2. \((0 \leq \text{S.Top})
3. \((\text{S.Top} \leq \text{Max_Depth})
4. \((1 \leq \lvert\text{Reverse(Iterated.Concatenation}(1, \text{S.Top}, \lambda (i : Z).(<\text{S.Contents}([\text{Universal}] i)>)))\rvert)\)

**VC 4_3**

Requires Clause of Difference: Array_Realiz.rb(22:17)

**Goal**\((s)\):

\((\text{min_int} \leq (\text{S.Top} - 1))

**Given**\((s)\):

1. \((R' = \text{S.Contents}(\text{S.Top}))
2. \((\text{S.Contents}' = \lambda (j : Z). (\text{R if } ([\text{Universal}] j = \text{S.Top}) \text{S.Contents}([\text{Universal}] j \text{ otherwise}))
3. \((\text{min_int} \leq 0)\)
4. \((0 \leq \text{S.Top})
5. \((\text{S.Top} \leq \text{Max_Depth})
6. \((1 \leq \lvert\text{Reverse(Iterated.Concatenation}(1, \text{S.Top}, \lambda (i : Z).(<\text{S.Contents}([\text{Universal}] i)>)))\rvert)\)
7. (1 <= Max_Depth)

**VC 4_4**

Requires Clause of Difference: Array_Realiz.rb(22:17)

**Goal**(s):

\[ \text{((S.Top - 1) <= max_int)} \]

**Given**(s):

1. \( (R' = S.Contents(S.Top)) \)
2. \( (S.Contents' = \lambda (j : Z).\{ \\
    R \quad \text{if} \quad ([\text{Universal}] j = S.Top) \\
    S.Contents([\text{Universal}] j) \quad \text{otherwise} \}) \)
3. (1 <= max_int)
4. (0 <= S.Top)
5. (S.Top <= Max_Depth)
6. (1 <= |Reverse(Iterated_Concatenation(1, S.Top, \lambda (i : Z).\langle S.Contents([\text{Universal}] i)\rangle))|)
7. (1 <= Max_Depth)

**VC 4_5**

Type Convention for Stack Generated by Pop: Array_Realiz.rb(20:14)

**Goal**(s):

\[ (0 <= (S.Top - 1)) \]

**Given**(s):

1. \( (R' = S.Contents(S.Top)) \)
2. \( (S.Contents' = \lambda (j : Z).\{ \\
    R \quad \text{if} \quad ([\text{Universal}] j = S.Top) \\
    S.Contents([\text{Universal}] j) \quad \text{otherwise} \}) \)
3. (0 <= S.Top)
4. (S.Top <= Max_Depth)
5. (1 <= |Reverse(Iterated_Concatenation(1, S.Top, \lambda (i : Z).\langle S.Contents([\text{Universal}] i)\rangle))|)
6. (1 <= Max_Depth)

**VC 4_6**

Type Convention for Stack Generated by Pop: Array_Realiz.rb(20:14)

**Goal**(s):

\[ \text{((S.Top - 1) <= Max_Depth)} \]

**Given**(s):

1. \( (R' = S.Contents(S.Top)) \)
2. \( (S.Contents' = \lambda (j : Z).\{ \\
    \}) \)
R if ([Universal] j = S.Top)
S.Contents([Universal] j) otherwise)
3. (1 <= Max_Depth)
4. (0 <= S.Top)
5. (S.Top <= Max_Depth)
6. (1 <= |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(S.Contents([Universal] i))))|)

VC 4_7

Ensures Clause of Pop: Array_Realiz.rb(20:14)

Goal(s):
(Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(S.Contents([Universal] i)))) = (<S.Contents(S.Top) > o Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(S.Contents'([Universal] i))))))

Given(s):
1. (S.Contents' = lambda (j : Z).(R if ([Universal] j = S.Top)
S.Contents([Universal] j) otherwise))
2. (0 <= S.Top)
3. (S.Top <= Max_Depth)
4. (1 <= |Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(S.Contents([Universal] i))))|)
5. (1 <= Max_Depth)

VC 5_1

Type Convention for Stack Generated by Depth: Array_Realiz.rb(25:14)

Goal(s):
(0 <= S.Top)

Given(s):
1. (0 <= S.Top)
2. (S.Top <= Max_Depth)
3. (1 <= Max_Depth)

VC 5_2

Type Convention for Stack Generated by Depth: Array_Realiz.rb(25:14)

Goal(s):
(S.Top <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)
2. \(0 \leq S.\text{Top}\)  
3. \(S.\text{Top} \leq \text{Max\_Depth}\)  

**VC 5_3**

Ensures Clause of Depth: Array\_Realiz.rb(25:14)

**Goal**

\[(S.\text{Top} = |\text{Reverse}(\text{Iterated\_Concatenation}(1, S.\text{Top}, \lambda (i : Z).(<S.\text{Contents}([\text{Universal}\ i]>))))|)\]

**Given**

1. \(0 \leq S.\text{Top}\)  
2. \(S.\text{Top} \leq \text{Max\_Depth}\)  
3. \(1 \leq \text{Max\_Depth}\)

**VC 5_4**

Ensures Clause of Depth (Condition from "RESTORES" parameter mode):  
Array\_Realiz.rb(25:29)

**Goal**

\[(\text{Reverse}(\text{Iterated\_Concatenation}(1, S.\text{Top}, \lambda (i : Z).(<S.\text{Contents}([\text{Universal}\ i]>)))) = \text{Reverse}(\text{Iterated\_Concatenation}(1, S.\text{Top}, \lambda (i : Z).(<S.\text{Contents}([\text{Universal}\ i]>))))\]

**Given**

1. \(0 \leq S.\text{Top}\)  
2. \(S.\text{Top} \leq \text{Max\_Depth}\)  
3. \(1 \leq \text{Max\_Depth}\)

**VC 6_1**

Requires Clause of Difference: Array\_Realiz.rb(30:24)

**Goal**

\[(\text{min\_int} \leq (\text{Max\_Depth} - S.\text{Top}))\]

**Given**

1. \(1 \leq \text{Max\_Depth}\)  
2. \(\text{min\_int} \leq 0\)  
3. \(0 \leq S.\text{Top}\)  
4. \(S.\text{Top} \leq \text{Max\_Depth}\)

**VC 6_2**

Requires Clause of Difference: Array\_Realiz.rb(30:24)

**Goal**

193
\((\text{Max} \_\text{Depth} - \text{S.\text{Top}}) \leq \text{max\_int}\)  

**Given\((s)\):**  
1. \((1 \leq \text{Max} \_\text{Depth})\)  
2. \((1 \leq \text{max\_int})\)  
3. \((0 \leq \text{S.\text{Top}})\)  
4. \((\text{S.\text{Top}} \leq \text{Max} \_\text{Depth})\)  

**VC 6\_3**  
Type Convention for Stack Generated by Rem\_Capacity: Array\_Realiz.rb(29:14)  

**Goal\((s)\):**  
\((0 \leq \text{S.\text{Top}})\)  

**Given\((s)\):**  
1. \((0 \leq \text{S.\text{Top}})\)  
2. \((\text{S.\text{Top}} \leq \text{Max} \_\text{Depth})\)  
3. \((1 \leq \text{Max} \_\text{Depth})\)  

**VC 6\_4**  
Type Convention for Stack Generated by Rem\_Capacity: Array\_Realiz.rb(29:14)  

**Goal\((s)\):**  
\((\text{S.\text{Top}} \leq \text{Max} \_\text{Depth})\)  

**Given\((s)\):**  
1. \((1 \leq \text{Max} \_\text{Depth})\)  
2. \((0 \leq \text{S.\text{Top}})\)  
3. \((\text{S.\text{Top}} \leq \text{Max} \_\text{Depth})\)  

**VC 6\_5**  
Ensures Clause of Rem\_Capacity: Array\_Realiz.rb(29:14)  

**Goal\((s)\):**  
\((\text{Max} \_\text{Depth} - \text{S.\text{Top}}) = (\text{Max} \_\text{Depth} - |\text{Reverse(Iterated\_Concatenation}(1, \text{S.\text{Top}}, \lambda \text{i} : \mathbb{Z}.\langle\text{S.\text{Contents}([\text{Universal}] \text{i})}\rangle)|)|)\)  

**Given\((s)\):**  
1. \((1 \leq \text{Max} \_\text{Depth})\)  
2. \((0 \leq \text{S.\text{Top}})\)  
3. \((\text{S.\text{Top}} \leq \text{Max} \_\text{Depth})\)  

**VC 6\_6**
Ensures Clause of Rem_Capacity (Condition from "RESTORES" parameter mode):
Array_Realiz.rb(29:36)

Goal(s):
(Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))) = Reverse(Iterated_Concatenation(1, S.Top, lambda (i : Z).(<S.Contents([Universal] i)>))))

Given(s):
1. (0 <= S.Top)
2. (S.Top <= Max_Depth)
3. (1 <= Max_Depth)

VC 7_1
Type Convention for Stack Generated by Clear: Array_Realiz.rb(33:14)

Goal(s):
(0 <= 0)

Given(s):

VC 7_2
Type Convention for Stack Generated by Clear: Array_Realiz.rb(33:14)

Goal(s):
(0 <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)
2. (S.Top <= Max_Depth)
3. (0 <= S.Top)

VC 7_3
Ensures Clause of Clear (Condition from "CLEARS" parameter mode):
Array_Realiz.rb(33:27)

Goal(s):
(Reverse(Iterated_Concatenation(1, 0, lambda (i : Z).(<S.Contents([Universal] i)>))) = Empty_String)

Given(s):
1. (0 <= S.Top)
2. (S.Top <= Max_Depth)
3. (1 <= Max_Depth)
D.2.3 Array Realization (With Initialization)

Realization Init_Array_Realiz for Stack_Template;

Type Stack is represented by Record
  Contents: Array 1..Max_Depth of Entry;
  Top: Integer;
end;
convention
  1 <= S.Top <= Max_Depth + 1;
correspondence
  Conc.S = Reverse(Iterated_Concatenation(1, S.Top - 1, lambda(i : Z).(<S.Contents(i)>)));
initialization
  S.Top := 1;
end;
end;

Procedure Push(alters E: Entry; updates S: Stack);
  E :=: S.Contents[S.Top];
  S.Top := S.Top + 1;
end Push;

Procedure Pop(replaces R: Entry; updates S: Stack);
  S.Top := S.Top - 1;
  R :=: S.Contents[S.Top];
end Pop;

Procedure Depth(restores S: Stack): Integer;
  Depth := S.Top - 1;
end Depth;

Procedure Rem_Capacity(restores S: Stack): Integer;
  Var Temp: Integer;
  Temp := Max_Depth + 1;
  Rem_Capacity := Temp - S.Top;
end Rem_Capacity;

Procedure Clear(clears S: Stack);
  S.Top := 1;
end Clear;
end Init_Array_Realiz;

D.2.4 Array Realization (With Initialization) VCs

VCs for Init_Array_Realiz.rb generated Tue Jul 17 15:09:50 EDT 2018

VC 0_1

Requires Clause for Static_Array_Template in Facility Instantiation Rule:
Init_Array_Realiz.rb(6:22)

Goal(s):
(1 <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)

VC 1_1

Well Defined Correspondence for Stack: Init_Array_Realiz.rb(5:9)

Goal(s):
(|Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))))| <= Max_Depth)

Given(s):
1. (1 <= Max_Depth)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))

VC 2_1

Convention for Stack Generated by Initialization Rule (Concept Type Realization): Init_Array_Realiz.rb(14:8)

Goal(s):
(1 <= 1)

Given(s):

VC 2_2

Convention for Stack Generated by Initialization Rule (Concept Type Realization): Init_Array_Realiz.rb(14:8)

Goal(s):
(1 <= (Max_Depth + 1))

Given(s):
1. (1 <= Max_Depth)

VC 2_3

Initialization Ensures Clause of Stack: Stack_Template.co(8:23)

Goal(s):
(Reverse(Iterated_Concatenation(1, (1 - 1), lambda (i : Z).(<S.Contents([Universal] i)>))) = Empty_String)

**Given (s):**

1. Array_Is_Initial_in_Range(S.Contents, Lower_Bound, Upper_Bound)
2. (S.Top = 0)

**VC 3_1**

Requires Clause of Swap_Entry: Init_Array_Realiz.rb(20:8)

**Goal (s):**

(1 <= S.Top)

**Given (s):**

1. (1 <= S.Top)
2. (S.Top <= (Max_Depth + 1))
3. (1 + |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))|) <= Max_Depth
4. (1 <= Max_Depth)

**VC 3_2**

Requires Clause of Swap_Entry: Init_Array_Realiz.rb(20:8)

**Goal (s):**

(S.Top <= Max_Depth)

**Given (s):**

1. (1 <= Max_Depth)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))
4. (1 + |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))|) <= Max_Depth

**VC 3_3**

Requires Clause of Sum: Init_Array_Realiz.rb(21:17)

**Goal (s):**

(min_int <= (S.Top + 1))

**Given (s):**

1. (E' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z).(E if ([Universal] j = S.Top) S.Contents([Universal] j) otherwise))
3. \( \text{min}_\text{int} \leq 0 \)
4. \( 1 \leq \text{S.Top} \)
5. \( \text{S.Top} \leq (\text{Max_Depth} + 1) \)
6. \( (1 + |\text{Reverse(Iterated.Concatenation(1, (\text{S.Top} - 1), lambda (i : Z).(<\text{S.Contents([Universal] i}>)})|)|) \leq \text{Max_Depth} \)
7. \( 1 \leq \text{Max_Depth} \)

**VC 3_4**

Requires Clause of Sum: Init_Array_Realiz.rb(21:17)

**Goal(s):**

\( ((\text{S.Top} + 1) \leq \text{max}\text{int}) \)

**Given(s):**

1. \( (E' = \text{S.Contents}(\text{S.Top})) \)
2. \( (\text{S.Contents}' = \lambda (j : Z).\begin{cases} E & \text{if } ([\text{Universal}] j = \text{S.Top}) \\ S.\text{Contents}([\text{Universal}] j) & \text{otherwise} \end{cases}) \)
3. \( (1 \leq \text{max}\text{int}) \)
4. \( (1 \leq \text{S.Top}) \)
5. \( (\text{S.Top} \leq (\text{Max_Depth} + 1)) \)
6. \( (1 + |\text{Reverse(Iterated.Concatenation(1, (\text{S.Top} - 1), lambda (i : Z).(<\text{S.Contents([Universal] i}>)})|)|) \leq \text{Max_Depth} \)
7. \( (1 \leq \text{Max_Depth}) \)

**VC 3_5**

Type Convention for Stack Generated by Push: Init_Array_Realiz.rb(19:14)

**Goal(s):**

\( (1 \leq (\text{S.Top} + 1)) \)

**Given(s):**

1. \( (E' = \text{S.Contents}(\text{S.Top})) \)
2. \( (\text{S.Contents}' = \lambda (j : Z).\begin{cases} E & \text{if } ([\text{Universal}] j = \text{S.Top}) \\ S.\text{Contents}([\text{Universal}] j) & \text{otherwise} \end{cases}) \)
3. \( (1 \leq \text{S.Top}) \)
4. \( (\text{S.Top} \leq (\text{Max_Depth} + 1)) \)
5. \( (1 + |\text{Reverse(Iterated.Concatenation(1, (\text{S.Top} - 1), lambda (i : Z).(<\text{S.Contents([Universal] i}>)})|)|) \leq \text{Max_Depth} \)
6. \( (1 \leq \text{Max_Depth}) \)

**VC 3_6**

Type Convention for Stack Generated by Push: Init_Array_Realiz.rb(19:14)

**Goal(s):**

\( ((\text{S.Top} + 1) \leq (\text{Max_Depth} + 1)) \)
Given(s):

1. \( (E' = \text{S.Content}(\text{S.Top})) \)
2. \( (\text{S.Content}' = \lambda (j : Z). (\text{E if } ([\text{Universal}] j = \text{S.Top}) \text{S.Content}([\text{Universal}] j) \text{ otherwise})) \)
3. \( (1 \leq \text{Max_Depth}) \)
4. \( (1 \leq \text{S.Top}) \)
5. \( (\text{S.Top} \leq (\text{Max_Depth} + 1)) \)
6. \( ((1 + |\text{Reverse(Iterated_Concatenation(1, (\text{S.Top} - 1), \lambda (i : Z).(<\text{S.Content}([\text{Universal}] i)>))|)}) \leq \text{Max_Depth}) \)

VC 3_7

Ensures Clause of Push: Init_Array_Realiz.rb(19:14)

Goal(s):

\((\text{Reverse(Iterated_Concatenation(1, ((\text{S.Top} + 1) - 1), \lambda (i : Z).(<\text{S.Content}'([\text{Universal}] i)>))|)}) = (\text{E} o \text{Reverse(Iterated_Concatenation(1, (\text{S.Top} - 1), \lambda (i : Z).(<\text{S.Content}([\text{Universal}] i)>))|}))\)

Given(s):

1. \( (E' = \text{S.Content}(\text{S.Top})) \)
2. \( (\text{S.Content}' = \lambda (j : Z). (\text{E if } ([\text{Universal}] j = \text{S.Top}) \text{S.Content}([\text{Universal}] j) \text{ otherwise})) \)
3. \( (1 \leq \text{S.Top}) \)
4. \( (\text{S.Top} \leq (\text{Max_Depth} + 1)) \)
5. \( ((1 + |\text{Reverse(Iterated_Concatenation(1, (\text{S.Top} - 1), \lambda (i : Z).(<\text{S.Content}([\text{Universal}] i)>))|)}) \leq \text{Max_Depth}) \)
6. \( (1 \leq \text{Max_Depth}) \)

VC 4_1

Requires Clause of Difference: Init_Array_Realiz.rb(25:17)

Goal(s):

\((\text{min_int} \leq (\text{S.Top} - 1))\)

Given(s):

1. \( (\text{min_int} \leq 0) \)
2. \( (1 \leq \text{S.Top}) \)
3. \( (\text{S.Top} \leq (\text{Max_Depth} + 1)) \)
4. \( ((1 + |\text{Reverse(Iterated_Concatenation(1, (\text{S.Top} - 1), \lambda (i : Z).(<\text{S.Content}([\text{Universal}] i)>))|)}) \leq \text{Max_Depth}) \)
5. \( (1 \leq \text{Max_Depth}) \)

VC 4_2
Requires Clause of Difference: Init_Array_Realiz.rb(25:17)

**Goal(s):**

((S.Top - 1) <= max_int)

**Given(s):**

1. (1 <= max_int)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))
4. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))))
5. (1 <= Max_Depth)

VC 4_3

Requires Clause of Swap_Entry: Init_Array_Realiz.rb(26:8)

**Goal(s):**

(1 <= (S.Top - 1))

**Given(s):**

1. (1 <= S.Top)
2. (S.Top <= (Max_Depth + 1))
3. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))))
4. (1 <= Max_Depth)

VC 4_4

Requires Clause of Swap_Entry: Init_Array_Realiz.rb(26:8)

**Goal(s):**

((S.Top - 1) <= Max_Depth)

**Given(s):**

1. (1 <= Max_Depth)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))
4. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))))

VC 4_5

Type Convention for Stack Generated by Pop: Init_Array_Realiz.rb(24:14)

**Goal(s):**

(1 <= (S.Top - 1))
Given(s):

1. (R' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z). (R if ([Universal] j = (S.Top - 1))
   S.Contents([Universal] j) otherwise))
3. (1 <= S.Top)
4. (S.Top <= (Max_Depth + 1))
5. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))))
6. (1 <= Max_Depth)

VC 4_6

Type Convention for Stack Generated by Pop: Init_Array_Realiz.rb(24:14)

Goal(s):

((S.Top - 1) <= (Max_Depth + 1))

Given(s):

1. (R' = S.Contents(S.Top))
2. (S.Contents' = lambda (j : Z). (R if ([Universal] j = (S.Top - 1))
   S.Contents([Universal] j) otherwise))
3. (1 <= Max_Depth)
4. (1 <= S.Top)
5. (S.Top <= (Max_Depth + 1))
6. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))))))

VC 4_7

Ensures Clause of Pop: Init_Array_Realiz.rb(24:14)

Goal(s):

(Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>)))) = (<S.Contents(S.Top) > o
 Reverse(Iterated_Concatenation(1, ((S.Top - 1) - 1), lambda (i : Z).(<S.Contents'([Universal] i)>))))))

Given(s):

1. (S.Contents' = lambda (j : Z). (R if ([Universal] j = (S.Top - 1))
   S.Contents([Universal] j) otherwise))
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))
4. (1 <= |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))))))
5. (1 <= Max_Depth)

VC 5_1

202
Requires Clause of Difference: Init_Array_Realiz.rb(30:17)

Goal(s):

\[(\text{min}_\text{int} \leq (S.\text{Top} - 1))\]

Given(s):

1. \[(\text{min}_\text{int} \leq 0)\]
2. \[(1 \leq S.\text{Top})\]
3. \[(S.\text{Top} \leq (\text{Max}_\text{Depth} + 1))\]
4. \[(1 \leq \text{Max}_\text{Depth})\]

VC 5_2

Requires Clause of Difference: Init_Array_Realiz.rb(30:17)

Goal(s):

\[((S.\text{Top} - 1) \leq \text{max}_\text{int})\]

Given(s):

1. \[(1 \leq \text{max}_\text{int})\]
2. \[(1 \leq S.\text{Top})\]
3. \[(S.\text{Top} \leq (\text{Max}_\text{Depth} + 1))\]
4. \[(1 \leq \text{Max}_\text{Depth})\]

VC 5_3

Type Convention for Stack Generated by Depth: Init_Array_Realiz.rb(29:14)

Goal(s):

\[(1 \leq S.\text{Top})\]

Given(s):

1. \[(1 \leq S.\text{Top})\]
2. \[(S.\text{Top} \leq (\text{Max}_\text{Depth} + 1))\]
3. \[(1 \leq \text{Max}_\text{Depth})\]

VC 5_4

Type Convention for Stack Generated by Depth: Init_Array_Realiz.rb(29:14)

Goal(s):

\[(S.\text{Top} \leq (\text{Max}_\text{Depth} + 1))\]

Given(s):

1. \[(1 \leq \text{Max}_\text{Depth})\]
2. \[(1 \leq S.\text{Top})\]
3. \((S.\text{Top} \leq (\text{Max}\_\text{Depth} + 1))\)

**VC 5_5**

Ensures Clause of Depth: Init\_Array\_Realiz.rb(29:14)

**Goal\(\langle s \rangle\):**

\[
\left( (S.\text{Top} - 1) = \text{Reverse}(\text{Iterated\_Concatenation}(1, (S.\text{Top} - 1), \lambda (i : Z).(<S.\text{Contents}([\text{Universal}] i)>))) \right)
\]

**Given\(\langle s \rangle\):**

1. \((1 \leq S.\text{Top})\)
2. \((S.\text{Top} \leq (\text{Max}\_\text{Depth} + 1))\)
3. \((1 \leq \text{Max}\_\text{Depth})\)

**VC 5_6**

Ensures Clause of Depth (Condition from "RESTORES" parameter mode):
Init\_Array\_Realiz.rb(29:29)

**Goal\(\langle s \rangle\):**

\[
(\text{Reverse}(\text{Iterated\_Concatenation}(1, (S.\text{Top} - 1), \lambda (i : Z).(<S.\text{Contents}([\text{Universal}] i)>))) = \text{Reverse}(\text{Iterated\_Concatenation}(1, (S.\text{Top} - 1), \lambda (i : Z).(<S.\text{Contents}([\text{Universal}] i)>))))
\]

**Given\(\langle s \rangle\):**

1. \((1 \leq S.\text{Top})\)
2. \((S.\text{Top} \leq (\text{Max}\_\text{Depth} + 1))\)
3. \((1 \leq \text{Max}\_\text{Depth})\)

**VC 6_1**

Requires Clause of Sum: Init\_Array\_Realiz.rb(35:16)

**Goal\(\langle s \rangle\):**

\[
(\text{min}\_\text{int} \leq (\text{Max}\_\text{Depth} + 1))
\]

**Given\(\langle s \rangle\):**

1. \((1 \leq \text{Max}\_\text{Depth})\)
2. \((\text{min}\_\text{int} \leq 0)\)
3. \((S.\text{Top} \leq (\text{Max}\_\text{Depth} + 1))\)
4. \((1 \leq S.\text{Top})\)

**VC 6_2**

Requires Clause of Sum: Init\_Array\_Realiz.rb(35:16)

**Goal\(\langle s \rangle\):**
\((\text{Max} \_\text{Depth} + 1) \leq \text{max} \_\text{int}\)

**Given**(s):  
1. \(1 \leq \text{Max} \_\text{Depth}\)  
2. \(1 \leq \text{max} \_\text{int}\)  
3. \((\text{S}.\text{Top} \leq (\text{Max} \_\text{Depth} + 1))\)  
4. \(1 \leq \text{S}.\text{Top}\)

**VC 6_3**

Requires Clause of Difference: Init\_Array\_Realiz.rb(36:24)

**Goal**(s):  
\((\text{min} \_\text{int} \leq ((\text{Max} \_\text{Depth} + 1) - \text{S}.\text{Top}))\)

**Given**(s):  
1. \(1 \leq \text{Max} \_\text{Depth}\)  
2. \(\text{min} \_\text{int} \leq 0\)  
3. \(1 \leq \text{S}.\text{Top}\)  
4. \((\text{S}.\text{Top} \leq (\text{Max} \_\text{Depth} + 1))\)

**VC 6_4**

Requires Clause of Difference: Init\_Array\_Realiz.rb(36:24)

**Goal**(s):  
\(((\text{Max} \_\text{Depth} + 1) - \text{S}.\text{Top}) \leq \text{max} \_\text{int}\)

**Given**(s):  
1. \(1 \leq \text{Max} \_\text{Depth}\)  
2. \(1 \leq \text{max} \_\text{int}\)  
3. \(1 \leq \text{S}.\text{Top}\)  
4. \((\text{S}.\text{Top} \leq (\text{Max} \_\text{Depth} + 1))\)

**VC 6_5**

Type Convention for Stack Generated by Rem\_Capacity: Init\_Array\_Realiz.rb(33:14)

**Goal**(s):  
\((1 \leq \text{S}.\text{Top})\)

**Given**(s):  
1. \(1 \leq \text{S}.\text{Top}\)  
2. \((\text{S}.\text{Top} \leq (\text{Max} \_\text{Depth} + 1))\)  
3. \(1 \leq \text{Max} \_\text{Depth}\)

**VC 6_6**
Type Convention for Stack Generated by Rem_Capacity: Init_Array_Realiz.rb(33:14)

Goal(s):

(S.Top <= (Max_Depth + 1))

Given(s):

1. (1 <= Max_Depth)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))

VC 6_7

Ensures Clause of Rem_Capacity: Init_Array_Realiz.rb(33:14)

Goal(s):

(((Max_Depth + 1) - S.Top) = (Max_Depth - |Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))))|)

Given(s):

1. (1 <= Max_Depth)
2. (1 <= S.Top)
3. (S.Top <= (Max_Depth + 1))

VC 6_8

Ensures Clause of Rem_Capacity (Condition from "RESTORES" parameter mode):
Init_Array_Realiz.rb(33:36)

Goal(s):

(Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))) = Reverse(Iterated_Concatenation(1, (S.Top - 1), lambda (i : Z).(<S.Contents([Universal] i)>))))

Given(s):

1. (1 <= S.Top)
2. (S.Top <= (Max_Depth + 1))
3. (1 <= Max_Depth)

VC 7_1

Type Convention for Stack Generated by Clear: Init_Array_Realiz.rb(39:14)

Goal(s):

(1 <= 1)

Given(s):
**VC 7_2**

Type Convention for Stack Generated by Clear: Init_Array_Realiz.rb(39:14)

**Goal(s):**

\[(1 \leq (Max\_Depth + 1))\]

**Given(s):**

1. \[(1 \leq Max\_Depth)\]
2. \[(S.Top \leq (Max\_Depth + 1))\]
3. \[(1 \leq S.Top)\]

**VC 7_3**

Ensures Clause of Clear (Condition from "CLEARS" parameter mode):

Init_Array_Realiz.rb(39:27)

**Goal(s):**

\[
(\text{Reverse(Iterated\_Concatenation}(1, (1 - 1), \lambda (i : 2).(<S.Contents([Universal\ i]>)))) = \text{Empty\_String})
\]

**Given(s):**

1. \[(1 \leq S.Top)\]
2. \[(S.Top \leq (Max\_Depth + 1))\]
3. \[(1 \leq Max\_Depth)\]
Appendix E  Globally Bounded Stack Collection

E.1 Concept

Concept Globally_Bounded_Stack_Template(type Entry);
   uses String_Theory;

   Type Family Stack is modeled by Str(Entry);
      exemplar S;
      initialization ensures S = Empty_String;
   end;

   Operation Push(alters E: Entry; updates S: Stack);
      ensures S = <#E> o #S;
   end Push;

   Operation Pop(replaces R: Entry; updates S: Stack);
      requires not(S = Empty_String);
      ensures #S = <R> o S;
   end Pop;

   Operation Is_Empty(restores S : Stack) : Boolean;
      ensures Is_Empty = (S = Empty_String);
   end Is_Empty;

   Operation Clear(clears S: Stack);
end Globally_Bounded_Stack_Template;

E.2 Concept Realizations

E.2.1 Globally Bounded List Realization

Realization GBList_Based_Realiz for Globally_Bounded_Stack_Template;
   uses String_Theory;

   Facility GB_List_Fac is Globally_Bounded_List_Template(Entry)
      externally realized by UVRT_Realiz;

   Type Stack = GB_List_Fac::List;
      convention
         S.Prec = Empty_String;
      correspondence
         Conc.S = S.Rem;
   end;

   Procedure Push(alters E: Entry; updates S: Stack);
      Insert(E, S);
   end Push;

   Procedure Pop(replaces R: Entry; updates S: Stack);
      Remove(R, S);
   end Pop;
Procedure Is_Empty(restores S : Stack) : Boolean;
    Is_Empty := Is_Rem_Empty(S);
end Is_Empty;

Procedure Clear(clears S: Stack);
    Clear(S);
end Clear;

end GBList_Based_Realiz;

E.2.2 Globally Bounded List Realization VCs

VCs for GBList_Based_Realiz.rb generated Sat Aug 18 13:08:19 EDT 2018

VC (s): Convention for Stack Generated by Initialization Rule (Concept Type
Realization): GBList_Based_Realiz.rb(7:1)

Goal(s):
(Empty_String = Empty_String)

VC 0_1

VC (s): Initialization Ensures Clause of Stack: Globally_Bounded_Stack_Template.co(7:17)

Goal(s):
(Empty_String = Empty_String)

VC 0_2

VC (s): Type Convention for Stack Generated by Push: GBList_Based_Realiz.rb(14:11)

Goal(s):
(S’.Prec = Empty_String)

VC 1_1

VC (s): Ensures Clause of Push: GBList_Based_Realiz.rb(14:11)

VC 1_2
Goal(s):

\((\langle E \rangle \circ S.'\text{Rem}) = (\langle E \rangle \circ S.'\text{Rem})\) 

Given(s):

1. (S.'Prec = Empty_String)

VC 2_1

Requires Clause of Remove [After Logical Reduction(s)]:

GBList_Based_Realiz.rb(19:2)

Goal(s):

(S'Rem = Empty_String)

Given(s):

1. (S'Rem = Empty_String)
2. (S.Prec = Empty_String)

VC 2_2

Type Convention for Stack Generated by Pop [After Logical Reduction(s)]:

GBList_Based_Realiz.rb(18:11)

Goal(s):

(S'.Prec = Empty_String) or 
(S'Rem = Empty_String)

Given(s):

1. (S'.Prec = Empty_String)
2. (R' = DeString(Prt_Btwn(0, 1, S'Rem)))
3. (S'.Rem = Prt_Btwn(1, |S'Rem|, S'Rem))

VC 2_3

Ensures Clause of Pop [After Logical Reduction(s)]:

GBList_Based_Realiz.rb(18:11)

Goal(s):

(S'Rem = (\langle DeString(Prt_Btwn(0, 1, S'Rem)) \rangle \circ Prt_Btwn(1, |S'Rem|, S'Rem))) or 
(S'Rem = Empty_String)

Given(s):

1. (S'.Prec = Empty_String)

VC 3_1

Type Convention for Stack Generated by Is_Empty: GBList_Based_Realiz.rb(22:11)

210
Goal(s):
(Empty_String = Empty_String)

Given(s):

VC 3_2
Ensures Clause of Is_Empty: GBList_Based_Realiz.rb(22:11)

Goal(s):
((S.Rem = Empty_String) = (S.Rem = Empty_String))

Given(s):
1. (S.Prec = Empty_String)

VC 3_3
Ensures Clause of Is_Empty (Condition from "RESTORES" parameter mode):
   GBList_Based_Realiz.rb(22:29)

Goal(s):
(S.Rem = S.Rem)

Given(s):
1. (S.Prec = Empty_String)

VC 4_1
Type Convention for Stack Generated by Clear: GBList_Based_Realiz.rb(26:11)

Goal(s):
(Empty_String = Empty_String)

Given(s):

VC 4_2
Ensures Clause of Clear (Condition from "CLEARS" parameter mode):
   GBList_Based_Realiz.rb(26:24)

Goal(s):
(Empty_String = Empty_String)

Given(s):
Appendix F  Communally Bounded Stack Collection

F.1  Concept

**Shared Concept**  Communally_Bounded_Stack_Template{
  type Entry;  evaluates Max_Capacity: Integer );
  uses String_Theory, Set_Theory, Integer_Ext_Theory;
  requires 1 <= Max_Capacity which entails Max_Capacity is in N;

  Shared Variables
  Abstract_Var Total_Size: N;

  constraint Total_Size <= Max_Capacity;
  initialization
    ensures Total_Size = 0;
  end;

  Type Family Stack is modeled by Str(Entry);
  exemplar S;
  initialization
    ensures S = Empty_String;
  finalization
    affects Total_Size;
    ensures Total_Size = #Total_Size - |#S|;
  end;

  Operation Push(alters E: Entry; updates S: Stack);
    affects Total_Size;
    requires 1 + Total_Size <= Max_Capacity;
    ensures S = <#E> o #S and
      Total_Size = #Total_Size + 1;

  Operation Pop(replaces R: Entry; updates S: Stack);
    affects Total_Size;
    requires 1 <= |S|;
    ensures #S = <R> o S and
      Total_Size = #Total_Size - 1;

  Operation Depth(restores S : Stack) : Integer;
    ensures Depth = (|S|);

  Operation Occupied_Size(): Integer;
    ensures Occupied_Size = ( Total_Size );

  Operation Clear(clears S: Stack);
    affects Total_Size;
    ensures Total_Size = #Total_Size - |#S|;

end Communally_Bounded_Stack_Template;
F.1.1 Concept VCs

VCs for Communally_Bounded_Stack_Template.co generated Sat Aug 18 16:37:55 EDT 2018

VC 0_1

Which_Entails Expression Located at Communally_Bounded_Stack_Template.co(3:10):
Communally_Bounded_Stack_Template.co(3:42)

Goal(s):
(Max_Capacity is_in N)

Given(s):
1. (1 <= Max_Capacity)

F.2 Concept Realizations

F.2.1 Communally Bounded List Realization

Realization CBList_Based_Realiz for Communally_Bounded_Stack_Template;
uses String_Theory;

Facility CB_List_Fac is Communally_Bounded_List_Template(
    Entry, Max_Capacity)
externally realized by UVRT_List_Realiz;

Shared Variables
    correspondence
        involves CB_List_Fac::Total_Size;
        Conc.Total_Size = CB_List_Fac::Total_Size;
end;

Type Stack = CB_List_Fac::List;
    convention
        S.Prec = Empty_String;
    correspondence
        Conc.S = S.Rem;
    finalization
        affects CB_List_Fac::Total_Size;
end;

Procedure Push(alters E: Entry; updates S: Stack);
affects CB_List_Fac::Total_Size;

    Insert(E, S);
end Push;
Procedure Pop(replaces R: Entry; updates S: Stack); 
  affects CB_List_Fac::Total_Size;
  Remove(R, S);
end Pop;

Procedure Depth(restores S : Stack) : Integer;
  Depth := Length_of_Rem(S);
end Depth;

Procedure Occupied_Size(): Integer;
  Occupied_Size := CB_List_Fac::Occupied_Size();
end Occupied_Size;

Procedure Clear(clears S: Stack);
  affects CB_List_Fac::Total_Size;
  CB_List_Fac::Clear(S);
end Clear;
end CBLList_Based_Realiz;

F.2.2 Communally Bounded List Realization VCs

Note: Some of the VCs are not provable, because -constraints flag has been used in generating reduced givens (e.g., VC 1_1).

VCs for CBLList_Based_Realiz.rb generated Sat Aug 18 16:42:02 EDT 2018

=================================
VC(s): ==================================

VC 0_1
Requires Clause for Communally_Bounded_List_Template in Facility Instantiation
Rule: CBLList_Based_Realiz.rb(4:25)

Goal(s):
(1 <= Max_Capacity)

Given(s):
1. (1 <= Max_Capacity)
2. (Max_Capacity is_in N)

VC 1_1
Well Defined Correspondence for Shared Variables: CBLList_Based_Realiz.rb(7:1)

Goal(s):
(CB_List_Fac::Total_Size <= Max_Capacity)
Given(s):

1. (1 <= Max_Capacity)
2. (Max_Capacity is in N)

VC 2_1

Initialization Ensures Clause of Shared Variables:
Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

(0 = 0)

Given(s):

VC 2_2

Ensures Clause of Shared Variables (Condition from Non-Affected Shared Variable): Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

(0 = 0)

Given(s):

VC 3_1

Convention for Stack Generated by Initialization Rule (Concept Type Realization): CBList_Based_Realiz.rb(13:1)

Goal(s):

(Empty_String = Empty_String)

Given(s):

VC 3_2

Initialization Ensures Clause of Stack:
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):

(Empty_String = Empty_String)

Given(s):

VC 3_3
Ensures Clause of Stack (Condition from Non-Affected Shared Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):

(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 3.4

Ensures Clause of Stack (Condition from Non-Affected Shared Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):

(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 4.1

Finalization Ensures Clause of Stack: Communally_Bounded_Stack_Template.co(19:3)

Goal(s):

((CB_List_Fac::Total_Size - (|Empty_String| + |S.Rem|)) =
(CB_List_Fac::Total_Size - |S.Rem|))

Given(s):

VC 5.1

Requires Clause of Insert: CBLList_Based_Realiz.rb(26:2)

Goal(s):

((1 + CB_List_Fac::Total_Size) <= Max_Capacity)

Given(s):

1. (1 <= Max_Capacity)
2. (Max_Capacity is_in N)
3. ((1 + CB_List_Fac::Total_Size) <= Max_Capacity)

VC 5.2

Type Convention for Stack Generated by Push: CBLList_Based_Realiz.rb(23:11)

Goal(s):

(S'.Prec = Empty_String)
Given(s):
1. (S’.Prec = Empty_String)
2. (S’.Rem = (<E> o S.Rem))

VC 5_3
Ensures Clause of Push: CBLlist_Based_Realiz.rb(23:11)

Goal(s):
((<E> o S.Rem) = (<E> o S.Rem))

Given(s):
1. (S’.Prec = Empty_String)

VC 5_4
Ensures Clause of Push: CBLlist_Based_Realiz.rb(23:11)

Goal(s):
((CB_List_Fac::Total_Size + 1) = (CB_List_Fac::Total_Size + 1))

Given(s):
1. ((1 + CB_List_Fac::Total_Size) <= Max_Capacity)
2. (1 <= Max_Capacity)
3. (Max_Capacity is_in N)

VC 6_1
Requires Clause of Remove: CBLlist_Based_Realiz.rb(32:2)

Goal(s):
(1 <= |S.Rem|)

Given(s):
1. (S.Prec = Empty_String)
2. (1 <= |S.Rem|)

VC 6_2
Type Convention for Stack Generated by Pop: CBLlist_Based_Realiz.rb(29:11)

Goal(s):
(S’.Prec = Empty_String)

Given(s):
1. \( (S'.Prec = \text{Empty\_String}) \)
2. \( (R' = \text{DeString(Prt\_Btwn(0, 1, S.Rem)}) \)
3. \( (S'.Rem = \text{Prt\_Btwn(1, \mid S.Rem\mid, S.Rem)} \)
4. \( (1 <= \mid S.Rem\mid) \)

**VC 6_3**

Ensures Clause of Pop: CBList\_Based\_Realiz.rb(29:11)

**Goal\( (s) : \)**

\( (S.Rem = (<\text{DeString(Prt\_Btwn(0, 1, S.Rem)}) \circ \text{Prt\_Btwn(1, \mid S.Rem\mid, S.Rem)}) \)

**Given\( (s) : \)**

1. \( (S'.Prec = \text{Empty\_String}) \)
2. \( (1 <= \mid S.Rem\mid) \)

**VC 6_4**

Ensures Clause of Pop: CBList\_Based\_Realiz.rb(29:11)

**Goal\( (s) : \)**

\( ((\text{CB\_List\_Fac::Total\_Size - 1}) = (\text{CB\_List\_Fac::Total\_Size - 1})) \)

**Given\( (s) : \)**

**VC 7_1**

Type Convention for Stack Generated by Depth: CBList\_Based\_Realiz.rb(35:11)

**Goal\( (s) : \)**

\( (\text{Empty\_String} = \text{Empty\_String}) \)

**Given\( (s) : \)**

**VC 7_2**

Ensures Clause of Depth: CBList\_Based\_Realiz.rb(35:11)

**Goal\( (s) : \)**

\( (\mid S.Rem\mid = \mid S.Rem\mid) \)

**Given\( (s) : \)**

1. \( (S.Prec = \text{Empty\_String}) \)

**VC 7_3**
Ensures Clause of Depth (Condition from "RESTORES" parameter mode):
  CBList_Based_Realiz.rb(35:26)

Goal(s):
(S.Rem = S.Rem)

Given(s):
1. (S.Prec = Empty_String)

VC 7_4

Ensures Clause of Depth (Condition from Non-Affected Shared Variable):
  CBList_Based_Realiz.rb(35:11)

Goal(s):
(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 7_5

Ensures Clause of Depth (Condition from Non-Affected Shared Variable):
  CBList_Based_Realiz.rb(35:11)

Goal(s):
(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 8_1

Ensures Clause of Occupied_Size: CBList_Based_Realiz.rb(39:11)

Goal(s):
(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 8_2

Ensures Clause of Occupied_Size (Condition from Non-Affected Shared Variable):
  CBList_Based_Realiz.rb(39:11)

Goal(s):
(CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)
Given(s):

VC 8_3
Ensures Clause of Occupied_Size (Condition from Non-Affected Shared Variable):
   CBList_Based_Realiz.rb(39:11)

Goal(s):
   (CB_List_Fac::Total_Size = CB_List_Fac::Total_Size)

Given(s):

VC 9_1
Type Convention for Stack Generated by Clear: CBList_Based_Realiz.rb(43:11)

Goal(s):
   (Empty_String = Empty_String)

Given(s):

VC 9_2
Ensures Clause of Clear: CBList_Based_Realiz.rb(43:11)

Goal(s):
   ((CB_List_Fac::Total_Size - (|Empty_String| + |S.Rem|)) =
   (CB_List_Fac::Total_Size - |S.Rem|))

Given(s):

VC 9_3
Ensures Clause of Clear (Condition from "CLEARS" parameter mode):
   CBList_Based_Realiz.rb(43:24)

Goal(s):
   (Empty_String = Empty_String)

Given(s):
F.2.3 Communal Bounded UVRT Realization

**Realization** UVRT\textunderscore Stack\_Realiz for Communally\_Bounded\_Stack\_Template;

uses Set\_Theory, Set\_App\_Op\_Ext;

**Facility** UVRT\_Fac is Communual\_UVR\_Template(Entry, Max\_Capacity)
externally realized by Communual\_Array\_Realiz;

**Shared Variables**
correspondence involves
UVRT\_Fac::Accessible\_Loc;
Conc.Total\_Size = ||UVRT\_Fac::Accessible\_Loc|| - 1;
end;

-- Ref\{}times\}(start)
-- Note: Future syntax is expected to include a suitable notation and
-- this definition will be elided.

**Definition** Iterated\_Apply(
f : Location -> Location, start : Location, times : Z) : Location;

**Type** Stack is represented by Record

Top\_Pos: UVRT\_Fac::Pos;
Depth: Integer;
end;

convention 0 <= S.Depth and Iterated\_Apply(UVRT\_Fac::Ref,
S.Top\_Pos, S.Depth) = UVRT\_Fac::Void;

independent correspondence involves UVRT\_Fac::Ref, UVRT\_Fac::Content;
Conc.S = Iterated\_Concatenation(1,
S.Depth,
lambda(i : Z).(<UVRT\_Fac::Content(
Iterated\_Apply(
UVRT\_Fac::Ref, S.Top\_Pos, i-1))>));

finalization affects UVRT\_Fac::Accessible\_Loc, UVRT\_Fac::Cast\_Accessible\_Loc,
UVRT\_Fac::Ref, UVRT\_Fac::Content;
end;

**Procedure** Push(alters E: Entry; updates S: Stack);
affects UVRT\_Fac::Accessible\_Loc, UVRT\_Fac::Cast\_Accessible\_Loc,
UVRT\_Fac::Ref, UVRT\_Fac::Content;

Var New\_Pos: Pos;

Give\_New\_Loc(New\_Pos);
Swap\_Content\_of(New\_Pos, E);
Redirect\_Ref\_at(New\_Pos, S.Top\_Pos);
S_Top\_Pos :=: New\_Pos;
S_Depth := S_Depth + 1;
end Push;
Procedure Pop(replaces R: Entry; updates S: Stack);
affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
        UVRT_Fac::Ref, UVRT_Fac::Content;
        Swap_Content_of(S.Top_Pos, R);
        Follow_Ref(S.Top_Pos); -- Let UVRT take care of finalization
        S.Depth := S.Depth - 1;
end Pop;

Procedure Depth(restores S : Stack) : Integer;
    Depth := S.Depth;
end Depth;

Procedure Occupied_Size(): Integer;
    Occupied_Size := UVRT_Fac::Occupied_Size() - 1;
end Occupied_Size;

Procedure Clear(clears S: Stack);
affects UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc,
        UVRT_Fac::Ref, UVRT_Fac::Content;
        Set_to_Void(S.Top_Pos); -- Let UVRT take care of finalization
        S.Depth := 0;
end Clear;

end UVRT_Stack_Realiz;

F.2.4 Communally Bounded UVRT Realization VCs

Note 1: Some of the VCs are not provable, because ~constraints flag has been used in generating
        reduced givens (e.g., VC 1_1).

Note 2: Some of the VCs are not provable for reasons discussed in Section 7.3.1 (e.g., finalization
        VC 4_1).

VCs for UVRT_Stack_Realiz.rb generated Sat Aug 18 20:16:31 EDT 2018

================================ VC(s): =================================

VC 0_1

Requires Clause for Communal_UVR_Template in Facility Instantiation Rule:
        UVRT_Stack_Realiz.rb(4:22)

Goal(s):
(1 <= Max_Capacity)

Given(s):
1. (1 <= Max_Capacity)
2. (Max_Capacity is_in N)
VC 1_1

Well Defined Correspondence for Shared Variables: UVRT_Stack_Realiz.rb(7:1)

Goal(s):

(((||UVRT_Fac::Accessible_Loc|| - 1) <= Max_Capacity)

Given(s):

1. (1 <= Max_Capacity)
2. (Max_Capacity is_in N)

VC 2_1

Initialization Ensures Clause of Shared Variables:
Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

(((||({Void} union Closure_for(Location, {UVRT_Fac::Ref},
  SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles)))|| - 1) = 0)

Given(s):

1. (SqBr(UVRT_Fac::Ref, Location) = {Void})
2. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
3. (UVRT_Fac::Cast_Accessible_Loc = ({Void} union Closure_for(Location,
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Pos.Receptacles = {})

VC 2_2

Ensures Clause of Shared Variables (Condition from Non-Affected Shared
Variable): Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

(UVRT_Fac::Ref = UVRT_Fac::Ref)

Given(s):

1. (SqBr(UVRT_Fac::Ref, Location) = {Void})
2. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
3. (UVRT_Fac::Cast_Accessible_Loc = ({Void} union Closure_for(Location,
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Cast_Accessible_Loc = ({Void} union Closure_for(Location,
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
5. (UVRT_Fac::Pos.Receptacles = {})

VC 2_3

Ensures Clause of Shared Variables (Condition from Non-Affected Shared
Variable): Communally_Bounded_Stack_Template.co(10:3)
Goal(s):

(UVRT_Fac::Content = UVRT_Fac::Content)

Given(s):

1. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
2. (UVRT_Fac::Accessible_Loc = ({Void} union Closure_for(Location, 
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
3. (UVRT_Fac::Cast_Accessible_Loc = ({Void} union Closure_for(Location, 
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Pos.Receptacles = { })
5. (SqBr(UVRT_Fac::Ref, Location) = {Void})

VC 2_4

Ensures Clause of Shared Variables (Condition from Non-Affected Definition Variable): Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

({Void} union Closure_for(Location, {UVRT_Fac::Ref}, 
   SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))) = ({Void} union 
   Closure_for(Location, {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, 
   UVRT_Fac::Pos.Receptacles)))

Given(s):

1. (SqBr(UVRT_Fac::Ref, Location) = {Void})
2. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
3. (UVRT_Fac::Cast_Accessible_Loc = ({Void} union Closure_for(Location, 
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Pos.Receptacles = { })

VC 2_5

Ensures Clause of Shared Variables (Condition from Non-Affected Definition Variable): Communally_Bounded_Stack_Template.co(10:3)

Goal(s):

({Void} union Closure_for(Location, {UVRT_Fac::Ref}, 
   SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))) = ({Void} union 
   Closure_for(Location, {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, 
   UVRT_Fac::Pos.Receptacles)))

Given(s):

1. (SqBr(UVRT_Fac::Ref, Location) = {Void})
2. (SqBr(Entry.Is_Initial, SqBr(UVRT_Fac::Content, Location)) = {true})
3. (UVRT_Fac::Accessible_Loc = ({Void} union Closure_for(Location, 
   {UVRT_Fac::Ref}, SqBr(UVRT_Fac::Pos.Val_in, UVRT_Fac::Pos.Receptacles))))
4. (UVRT_Fac::Pos.Receptacles = { })

VC 3_1

224
Convention for Stack Generated by Initialization Rule (Concept Type
Realization): UVRT_Stack_Realiz.rb(16:1)

Goal(s):
(0 <= 0)

Given(s):

VC 3_2

Convention for Stack Generated by Initialization Rule (Concept Type
Realization): UVRT_Stack_Realiz.rb(16:1)

Goal(s):
(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, 0) = UVRT_Fac::Void)

Given(s):
1. (S.Top_Pos = Void)

VC 3_3

Initialization Ensures Clause of Stack:
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):
(Iterated_Concatenation(1, 0, lambda (i :
Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
([Universal] i - 1)))>)) = Empty_String)

Given(s):
1. (S.Top_Pos = Void)

VC 3_4

Ensures Clause of Stack (Condition from Non-Affected Shared Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):
(||UVRT_Fac::Accessible_Loc|| - 1) = (||UVRT_Fac::Accessible_Loc|| - 1))

Given(s):

VC 3_5

Ensures Clause of Stack (Condition from Non-Affected Shared Variable):
Communally_Bounded_Stack_Template.co(16:3)
Goal(s):
(UVRT_Fac::Ref = UVRT_Fac::Ref)

Given(s):

VC 3_6
Ensures Clause of Stack (Condition from Non-Affected Shared Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):
(UVRT_Fac::Content = UVRT_Fac::Content)

Given(s):

VC 3_7
Ensures Clause of Stack (Condition from Non-Affected Definition Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):
(UVRT_Fac::Accessible_Loc = UVRT_Fac::Accessible_Loc)

Given(s):

VC 3_8
Ensures Clause of Stack (Condition from Non-Affected Definition Variable):
Communally_Bounded_Stack_Template.co(16:3)

Goal(s):
(UVRT_Fac::Cast_Accessible_Loc = UVRT_Fac::Cast_Accessible_Loc)

Given(s):

VC 4_1
Finalization Ensures Clause of Stack: Communally_Bounded_Stack_Template.co(19:3)

Goal(s):

((||UVRT_Fac::Accessible_Loc'|| - 1) = (((||UVRT_Fac::Accessible_Loc|| - 1) -
Iterated_Concatenation(1, S.Depth, lambda (i :
2).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
([Universal] i - 1))))>)))})
Given(s):
1. (0 <= S.Depth)
2. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)

VC 5_1
Requires Clause of Give_New_Loc: UVRT_Stack_Realiz.rb(36:2)

Goal(s):
(New_Pos = Void)

Given(s):
1. (New_Pos = Void)

VC 5_2
Requires Clause of Give_New_Loc: UVRT_Stack_Realiz.rb(36:2)

Goal(s):
((1 + ||UVRT_Fac::Accessible_Loc||) <= Max_Capacity)

Given(s):
1. (1 <= Max_Capacity)
2. (Max_Capacity is_in N)
3. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)

VC 5_3
Requires Clause of Swap_Content_of: UVRT_Stack_Realiz.rb(37:2)

Goal(s):
(New_Pos’ /= Void)

Given(s):
1. (New_Pos’ is_not_in UVRT_Fac::Accessible_Loc)
2. (New_Pos = Void)
3. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)
4. (1 <= Max_Capacity)
5. (Max_Capacity is_in N)

VC 5_4
Requires Clause of Redirect_Ref_at: UVRT_Stack_Realiz.rb(38:2)

Goal(s):
(New_Pos’ is_not_in Closure_for(Location, UVRT_Fac::Ref, S.Top_Pos))
Given(s):

1. (E' = UVRT_Fac::Content(New_Pos'))
2. (UVRT_Fac::Content' = lambda (q : Location).{
   E if ([Universal] q = New_Pos')
   UVRT_Fac::Content([Universal] q) otherwise})
3. (New_Pos' is_not_in UVRT_Fac::Accessible_Loc)
4. (0 <= S.Depth)
5. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
6. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)
7. (1 <= Max_Capacity)
8. (Max_Capacity is_in N)

VC 5_5

Requires Clause of Sum: UVRT_Stack_Realiz.rb(40:13)

Goal(s):

(min_int <= (S.Depth + 1))

Given(s):

1. (UVRT_Fac::Ref' = lambda (q : Location).{
   S.Top_Pos if ([Universal] q = New_Pos')
   UVRT_Fac::Ref([Universal] q) otherwise})
2. (S.Top_Pos' = UVRT_Fac::Ref(New_Pos'))
3. (E' = UVRT_Fac::Content(New_Pos'))
4. (UVRT_Fac::Content' = lambda (q : Location).{
   E if ([Universal] q = New_Pos')
   UVRT_Fac::Content([Universal] q) otherwise})
5. (New_Pos' is_not_in UVRT_Fac::Accessible_Loc)
6. (min_int <= 0)
7. (0 <= S.Depth)
8. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
9. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)
10. (1 <= Max_Capacity)
11. (Max_Capacity is_in N)

VC 5_6

Requires Clause of Sum: UVRT_Stack_Realiz.rb(40:13)

Goal(s):

((S.Depth + 1) <= max_int)

Given(s):

1. (UVRT_Fac::Ref' = lambda (q : Location).{
   S.Top_Pos if ([Universal] q = New_Pos')
   UVRT_Fac::Ref([Universal] q) otherwise})
2. (S.Top_Pos' = UVRT_Fac::Ref(New_Pos'))
3. (E' = UVRT_Fac::Content(New_Pos'))
4. (UVRT_Fac::Content' = lambda (q : Location).{

228
5. (New_Pos' is_not_in UVRT_Fac::Accessible_Loc)
6. (1 <= max_int)
7. (0 <= S.Depth)
8. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
9. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)
10. (1 <= Max_Capacity)
11. (Max_Capacity is_in N)

VC 5_7

Type Convention for Stack Generated by Push: UVRT_Stack_Realiz.rb(31:11)

Goal(s):
(0 <= (S.Depth + 1))

Given(s):

1. (UVRT_Fac::Ref'' = lambda (q : Location).(
    S.Top_Pos if ([Universal] q = New_Pos')
    UVRT_Fac::Ref([Universal] q) otherwise))
2. (S.Top_Pos' = UVRT_Fac::Ref(New_Pos'))
3. (E' = UVRT_Fac::Content(New_Pos'))
4. (UVRT_Fac::Content'' = lambda (q : Location).(
    E if ([Universal] q = New_Pos')
    UVRT_Fac::Content([Universal] q) otherwise))
5. (New_Pos’ is_not_in UVRT_Fac::Accessible_Loc)
6. (0 <= S.Depth)
7. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
8. ((1 + (||UVRT_Fac::Accessible_Loc|| - 1)) <= Max_Capacity)
9. (1 <= Max_Capacity)
10. (Max_Capacity is_in N)

VC 5_8

Type Convention for Stack Generated by Push: UVRT_Stack_Realiz.rb(31:11)

Goal(s):

(Iterated_Apply(lambda (q : Location).(
    UVRT_Fac::Ref''([Universe] q) if ([Universal] q is_in (Void union
    Closure_for(Location, lambda (q : Location).(
        S.Top_Pos if ([Universal] q = New_Pos')
        UVRT_Fac::Ref([Universal] q) otherwise), SqBr(Pos.Val_in,
        (Pos.Receptacles without recp.p)))))
    Void otherwise), New_Pos’, (S.Depth + 1)) =
    Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth))

Given(s):

1. (S.Top_Pos’ = UVRT_Fac::Ref(New_Pos'))
2. (E’ = UVRT_Fac::Content(New_Pos’))
3. (UVRT_Fac::Content’’ = lambda (q : Location).(
Ensures Clause of Push: UVRT_Stack_Realiz.rb(31:11)

Goal(s):

\[
\text{Iterated_Concatenation}(1, (\text{S.Depth} + 1), \lambda (i : Z).(<\text{UVRT_Fac::Content}'(\text{Iterated_Apply}(\lambda (q : \text{Location}).(\text{UVRT_Fac::Ref}'([\text{Universal}] q) \text{ if } ([\text{Universal}] q \text{ is_in } (\text{Void union}
\text{Closure_for(Location, lambda (q : Location).(S.Top_Pos \text{ if } ([\text{Universal}] q = \text{New_Pos'})
\text{UVRT_Fac::Ref([Universal]} q \text{ otherwise}, \text{SqBr(Pos.Val_in,}
\text{Pos.Receptacles without recp.p)))))
\text{Void otherwise), \text{New_Pos'}}, ([\text{Universal}] i - 1))))>) = (\langle E \rangle \circ
\text{Iterated_Concatenation}(1, \text{S.Depth}, \lambda (i : Z).(<\text{UVRT_Fac::Content}'(\text{Iterated_Apply}(\text{UVRT_Fac::Ref}, \text{S.Top_Pos,}
([\text{Universal}] i - 1))))>))\]

Given(s):

1. (Fn_Restrict_to(\text{UVRT_Fac::Content}'', \text{UVRT_Fac::Cast Accessible_Loc}'') =
   \text{Fn_Restrict_to}(\lambda (q : \text{Location}).(\text{E if } ([\text{Universal}] q = \text{New_Pos'})
   \text{UVRT_Fac::Content}([\text{Universal}] q) \text{ otherwise},
   \text{UVRT_Fac::Cast Accessible_Loc}''))
2. (E' = \text{UVRT_Fac::Content'(New_Pos'})
3. (New_Pos' is_not_in \text{UVRT_Fac::Accessible_Loc})
4. (0 <= \text{S.Depth})
5. (\text{Iterated_Apply}(\text{UVRT_Fac::Ref}, \text{S.Top_Pos, S.Depth} = \text{UVRT_Fac::Void})
6. ((1 + (||\text{UVRT_Fac::Accessible_Loc}|| - 1)) <= \text{Max_Capacity})
7. (1 <= \text{Max_Capacity})
8. (\text{Max_Capacity is_in N})

VC 5_9

Ensures Clause of Push: UVRT_Stack_Realiz.rb(31:11)

Goal(s):

\[
((||\text{UVRT_Fac::Accessible_Loc}|| - 1) = (||\text{UVRT_Fac::Accessible_Loc}|| - 1) + 1))
\]

Given(s):

1. ((1 + (||\text{UVRT_Fac::Accessible_Loc}|| - 1)) <= \text{Max_Capacity})
2. (1 <= \text{Max_Capacity})
3. (\text{Max_Capacity is_in N})

230
VC 6_1

Requires Clause of Swap_Content_of: UVRT_Stack_Realiz.rb(46:2)

Goal(s):

(S.Top_Pos /= Void)

Given(s):

1. (0 <= S.Depth)
2. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
3. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i : Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, ([Universal] i - 1)))>))|)

VC 6_2

Requires Clause of Follow_Ref: UVRT_Stack_Realiz.rb(47:2)

Goal(s):

(S.Top_Pos /= Void)

Given(s):

1. (R' = UVRT_Fac::Content(S.Top_Pos))
2. (UVRT_Fac::Content'' = lambda (q : Location).(
   R if ([Universal] q = S.Top_Pos)
   UVRT_Fac::Content([Universal] q) otherwise))
3. (0 <= S.Depth)
4. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
5. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i : Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, ([Universal] i - 1)))>))|)

VC 6_3

Requires Clause of Difference: UVRT_Stack_Realiz.rb(48:13)

Goal(s):

(min_int <= (S.Depth - 1))

Given(s):

1. (S.Top_Pos' = UVRT_Fac::Ref(S.Top_Pos))
2. (UVRT_Fac::Ref' = lambda (q : Location).(Void if (([Universal] q = S.Top_Pos) and (S.Top_Pos is_not_in Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in, (Pos.Receptacles without recp.UVRT_Fac::Ref(S.Top_Pos)))))))
3. (R' = UVRT_Fac::Content(S.Top_Pos))
4. (UVRT_Fac::Content'' = lambda (q : Location).{
\begin{verbatim}
R if ([Universal] q = S.Top_Pos)
UVRT_Fac::Content([Universal] q) otherwise)

5. (min_int <= 0)
6. (0 <= S.Depth)
7. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
8. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i : Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
([Universal] i - 1))>))>)|)

VC 6.4

Requires Clause of Difference: UVRT_Stack_Realiz.rb(48:13)

Goal(s):

((S.Depth - 1) <= max_int)

Given(s):

1. (S.Top_Pos' = UVRT_Fac::Ref(S.Top_Pos))
2. (UVRT_Fac::Ref' = lambda (q : Location).(Void if ((((Universal] q = S.Top_Pos) and (S.Top_Pos is_not_in
Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in,
(Pos.Receptacles without recp.UVRT_Fac::Ref(S.Top_Pos))))))))
UVRT_Fac::Ref([Universal] q) otherwise))
3. (R' = UVRT_Fac::Content(S.Top_Pos))
4. (UVRT_Fac::Content'' = lambda (q : Location).(R if ([Universal] q = S.Top_Pos)
UVRT_Fac::Content([Universal] q) otherwise))
5. (1 <= max_int)
6. (0 <= S.Depth)
7. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
8. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i : Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
([Universal] i - 1))>))>)|)

VC 6.5

Type Convention for Stack Generated by Pop: UVRT_Stack_Realiz.rb(43:11)

Goal(s):

(0 <= (S.Depth - 1))

Given(s):

1. (S.Top_Pos' = UVRT_Fac::Ref(S.Top_Pos))
2. (UVRT_Fac::Ref' = lambda (q : Location).(Void if ((([Universal] q = S.Top_Pos) and (S.Top_Pos is_not_in
Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in,
(Pos.Receptacles without recp.UVRT_Fac::Ref(S.Top_Pos))))))))
UVRT_Fac::Ref([Universal] q) otherwise))
3. (R' = UVRT_Fac::Content(S.Top_Pos))
4. (UVRT_Fac::Content'' = lambda (q : Location).(R if ([Universal] q = S.Top_Ptr))
\end{verbatim}
UVRT_Fac::Content([Universal] q) otherwise)
5. (0 <= S.Depth)
6. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
7. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i : Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, ([Universal] i - 1))>))|))

VC 6_6

Type Convention for Stack Generated by Pop: UVRT_Stack_Realiz.rb(43:11)

Goal(s):

(Iterated_Apply(lambda (q : Location).(  
    Void if ((([Universal] q = S.Top_Pos) and (S.Top_Pos is_not_in  
        Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in,  
        (Pos.Receptacles without recip.UVRT_Fac::Ref(S.Top_Pos)))))  
    UVRT_Fac::Ref([Universal] q) otherwise), UVRT_Fac::Ref(S.Top_Pos),  
    (S.Depth - 1)) = Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth))

Given(s):

1. (R' = UVRT_Fac::Content(S.Top_Pos))
2. (UVRT_Fac::Content'' = lambda (q : Location).(  
    R if ((([Universal] q = S.Top_Pos)  
    UVRT_Fac::Content([Universal] q) otherwise))
3. (0 <= S.Depth)
4. (1 <= |Iterated_Concatenation(1, S.Depth, lambda (i :  
    Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,  
    ([Universal] i - 1))>))|))

VC 6_7

Ensures Clause of Pop: UVRT_Stack_Realiz.rb(43:11)

Goal(s):

(Iterated_Concatenation(1, S.Depth, lambda (i :  
    Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,  
    ([Universal] i - 1))>)) = (<UVRT_Fac::Content(S.Top_Pos)> o  
Iterated_Concatenation(1, (S.Depth - 1), lambda (i :  
    Z).(<UVRT_Fac::Content''(Iterated_Apply(lambda (q : Location).(  
    Void if ((([Universal] q = S.Top_Pos) and (S.Top_Pos is_not_in  
        Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in,  
        (Pos.Receptacles without recip.UVRT_Fac::Ref(S.Top_Pos)))))  
    UVRT_Fac::Ref([Universal] q) otherwise), UVRT_Fac::Ref(S.Top_Pos),  
    ([Universal] i - 1))>)))))

Given(s):

1. (Fn_Restrict_to(UVRT_Fac::Content', UVRT_Fac::Cast_Accessible_Loc') =  
    Fn_Restrict_to(lambda (q : Location).(  
    R if ((([Universal] q = S.Top_Pos)  
    UVRT_Fac::Content([Universal] q) otherwise),  
    UVRT_Fac::Cast_Accessible_Loc')))
2. \((0 \leq S.\text{Depth})\)
3. \((\text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, S.\text{Depth}) = \text{UVRT}\_\text{Fac}::\text{Void})\)
4. \((1 \leq |\text{Iterated}_{\text{Concatenation}}(1, S.\text{Depth}, \lambda (i : \mathbb{Z}).(<\text{UVRT}\_\text{Fac}::\text{Content}(\text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, ([\text{Universal}] i - 1))))>)))\)

**VC 6_8**

Ensures Clause of Pop: UVRT_Stack_Realiz.rb(43:11)

**Goal** (s):

\(((||\text{UVRT}\_\text{Fac}::\text{Accessible}\_\text{Loc}|| - 1) = ((||\text{UVRT}\_\text{Fac}::\text{Accessible}\_\text{Loc}|| - 1) - 1))\)

**Given** (s):

**VC 7_1**

Type Convention for Stack Generated by Depth: UVRT_Stack_Realiz.rb(51:11)

**Goal** (s):

\((0 \leq S.\text{Depth})\)

**Given** (s):

1. \((0 \leq S.\text{Depth})\)
2. \((\text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, S.\text{Depth}) = \text{UVRT}\_\text{Fac}::\text{Void})\)

**VC 7_2**

Type Convention for Stack Generated by Depth: UVRT_Stack_Realiz.rb(51:11)

**Goal** (s):

\((\text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, S.\text{Depth}) = \\
\quad \text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, S.\text{Depth}))\)

**Given** (s):

1. \((0 \leq S.\text{Depth})\)

**VC 7_3**

Ensures Clause of Depth: UVRT_Stack_Realiz.rb(51:11)

**Goal** (s):

\((S.\text{Depth} = |\text{Iterated}_{\text{Concatenation}}(1, S.\text{Depth}, \lambda (i : \mathbb{Z}).(<\text{UVRT}\_\text{Fac}::\text{Content}(\text{Iterated}_{\text{Apply}}(\text{UVRT}\_\text{Fac}::\text{Ref}, S.\text{Top}\_\text{Pos}, ([\text{Universal}] i - 1))))>)))\)

**Given** (s):
1. (0 <= S.Depth)
2. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)

VC 7_4

Ensures Clause of Depth (Condition from "RESTORES" parameter mode):
UVRT_Stack_Realiz.rb(51:26)

Goal(s):

(Iterated_Concatenation(1, S.Depth, lambda (i :
  Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
  ([Universal] i - 1))>))) = Iterated_Concatenation(1, S.Depth, lambda (i :
  Z).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos,
  ([Universal] i - 1))>))))

Given(s):

1. (0 <= S.Depth)
2. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)

VC 7_5

Ensures Clause of Depth (Condition from Non-Affected Shared Variable):
UVRT_Stack_Realiz.rb(51:11)

Goal(s):

(||UVRT_Fac::Accessible_Loc|| - 1) = (||UVRT_Fac::Accessible_Loc|| - 1))

Given(s):

VC 7_6

Ensures Clause of Depth (Condition from Non-Affected Shared Variable):
UVRT_Stack_Realiz.rb(51:11)

Goal(s):

(UVRT_Fac::Ref = UVRT_Fac::Ref)

Given(s):

1. (Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth) = UVRT_Fac::Void)
2. (0 <= S.Depth)

VC 7_7

Ensures Clause of Depth (Condition from Non-Affected Shared Variable):
UVRT_Stack_Realiz.rb(51:11)

Goal(s):

(UVRT_Fac::Content = UVRT_Fac::Content)
**Given(s):**

**VC 7_8**

Ensures Clause of Depth (Condition from Non-Affected Definition Variable):  
UVRT_Stack_Realiz.rb(51:11)

**Goal(s):**

(UVRT_Fac::Accessible_Loc = UVRT_Fac::Accessible_Loc)

**Given(s):**

**VC 7_9**

Ensures Clause of Depth (Condition from Non-Affected Definition Variable):  
UVRT_Stack_Realiz.rb(51:11)

**Goal(s):**

(UVRT_Fac::Cast_Accessible_Loc = UVRT_Fac::Cast_Accessible_Loc)

**Given(s):**

**VC 8_1**

Requires Clause of Difference: UVRT_Stack_Realiz.rb(56:19)

**Goal(s):**

(min_int <= (||UVRT_Fac::Accessible_Loc|| - 1))

**Given(s):**

1. (min_int <= 0)

**VC 8_2**

Requires Clause of Difference: UVRT_Stack_Realiz.rb(56:19)

**Goal(s):**

((||UVRT_Fac::Accessible_Loc|| - 1) <= max_int)

**Given(s):**

1. (1 <= max_int)

**VC 8_3**

Ensures Clause of Occupied_Size: UVRT_Stack_Realiz.rb(55:11)
Goal(s):

\(((||UVRT\_Fac::Accessible\_Loc|| - 1) = (||UVRT\_Fac::Accessible\_Loc|| - 1))\)

Given(s):

VC 8_4

Ensures Clause of Occupied\_Size (Condition from Non-Affected Shared Variable):
   UVRT\_Stack\_Realiz.rb(55:11)

Goal(s):

\(((||UVRT\_Fac::Accessible\_Loc|| - 1) = (||UVRT\_Fac::Accessible\_Loc|| - 1))\)

Given(s):

VC 8_5

Ensures Clause of Occupied\_Size (Condition from Non-Affected Shared Variable):
   UVRT\_Stack\_Realiz.rb(55:11)

Goal(s):

(UVRT\_Fac::Ref = UVRT\_Fac::Ref)

Given(s):

VC 8_6

Ensures Clause of Occupied\_Size (Condition from Non-Affected Shared Variable):
   UVRT\_Stack\_Realiz.rb(55:11)

Goal(s):

(UVRT\_Fac::Content = UVRT\_Fac::Content)

Given(s):

VC 8_7

Ensures Clause of Occupied\_Size (Condition from Non-Affected Definition
   Variable): UVRT\_Stack\_Realiz.rb(55:11)

Goal(s):

(UVRT\_Fac::Accessible\_Loc = UVRT\_Fac::Accessible\_Loc)

Given(s):
VC 8_8
Ensures Clause of Occupied_Size (Condition from Non-Affected Definition Variable): UVRT_Stack_Realiz.rb(55:11)

Goal(s):
(UVRT_Fac::Cast_Accessible_Loc = UVRT_Fac::Cast_Accessible_Loc)

Given(s):

VC 9_1
Type Convention for Stack Generated by Clear: UVRT_Stack_Realiz.rb(59:11)

Goal(s):
(0 <= 0)

Given(s):

VC 9_2
Type Convention for Stack Generated by Clear: UVRT_Stack_Realiz.rb(59:11)

Goal(s):

(Iterated_Apply(lambda (q : Location).(UVRT_Fac::Ref([Universal] q) if ([Universal] q is_in (Void union Closure_for(Location, UVRT_Fac::Ref, SqBr(Pos.Val_in, (Pos.Receptacles without recp.p)))))) Void otherwise), S.Top_Pos', 0) = Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, S.Depth))

Given(s):
1. (S.Top_Pos’ = Void)
2. (0 <= S.Depth)

VC 9_3
Ensures Clause of Clear: UVRT_Stack_Realiz.rb(59:11)

Goal(s):

((||UVRT_Fac::Accessible_Loc'|| - 1) = ((||UVRT_Fac::Accessible_Loc|| - 1) - |Iterated_Concatenation(1, S.Depth, lambda (i : 2).(<UVRT_Fac::Content(Iterated_Apply(UVRT_Fac::Ref, S.Top_Pos, ([Universal] i - 1)))>)))|))

Given(s):
1. \(0 \leq \text{S.Depth}\)
2. \(\text{Iterated_Apply(UVRT}_{\text{Fac}}::\text{Ref}, \text{S.Top_Pos}, \text{S.Depth}) = \text{UVRT}_{\text{Fac}}::\text{Void}\) 

**VC 9-4**

Ensures Clause of Clear (Condition from "CLEARS" parameter mode):
UVRT_Stack_Realiz.rb(59:24)

**Goal(s)**:

\[(\text{Iterated_Concatenation(1, 0, lambda (i : (Z).(<\text{UVRT}_{\text{Fac}}::\text{Content'}}\text{Iterated_Apply(lambdas (q : \text{Location}).(\text{UVRT}_{\text{Fac}}::\text{Ref}}[[\text{Universal}] q] \text{ if } ([\text{Universal}] q \text{ is in (Void union}\text{Closure_for(Location, UVRT}_{\text{Fac}}::\text{Ref, SqBr(Pos.Val_in, (Pos.Receptacles without recip.p))})\text{Void otherwise), S.Top_Pos', ([Universal] i - 1)))>)) = Empty_String)}\]

**Given(s)**:

1. \((\text{Fn_Restrict_to(UVRT}_{\text{Fac}}::\text{Content'}}\text{, UVRT}_{\text{Fac}}::\text{Cast Accessible Loc'}) = \text{Fn_Restrict_to(UVRT}_{\text{Fac}}::\text{Content}, UVRT}_{\text{Fac}}::\text{Cast Accessible Loc'})\)
Appendix G  Communally Bounded UVRT Collection

G.1  Concept

Shared Concept Communal_UVR_Template(
    type Info; evaluates Max_Capacity: Integer );
uses Set_App_Op_Ext, Closure_Op_Ext, Terminal_Range_Op_Ext,
    Integer_Ext_Theory;
requires 1 <= Max_Capacity which_entails Max_Capacity is_in N;

-- Some of the which_entails clauses have been commented out
-- (because they are not directly useful for proving the VCs in
-- the examples and turn out to be a distraction)

Defines Location: SSet;
Defines Void: Location;

Shared Variables
    Abstract_Var Ref: Location -> Location;
    Abstract_Var Content: Location -> Info;

    constraint Terminal_Range(Location, {Ref}, Location) is_subset_of {Void}
        which_entails Ref(Void) = Void;

initialization
    ensures SqBr(Ref, Location) = {Void} and
        SqBr(Info.Is_Initial, SqBr(Content, Location)) = {true};
end;

Type Family Pos is modeled by Location;
    exemplar p;

Def Var Accessible_Loc : FinPowerset(Location) = ( {Void} union
    Closure_for(Location, {Ref}, SqBr(Pos.Val_in,
        Pos.Receptacles)));

Def Var Cast_Accessible_Loc : Powerset(Location) = Accessible_Loc;

constraint Cast_Accessible_Loc = Accessible_Loc and
    SqBr(Info.Is_Initial, SqBr(Content, Location without
        Accessible_Loc)) is_in {true} and
    SqBr(Ref, Location without Accessible_Loc) is_in {Void} and
    ||Accessible_Loc|| <= Max_Capacity;

initialization
    ensures p = Void;
finalization
    affects Ref, Content, Accessible_Loc, Cast_Accessible_Loc;
    ensures Ref = lambda (q : Location).
        { ||#Ref(q) if q is_in {{Void} union
            Closure_for(Location, (#Ref),
                SqBr(#Pos.Val_in,
                    #Pos.Receptacles without
                        {recp.p}))}};
        Void otherwise;}})
Operation Give_New_Loc(updates p: Pos);
  affects Accessible_Loc, Cast_Accessible_Loc;
  requires p = Void and 1 + ||Accessible_Loc|| <= Max_Capacity;
  ensures p is_not_in #Accessible_Loc;

Operation Occupied_Size() : Integer;
  ensures Occupied_Size = ( ||Accessible_Loc|| );

Operation Redirect_Ref_at(preserves p: Pos; updates referent: Pos);
  affects Ref;
  requires p is_not_in Closure_for(Location, {Ref}, {referent});
  ensures Ref = lambda (q : Location).({#referent if q = p; #Ref(q) otherwise;});

Operation Follow_Ref(updates p: Pos);
  affects Ref, Accessible_Loc, Cast_Accessible_Loc, Content;
  requires p /= Void;
  ensures p = #Ref(#p) and Ref = lambda (q : Location).({#I if q = p; #Content(q) otherwise;});

Operation Swap_Content_of(preserves p: Pos; updates I: Info);
  affects Content;
  requires p /= Void;
  ensures I = #Content(p) and Content = lambda (q : Location).({#I if q = p; #Content(q) otherwise;});

Operation Relocate_to(preserves New_L: Pos; replaces p: Pos);
  affects Ref, Accessible_Loc, Cast_Accessible_Loc, Content;
  requires p = New_L and
  ensures Ref = lambda (q : Location).({#Ref(q) if q is_in ((Void) union
Closure_for(Location, {#Ref}, SqBr(#Pos.Val_in, #Pos.Receptacles without {recp.p}));
Void otherwise;}))

and Fn_Restrict_to(Content, Cast_Accessible_Loc) =
Fn_Restrict_to(#Content, Cast_Accessible_Loc);
-- which_entails p is_in Closure_for(Location, {Ref},
SqBr(#Pos.Val_in, #Pos.Receptacles without {recp.p}))
implies
-- Ref = #Ref and Accessible_Loc = #Accessible_Loc and
Content = #Content;

Operation Are_Colocated(preserves p, q: Pos): Boolean;
ensures Are_Colocated = (p = q);

Operation Is_Almost_Inaccessible(preserves p: Pos): Boolean;
ensures Is_Almost_Inaccessible = (
  p is_not_in {{Void} union Closure_for(Location, {Ref},
    SqBr(#Pos.Val_in, #Pos.Receptacles without {recp.p}))});

Operation Is_Void(preserves p: Pos): Boolean;
ensures Is_Void = (p = Void);

Operation Set_to_Void(clears p: Pos);
affects Ref, Accessible_Loc, Cast_Accessible_Loc, Content;
ensures Ref = lambda (q : Location).(1
  {{#Ref(q) if q is_in {{Void} union
    Closure_for(Location, {Ref}, SqBr(#Pos.Val_in,
      #Pos.Receptacles without {recp.p}))};
  Void otherwise;}})

and Fn_Restrict_to(Content, Cast_Accessible_Loc) =
Fn_Restrict_to(#Content, Cast_Accessible_Loc);
-- which_entails p is_in {{Void} union Closure_for(Location, {Ref},
  SqBr(#Pos.Val_in, #Pos.Receptacles without {recp.p}))} implies
-- Ref = #Ref and Accessible_Loc = #Accessible_Loc and
Content = #Content;

end Communal_UVR_Template;

G.1.1 Concept VCs

VCs for Communal_UVR_Template.co generated Sat Aug 18 13:57:18 EDT 2018

VC(0:1)

Which_Entails Expression Located at Communal_UVR_Template.co(12:13):
Communal_UVR_Template.co(13:21)

Goal(s):
(Ref(Void) = Void)
Given(s):

1. \((\text{Terminal\_Range}(\text{Location, \{Ref\}, Location}) \subseteq \{\text{Void}\})\)

VC 1_1

Which_Entails Expression Located at Communal_UVR_Template.co(3:10):
Communal_UVR_Template.co(3:42)

Goal(s):

\((\text{Max\_Capacity is in N})\)

Given(s):

1. \((1 \leq \text{Max\_Capacity})\)

G.2 Facilities

G.2.1 Example Client Program

```
Facility Inject_Front_Example;
   uses Integer_Theory, Set_App_Op_Ext;

   Facility UVRT_Fac is Communal_UVR_Template(Integer, 4)
      externally realized by Communal_Array_Realiz;

   Operation Inject_Front(updates p: Pos; alters i: Integer);
      affects UVRT_Fac::Ref, UVRT_Fac::Content,
              UVRT_Fac::Accessible_Loc, UVRT_Fac::Cast_Accessible_Loc;
      requires 1 + ||UVRT_Fac::Accessible_Loc|| \leq 4;
      ensures UVRT_Fac::Ref(p) = \#p and UVRT_Fac::Content(p) = \#i;
   Procedure
      Var New_Pos: UVRT_Fac::Pos;
         Give_New_Loc(New_Pos);
         Swap_Content_of(New_Pos, i);
         Redirect_Ref_at(New_Pos, p);
         New_Pos := p;
      end Inject_Front;
   end

end Inject_Front_Example;
```
VC 0_1
Requires Clause for Communal_UVR_Template in Facility Instantiation Rule:
Inject_Front_Example.fa(4:25)

Goal(s):
(1 <= 4)

Given(s):

VC 1_1
Requires Clause of Give_New_Loc: Inject_Front_Example.fa(15:8)

Goal(s):
(New_Pos = Void)

Given(s):
1. (New_Pos = Void)

VC 1_2
Requires Clause of Give_New_Loc: Inject_Front_Example.fa(15:8)

Goal(s):
((1 + ||UVRT_Fac::Accessible_Loc||) <= 4)

Given(s):
1. ((1 + ||UVRT_Fac::Accessible_Loc||) <= 4)

VC 1_3
Requires Clause of Swap_Content_of: Inject_Front_Example.fa(16:8)

Goal(s):
(New_Pos’ /= Void)

Given(s):
1. (New_Pos’ is_not_in UVRT_Fac::Accessible_Loc)
2. (New_Pos = Void)
3. \((1 + |\text{UVRT Fac::Accessible_Loc}|) \leq 4\)

**VC 1_4**

Requires Clause of Redirect_Ref_at: Inject_Front_Example.fa(17:8)

**Goal(s):**

\((\text{New_Pos'} \text{ is not in } \text{Closure_for(Location, UVRT Fac::Ref, p)})\)

**Given(s):**

1. \((i' = \text{UVRT Fac::Content}(\text{New_Pos'}))\)
2. \((\text{UVRT Fac::Content''} = \lambda (q : \text{Location}).\{
   i \quad \text{if } ([\text{Universal}] \ q = \text{New_Pos'}),
   \text{UVRT Fac::Content}([\text{Universal}] \ q) \quad \text{otherwise}\})\)
3. \((\text{New_Pos'} \text{ is not in } \text{UVRT Fac::Accessible_Loc})\)
4. \((1 + |\text{UVRT Fac::Accessible_Loc}|) \leq 4\)

**VC 1_5**

Ensures Clause of Inject_Front: Inject_Front_Example.fa(7:14)

**Goal(s):**

\((\text{UVRT Fac::Ref'}(\text{New_Pos'}) = p)\)

**Given(s):**

1. \((\text{UVRT Fac::Ref'} = \lambda (q : \text{Location}).\{
   \text{UVRT Fac::Ref'}([\text{Universal}] \ q) \quad \text{if } ([\text{Universal}] \ q \text{ is in } (\text{Void union } \text{Closure_for(Location, lambda (q : Location).} (\text{p} \quad \text{if } ([\text{Universal}] \ q = \text{New_Pos'}), \text{UVRT Fac::Ref}([\text{Universal}] \ q) \quad \text{otherwise})), \text{SqBr(Pos.Val_in, (Pos.Receptacles without recp.New_Pos')))\)),
   \text{Void} \quad \text{otherwise})\})\)
2. \((p' = \text{UVRT Fac::Ref}(\text{New_Pos'})\)
3. \((i' = \text{UVRT Fac::Content}(\text{New_Pos'})\)
4. \((\text{UVRT Fac::Content''} = \lambda (q : \text{Location}).\{
   i \quad \text{if } ([\text{Universal}] \ q = \text{New_Pos'}),
   \text{UVRT Fac::Content}([\text{Universal}] \ q) \quad \text{otherwise}\})\)
5. \((\text{New_Pos'} \text{ is not in } \text{UVRT Fac::Accessible_Loc})\)
6. \((\text{New_Pos} = \text{Void})\)
7. \((1 + |\text{UVRT Fac::Accessible_Loc}|) \leq 4\)

**VC 1_6**

Ensures Clause of Inject_Front: Inject_Front_Example.fa(7:14)

**Goal(s):**

\((\text{UVRT Fac::Content'}(\text{New_Pos'}) = i)\)

**Given(s):**
1. \( \text{Fn\_Restrict\_to}(\text{UVRT\_Fac::Content'}, \text{UVRT\_Fac::Cast\_Accessible\_Loc'}) = \text{Fn\_Restrict\_to}(\lambda q: \text{Location}. (\neg q' = \text{New\_Pos'} \rightarrow \text{UVRT\_Fac::Content}([\text{Universal}] q) \text{ otherwise}), \text{UVRT\_Fac::Cast\_Accessible\_Loc'}) \)

2. \( i' = \text{UVRT\_Fac::Content}(\text{New\_Pos'}) \)

3. \( \text{New\_Pos'} \text{ is_not_in UVRT\_Fac::Accessible\_Loc} \)

4. \( (1 + ||\text{UVRT\_Fac::Accessible\_Loc}||) \leq 4) \)
Appendix H  Mathematical Theories

H.1  String Theory

Precis  String_Theory;
uses  Integer_Theory;

--The type of all strings of heterogenous type
Definition  SStr : Cls;
Definition  Empty_String  : SStr;

--A function that restricts SStr to the type of all strings of
--some homogenous type
Definition  Str  : Cls -> Cls;
Definition  ext(S : SStr, x : Entity) : SStr;

Type Theorem  Empty_String_In_All_S strs:
  For all  T : Cls,
    Empty_String  : Str(T);

Type Theorem  All_S strs_In_SStr:
  For all  T : Cls,
    For all  S : Str(T),
      S  : SStr;

--If R is a subset of T, then Str(R) is a subset of Str(T)
Type Theorem  Str_Subsets:
  For all  T : Cls,
    For all  R : Powerclass(T),
      For all  s : Str(R),
        s  : Str(T);

Definition  DeString(s : SStr)  : Entity;
Definition  Iterated_Concatenation(l : Z, m : Z, F: Z->SStr) : SStr;  -- Big Pi
Definition  (s : SStr) o (t : SStr) : SStr;

Type Theorem  Concatenation_Preserves_Generic_Type:
  For all  T : Cls,
    For all  U, V : Str(T),
      U o V  : Str(T);

Type Theorem  DeString_Extracts_Generic_Type:
  For all  T : Cls,
    For all  S : Str(T),
      DeString(S)  : Str(T);

Definition  Reverse(s : SStr) : SStr;

Type Theorem  Reverse_Preserves_Generic_Type:
  For all  T : Cls,
    For all  S : Str(T),
      Reverse(S)  : Str(T);
**Definition** \( \min(m : \mathbb{Z}, n : \mathbb{Z}) : \mathbb{Z} ; \)

**Definition** \( \max(m : \mathbb{Z}, n : \mathbb{Z}) : \mathbb{Z} ; \)

**Corollary** Concatenation_1_a: -- Is_Identity_for(o,Empty_String);
   For all \( S : \mathbb{SStr}, \)
       \( \operatorname{Empty\_String} \circ S = S ; \)

**Corollary** Concatenation_1_b: -- Is_Identity_for(o,Empty_String);
   For all \( S : \mathbb{SStr}, \)
       \( S \circ \operatorname{Empty\_String} = S ; \)

**Corollary** Concatenation_2: -- Is_Associative(o);
   For all \( S, T, U : \mathbb{SStr}, \)
       \( S \circ (T \circ U) = (S \circ T) \circ U ; \)

**Corollary** Concatenation_3: -- Is_Right_Cancellative(o)
   For all \( S, T, U : \mathbb{SStr}, \)
       \( ((S \circ U) = (T \circ U)) = (S = T) ; \)

**Definition** \( |(alpha : \mathbb{SStr})| : \mathbb{N} ; \)

**Theorem** Str_Length_Expanded_Def_i:
   \( |\operatorname{Empty\_String}| = 0 ; \)

**Corollary** Str-Length_1_a:
   For all \( \alpha : \mathbb{SStr}, \)
       \( (|\alpha| = 0) = (\alpha = \operatorname{Empty\_String}) ; \)

**Corollary** Str-Length_1b:
   For all \( \alpha : \mathbb{SStr}, \)
       \( \neg(\alpha = \operatorname{Empty\_String}) = (1 \leq |\alpha|) ; \)

**Corollary** Str-Length_2:
   For all \( \alpha, \beta : \mathbb{SStr}, \)
       \( |\alpha \circ \beta| = |\alpha| + |\beta| ; \)

**Corollary** Str-Length2_without_Length_Op:
   For all \( \alpha, \beta, \gamma : \mathbb{SStr}, \)
       \( \alpha \circ \beta = \gamma \implies |\gamma| = |\alpha \circ \beta| ; \)

**Corollary** Str-Length_Lt:
   For all \( \alpha, \beta, \gamma : \mathbb{SStr}, \)
       \( |\alpha \circ \beta| = |\gamma| \text{ and } 1 \leq |\beta| \implies 1 + |\alpha| \leq |\gamma| ; \)

**Corollary** Str-Length_2_1:
   For all \( \alpha, \beta, \gamma, \delta : \mathbb{SStr}, \)
       \( ((\alpha \circ \beta) = (\gamma \circ \delta) \text{ and } |\beta| = |\delta|) \implies (\beta = \delta \text{ and } \alpha = \gamma) ; \)
Corollary Str_Length_3_2:
For all alpha, beta, gamma, delta : SStr,
((alpha o beta) = (gamma o delta) and |alpha| = |gamma|)
implies (beta = delta and alpha = gamma);

Definition Prime_Str : Cls;
Definition <(e : Entity)> : Prime_Str;
Type Theorem Prime_Str_is_SSTR:
For all p : Prime_Str,
p : SStr;

Type Theorem Singleton_Preserves_Generic_Type:
For all T : Cls,
For all e : T,
<e> : Str(T);

Corollary Singleton_Str_1:
For all p : Prime_Str,
not (p = Empty_String);

Corollary Singleton_Str_2:
For all p : Prime_Str,
|p| = 1;

Corollary Singleton_Str_3a: -- Is_Bijective(op<>); Changed from IsInjective
For all x, y : Entity,
(<x> = <y>) = (x = y);

Theorem Reverse_Expanded_Definition_i:
Reverse(Empty_String) = Empty_String;

Corollary Reverse_1:
For all p : Prime_Str,
Reverse(p) = p;

Corollary Reverse_2:
For all alpha, beta : SStr,
Reverse(alpha o beta) = Reverse(beta) o Reverse(alpha);

Corollary Reverse_3:
For all alpha : SStr,
Reverse(Reverse(alpha)) = alpha;

Corollary Reverse_4: -- Is_Bijective(Reverse);
For all alpha, beta : SStr,
(Reverse(alpha) = Reverse(beta)) = (alpha = beta);

Corollary Reverse_5:
For all alpha, beta : SStr,
Reverse(alpha) = beta implies Reverse(beta) = alpha;

Corollary Reverse_6: -- Is_Left_Cancellative( o )
For all S, T, U : SStr,
((U o S) = (U o T)) = (S = T);
Corollary Reverse_8:
  For all \( \alpha : SStr \),
  \[ |\text{Reverse}(\alpha)| = |\alpha|; \]

Definition Prt_Btwn(m : Z, n : Z, \( \alpha : SStr \)) : SStr;

Theorem Prt_Btn Expanded Def_i:
  For all \( m, n : Z \),
  \( \text{Prt_Btwn}(m,n,\text{Empty_String}) = \text{Empty_String}; \)

Corollary Prt_Btwn_1:
  For all \( \alpha, \beta : SStr \),
  For all \( n : Z \),
  \( (|\alpha| \leq n) \text{ and } \text{Prt_Btwn}(0,n,\alpha) = \beta \) implies \( \beta = \alpha; \)

Corollary Prt_Btwn_2:
  For all \( \alpha : SStr \),
  For all \( m, n : Z \),
  \( (\text{Prt_Btwn}(m,n,\alpha) = \alpha \text{ and } \alpha \neq \text{Empty_String}) \) implies
  \( m < 0 \text{ and } |\alpha| < n; \)

Corollary Prt_Btwn_3:
  For all \( \alpha : SStr \),
  For all \( m, n : Z \),
  \( (\text{Prt_Btwn}(m,n,\alpha) = \text{Empty_String}) = (\alpha = \text{Empty_String} \text{ or } |\alpha| \leq m \text{ or } n < m; \)

Corollary Prt_Btwn_4:
  For all \( \alpha : SStr \),
  For all \( n : Z \),
  \( \text{Prt_Btwn}(n,n,\alpha) = \text{Empty_String}; \)

Corollary Prt_Btwn_5:
  For all \( \alpha : SStr \),
  For all \( m, n : Z \),
  \( |\text{Prt_Btwn}(m,n,\alpha)| = \max(\min(n,|\alpha|) + -\{\max(m,0)\},0); \)

Corollary Prt_Btwn_6a:
  For all \( \alpha : SStr \),
  For all \( i, m, n : N \),
  \( (|\text{Prt_Btwn}(m,n,\alpha)| = i \text{ and } m \leq n \leq |\alpha|) \) implies
  \( i = n + (-m); \)

Corollary Prt_Btwn_6b: -- 6a without negatives
  For all \( \alpha : SStr \),
  For all \( i, m, n : N \),
  \( (|\text{Prt_Btwn}(m,n,\alpha)| = i \text{ and } m \leq n \leq |\alpha|) \) implies
  \( i \leq n; \)

Corollary Prt_Btwn_6c: -- 6a without negatives
  For all \( \alpha : SStr \),
  For all \( i, m, n : N \),
  \( (m + |\text{Prt_Btwn}(m,n,\alpha)| = i \text{ and } m \leq n \leq |\alpha|) \) implies
  \( i = n; \)

Corollary Prt_Btwn_7:
  For all \( \alpha : SStr \),
  \[ 250 \]
For all $m, n : \mathbb{Z}$,
\[
\text{Prt}_\text{Btwn}(0, m, \alpha) \circ \text{Prt}_\text{Btwn}(m, n, \alpha) \circ \\
\text{Prt}_\text{Btwn}(\max(m, n), |\alpha|, \alpha) = \alpha;
\]

**Corollary** $\text{Prt}_\text{Btwn}_8$:
For all $\alpha : \text{SStr}$,
For all $n : \mathbb{Z}$,
\[
\text{Prt}_\text{Btwn}(0, n, \alpha) \circ \text{Prt}_\text{Btwn}(n, |\alpha|, \alpha) = \alpha;
\]

**Corollary** $\text{Prt}_\text{Btwn}_9_a$:
For all $\alpha, \beta, \gamma, \delta : \text{SStr}$,
For all $m, n : \mathbb{Z}$,
\[
\text{Prt}_\text{Btwn}(m, n, \alpha \circ \beta) = \gamma \quad \text{and} \quad \\
\text{Prt}_\text{Btwn}(m, n, \alpha) = \delta \quad \text{and} \quad n \leq |\alpha| \\
\implies \gamma = \delta;
\]

**Corollary** $\text{Prt}_\text{Btwn}_9_b$:
For all $\alpha, \beta, \gamma, \delta : \text{SStr}$,
For all $m, n : \mathbb{Z}$,
\[
\text{Prt}_\text{Btwn}(m, n, \alpha \circ \beta) = \gamma \quad \text{and} \quad \\
\text{Prt}_\text{Btwn}(m + (- |\alpha|), n + (- |\alpha|), \beta) = \delta \quad \text{and} \quad |\alpha| \leq m \\
\implies \gamma = \delta;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{10_a}$:
For all $\alpha, \beta : \text{SStr}$,
\[
\text{Prt}_\text{Btwn}(0, |\alpha|, \alpha \circ \beta) = \alpha;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{10_b}$:
For all $\alpha, \beta : \text{SStr}$,
\[
\text{Prt}_\text{Btwn}(|\alpha|, |\alpha \circ \beta|, \alpha \circ \beta) = \beta;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{11_a}$:
For all $\alpha, \beta : \text{SStr}$,
For all $x : \text{Entity}$,
\[
\text{Prt}_\text{Btwn}(|\alpha|, |\alpha| + 1, \alpha \circ <x>) = <x>;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{11_b}$:
For all $\alpha, \beta : \text{SStr}$,
For all $x : \text{Entity}$,
\[
\text{Prt}_\text{Btwn}(0, 1, <x> \circ \alpha) = <x>;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{11_c}$:
For all $\alpha, \beta : \text{SStr}$,
For all $x : \text{Entity}$,
\[
\text{Prt}_\text{Btwn}(0, |\alpha|, \alpha \circ <x>) = \alpha;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{11_d}$:
For all $\alpha, \beta : \text{SStr}$,
For all $x : \text{Entity}$,
\[
\text{Prt}_\text{Btwn}(1, |\alpha| + 1, <x> \circ \alpha) = \alpha;
\]

**Corollary** $\text{Prt}_\text{Btwn}_{12_a}$:
For all $\alpha : \text{SStr}$,
For all $m, n, p, q : \mathbb{Z}$,
\[
\text{Prt}_\text{Btwn}(m, n, \text{Prt}_\text{Btwn}(p, q, \alpha)) = \text{Prt}_\text{Btwn}(m + p, \min(n + p, q)),
\]
Corollary Prt_Btwn_12_b:
  For all alpha : SStr,
  For all n : Z,
  Reverse(Prt_Btwn(n,n+1,alpha)) = Prt_Btwn(n,n+1,alpha);

-- Determines if for every pairing of elements from s and t,
-- the given predicate holds
Definition Is_Universally_Related(s : SStr, t : SStr,
  f : (Entity * Entity) -> B) : B;

Theorem DeString_Expanded_Definition:
  For all rho: Prime_Str,
  <DeString(rho)> = rho;

Corollary DeString_1:
  For all x : Entity,
  DeString(<x>) = x;

Corollary DeString_2:
  For all alpha : SStr,
  For all n : Z,
  1 <= n + 1 <= |alpha| implies <DeString(Prt_Btwn(n, n + 1, alpha))> = Prt_Btwn(n, n+1, alpha);

Corollary DeString_2_no_addition_no_Length:
  For all alpha : SStr,
  not (alpha = Empty_String) implies <DeString(Prt_Btwn(0, 1, alpha))> = Prt_Btwn(0, 1, alpha);

Definition Is_Substring(a : SStr, b : SStr) : B;

Corollary Is_Substring_1a:
  For all a, b : SStr,
  Is_Substring(a, a o b);

Corollary Is_Substring_1b:
  For all a, b : SStr,
  Is_Substring(b, a o b);

Corollary Is_Substring_3_reflexive:
  For all a : SStr,
  Is_Substring(a, a);

Corollary Is_Substring_3_transitive:
  For all a, b, c : SStr,
  Is_Substring(a, b) and Is_Substring(b, c) implies Is_Substring(a, c);

-- Will be able to state these as: (not(Is_Substring(a,c)) implies Is_Substring(a, b o c) = Is_Substring(a, b);

Corollary Is_Substring_3_transitive_contrapositive_a:
  For all a, b, c : SStr,
  For all p : B,
\begin{verbatim}
(not(Is_Substring(a, c)) and Is_Substring(a, b o c) = p ) implies p = Is_Substring(a, b);

Corollary Is_Substring_3_transitive_contraposition_b:
For all a, b, c : SStr,
For all p : B,
(not(Is_Substring(a, c)) and Is_Substring(a, c o b) = p ) implies p = Is_Substring(a, b);

Corollary Is_Substring_3_antisymmetric:
For all a, b : SStr,
(Is_Substring(a, b) and Is_Substring(b, a)) = (a = b);

Corollary Is_Substring_4:
For all a, b : SStr,
Is_Substring(a, b) implies |a| <= |b|;

Corollary Is_Substring_4_Without_Length_Operator:
For all a : SStr,
Is_Substring(a, Empty_String) = (a = Empty_String);

Corollary Is_Substring_5:
For all a : SStr,
For all m, n : N,
Is_Substring( Prt_Btwn(m, n, a), a );

-- These are specialized versions for Prime_Str

Corollary Is_Substring_Primes_1:
For all p, s : Prime_Str,
(p = s) = Is_Substring(p, s);

Corollary Not_Eq_Str_Length:
For all S, T : SStr,
|S| /= |T| implies S /= T;

end String_Theory;

H.2 Set Theory

Precis Set_Theory;
uses Cls_Theory;

-- This is a place holder theory used for VC generation
-- It is missing definitions and theorems
Type Theorem Powerset_1:
For all S : SSet,
For all T : Powerclass(S),
T: SSet;

Type Theorem Powerset_2:
For all S : SSet,
For all T : Powerclass(S),
T: Powerset(S);
\end{verbatim}
Definition (S : SSet) union (T : SSet) : SSet;
Definition (S : SSet) intersection (T : SSet) : SSet;
Definition (e : Entity) is_in (S : SSet) : B;
Definition (e : Entity) is_not_in (S : SSet) : B; -- = not (e is_in S);
Definition complement (S : SSet) : SSet;
Definition (S : SSet) without (T : SSet) : SSet;
Definition (S : SSet) is_subset_of (T : SSet) : B;
Definition (S : SSet) is_not_subset_of (T : SSet) : B;
Definition (S : SSet) is_proper_subset_of (T : SSet) : B;
Definition (S : SSet) is_not_proper_subset_of (T : SSet) : B;
Definition Singleton(e : Entity) : SSet; -- {(e: Entity)}

Corollary Union_1_a: -- Is_Identity_for(union, Empty_Set);
For all S: SSet,
   Empty_Set union S = S;

Corollary Union_1_b: -- Is_Identity_for(o, Empty_String);
For all S: SSet,
   S union Empty_Set = S;

Corollary Concatenation_2: -- Is_Associative(union);
For all S, T, U: SSet,
   S union (T union U) = (S union T) union U;

end Set_Theory;

H.3 Set_App_Op_Ext

Precis Set_App_Op_Ext;
uses Set_Theory, Natural_Number_Theory;

-- This is a place holder theory used for VC generation
-- It is missing definitions and theorems
-- Some definitions and results are included in the dissertation chapters

Definition Card: SSet;
Definition ||{|S:SSet}||: N; -- YS: Should be a Card.
Definition FinPowerset(S: SSet): SSet;

Type Theorem Fin_0:
For all S: SSet,
   For all T: FinPowerset(S),
      T: SSet;

Type Theorem Fin_1:
For all S: SSet,
   For all T: FinPowerset(S),
      T: Powerset(S);

Definition Fn_Restrict_to(f : (D : SSet) -> (R : SSet), S : Powerset(D)) : S -> R;

254
-- Since the compiler does not parse the original definition below,
-- we create an alternative version.
-- Definition (f : (D : SSet) -> (R : SSet)) [S : Powerset(D)] : Powerset(R);
Definition SqBr(f : (D : SSet) -> (R : SSet), S : Powerset(D)) :
  Powerset(R);

end Set_App_Op_Ext;

H.4 Closure_Op_Ext

Precis Closure_Op_Ext;
  uses Set_Theory, Set_App_Op_Ext;

-- This is a place holder theory used for VC generation
-- It is missing definitions and theorems
-- Some definitions and results are included in the dissertation chapters

Definition Is_Closed_wrt(U : SSet, FC: Powerset(U -> U), S: Powerset(U)) :
  B;

Definition Closure_for(U: SSet, FC: Powerset(U -> U), G: Powerset(U)) :
  Powerset(U);

end Closure_Op_Ext;

H.5 Terminal_Range_Op_Ext

Precis Terminal_Range_Op_Ext;
  uses Set_Theory, Set_App_Op_Ext, Closure_Op_Ext;

-- This is a place holder theory used for VC generation
-- It is missing definitions and theorems
-- Some definitions and results are included in the dissertation chapters

Definition Is_Stable_wrt(U: SSet, FC: Powerset(U -> U), S : Powerset(U)) :
  B;

Definition Terminal_Range(U : SSet, FC : Powerset(U -> U), G : Powerset(U)) :
  Powerset(U);

end Terminal_Range_Op_Ext;
Bibliography


