Enhanced Specification Expressivity for Verification with Separation Logic

Cristina David

(B.Sc. in Computer Engineering, University Politehnica of Bucharest, Romania)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

May 2012

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Dr. Chin Wei-Ngan, who's thoughtful guidance helped me find my path in the research world. His patience, kindness, knowledge profoundly touched me during my Ph.D. years. I surely would have been lost without his encouragement, support, advice. There is no doubt in my mind that I had the best advisor ever!

I am also thankful to my Ph.D. committee, Dr. Khoo Siau Cheng and Dr. Jin Song Dong, for their helpful comments throughout my Ph.D. candidature. I wish to thank Dr. Shengchao Qin for a prosperous research collaboration, and Dr. Kwangkeun Yi for giving me the opportunity to visit Seoul National University, which was an enriching experience.

I am grateful to my colleagues from the PLS lab for providing a remarkable learning environment. I especially want to thank Andreea, Cristi, David, Florin, Hai, Narcisa, Yami. Our discussions helped me find answers to my research questions, and made my days at work extremely joyful. Working with them was a real pleasure. I am mostly thankful to Corneliu for his support and valuable comments on my thesis.

I would also want to thank Alberto, Aleks, Dmitry, Mihai, Nandini, Sara, Trang, for making Singapore feel like home. They helped me understand that, when it comes to friendship, there is no such thing as cultural differences. Throughout the years, they became my remote family!

I am also grateful to my yoga friends Alice, Celeste, Hwee Koon, Jane, Karry, Wai Ching, Mr. and Mrs. Chua for always encouraging me and making me feel cared after. Furthermore, Mayuko always inspired me through her determination to overcome any obstacle. I especially want to thank my yoga teacher, master Vicky, who taught me the passion for yoga, and the meaning of dedication, perseverance, integrity. His advices helped me see that all the limitations live in my mind, and there is nothing that I cannot achieve, once I set my mind free.

I wish to thank my parents for providing a loving environment and always supporting me.

Thank you everyone!

TABLE OF CONTENTS

SUMMARY

Traditionally, the focus of specification mechanism has been on improving its ability to cover a wider range of problems more accurately, while the effectiveness of verification is left to the underlying provers. In this thesis, we attempt a novel approach, where the focus is on determining a good specification mechanism to achieve better expressivity (the specification should capture more accurately and concisely the functionality and applicability of the corresponding code) and verifiability (the verification process should succeed in more scenarios than the corresponding verification without the specification enhancements, with better or at least similar performance). In particular, we develop three new specification mechanisms, which, besides improving the specification, are meant to assist during the verification process itself.

We begin by investigating the benefits of immutability annotations in the specification for allowing more flexible handling of aliasing, as well as more precise and concise specifications. Our approach supports finer levels of control that can localize and mark parts of a data structure as being immutable through the use of annotations on predicate and data declarations. By using such annotations to encode immutability guarantees, we expect to obtain better specifications that can more accurately describe the intentions, as well as prohibitions, of the method. Ultimately, our goal is improving the precision of the verification process. We have designed and implemented a new entailment procedure to formally and automatically reason about immutability enhanced specifications. We have also formalised the soundness for our new procedure through an operational semantics with mutability assertions on the heap. Additionally, we have carried out a set of experiments to both validate and affirm the utility of our current proposal on immutability enhanced specification mechanism.

Secondly, we notice that, often, a user has an intuition about the proving process. This thesis provides the necessary utensils for integrating this intuition in the specification. Instead of writing a flat (unstructured) specification, the user can use insights about the proof for writing a structured specification that will trigger different techniques during the proving process: (i) case analysis can be invoked to take advantage of disjointness conditions in the logic. (ii) early,

as opposed to late, instantiation can minimise on the use of existential quantification. (iii) formulae that are staged provide better reuse of the verification process. Initial experiments have shown that structured specifications can lead to more precise verification without incurring any performance overhead.

Lastly, we observe that one major issue about writing specifications for object-oriented (OO) programs is the fact that such specifications must adhere to behavioral subtyping in support of class inheritance and method overriding. However, this requirement inherently weakens the specifications of overridden methods in superclasses, leading to imprecision in program reasoning. To address this, we advocate for two types of specifications, one type that caters to calls with static dispatching, and one for calls with dynamic dispatching. We formulate a novel specification subsumption that can avoid code re-verification, where possible. Using a predicate mechanism, we propose a flexible scheme for supporting class invariant and lossless casting.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Computer programs (software) are present everywhere in our day to day life, and it is crucial for them to be dependable, especially in critical environments (aeronautics, automotive industry, banking, etc.). In 2002, the US Department of Commerce estimated that the cost to the US economy of avoidable software errors is between 20 and 60 billion dollars every year [117]. Consequently, a great effort has been put into software verification, in order to prove that software fully satisfies the expected requirements.

Software verification appears in two flavors, static and dynamic [38]. Dynamic verification (analysis) works by inspecting the executions of a given program. Examples of standard dynamic analysis are testing and profiling. The disadvantage of dynamic analysis is that it might not generalize to all the possible runs. The fact that the program has been found to behave in a certain manner for a set of possible inputs, might not signify that the behavior can be generalized for all the possible inputs.

While dynamic verification requires the running code, static verification (analysis) works at the program code level in order to reason about all possible behaviors that might arise at run time, regardless of the inputs provided or of the environment in which the program is being run [91, 48]. Hence, it can be applied earlier in development. One example of static analysis are the compiler optimizations. In order to cover all the possible execution paths, static analysis typically uses an abstracted model of the program state, which might lose some information. Consequently, the result of the analysis, while sound, might be less precise, providing false positives (issues which are reported but are not really defects). The goal of the research community is constructing a program verifier, which by using logical proof, can give an automatic check of the correctness of programs submitted to it [117, 54].

First formulations of the usage of logic for program verification were given by Floyd [43], and Hoare [53]. The main feature of Hoare logic is the Hoare triple, $\{p\}c\{q\}$, describing how the execution of a command c changes the state of the program from p to q. A problem faced by Hoare logic is establishing the correctness of programs that mutate data structures. These

programs typically require a storage that persists outside the call stack, namely the heap, and their correctness usually depends upon complex restrictions on the sharing in the data structures. As Hoare logic has to explicitly handle all the possible aliasing on the heap, scalability issues are likely to arise [107].

In order to deal with this shortcoming, Ishtiaq and O'Hearn [56] and Reynolds [107] designed separation logic, an extension to Hoare logic for reasoning about shared mutable data structures, i.e. data structures with updatable fields that can be referenced from more than one point. Separation logic assertions describe states, which contain both the store (stack) and the heap. In order to simplify the aliasing issue, separation logic adds two new logical connectives, interpreted as follows:

- $p_1 * p_2$, where $*$ represents the separating conjunction, and denotes the fact that the heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other. Basically, the separating conjunction has the non-aliasing information built in.

- $p_1 -\!\!* p_2$, where $-\!\!*$ represents the separating implication for denoting the fact that if the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

For illustration, if we compare $p_1 * p_2$ and $p_1 \wedge p_2$, the novelty introduced by the separating conjunction over the logical conjunction is the fact that, in the former case, p_1 and p_2 are required to point to disjoint pieces of heap. Thus, there is no need to explicitly consider the aliasing between them. On the other hand, in the latter case, p_1 and p_2 can be either aliased, or disjoint.

By using the separating conjunction, local specifications can be extended, as illustrated by the frame rule:

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

where no variable occurring free in r is modified by c.

With the help of the frame rule, a local specification can be extended with arbitrary predicates about variables and heap cells that are not modified by command c. By local specification we mean a specification involving only the variables and heap cells that are actually used by the command c (the footprint of c). Basically, the frame rule says that in order to understand how a program works, the specification should only refer to the cells that the program actually

accesses. All the other heap cells automatically remain unchanged. Through this frame rule, a specification of the heap being used by c can be arbitrarily extended as long as free variables of the extended part are not modified by c.

By the use of separation logic, the heap memory assertions can be made more precise (with the help of must-aliases implied by the separating conjunction) and concise (with the help of frame conditions).

From the moment when separation logic was proposed, lots of automated reasoning tools based on this logic were developed [8, 45, 92, 57]. The use of the separation logic formalism has been further extended for termination proofs [15], concurrency [119, 120, 47, 46], interprocedural shape analysis [45, 18, 109], verifying overlapping structures [76, 52], Java verification [35, 101, 35].

1.1 About This Thesis

The current thesis applies to the area of static program verification, and makes use of the formalism of separation logic in order to verify properties of mutable data structures. The starting point of this thesis is the automated verification system proposed in [92]. As opposed to other works [7, 33], which have designed specialised solvers that work for a fixed set of predicates (e.g. the predicate `lseg` to describe a segment of linked-list nodes), the approach in [92] describes a verifier that works for user-defined shape predicates. Shape predicates are predicates specifying data structure shapes, as well as certain numerical properties of data structures, such as size and reachability.

The main concern of the current thesis is improving the precision and expressivity of the verification process. We start from the remark that most efforts on improving the verification process have been confined to the verification technology, an approach that may lead to more reliance on clever heuristics from the verification tools, and also more complex implementation for the verification tools themselves. In this thesis, we shall propose a novel approach towards improving the verification process that focuses on enhancing the specification mechanism instead. In particular, we advocate for enhancing the specification in order to capture the intention of the corresponding code in a more precise and concise manner:

- a more precise specification should capture more accurately the functionality and applicability of the corresponding code.

- more concise specification should be shorter than the specification prior to the enhancement.

This specification restructuring is not meant to only increase the readability of the specifications, but it should assist in the verification process. Correspondingly, the results in the current thesis provide evidence that, when put to good use, a more precise and concise specification mechanism leads to a more precise and more efficient verification:

- more precise verification means that it should succeed in more scenarios than the corresponding verification without the specification enhancements.
- more efficient verification means that it should be faster.

Next, we will illustrate the specification enhancements proposed by the current thesis through a running example, which verifies properties of an AVL tree. An AVL tree is a binary search tree such that, for each of its nodes, the balance factor is between -1 and 1 (the balance factor of a node is the difference between the height of its right subtree and the height of its left subtree). We first define a data node `node2`, as follows:

$$\texttt{data node2}\,\{\,\texttt{int val; int height; node2 right; node2 left;}\,\}$$

Each node is used to store the actual data in the `val` field, the maximum height of its subtrees in the `height` field, and references to the right and left subtrees in the `right` and `left` fields, respectively. Next, we provide a shape predicate for the AVL tree. The first version of this shape predicate conforms to the approach in [92], and it is given below. Subsequently, we will enhance this specification according to the directions pursued in the current thesis. The name of the predicate is `avl` and it captures the size property via `s`, the height via `h`, and the balance factor via `b`.

$$\begin{aligned}
&\texttt{root::avl}\langle \texttt{h},\texttt{s},\texttt{b}\rangle \equiv \texttt{root=null} \wedge \texttt{h=0} \wedge \texttt{s=0} \wedge \texttt{b=0}\\
&\vee\ \texttt{root::node2}\langle _,\texttt{h},\texttt{r},\texttt{l}\rangle * \texttt{r::avl}\langle \texttt{h}_1,\texttt{s}_1,\texttt{b}_1\rangle * \texttt{l::avl}\langle \texttt{h}_2,\texttt{s}_2,\texttt{b}_2\rangle \wedge \texttt{h=max}(\texttt{h}_1,\texttt{h}_2)\texttt{+1}\\
&\wedge\ \texttt{s=}\texttt{s}_1\texttt{+}\texttt{s}_2\texttt{+1} \wedge \underline{\texttt{h}_1\texttt{=}\texttt{h}_2\texttt{+1} \wedge \texttt{b=-1}}\\
&\vee\ \texttt{root::node2}\langle _,\texttt{h},\texttt{r},\texttt{l}\rangle * \texttt{r::avl}\langle \texttt{h}_1,\texttt{s}_1,\texttt{b}_1\rangle * \texttt{l::avl}\langle \texttt{h}_2,\texttt{s}_2,\texttt{b}_2\rangle \wedge \texttt{h=max}(\texttt{h}_1,\texttt{h}_2)\texttt{+1}\\
&\wedge\ \texttt{s=}\texttt{s}_1\texttt{+}\texttt{s}_2\texttt{+1} \wedge \underline{\texttt{h}_1\texttt{+1=}\texttt{h}_2 \wedge \texttt{b=1}}\\
&\vee\ \texttt{root::node2}\langle _,\texttt{h},\texttt{r},\texttt{l}\rangle * \texttt{r::avl}\langle \texttt{h}_1,\texttt{s}_1,\texttt{b}_1\rangle * \texttt{l::avl}\langle \texttt{h}_2,\texttt{s}_2,\texttt{b}_2\rangle \wedge \texttt{h=max}(\texttt{h}_1,\texttt{h}_2)\texttt{+1}\\
&\wedge\ \texttt{s=}\texttt{s}_1\texttt{+}\texttt{s}_2\texttt{+1} \wedge \underline{\texttt{h}_1\texttt{=}\texttt{h}_2 \wedge \texttt{b=0}}
\end{aligned}$$

Formula $\texttt{p::c}\langle\texttt{v}^*\rangle$ may denote either a points-to fact of the heap where c is a data node, or a shape (heap) predicate where c is a named, parameterized assertion over the heap. For both cases, $\texttt{v}^*$ denotes the arguments, and _ denotes an anonymous variable. For each shape predicate and data node, we distinguish the first parameter root, denoting a pointer to the specified data structure that guides data traversal.

The aforementioned inductive definition of the AVL tree consists of a base case corresponding to the situation when the tree is null ($\texttt{root=null}\wedge\texttt{h=0}\wedge\texttt{s=0}\wedge\texttt{b=0}$), and an inductive case consisting of the last three disjuncts in the definition. The constraints $\texttt{b=0}$, $\texttt{b=1}$, and $\texttt{b=}-1$ state that the tree is balanced, while constraints $\texttt{s=}\texttt{s}_1\texttt{+}\texttt{s}_2\texttt{+1}$ and $\texttt{h=max}(\texttt{h}_1,\texttt{h}_2)\texttt{+1}$ compute the size and height of the tree pointed by root, respectively. The $*$ connector ensures that the head node, the right and left subtrees reside in disjoint heaps. Existential quantifiers for local values and pointers, such as $\texttt{r},\texttt{l},\texttt{h}_1,\texttt{h}_2,\texttt{s}_1,\texttt{s}_2$ are implicitly assumed.

Let us first address the issue of the lack of structure of the aforementioned specification (this reaserch direction is pursued in more detail in Chapter 5). With a closer inspection, the reader might notice that the specification contains significant redundancy. More specifically, the only part changing between the last three disjuncts is the relation between the heights of the left and right subtrees, and, consequently, the balance factor (the underlined formula). Everything else is left unchanged. In order to remove the redundancy, the user may rewrite the same inductive definition as follows:

$$
\begin{array}{l}
\texttt{root::avl}\langle\texttt{h},\texttt{s},\texttt{b}\rangle \equiv \texttt{case}\\
\qquad\{\texttt{root=null} \Rightarrow \texttt{h=0}\wedge\texttt{s=0}\wedge\texttt{b=0};\\
\qquad\ \texttt{root}\neq\texttt{null} \Rightarrow \texttt{root::node2}\langle\_,\texttt{h},\texttt{r},\texttt{l}\rangle{*}\texttt{r::avl}\langle\texttt{h}_1,\texttt{s}_1,\texttt{b}_1\rangle{*}\texttt{l::avl}\langle\texttt{h}_2,\texttt{s}_2,\texttt{b}_2\rangle\\
\qquad\qquad\qquad\wedge\ \texttt{h=max}(\texttt{h}_1,\texttt{h}_2)\texttt{+1}\wedge\texttt{s=}\texttt{s}_1\texttt{+}\texttt{s}_2\texttt{+1}\ \texttt{then}\\
\qquad\qquad\qquad\texttt{case}\ \{\texttt{h}_1\texttt{=}\texttt{h}_2\texttt{+1}\Rightarrow\texttt{b=1};\\
\qquad\qquad\qquad\qquad\texttt{h}_1\texttt{+1=}\texttt{h}_2\Rightarrow\texttt{b=}-1;\\
\qquad\qquad\qquad\qquad\texttt{h}_1\texttt{=}\texttt{h}_2\Rightarrow\texttt{b=0}\}\}
\end{array}
$$

In the latter definition for the AVL tree, the following constructs proposed by the current thesis are being used:

- case constructs (denoted by the keyword case): employed in order to highlight the disjointedness conditions root=null and root≠null, and $\texttt{h}_1\texttt{=}\texttt{h}_2\texttt{+1}$, $\texttt{h}_1\texttt{+1=}\texttt{h}_2$, and

$\mathtt{h}_1{=}\mathtt{h}_2$, respectively.

- staged formula (denoted by the keyword `then`): ensures the reuse of the logical formula before the keyword,

$$\mathtt{root}{::}\mathtt{node2}\langle \_, \mathtt{h}, \mathtt{r}, \mathtt{l}\rangle * \mathtt{r}{::}\mathtt{avl}\langle \mathtt{h}_1, \mathtt{s}_1, \mathtt{b}_1\rangle * \mathtt{l}{::}\mathtt{avl}\langle \mathtt{h}_2, \mathtt{s}_2, \mathtt{b}_2\rangle$$
$$\wedge \mathtt{h}{=}\mathtt{max}(\mathtt{h}_1, \mathtt{h}_2){+}1 \wedge \mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}1,$$

  between the three branches corresponding to $\mathtt{h}_1{=}\mathtt{h}_2{+}1$, $\mathtt{h}_1{+}1{=}\mathtt{h}_2$ and $\mathtt{h}_1{=}\mathtt{h}_2$, respectively.

Take note that, at this point, we only highlight the syntactic implications of the new constructs in making the specification more readable and minimizing the redundancy. In Chapter 5, we will explain how the new constructs assist in obtaining a better verification from the point of efficiency and precision. Additionally, in Chapter 5, we will explain the third specification structuring enhancement, which provides a way for the user to specify the type of instantiation to be used for a given logical variable.

Another contribution of this thesis relies on the observation that, while the shape predicates in the specifications denote resources that can be always consumed, some data structures are only being read from (direction pursued in Chapter 4). Hence, we enhance the specification mechanism for capturing the immutability property of data structures and investigate how the verification process can take advantage of this knowledge. Consequently, our approach enables a more restricted access to data structures. Assuming one of the aforementioned definitions of the AVL tree, let us try to specify a method which computes the balance factor of the head node of an AVL tree. For this purpose, we define two methods:

- `get_height`, which returns the height of the AVL tree received as argument.
- `get_balance`, which computes the balance factor.

Note that the keyword `requires` introduces the method's precondition (the program state that must hold prior to the method's execution), whereas `ensures` precedes the method's postcondition (the program state that must hold just after the method's execution). Additionally, `res` is a special identifier used in the postcondition to denote the result of a method.

```
int get_height(node2 x)
  requires x::avl⟨h, s, b⟩
  ensures x::avl⟨h, s, b⟩ ∧ res=h ;
  { int lh, rh;
   if (x==null) then return 0;
  else {rh=get_height(x.right);
       lh=get_height(x.left);
       { if (rh≥lh) then return 1+rh else return 1+lh}}

int get_balance(node2 x)
  requires x::avl⟨h, s, b⟩
  ensures x::avl⟨h, s, b⟩ ∧ res=b ∧ −1≤b≤1;
  { return get_height(x.right)−get_height(x.left)}
```

The precondition of both `get_height` and `get_balance` assume an AVL tree of height `h`, size `s`, and balance factor `b`. The same predicate, $\texttt{x::avl}\langle \texttt{h}, \texttt{s}, \texttt{b}\rangle$, is also present in their postconditions, which suggests that an AVL tree of the same height, size and balance factor is being preserved by both methods. However, as these methods do not mutate the input tree, we would want to express a stronger property stating that exactly the same tree from the method's entry is being preserved at the method's exit. We propose to use an immutability annotation of the form `@I` to annotate the specification of the `AVL` tree in order to indicate that the tree pointed by `x` is not mutated by its method:

```
int get_height(node2 x)
  requires x::avl⟨h, s, b⟩@I
  ensures res=h ;

int get_balance(node2 x)
  requires x::avl⟨h, s, b⟩@I
  ensures res=b ∧ −1≤b≤1;
```

Each precondition states that the AVL tree pointed by `x` will only be read by the corresponding method. This indirectly ensures the preservation of the input AVL tree, which does not

need to be re-proven in the postcondition. The latter specifications given for `get_height` and `get_balance` methods are:

- more concise (or shorter) since there are fewer predicate in the postconditions.
- more precise (or accurate) since they capture the total preservation of the input AVL tree without resorting to the use of a more complex predicate.

As the final research direction of this thesis, we investigate the specification mechanism in an object oriented (OO) setting (direction pursued in Chapter 6). One major issue to consider when verifying OO programs is how to design a specification for a method that may be overridden by another method down the class hierarchy (a subclass might provide a specific implementation of the method), such that it conforms to behavioral subtyping. According to the behavioral subtyping requirement, an object of a subclass can always be passed to a location where an object of its superclass is expected, as the object from each subclass must subsume the entire set of behaviors from its superclass [80]. This requirement may lead to imprecision during program reasoning.

For illustration, let us define the AVL tree in our running example in an OO setting. For this purpose, we will provide three classes:

- a class Node2 denoting an element of the tree. This class has three fields: `val` representing the value stored in the node, and `right` and `left` for denoting the references to the right and left subtrees, respectively.
- a class BinaryTree with four fields: `root` denoting the reference to the head node, `h` representing the height of the tree, `s` for denoting the size of the tree, and `b` for denoting the balance factor of the head node. The class also provides a method `get_balance`, which returns the balance factor of the head node.

```
class Node2 {
  int val;
  Node2 right, left;
  Node2(int v) {
    val=v; right=null; left=null;
  }}

class BinaryTree {
  Node2 root;
  int h, s, b;
  BinaryTree(){
    root=null; h=0; s=0; b=0;
  }
  int get_balance() {
    return b;
  }}
```

Next, we design the specification of the `get_balance` method in class `BinaryTree` without worrying about any potential subclass that might override it. The `this` variable denotes the receiver of the method.

```
int BinaryTree.get_balance()
  requires this::BinaryTree⟨h, s, b⟩
  ensures this::BinaryTree⟨h, s, b⟩ ∧ res=b;
```

This specification is very precise as it was considered statically on a per method basis without concern for method overriding, and can be used whenever the actual type of the receiver is known (static dispatch). Now, let us assume we have a subclass `AVLTree` extending `BinaryTree`, which inherits method `get_balance`, but adds an additional constraint in its specification in order to make sure that the balance factor is between -1 and 1.

```
class AVLTree extends BinaryTree {
int get_balance()
  requires this::AVLTree⟨h, s, b⟩
  ensures this::AVLTree⟨h, s, b⟩ ∧ res=b ∧ −1≤b≤1;
}}
```

Getting back to the specification of `get_balance` in class `BinaryTree`, if we take into account the overriding of the `get_balance` method by its corresponding method in the `AVLTree` subclass, in order to adhere to behavioral subtyping, we may have to weaken the postcondition of `BinaryTree.get_balance` by adding the constraint $-1 \leq b \leq 1$.

```
void BinaryTree.get_balance()
  requires this::BinaryTree⟨s, h, b⟩
  ensures this::BinaryTree⟨s, h, b⟩ ∧ res=b ∧ −1≤b≤1;
```

Such changes make the specifications of the methods in superclasses less precise, and are carried out to ensure behavioral subtyping in order to handle calls with dynamic dispatch. Furthermore, these specifications must also cater to potential modifications that may occur in the extra fields of the subclasses. To address this, we advocate a new specification mechanism for the OO setting that focuses on the distinction and relation between specifications that cater to calls with static dispatching from those for calls with dynamic dispatching.

1.2 Contributions of the Thesis

After providing a short description of the research directions pursued by the current thesis, we highlight its contributions:

- **Immutability enhanced specifications (Chapter 4, first proposed in [28]).** We provide a more concise and precise specification mechanism that allows immutability annotations and heap sharing. We show how our proposal enables better precision and applicability of the specifications, as well as preservation of cut-points in support of modular analysis. In order to support and make use the immutability enhanced specifications, we make the following related contributions:

  - **Immutability Guarantees** We discuss several immutability guarantees that can be enforced through our approach. Among them, we differentiate between total immutability and partial immutability.
  - **Entailment Procedure** We have designed a new entailment procedure to automatically reason about immutability enhanced specifications and have carried out experiments for validating the proposal.

- **Structured specifications (Chapter 5, first proposed in [44]).** We propose to add new structures to specifications to achieve a better outcome for the verification of pointer-based programs. We have designed and implemented a new entailment procedure to formally and automatically reason about our enhanced specifications. The three new specification mechanisms that we propose are described next:

  The experimental results have shown that our proposal can lead to more precise verification with a performance gain.

  - **Case constructs** allow capturing different contexts of use by highlighting disjointedness conditions. Case analysis is conventionally captured as part of the proving process. The user typically indicates the program location where case analysis is to be performed [123]. This corresponds to performing a case analysis on some program state (or antecedent) of the proving process. In our approach, we provide a case construct to distinguish the input states of pre/post specifications instead. This richer specification can be directly used to guide the verification process.
  - **Staged formulae** allow the specification to be made more concise through sharing of common sub-formulae. Apart from better sharing, this also allows verification to be carried out incrementally over multiple (smaller) stages, instead of a single (larger) stage.
  - **Early vs. late instantiations** denote different types of bindings for the logical variables (of consequent) during the entailment proving process. Early instantiation is an instantiation that occurs at the first occurrence of its logical variable, while late instantiation occurs at the last occurrence of its logical variable. While late instantiation can be more accurate for variables that are constructed from inequality

constraints, early instantiation can typically be done with fewer existential quantifiers since instantiation converts these existential logical variables to quantifier-free form at an earlier point. We propose to use early instantiation, by default, and only to resort to late instantiation when explicitly requested by the programmer.

- **Static and dynamic specifications (Chapter 6, first proposed in [22]).** We advocate for the coexistence of static and dynamic specifications, with an emphasis on the former. This technique is important as the majority of method dispatch operations (71%) are indeed statically known [3]. We impose an important subsumption relation between the static and the dynamic specifications. This principle allows for improved precision, while keeping code re-verifications to a minimum. While building up the necessary framework for the use of static and dynamic specifications, the following related contributions were achieved:

  - **Enhanced Specification Subsumption :** We improve on a classical specification subsumption relation. Apart from the usual checking for contravariance on preconditions and covariance on postconditions, we allow postcondition checking to be strengthened with the residual heap state from precondition checking. This enhancement is courtesy of the frame rule from separation logic which can improve modularity.
  - **Lossless Casting :** We use a new object format that allows lossless casting to be performed. This format supports both partial views and full views for objects of classes that are suitable for static and dynamic specifications, respectively.
  - **Statically-Inherited Methods :** New specifications may be given for inherited methods but must typically be re-verified. To avoid the need for re-verification, we propose for specification subsumption to be checked between each new static specification of the inherited method in a subclass against the static specification of the original method in the superclass. We identify a special category of statically-inherited methods that can safely avoid code re-verification for static specifications.
  - **Deriving Specifications :** We propose techniques to derive dynamic specifications from static specifications, and show how refinement can be carried out to ensure behavioral subtyping.

1.3 Thesis Overview

After introducing some background notions in Chapter 2, the subsequent chapters will describe our specification enhancements, as follows:

- Chapter 3 presents a summary of the most relevant related works.
- Chapter 4 investigates the benefits of immutability annotations for allowing more flexible handling of aliasing, as well as more precise and concise specifications;
- Chapter 5 presents our work on introducing structured specifications;
- Chapter 6 describes our distinction between static and dynamic specifications.

Note that we consider the programming language in Sec 2.1, the specification language in Sec 2.2, the forward verification rules in Sec 2.3, and entailment checking rules in Sec 2.4 as a reference for the verification techniques developed in this thesis. Accordingly, the corresponding sections of Chapters 4, 5, and 6 will only present the differences/enhancements from this reference point.

CHAPTER II

TECHNICAL BACKGROUND

In the current chapter, we provide a summary of the relevant technical background. We assume the reader is familiar with first-order logic, Presburger arithmetic, bag theory. More specifically, we explain some of the technical notions that we use in the current dissertation:

- the programming language.
- the specification language, with an emphasis on user-defined predicates.
- the forward verification procedure.
- the entailment checking procedure.
- semantic issues, including the storage model, the semantic model, and the dynamic semantics.

2.1 Programming Language

In this section, we introduce a core imperative language, which is given in Figure 2.1.

For simplicity, we shall assume that programs and specification formulas we use are well-typed. To simplify the presentation but without loss of expressiveness, we allow only one-level field access like `v.f` (rather than $\texttt{v.f}_1\texttt{.f}_2$...), and we allow only boolean variables (but not expressions) to be used as the test conditions for conditionals. The language supports data type declaration via *datat*, and shape predicate definition via *spred*. The syntax for shape predicates is given in the next section.

The following data node declarations can be expressed in our language and will be used as examples throughout this chapter. Note that they are recursive data declarations with different numbers of fields.

```
data node { int val; node next }
data node2 { int val; node2 prev; node2 next }
data node3 { int val; node3 left; node3 right; node3 parent }
```

$$
\begin{array}{lll}
P & ::= & \mathit{tdecl}^* \ \mathit{meth}^* \\
\mathit{tdecl} & ::= & \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= & \texttt{data}\ c\ \{\ \mathit{field}^*\ \} \\
\mathit{field} & ::= & t\ v \\
t & ::= & c \mid \tau \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{void} \\
\mathit{meth} & ::= & t\ \mathit{mn}\ ((\texttt{ref}\ t\ v)^*, (t\ v)^*)\ \mathit{mspec}\ \{e\} \\
e & ::= & \texttt{null} \mid k^{\tau} \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \texttt{new}\ c(v^*) \\
 & & \mid e_1; e_2 \mid t\ v;\ e \mid \mathit{mn}(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{return}\ e
\end{array}
$$

Figure 2.1: A Core Imperative Language

Each method *meth* is associated with a pre/post specification *mspec*, the syntax of which will be given in the next section. For simplicity, we assume that variable names declared inside each method are all distinct.

Pass-by-reference parameters are marked with `ref`. In a pass-by-reference evaluation, a method receives a reference to a variable used as argument, rather than a copy of its value. For formalization convenience, all the pass-by-reference parameters are grouped together. As an example of pass-by-reference parameters, the following function allows the actual parameters of $\{\texttt{x}, \texttt{y}\}$ to be swapped at its callers' sites.

$$
\begin{array}{l}
\texttt{void swap(ref node2 x, ref node2 y)} \\
\quad \ldots \\
\qquad \{\ \texttt{node2 z:=x;\ x:=y;\ y:=z}\ \}
\end{array}
$$

Furthermore, these parameters allow each iterative loop to be directly converted to an equivalent tail-recursive method, where mutation on parameters are made visible to the caller via pass-by-reference. This technique of translating away iterative loops is standard and is helpful in further minimising our core language. Note that we use an expression-oriented language where the last subexpression (e.g. $\texttt{e}_2$ from $\texttt{e}_1;\texttt{e}_2$) denotes the result of an expression. The missing method specifications, denoted by $mspec$, are described in the next section.

The standard insertion sort algorithm can be written in our language as follows:

```
node insert(node x, node vn)
{ if (vn.val≤x.val)
  then { vn.next:=x; return vn }
  else if (x.next=null) then
    { x.next:=vn; vn.next:=null; return x }
  else { x.next:=insert(x.next, vn); return x }}

 node insertion_sort(node y)
{ if (y.next=null) then return y
  else {
   y.next:=insertion_sort(y.next);
  return insert(y.next, y)}}
```

The `insert` method takes a sorted list `x` and a node `vn` that is to be inserted in the correct location of its sorted list. The `insertion_sort` method recursively applies itself (sorting) to the tail of its input list, namely `y.next`, before inserting the first node, namely `y`, into its now sorted tail.

2.2 Specification Language

A program P consists of declarations $tdecl$ and methods $meth$. Declarations can be shape predicates $spred$ or object types $objt$. Each method is decorated with the specification $\{\texttt{requires}\ \Phi^i_{pr}\ \ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}$, which is made up of a collection of pre- and post-condition pairs. The reason for supporting multiple pre- and post-conditions is that, given the rich variety of shapes that can be specified, there are often multiple ways of viewing a methods behaviour. The intended meaning is that whenever the method is called in a program state satisfying precondition Φ^i_{pr} and if the method terminates, the resulting state will satisfy the corresponding postcondition Φ^i_{po}. We handle `while` loop in a similar way. Other constructs are standard.

Primed notation is used to capture the latest value of local variables and may appear in the

Shape pred.	*spred*	$::= c\langle v^* \rangle \equiv \Phi$ `inv` π
Method spec.	*mspec*	$::= \{$`requires` Φ^i_{pr} `ensures` $\Phi^i_{po}\}_{i=1}^{p}$
Formula	Φ	$::= \bigvee(\exists v^* \cdot \kappa \wedge \pi)^*$
Pure formula	π	$::= \gamma \wedge \phi$
Ptr. equality/disequality	γ	$::= v_1{=}v_2 \mid v{=}$`null` $\mid v_1{\neq}v_2 \mid v{\neq}$`null` $\mid \gamma_1 \wedge \gamma_2$
Heap formula	κ	$::=$ `emp` $\mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2$
	Δ	$::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta$
	ϕ	$::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$
	b	$::=$`true` $\mid$ `false` $\mid v \mid b_1{=}b_2$
	a	$::= s_1{=}s_2 \mid s_1{\leq}s_2$
Presburger arith.	s	$::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1{+}s_2 \mid -s$ $\mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid \lvert B \rvert$
Bag constraint	φ	$::= v{\in}\mathcal{B} \mid \mathcal{B}_1{=}\mathcal{B}_2 \mid \mathcal{B}_1 \sqsubset \mathcal{B}_2 \mid \forall v{\in}\mathcal{B}{\cdot}\phi \mid \exists v{\in}\mathcal{B}{\cdot}\phi$
	$\mathcal{B}$	$::= \mathcal{B}_1 \sqcup \mathcal{B}_2 \mid \mathcal{B}_1 \sqcap \mathcal{B}_2 \mid \mathcal{B}_1 - \mathcal{B}_2 \mid \{\} \mid \{v\}$

Figure 2.2: The Specification Language

postcondition of loops. For example :

```
while x<0
requires true
ensures (x>0∧x′=x) ∨ (x≤0∧x′=0);
      do { x:=x+1 }
```

Here x and x′ denote the old and new values of variable x at the entry and exit of the loop, respectively.

The separation formulas we use are in a disjunctive normal form (eg. Φ, Φ_{pr}, Φ_{po} in Figure 2.2). Each disjunct consists of a $*$-separated heap constraint κ, referred to as heap part, and a heap-independent formula π, referred to as pure part. The pure part does not contain any heap nodes and is presently restricted to pointer equality/disequality γ, Presburger arithmetic s, ϕ ([105]) and bag constraint φ, ϕ. Furthermore, Δ denotes a composite formula that could always be safely translated into the Φ form which captures a disjunct of heap states, denoted

by κ, that are in separation conjunction.[1] The constraint domains ϕ for properties are currently chosen, due to the availability of the corresponding solvers.

2.2.1 User-defined Predicates

In order to verify properties of the linked data structures handled by a program, we must have a description/specification of those properties. A shape predicate is such a possibly inductive definition of the consistency and correctness properties of a data structure. Throughout the thesis we might refer to a shape predicate as a heap predicate, or simply a predicate.

Some automated reasoning systems [7, 10] are designed to work with only a small set of fixed predicates. However, it is impossible to provide specifications for all possible data structures. In our approach, we allow users to define their own specifications for data structures. User-definable shape predicates provide us with more flexibility than other automated reasoning systems [7, 10] as users can capture multiple aspects of linked data structures, such as their shapes, their numerical constraints and their contents constraints.

We provide below the predicate for an acyclic linked list (that terminates with a `null` reference):

$$\begin{array}{lcl} \texttt{root::ll}\langle\texttt{n}\rangle & \equiv & (\texttt{root=null}\wedge\texttt{n=0})\vee \\ & & (\exists \texttt{i},\texttt{m},\texttt{q}\cdot \texttt{root::node}\langle\texttt{i},\texttt{q}\rangle{*}\texttt{q::ll}\langle\texttt{m}\rangle\wedge\texttt{n=m+1}) \\ \texttt{inv n}\geq\texttt{0} & & \end{array}$$

From the notation point of view, in separation logic [107, 56], the formula $\texttt{p}\mapsto[\texttt{val}:\texttt{3},\texttt{next}:\texttt{l}]$ represents a singleton heap referred to by `p`, where $[\texttt{val}:\texttt{3},\texttt{next}:\texttt{l}]$ is a data record containing fields `val` and `next`. On the other hand, separation logic also uses predicate formulas to denote more complicated shapes, e.g. $\texttt{lseg}(\texttt{p},\texttt{q})$ represents list segments from `p` to `q`. In our system, we unify these two different representations into one form: $\texttt{p::c}\langle\texttt{v}^*\rangle$. When `c` is a data type name, $\texttt{p::c}\langle\texttt{v}^*\rangle$ stands for a singleton heap $\texttt{p}\mapsto[(\texttt{f:v})^*]$ where $\texttt{f}^*$ are fields of data declaration `c`. When `c` is a predicate name, $\texttt{p::c}\langle\texttt{v}^*\rangle$ stands for the predicate formula $\texttt{c}(\texttt{p},\texttt{v}^*)$. The reason we distinguish the first parameter from the rest is that each predicate has an implicit parameter `root` as its first parameter. Effectively, this is a "root" pointer to the specified data structure that guides data traversal and facilitates the definition of well-founded predicates (given in Sec 2.2).

Getting back to the shape predicate $\texttt{root::ll}\langle\texttt{n}\rangle$, the parameter `n` captures a *derived* value

[1] This translation is elaborated later in Figure 2.4.

that denotes the length of the acyclic list starting from root pointer. The above definition asserts that an ll list can be empty (the base case root=null) or consists of a head data node (specified by root::node⟨i, q⟩) and a separate tail data structure which is also an ll list (q::ll⟨m⟩). The $*$ connector ensures that the head node and the tail reside in disjoint heaps. We also specify a default invariant n≥0 that holds for all ll lists. (This invariant can be verified by checking that each disjunctive branch of the predicate definition always implies its stated invariant. In the case of ll predicate, the disjunctive branch with $n = 0$ implies the given invariant n≥0. Similarly, the $n = m + 1$ branch together with m≥0 from the invariant of q::ll⟨m⟩ also implies the given invariant n≥0.) Our predicate uses existential quantifiers for local values and pointers, such as i, m, q. The syntax for inductive shape predicates is given in Figure 2.2. For each shape definition *spred*, the heap-independent invariant π over the parameters $\{root, v^*\}$ holds for each instance of the predicate. Types need not be given in our specification as we have an inference algorithm to automatically infer non-empty types for specifications that are well-typed. For the ll predicate, our type inference can determine that m, n, i are of int type, while root, q are of the node type. As the construction of type inference algorithm is quite standard for a language without polymorphism, its description is omitted in the current thesis.

Regarding the notation, in the rest of the thesis we use underscore _ to denote an anonymous variable. Non-parameter variables (including anonymous variables) in the RHS of the shape definition, such as q, are existentially quantified. Furthermore, terms may be directly written as arguments of shape predicate or data node, while the root parameter on the LHS can be omitted as it is an implicit parameter that must be present for each of our predicate definitions. By using these conventions, a more complex shape, doubly linked-list with length n, is described by:

$$\texttt{dll}\langle \texttt{p}, \texttt{n}\rangle \quad \equiv \quad (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0) \vee (\texttt{root::node2}\langle \_, \texttt{p}, \texttt{q}\rangle * \texttt{q::dll}\langle \texttt{root}, \texttt{n}{-}1\rangle)$$
$$\textbf{inv}\ \texttt{n}{\geq}0$$

The dll shape predicate has a parameter p that represents the prev field of the first node of the doubly linked-list. It captures a chain of nodes that are to be traversed via the next field starting from the current node root. The nodes accessible via the prev field of the root node are not part of the dll list. This example also highlights some shortcuts we may use to make shape specifications shorter. Our shape predicates can describe not only the *shape* of data structures, but also their *size* and *bag* properties. (Examples with bag properties will be described later in Sec 2.2.3.) This capability enables many applications, including those requiring the support for

data structures with more complex invariants. For example, we may define a non-empty sorted list as below. The predicate also tracks the length, the minimum and maximum elements of the list.

$$\texttt{sortl}\langle \texttt{n}, \texttt{min}, \texttt{max}\rangle \equiv (\texttt{root::node}\langle \texttt{min}, \texttt{null}\rangle \wedge \texttt{min}{=}\texttt{max} \wedge \texttt{n}{=}1)$$
$$\vee (\texttt{root::node}\langle \texttt{min}, \texttt{q}\rangle * \texttt{q::sortl}\langle \texttt{n}{-}1, \texttt{k}, \texttt{max}\rangle \wedge \texttt{min}{\leq}\texttt{k})$$
$$\textbf{inv}\ \ \texttt{min}{\leq}\texttt{max} \wedge \texttt{n}{\geq}1$$

The constraint $\texttt{min}{\leq}\texttt{k}$ guarantees that sortedness property is adhered between any two adjacent nodes in the list. We may now specify (and then verify) the insertion sort algorithm mentioned earlier (see Sec 2.1 for the code) :

```
node insert(node x, node vn) where
  requires x::sortl⟨n, mi, ma⟩ * vn::node⟨v, _⟩
  ensures res::sortl⟨n+1, min(v, mi), max(v, ma)⟩;
node insertion_sort(node y)
  requires y::ll⟨n⟩ ∧ n>0
  ensures res::sortl⟨n, _, _⟩;
```

A special identifier `res` is used in the postcondition to denote the result of a method. The postcondition of `insertion_sort` shows that the output list is sorted and has the same number of nodes as the input list.

In this chapter, we use only separation conjunction, as we focus on only forward reasoning. This extension can help support more precise and concise reasoning for heap memory, as it can easily support must-aliasing and local reasoning. For example, when we specify that $\texttt{x::node}\langle 3, \texttt{y}\rangle * \texttt{y::node}\langle 5, \texttt{x}\rangle$ to be a precondition of some method, we can immediately determine that $\texttt{x}, \texttt{y}$ are non-aliased, namely $\texttt{x}{\neq}\texttt{y}$ due to the use of the separation conjunction, while $\texttt{x.next} = \texttt{y}$ and $\texttt{y.next} = \texttt{x}$ are must-aliases for the two fields from the heap formula. In contrast, if we had used the formula $\texttt{x::node}\langle 3, \texttt{y}\rangle \wedge \texttt{y::node}\langle 5, \texttt{x}\rangle$, we may not be able to determine if $\texttt{x}, \texttt{y}$ are aliased with each other, or not, Furthermore, due to the use of local reasoning, we can assume that *only* the heap memory specified in the precondition of each method is ever possibly modified by its method's body. This makes specifications using separation logic shorter by omitting the need to write **modifies** clauses that are necessary in traditional specification languages, such as JML [71] or Spec$^\sharp$[5]. In Chapter 4 we will relax the explicit aliasing requirement in order to allow arbitrary aliasing. Consequently, the use of $\wedge$ in the heap description will be

allowed for some cases.

2.2.2 Well-formedness Notions

As we have already seen, separation formulas are used in pre/post conditions and shape definitions. In order to handle them correctly without running into unmatched residual heap nodes, we require each separation constraint to be *well-formed*, as given by the following definitions:

Definition 2.2.1 (Accessible)**.** *A variable is* accessible *if it is a method parameter, or it is a special variable, either* `root` *or* `res`*.*

Definition 2.2.2 (Reachable)**.** *Given a heap constraint κ and a pointer constraint γ, the set of heap nodes in κ that are reachable from a set of pointers* `S` *can be computed by the following function.*

$$\begin{array}{lll} \texttt{reach}(\kappa,\gamma,\texttt{S}) & =_{df} & \texttt{p::c}\langle\texttt{v}^*\rangle * \texttt{reach}(\kappa - (\texttt{p::c}\langle\texttt{v}^*\rangle),\gamma,\texttt{S} \cup \{\texttt{v} | \texttt{v} \in \{\texttt{v}^*\}, \mathit{IsPtr}(\texttt{v})\}) \\ & & \texttt{if } \exists \texttt{q} \in \texttt{S} \cdot (\gamma \Longrightarrow \texttt{p=q}) \ \wedge \ \texttt{p::c}\langle\texttt{v}^*\rangle \in \kappa \\ \texttt{reach}(\kappa,\gamma,\texttt{S}) & =_{df} & \texttt{emp, otherwise} \end{array}$$

Note that $\kappa - (\texttt{p::c}\langle\texttt{v}^*\rangle)$ removes a term $\texttt{p::c}\langle\texttt{v}^*\rangle$ from κ, while IsPtr(`v`*) determines if* `v` *is of pointer type.*

Definition 2.2.3 (Well-Formed Formulas)**.** *A separation formula is well-formed if*

- *it is in a disjunctive normal form $\bigvee(\exists v^* \cdot \kappa_i \wedge \gamma_i \wedge \phi_i)^*$ where κ_i is for heap formula, and $\gamma_i \wedge \phi_i$ is for pure, i.e. heap-independent, formula, and*
- *all occurrences of heap nodes are reachable from its accessible variables,* `S`*. That is, we have $\forall i \cdot \kappa_i = \texttt{reach}(\kappa_i, \gamma_i, \texttt{S})$, modulo associativity and commutativity of the separation conjunction $*$.*

We also ensure that `root` can appear only in predicate bodies, `res` in postconditions. The primary significance of the *well-formed* condition is that all heap nodes of a heap constraint are reachable from accessible variables. This allows the entailment checking procedure to correctly match nodes from the consequent with nodes from the antecedent of an entailment relation.

Arbitrary recursive shape relations can lead to non-termination in unfold/fold reasoning. To avoid that problem, we propose to use only well-founded shape predicates in our framework.

Definition 2.2.4 (Well-Founded Predicates)**.** *A shape predicate is said to be well-founded if it satisfies the following conditions:*

- *its body is a well-formed formula,*
- *for all heap nodes p::$c\langle v^* \rangle$ occurring in the body, c is a data type name iff $p = \texttt{root}$.*

Note that the definitions above are syntactic and can easily be enforced. An example of well-founded shape predicates is `avl` - binary tree with near balanced heights, as follows :

$$\begin{array}{l} \texttt{avl}\langle \texttt{n}, \texttt{h} \rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{h}{=}0) \\ \quad \vee (\texttt{root}{::}\texttt{node2}\langle \_, \texttt{p}, \texttt{q} \rangle * \texttt{p}{::}\texttt{avl}\langle \texttt{n}_1, \texttt{h}_1 \rangle * \texttt{q}{::}\texttt{avl}\langle \texttt{n}_2, \texttt{h}_2 \rangle \\ \quad \wedge \texttt{n}{=}1{+}\texttt{n}_1{+}\texttt{n}_2 \wedge \texttt{h}{=}1{+}\texttt{max}(\texttt{h}_1, \texttt{h}_2) \wedge -1{\leq}\texttt{h}_1{-}\texttt{h}_2{\leq}1) \quad \textbf{inv } \texttt{n}, \texttt{h}{\geq}0; \end{array}$$

In contrast, the following three shape definitions are not well-founded.

$$\begin{array}{lcl} \texttt{foo}\langle \texttt{n} \rangle & \equiv & \texttt{root}{::}\texttt{foo}\langle \texttt{m} \rangle \wedge \texttt{n}{=}\texttt{m}{+}1 \\ \texttt{goo}\langle \rangle & \equiv & \texttt{root}{::}\texttt{node}\langle \_, \_ \rangle * \texttt{q}{::}\texttt{goo}\langle \rangle \\ \texttt{too}\langle \rangle & \equiv & \texttt{root}{::}\texttt{node}\langle \_, \texttt{q} \rangle * \texttt{q}{::}\texttt{node}\langle \_, \_ \rangle \end{array}$$

For `foo`, the `root` identifier is bound to a shape predicate. For `goo`, the heap node pointed by `q` is not reachable from variable `root`. For `too`, an extra data node is bound to a non-`root` variable. The first example may cause infinite unfolding, while the second example captures an unreachable (junk) heap that cannot be located by our entailment procedure. The last example illustrates the syntactic restriction imposed to facilitate termination of proof reasoning, which can be easily overcome by introducing intermediate predicates. For example, we may use:

$$\begin{array}{lcl} \texttt{too}\langle \rangle & \equiv & \texttt{root}{::}\texttt{node}\langle \_, \texttt{q} \rangle * \texttt{q}{::}\texttt{tmp}\langle \rangle \\ \texttt{tmp}\langle \rangle & \equiv & \texttt{root}{::}\texttt{node}\langle \_, \_ \rangle \end{array}$$

where `tmp` is the intermediate predicate added to satisfy our well-founded condition.

Our specification language allows bag/multiset properties to be specified in shape predicates and method specifications. This extra expressivity will be illustrated in Sec 2.2.3 by some examples.

2.2.3 Bag of Values/Addresses

The earlier specification of sorting from Sec 2.2 captures neither the in-situ reuse of memory cells nor the fact that all the elements of the list are preserved by sorting. The reason is that the

shape predicate captures only pointers and numbers but does not capture the set of reachable nodes in a heap predicate. A possible solution to this problem is to extend our specification mechanism to capture either a set or a bag of values. For generality and simplicity, we propose to only use the bag (or multi-set) notation that permits duplicates, though set notation could also be supported. In the rest of the thesis, we will use the following bag operators: bag union $\sqcup$, bag intersection $\sqcap$, bag subsumption $\sqsubset$, and bag cardinality $|\mathtt{B}|$. The shape specifications from the previous section are revised as follows:

$$\begin{array}{l}\mathtt{ll2}\langle \mathtt{n}, \mathtt{B}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\wedge \mathtt{n}{=}0\wedge \mathtt{B}{=}\{\})\\ \quad \vee(\mathtt{root}{::}\mathtt{node}\langle \_, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ll2}\langle \mathtt{n}{-}1, \mathtt{B}_1\rangle \wedge \mathtt{B}{=}\mathtt{B}_1\sqcup\{\mathtt{root}\})\\ \quad \textbf{inv}\ \mathtt{n}{\geq}0\wedge|\mathtt{B}|{=}\mathtt{n};\end{array}$$

$$\begin{array}{l}\mathtt{sortl2}\langle \mathtt{B}, \mathtt{mi}, \mathtt{ma}\rangle \equiv (\mathtt{root}{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{null}\rangle \wedge \mathtt{mi}{=}\mathtt{ma}\wedge \mathtt{B}{=}\{\mathtt{root}\})\\ \quad \vee\ (\mathtt{root}{::}\mathtt{node}\langle \mathtt{mi}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl2}\langle \mathtt{B}_1, \mathtt{k}, \mathtt{ma}\rangle \wedge \mathtt{B}{=}\mathtt{B}_1\sqcup\{\mathtt{root}\} \wedge\ \mathtt{mi}{\leq}\mathtt{k})\\ \quad \textbf{inv}\ \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{B}{\neq}\{\};\end{array}$$

Each predicate of the form $\mathtt{ll2}\langle \mathtt{n}, \mathtt{B}\rangle$ or $\mathtt{sortl2}\langle \mathtt{B}, \mathtt{mi}, \mathtt{ma}\rangle$ now captures a bag of addresses $\mathtt{B}$ for all the data nodes of its data structure (or heap predicate).With this extension, we can provide a more comprehensive specification for in-situ sorting, as follows :

$$\begin{array}{l}\mathtt{node\ insert(node\ x,\ node\ vn)\ where}\\ \quad \mathtt{requires\ x{::}sortl2}\langle \mathtt{B}, \mathtt{mi}, \mathtt{ma}\rangle * \mathtt{vn{::}node}\langle \mathtt{v}, \_\rangle\\ \quad \mathtt{ensures\ res{::}sortl2}\langle \mathtt{B}\sqcup\{\mathtt{vn}\}, \mathtt{min}(\mathtt{v}, \mathtt{mi}), \mathtt{max}(\mathtt{v}, \mathtt{ma})\rangle;\\ \{\cdots\}\\ \mathtt{node\ insertion\_sort(node\ y)\ where}\\ \quad \mathtt{requires\ y{::}ll2}\langle \mathtt{n}, \mathtt{B}\rangle \wedge \mathtt{B}{\neq}\{\}\\ \quad \mathtt{ensures\ res{::}sortl2}\langle \mathtt{B}, \_, \_\rangle;\\ \{\cdots\}\end{array}$$

We stress that this bag mechanism to capture the reachable nodes in a shape predicate is quite general. For example, instead of heap addresses, we may also revise our linked list view to capture a bag of reachable values, and its length, as follows:

$$\begin{array}{l}\mathtt{ll3}\langle \mathtt{n}, \mathtt{B}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\wedge \mathtt{n}{=}0\wedge \mathtt{B}{=}\{\})\vee\\ \quad (\mathtt{root}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ll3}\langle \mathtt{n}{-}1, \mathtt{B}_1\rangle \wedge \mathtt{B}{=}\mathtt{B}_1\sqcup\{\mathtt{a}\})\\ \quad \textbf{inv}\ \mathtt{n}{\geq}0 \wedge |\mathtt{B}|{=}\mathtt{n};\end{array}$$

Capturing a bag of values allows us to reason about the collection of values in a data structure, and permits relevant properties to be specified and automatically verified (when equipped with an appropriate constraint solver), as highlighted by two examples below:

```
data pair{node v1; node v2}

pair partition(node x, int p)
requires x::ll3⟨n, A⟩
ensures res::pair⟨r1, r2⟩ * r1::ll3⟨n1, B1⟩*r2::ll3⟨n2, B2⟩
∧A=B1⊔B2 ∧ n=n1 + n2 ∧ (∀a∈B1·a≤p)∧(∀a∈B2·a>p);
{ if (x=null) then new pair(null, null)
  else { pair t; t:=partition(x.next, p);
         if (x.val≤p) then { x.next:=t.v1; t.v1:=x }
                      else { x.next:=t.v2; t.v2:=x };
         t } }

bool allPos(node x) where
requires x::ll3⟨n, B⟩
ensures x::ll3⟨n, B⟩ ∧ ((∀a∈B·a≥0)∧res ∨ (∃a∈B·a<0)∧¬res);
{ if (x=null) then true
  else if (x.val<0) then false else allPos(x.next) }
```

Note that both universal and existential properties over bags can be expressed. The first example returns a pair of lists that have been partitioned from a single input list according to an integer pivot. This partition function and its pre/post specification can be used to prove the total correctness of the quicksort algorithm. The second example uses existentially and universally quantified formulae to determine if at least one negative number is present in an input list, or not. These specifications are somewhat expressive, but can be easily handled by our separation logic prover in conjunction with relevant classical provers, such as MONA [95] and Isabelle [64].

2.3 Forward Verification

The front-end of the system is a standard Hoare-style forward verifier, which invokes the entailment prover. In this section, we present the forward verifier which comprises a set of forward

verification rules to systematically check that the precondition is satisfied at each call site, and that the declared postcondition is successfully verified (assuming the given precondition) for each method definition. The back-end entailment prover will be given in Sec 2.4.

Program verification is typically formalised using Hoare triples of form $\{pre\}code\{post\}$, where pre and post are the initial and final states of the program code in some logic (separation logic in our case). We use P to denote the program being checked. With pre/post conditions declared for each method in P, we can now apply modular verification to its body using Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$. These are forward verification rules as we expect Δ_1 to be given before computing Δ_2. To capture proof search, we generalize the forward rule to the form $\vdash \{\Delta_1\}\, e\, \{S\}$ where S is a set of heap states, discovered by a search-based verification process. When S is empty, the forward verification is said to have failed for Δ as prestate.

For convenience, we also provide lifted variant of the forward verifier to take a set of prestates. Verification in such a case succeeds if any of the prestates gives rise to a successful verification, that is if at least one of the S_i is non-empty. This rule is useful when the forward verifier has processed at least one subexpression, potentially giving rise to a set of residual states.

$$\frac{\forall i \in 1..n \cdot \{\Delta_i\}\ \text{code}\ \{S_i\}}{\vdash \{\{\Delta_1, .., \Delta_n\}\}\ \text{code}\ \{\bigcup_{i=1}^{n} S_i\}}$$

Verification of a method starts with each precondition, and proves that the corresponding postcondition is guaranteed at the end of the method. The verification is formalized in the following rule:

$$\frac{\begin{array}{c}[\mathbf{FV{-}METH}]\\ V{=}\{v_m..v_n\} \quad W{=}\mathit{prime}(V)\\ \forall i = 1, .., p \ \cdot\ (\ \vdash \{\Phi^i_{pr} \wedge \mathit{nochange}(V)\}\, e\, \{S^i_1\}\\ (\exists W{\cdot}S^i_1) \vdash \Phi^i_{po} * S^i_2 \qquad S^i_2 \neq \{\})\end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\}}$$

The function *prime(V)* returns $\{v' \mid v \in V\}$. The predicate *nochange(V)* returns $\bigwedge_{v \in V}(v = v')$. If $V = \{\}$, *nochange(V)=true*. $\exists W \cdot S$ returns $\{\exists W \cdot S_i | S_i \in S\}$. The entailment $(\exists W{\cdot}S^i_1) \vdash \Phi^i_{po} * S^i_2$ is discharged by the entailment prover described in the next subsection.

At a method call, each of the method's precondition is checked. The combination of the residue S_i and the postcondition is added to the poststate. If a precondition is not entailed by the program state Δ, the corresponding residue is not added to the set of states. The test $\texttt{S}{\neq}\{\}$ ensures that at least one precondition is satisfied.

$$[\textbf{FV–CALL}]$$

$$\frac{\begin{array}{c} t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi_{pr}^i\ \ \texttt{ensures}\ \Phi_{po}^i\}_{i=1}^{p}\ \{e\} \in P \\ \rho{=}[v'_j/v_j]_{j=m}^{n} \qquad \Delta{\vdash}\rho\Phi_{pr}^i * S_i \quad \forall i{=}1,..,p \\ S = \bigcup_{i=1}^{p} S_i * \Phi_{po}^i \qquad S \neq \{\} \end{array}}{\vdash \{\Delta\}\, m(v_1..v_n)\, \{S\}}$$

Note that the verification rule also invokes the entailment prover to discharge $\Delta{\vdash}\rho\Phi_{pr}^i * S_i$, where ρ represents a substitution of v_j by v'_j, for all $j = 1,..,n$.. The lifted separation conjunction $*$ over a set (i.e., $S_i * \Phi_{po}^i$) is defined in Fig. 2.4.

Our verifier also ensures that each field access is safe from null dereferencing. This is shown in the field access rules in Fig. 2.3 which also includes other forward verification rules for the language. The verification rules attempt to track heap states, as accurately as possible, with path-sensitivity captured by [FV–IF] rule, flow-sensitivity by [FV–SEQ] rule and context sensitivity by the [FV–CALL] rule. In a nutshell, verification is carried out at three places. For each call site, the [FV–CALL] rule (mentioned earlier) ensures that at least one of its method's preconditions is satisfied. At each method definition, the [FV–METH] rule checks that every postcondition holds for the method body assuming its respective precondition. At each shape definition, [FV–SPRED] checks that its given invariant π_{inv} is sound w.r.t. (i.e. semantic consequence of) the well-formed heap formula Φ. (The rule for `while` loop is omitted but is essentially similar to the mechanics for handling tail-recursive methods.) The function $\mathit{XPure}_0(\Phi)$ generates a sound and heap-independent approximation of the heap constraint Φ. For instance,

$$\mathit{XPure}_0(\texttt{x::node}\langle\_,\_\rangle) \equiv \texttt{x} > 0$$

$$\mathit{XPure}_0(\texttt{x::node}\langle\_,\_\rangle * \texttt{y::node}\langle\_,\_\rangle) \equiv \texttt{x}{>}0{\wedge}\texttt{y}{>}0{\wedge}\texttt{x}{\neq}\texttt{y}$$

$$\mathit{XPure}_0(\texttt{x::lseg}\langle\texttt{p},\texttt{n}\rangle) \equiv \texttt{n} \geq 0$$

For the shape predicate case above, we can get a more precise approximation by unrolling the predicate definition once, for example:

$$\mathit{XPure}_1(\texttt{x::lseg}\langle\texttt{p},\texttt{n}\rangle) \equiv\ (\texttt{x}{=}\texttt{p}{\wedge}\texttt{n}{=}0 \vee \texttt{x}{>}0{\wedge}\texttt{n}{>}0)$$

$$\frac{\textit{XPure}_0(\Phi) \Longrightarrow [0/\texttt{null}](\pi_{inv})}{\vdash\ c\langle v^*\rangle \equiv \Phi\ \texttt{inv}\ \pi_{inv}}\ [\textbf{FV-SPRED}]$$

$$\frac{S{=}\{\Delta{\wedge}\texttt{res}{=}v'\}}{\vdash \{\Delta\}\, v\, \{S\}}\ [\textbf{FV-VAR}]$$

$$\frac{S = \{\Delta{\wedge}eq_\tau(\texttt{res}, k)\}}{\vdash \{\Delta\}\, k^\tau\, \{S\}}\ [\textbf{FV-CONST}]$$

$$\frac{\vdash \{\Delta\}\, e\, \{S\}}{\vdash \{\Delta\}\, \{t\ v;\ e\}\, \{\exists\, v, v'{\cdot}S\}}\ [\textbf{FV-LOCAL}]$$

$$\frac{\vdash \{\Delta{\wedge}v'\}\, e_1\, \{S_1\} \quad \vdash \{\Delta{\wedge}\neg v'\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{S_1{\vee}S_2\}}\ [\textbf{FV-IF}]$$

$$\frac{S{=}\{\Delta * \texttt{res}{::}c\langle v'_1, .., v'_n\rangle\}}{\vdash \{\Delta\}\, \texttt{new}\ c(v_1, .., v_n)\, \{S\}}\ [\textbf{FV-NEW}]$$

$$\frac{\vdash \{\Delta\}\, e_1\, \{S_1\} \quad \vdash \{S_1\}\, e_2\, \{S_2\}}{\vdash \{\Delta\}\, e_1; e_2\, \{S_2\}}\ [\textbf{FV-SEQ}]$$

$$\frac{\vdash \{\Delta\}\, e\, \{S_1\} \quad S_2{=}\exists\texttt{res}{\cdot}(S_1{\wedge}_{\{v\}}v'{=}\texttt{res})}{\vdash \{\Delta\}\, v{:=}e\, \{S_2\}}\ [\textbf{FV-ASSIGN}]$$

$$\frac{\Delta{\vdash}v'{::}c\langle v_{1..n}\rangle * S_1 \quad S_1{\neq}\{\} \quad \textit{fresh}\ v_1..v_n \quad S_2 = \exists v_1..v_n{\cdot}(S_1 * v'{::}c\langle v_{1..n}\rangle{\wedge}\texttt{res}{=}v_i)}{\vdash \{\Delta\}\, v.f_i\, \{S_2\}}\ [\textbf{FV-FIELD-READ}]$$

$$\frac{\Delta{\vdash}v'{::}c\langle v_{1..n}\rangle * S_1 \quad S_1{\neq}\{\} \quad \textit{fresh}\ v_1..v_n \quad S_2 = \exists v_1..v_n{\cdot}(S_1 * v'{::}[v'_0/v_i]c\langle v_{1..n}\rangle)}{\vdash \{\Delta\}\, v.f_i{:=}v_0\, \{S_2\}}\ [\textbf{FV-FIELD-UPDATE}]$$

Figure 2.3: Forward Verification Rules with Non-Determinism

The definition for the general approximation procedure $\textit{XPure}_n(\Phi)$ (also used in the entailment prover) can be found in Sec 2.4.4, where n denotes the number of unrollings done on the shape predicates.

The operators $\wedge_{\{v\}}$ (in assignment rule) and $*_W$ (in while rule) are *composition with update* operators. Given a state Δ_1, a state change Δ_2, and a set o'f variables to be updated $X{=}\{x_1, .., x_n\}$, the composition operator $\oplus_X$ is defined as:

$\Delta_1 \oplus_X \Delta_2 =_{df} \exists\, r_1..r_n \cdot \rho_1\, \Delta_1 \oplus \rho_2\, \Delta_2$

where $r_1, \ldots, r_n$ are fresh variables;

$\rho_1 = [r_i/x'_i]_{i=1}^n$; $\rho_2 = [r_i/x_i]_{i=1}^n$

Note that ρ_1 and ρ_2 are substitutions that link each latest value of x'_i in Δ_1 with the corresponding initial value x_i in Δ_2 via a fresh variable r_i. The binary operator $\oplus$ is either $\wedge$ or $*$.

Normalization rules for separation constraints and lifted operators over sets of states are given in Fig. 2.4. Note that the separation conjunction operator $*$ is commutative, associative, and distributive over disjunction. In separation logic, the separation conjunction between a formula and a pure (i.e. heap independent) formula is logically equivalent to a normal conjunction, i.e., $\Delta * \pi = \Delta \wedge \pi$. This justifies the third translation rule.

$$
\begin{array}{ll}
(\Delta_1 \vee \Delta_2) \wedge \pi & \rightsquigarrow (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi) \\
(\Delta_1 \vee \Delta_2) * \Delta & \rightsquigarrow (\Delta_1 * \Delta) \vee (\Delta_2 * \Delta) \\
(\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2) & \rightsquigarrow (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2) \\
(\kappa_1 \wedge \pi_1) \wedge (\pi_2) & \rightsquigarrow \kappa_1 \wedge (\pi_1 \wedge \pi_2) \\
(\gamma_1 \wedge \phi_1) \wedge (\gamma_2 \wedge \phi_2) & \rightsquigarrow (\gamma_1 \wedge \gamma_2) \wedge (\phi_1 \wedge \phi_2) \\
(\exists x \cdot \Delta) \wedge \pi & \rightsquigarrow \exists y \cdot ([y/x]\Delta \wedge \pi) \\
(\exists x \cdot \Delta_1) * \Delta_2 & \rightsquigarrow \exists y \cdot ([y/x]\Delta_1 * \Delta_2) \\
(S_1 \vee S_2) & \rightsquigarrow \{\Delta_1 \vee \Delta_2 \mid \Delta_1 \in S_1,\, \Delta_2 \in S_2\} \\
F(S) & \rightsquigarrow \{F(\Delta) \mid \Delta \in S\}
\end{array}
$$

where

$F(\mathcal{A}) ::= \mathcal{A} \wedge \pi \mid \mathcal{A} \wedge_W \pi \mid \mathcal{A} * \Delta \mid \mathcal{A} *_W \Delta \mid \exists x \cdot \mathcal{A}$

y denotes fresh variable

Figure 2.4: Normalization Rules for Separation Constraints and with Operators Lifted to a Set

2.3.1 Forward Verification Example

We present the detailed verification of the first branch of the `insert` method from Sec 2.1. Note that program variables appear primed in formulae to denote the latest values, whereas logical variables are always unprimed. The verification is straightforward, except for the last step, where a folding operation is invoked when we check an obtained disjunctive formula establishes the method's postcondition. The procedure to perform the unfolding/folding operations

is presented in Sec 2.4.

$\{\{\mathtt{x'}{::}\mathtt{sortl}\langle \mathtt{n},\mathtt{mi},\mathtt{ma}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\_\rangle\}\}$ // [FV−METH](*initialize precondition*)

if $(\mathbf{vn.val} \leq \mathbf{x.val})$ **then** $\{$

$\{\{(\mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\_\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi})$

$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\_\rangle$

$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi})\}\}$ // [FV−IF], [UNFOLDING](*Sec 2.4*)[2]

$\mathbf{vn.next} := \mathbf{x};$

$\{\{(\mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\mathtt{x'}\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi})$

$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\mathtt{x'}\rangle$

$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi})\}\}$ // [FV−FIELD−UPDATE]

$\mathbf{vn}$

$\{\{(\mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{null}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\mathtt{x'}\rangle \wedge \mathtt{mi}{=}\mathtt{ma} \wedge \mathtt{n}{=}1 \wedge \mathtt{v}{\leq}\mathtt{mi} \wedge \mathtt{res}{=}\mathtt{vn'})$

$\vee\ (\exists \mathtt{q},\mathtt{k}\cdot \mathtt{x'}{::}\mathtt{node}\langle \mathtt{mi},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sortl}\langle \mathtt{n}{-}1,\mathtt{k},\mathtt{ma}\rangle * \mathtt{vn'}{::}\mathtt{node}\langle \mathtt{v},\mathtt{x'}\rangle$

$\wedge \mathtt{mi}{\leq}\mathtt{k} \wedge \mathtt{mi}{\leq}\mathtt{ma} \wedge \mathtt{n}{\geq}2 \wedge \mathtt{v}{\leq}\mathtt{mi} \wedge \mathtt{res}{=}\mathtt{vn'})\}\}$ // [FV−VAR]

$\}$

$\{\{\mathtt{res}{::}\mathtt{sortl}\langle \mathtt{n}{+}1,\mathtt{min}(\mathtt{v},\mathtt{mi}),\mathtt{max}(\mathtt{v},\mathtt{ma})\rangle\}\}$

// [FV−METH](*checking postcondition*), [FOLDING](*Sec 2.4*)

2.4 Entailment Checking

Our prover for checking the entailment relation of separation formulae makes use the set of heap states to support non-deterministic entailment. By non-determinism, we mean a search process that returns multiple answers, any one of which indicates a successful verification. Our entailment prover is of the form $\Delta_A \vdash^{\kappa}_{V} \Delta_C * S$ which denotes $\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \Delta_C) * S$, where S is a set of possible residual poststates. The purpose of heap entailment is to check that heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and to compute the set of possible residual poststates S. The entailment succeeds when S is non-empty, otherwise it is deemed to have failed. κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified

[2]The rule [UNFOLDING] is to replace the predicate $\mathtt{x'}{::}\mathtt{sortl}\langle \mathtt{n},\mathtt{mi},\mathtt{ma}\rangle$ by its definition. The rule [FOLDING] used in the last step is to fold a formula which matches with a predicate's definition back to the predicate. Both rules will be discussed in detail in Sec 2.4.

variables from the consequent. Note that κ and V are derived. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. The entailment proving rules are given in Fig 2.5. We now briefly discuss the key steps that we may use in such an entailment proof.

2.4.1 Matching up heap nodes from the antecedent and the consequent

The procedure works by successively matching up heap nodes that can be proven aliased. As the matching process is incremental, we keep the successfully matched nodes from antecedent in κ for better precision. For example, consider the following entailment proof:

$$\dfrac{\dfrac{\dfrac{(((\texttt{p}{=}\texttt{null} \wedge \texttt{n}{=}0) \vee (\texttt{p}{\neq}\texttt{null} \wedge \texttt{n}{>}0)) \wedge \texttt{n}{>}0 \wedge \texttt{m}{=}\texttt{n}) \implies \texttt{p}{\neq}\texttt{null}}{(\mathit{XPure}_1(\texttt{p::ll}\langle\texttt{n}\rangle) \wedge \texttt{n}{>}0 \wedge \texttt{m}{=}\texttt{n} \implies \texttt{p}{\neq}\texttt{null}) \qquad S = \{(\texttt{n}{>}0 \wedge \texttt{m}{=}\texttt{n})\}}}{\texttt{n}{>}0 \wedge \texttt{m}{=}\texttt{n} \vdash^{\texttt{p::ll}\langle\texttt{n}\rangle} \texttt{p}{\neq}\texttt{null} * S}}{\texttt{p::ll}\langle\texttt{n}\rangle \wedge \texttt{n}{>}0 \vdash \texttt{p::ll}\langle\texttt{m}\rangle \wedge \texttt{p}{\neq}\texttt{null} * S}$$

Had the predicate $\texttt{p::ll}\langle\texttt{n}\rangle$ not been kept and used, the proof would not have succeeded since we require this predicate and $\texttt{n}{>}0$ to determine that $\texttt{p}{\neq}\texttt{null}$. Such an entailment would be useful when, for example, a list with positive length n is used as input for a function that requires a non-empty list.

Another feature of the entailment procedure is exemplified by the transfer of $\texttt{m}{=}\texttt{n}$ to the antecedent (and subsequently to the residue). In general, when a match occurs (rule [ENT−MATCH]) and an argument of the heap node coming from the consequent is free, the entailment procedure binds the argument to the corresponding variable from the antecedent and moves the equality to the antecedent. In our system, free variables in consequent are variables from method preconditions. These bindings play the role of parameter instantiations during forward reasoning, and can be accumulated into the antecedent to allow the subsequent program state (from residual heap) to be aware of their instantiated values. This process is formalized by the function *freeEqn* below, where V is the set of existentially quantified variables:

$$\mathit{freeEqn}([u_i/v_i]_{i=1}^n, V) =_{df} \texttt{ let } \pi_i = (\texttt{if } v_i{\in}V \texttt{ then true else } v_i{=}u_i) \texttt{ in } \bigwedge_{i=1}^n \pi_i$$

For soundness, we perform a preprocessing step to ensure that variables appearing as arguments of heap nodes and predicates are i) distinct and ii) if they are free, they do not appear in

the antecedent by adding (existentially quantified) fresh variables and equalities. This guarantees that the formula generated by *freeEqn* does not introduce any additional constraints over existing variables in the antecedent, as one side of each equation does not appear anywhere else in the antecedent.

Apart from the matching operation, another two essential operations that may be required in an entailment proof are (1) unfolding a shape predicate and (2) folding some data nodes back to a shape predicate. Unfold/fold operations can be used to handle well-founded inductive predicates in a deductive manner. They are normally invoked before a matching operation is invoked. In particular, we can unfold a predicate that appears in the antecedent if it co-relates (via aliasing) with a data node in the consequent. Correspondingly, if a predicate that appears in the consequent co-relates (via aliasing) with a data node in the antecedent, then we can fold the data node (perhaps together with other nodes) in the antecedent back to a shape predicate so that it can match with the predicate in the consequent. The well-founded condition is sufficient to ensure termination. We shall now use some examples to illustrate these two key steps.

2.4.2 Unfolding a shape predicate in the antecedent

Consider:

$$\mathtt{x}{::}\mathtt{ll3}\langle \mathtt{n}, \mathtt{B}\rangle \wedge \mathtt{n}{>}2 \;\vdash\; (\exists \mathtt{r} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{r}, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null} \wedge \mathtt{r} \in \mathtt{B}) * S$$

where S captures the set of possible residual states. Note that a predicate $\mathtt{x}{::}\mathtt{ll3}\langle \mathtt{n}, \mathtt{B}\rangle$ from the antecedent and a data node $\mathtt{x}{::}\mathtt{node}\langle \mathtt{r}, \mathtt{y}\rangle$ from the consequent are co-related via the same variable $\mathtt{x}$. For the entailment to succeed, we would first unfold the $\mathtt{ll3}\langle \mathtt{n}, \mathtt{B}\rangle$ predicate in the antecedent:

$$\begin{array}{l} \exists \mathtt{q}_1, \mathtt{v} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}_1\rangle * \mathtt{q}_1{::}\mathtt{ll3}\langle \mathtt{n}{-}1, \mathtt{B}_1\rangle \wedge \mathtt{n}{>}2 \wedge \mathtt{B}{=}\mathtt{B}_1 \cup \{\mathtt{v}\} \\ \quad \vdash (\exists \mathtt{r} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{r}, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null} \wedge \mathtt{r} \in \mathtt{B}) * S \end{array}$$

After removing the existential quantifiers, we obtain:

$$\begin{array}{l} \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}_1\rangle * \mathtt{q}_1{::}\mathtt{ll3}\langle \mathtt{n}{-}1, \mathtt{B}_1\rangle \wedge \mathtt{n}{>}2 \;\wedge\; \mathtt{B}{=}\mathtt{B}_1 \cup \{\mathtt{v}\} \\ \quad \vdash (\mathtt{x}{::}\mathtt{node}\langle \mathtt{r}, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null} \wedge \mathtt{r} \in \mathtt{B}) * S \end{array}$$

The data node in the consequent is then matched up, giving:

$$\mathtt{q}_1{::}\mathtt{ll3}\langle \mathtt{n}{-}1, \mathtt{B}_1\rangle \wedge \mathtt{n}{>}2 \wedge \mathtt{B}{=}\mathtt{B}_1 \cup \{\mathtt{v}\} \wedge \mathtt{q}_1{=}\mathtt{y} \;\vdash\; (\mathtt{q}_1{\neq}\mathtt{null} \wedge \mathtt{v} \in \mathtt{B}) * S$$

Due to the well-founded condition, each unfolding exposes a data node that matches with the data node in the consequent. Thus a reduction of the consequent immediately follows, which contributes to the termination of the entailment proving. A formal definition of the unfolding operation is given by the [UNFOLDING] rule in Figure 2.5.

2.4.3 Folding against a shape predicate in the consequent

Consider:

$$\texttt{x::node}\langle 1, \texttt{q}_1\rangle * \texttt{q}_1\texttt{::node}\langle 2, \texttt{null}\rangle * \texttt{y::node}\langle 3, \texttt{null}\rangle \vdash\ (\texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle \wedge \texttt{n}{>}1 \wedge 1 \in \texttt{B}) * S$$

The data node $\texttt{x::node}\langle 1, \texttt{q}_1\rangle$ from the antecedent and the predicate $\texttt{x::ll3}\langle \texttt{n}, \texttt{B}\rangle$ from the consequent are co-related by the variable x. In this case, we apply the folding operation to the first two nodes from the antecedent against the shape predicate from the consequent. After that, a matching operation is invoked since the folded predicate now matches with the predicate in the consequent.

The fold step may be recursively applied but is guaranteed to terminate for well-founded predicates as it will reduce a data node in the antecedent for each recursive invocation. This reduction in the antecedent cannot go on forever. Furthermore, the fold operation may introduce bindings for the parameters of the folded predicate. In the above, we obtain $\exists \texttt{n}_1, \texttt{n}_2 \cdot \texttt{n}{=}\texttt{n}_1{+}1 \wedge \texttt{n}_1{=}\texttt{n}_2{+}1 \wedge \texttt{n}_2{=}0$ and $\exists \texttt{B}_1, \texttt{B}_2 \cdot \texttt{B}{=}\texttt{B}_1 \cup \{2\} \wedge \texttt{B}_1{=}\{1\} \cup \texttt{B}_2 \wedge \texttt{B}_2{=}\{\}$, where $\texttt{n}_1, \texttt{n}_2, \texttt{B}_1, \texttt{B}_2$ are existential variables introduced by the folding process, and are subsequently eliminated. These binding formulae may be transferred to the antecedent if n and B are free (for instantiation). Otherwise, they will be kept in the consequent. Since n and B are indeed free, our folding operation would finally derive:

$$\texttt{y::node}\langle 3, \texttt{null}\rangle \wedge \texttt{n} = 2 \wedge \texttt{B} = \{1, 2\} \vdash\ (\texttt{n}{>}1 \wedge 1 \in \texttt{B}) * S$$

The effects of folding may seem similar to unfolding the predicate in the consequent. However, there is a subtle difference in their handling of bindings for free derived variables. If we choose to use unfolding on the consequent instead, these bindings may not be transferred to the antecedent. Consider the example below where n is free :

$$\texttt{z}{=}\texttt{null} \vdash\ \texttt{z::ll3}\langle \texttt{n}, \texttt{B}\rangle \wedge \texttt{n}{>}{-}1 * S$$

By unfolding the predicate $\texttt{ll3}\langle\texttt{n}\rangle$ in the consequent, we obtain :

$$\texttt{z=null} \vdash (\texttt{z=null}\wedge\texttt{n=0}\wedge\texttt{B} = \{\}\wedge\texttt{n}>-1)$$
$$\vee(\exists \texttt{q}, \texttt{v}\cdot\texttt{z::node}\langle\texttt{v},\texttt{q}\rangle * \texttt{q::ll3}\langle\texttt{n}-1, \texttt{B}_1\rangle\wedge\texttt{B} = \texttt{B}_1 \cup \{\texttt{v}\}\wedge\texttt{n}>-1) * S$$

There are now two disjuncts in the consequent. The second one fails because it mismatches. The first one matches but still fails as the derived binding $\texttt{n=0}$ was not transferred to the antecedent.

When a fold against a predicate $p_2{::}c_2\langle v_2^*\rangle$ is performed, the constraints related to variables v_2^* are significant. The *split* function projects these constraints out and differentiates those constraints based on free variables. These constraints on free variables can be transferred to the antecedent to support the variables' instantiations.

$$\begin{aligned} \mathit{split}_V^{\{v_2^*\}}(\bigwedge_{i=1}^{n} \pi_i^r) \equiv \quad & \texttt{let } \pi_i^a, \pi_i^c = \texttt{if } \mathit{FV}(\pi_i^r) \cap v_2^* = \emptyset \texttt{ then } (true, true) \\ & \quad \texttt{else if } \mathit{FV}(\pi_i^r) \cap V = \emptyset \texttt{ then } (\pi_i^r, true) \texttt{ else } (true, \pi_i^r) \\ & \texttt{in } (\textstyle\bigwedge_{i=1}^{n} \pi_i^a, \bigwedge_{i=1}^{n} \pi_i^c) \end{aligned}$$

A formal definition of folding is specified by the rule [FOLDING] in Figure 2.5. Some heap nodes from κ are removed by the entailment procedure so as to match with the heap formula of the predicate $p{::}c\langle v^*\rangle$. This requires a special version of entailment that returns three extra things:

- consumed heap nodes,
- existential variables used,
- final consequent.

The final consequent is used to return a constraint for $\{v^*\}$ via $\exists W_i \cdot \pi_i$. A set of answers is returned by the fold step as we allow it to explore multiple ways of matching up with its disjunctive heap state. Our entailment also handles empty predicates correctly with a couple of specialised rules.

2.4.4 Approximating separation formula by pure formula.

In our entailment proof, the entailment between separation formulae is reduced to entailment between pure formulae by successively removing heap nodes from the consequent until only a pure formula remains. When this happens, the heap formula in the antecedent can be soundly

approximated by function *XPure*$_n$. The function *XPure*$_n(\Phi)$, whose definition is given in Fig 4.6, returns a sound approximation of Φ as a formula of the form: $\beta ::= \mathbf{ex}\ i \cdot \beta \mid \bigvee(\exists v^* \cdot \pi)^*$ where $\mathbf{ex}\ i$ construct is being used to capture a distinct symbolic address i that has been abstracted from a heap node or predicate Φ. The function *IsData*(c) returns `true` if c is a data node, while *IsPred*(c) returns `true` if c is a shape predicate.

We illustrate how the approximation functions work by computing *XPure*$_1$(p::ll⟨n⟩). Let Φ be the body of the `ll` predicate, i.e. $\Phi \equiv (\texttt{root=null} \wedge \texttt{n=0}) \vee (\texttt{root::node}\langle \_, \texttt{r}\rangle * \texttt{r::ll}\langle \texttt{n-1}\rangle)$.

$$
\begin{array}{lll}
\mathit{Inv}_0(\texttt{p::ll}\langle\texttt{n}\rangle) & =_{df} & \texttt{n} \geq \texttt{0} \\
\mathit{XPure}_0(\Phi) & =_{df} & \mathbf{ex}\ \texttt{j} \cdot (\texttt{root=0} \wedge \texttt{n=0}) \vee (\texttt{root=j} \wedge \texttt{j>0} \wedge \mathit{Inv}_0(\texttt{r::ll}\langle\texttt{n-1}\rangle)) \\
 & = & \mathbf{ex}\ \texttt{j} \cdot (\texttt{root=0} \wedge \texttt{n=0}) \vee (\texttt{root=j} \wedge \texttt{j>0} \wedge \texttt{n-1} \geq \texttt{0}) \\
\mathit{Inv}_1(\texttt{p::ll}\langle\texttt{n}\rangle) & =_{df} & [\texttt{p/root}]\mathit{XPure}_0(\Phi) \\
 & = & \mathbf{ex}\ \texttt{j} \cdot (\texttt{p=0} \wedge \texttt{n=0}) \vee (\texttt{p=j} \wedge \texttt{j>0} \wedge \texttt{n-1} \geq \texttt{0}) \\
\mathit{XPure}_1(\texttt{p::ll}\langle\texttt{n}\rangle) & =_{df} & \mathbf{ex}\ \texttt{i} \cdot [\texttt{i/j}]\mathit{Inv}_1(\texttt{p::ll}\langle\texttt{n}\rangle) \\
 & = & \mathbf{ex}\ \texttt{i} \cdot (\texttt{p=0} \wedge \texttt{n=0}) \vee (\texttt{p=i} \wedge \texttt{i>0} \wedge \texttt{n-1} \geq \texttt{0})
\end{array}
$$

The following normalization rules are also used to propagate $\mathbf{ex}$ to the leftmost :

$$
\begin{array}{ll}
(\mathbf{ex}\ I \cdot \phi_1) \vee (\mathbf{ex}\ J \cdot \phi_2) & \rightsquigarrow \mathbf{ex}\ I \cup J \cdot (\phi_1 \vee \phi_2) \\
\exists\, v \cdot (\mathbf{ex}\ I \cdot \phi) & \rightsquigarrow \mathbf{ex}\ I \cdot (\exists\, v \cdot \phi) \\
(\mathbf{ex}\ I \cdot \phi_1) \wedge (\mathbf{ex}\ J \cdot \phi_2) & \rightsquigarrow \mathbf{ex}\ I \cup J \cdot \phi_1 \wedge \phi_2 \wedge \bigwedge_{i \in I, j \in J} i \neq j
\end{array}
$$

The $\mathbf{ex}\ i^*$ construct is converted to $\exists\, i^*$ when the formula is used as a pure formula. For instance, the above *XPure*$_1$(p::ll⟨n⟩) is converted to $\exists \texttt{i} \cdot (\texttt{p=0} \wedge \texttt{n=0}) \vee (\texttt{p=i} \wedge \texttt{i>0} \wedge \texttt{n-1} \geq \texttt{0})$, which is further reduced to $(\texttt{p=0} \wedge \texttt{n=0}) \vee (\texttt{p>0} \wedge \texttt{n-1} \geq \texttt{0})$.

The soundness of the heap approximation (given in the next section) ensures that it is safe to approximate an antecedent by using *XPure*, starting from a given sound invariant (checked by [FV–PRED] in Sec 2.3). The heap approximation also allows the possibility of obtaining a more precise invariant by unfolding the definition of a predicate one or more times, prior to applying the *XPure*$_0$ approximation with the predicate's invariant. For example, when given a pure invariant $\texttt{n} \geq \texttt{0}$ for the predicate ll⟨n⟩, the *XPure*$_0$ approximation is simply the pure invariant $\texttt{n} \geq \texttt{0}$ itself. However, the *XPure*$_1$ approximation would invoke a single unfold before the *XPure*$_0$ approximation is applied, yielding $\mathbf{ex}\ \texttt{i} \cdot (\texttt{root=0} \wedge \texttt{n=0} \vee \texttt{root=i} \wedge \texttt{i>0} \wedge \texttt{n-1} \geq \texttt{0})$, which is sound and more precise than $\texttt{n} \geq \texttt{0}$, since the former can relate the nullness of the `root` pointer with the size `n` of the list.

The invariants associated with shape predicates play an important role in our system. Without the knowledge $\texttt{m}{\geq}\texttt{0}$, the proof search for the entailment $\texttt{x::node}\langle \_, \texttt{y}\rangle * \texttt{y::ll}\langle \texttt{m}\rangle \vdash \texttt{x::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{\geq}\texttt{1}$ would not have succeeded (failing to establish $\texttt{n}{\geq}\texttt{1}$). Without a more precisely derived invariant using *XPure*$_1$ on predicate `ll`, the proof search for the entailment $\texttt{x::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{>}\texttt{0} \vdash \texttt{x}{\neq}\texttt{null}$ would not have succeeded either.

2.5 Storage Model

The semantics of our separation heap formula is similar to the model given for separation logic [107], except that we have extensions to handle our user-defined shape predicates.

To define the storage model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in$ *Val* denoting `null`, *Var* of variables (program and logical variables), and *ObjVal* of object values stored in the heap, with $c[f_1{\mapsto}\nu_1, .., f_n{\mapsto}\nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. with h, s from the following concrete domains:

$$\begin{array}{rcll} h & \in & \mathit{Heaps} & =_{df} \quad \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal} \\ s & \in & \mathit{Stacks} & =_{df} \quad \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc} \end{array}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [107, 56]. Function $\mathit{dom}(f)$ returns the domain of function f. Note that we use $\mapsto$ to denote mappings, not the points-to assertion in separation logic, which has been replaced by $p{::}c\langle v^* \rangle$ in our notation.

2.6 Semantic Model

Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ. The model relation for separation heap formulas is defined below. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

Definition 2.6.1. *[Model for Separation Constraint]*

$$
\begin{array}{lll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \textit{ or } s,h \models \Phi_2 \\
s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi & \texttt{iff} & \exists \nu_{1..n} \cdot (s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n], h \models \kappa \textit{ and } s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n] \models \pi) \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot\ h_1 \perp h_2 \textit{ and } h = h_1 \cdot h_2 \textit{ and} \\
 & & \quad s, h_1 \models \kappa_1 \textit{ and } s, h_2 \models \kappa_2 \\
s,h \models \texttt{emp} & \texttt{iff} & \mathit{dom}(h) = \emptyset \\
s,h \models p::c\langle v_{1..n} \rangle & \texttt{iff} & \texttt{data}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \in P,\ h = [s(p) \mapsto r], \\
 & & \quad \textit{and } r = c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)] \\
 & & \texttt{or}\ (c\langle v_{1..n} \rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P \textit{ and } s,h \models [p/\texttt{root}]\Phi
\end{array}
$$

Note that $h_1 \perp h_2$ indicates h_1 and h_2 are domain-disjoint, i.e. $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset$. $h_1 \cdot h_2$ denotes the union of disjoint heaps h_1 and h_2. The definition for $s,h \models p::c\langle v^* \rangle$ is split into two cases: (1) c is a data node defined in the program *P*; (2) c is a shape predicate defined in the program *P*. In the first case, h has to be a singleton heap. In the second case, the shape predicate c may be inductively defined.

2.7 Dynamic Semantics

This section presents a small-step operational semantics for our language given in Fig. 2.1. The machine configuration is represented by $\langle s, h, e \rangle$, where s denotes the current stack, h denotes the current heap, and e denotes the current program code. Each reduction step is formalized as a transition of the form: $\langle s, h, e \rangle \hookrightarrow \langle s_1, h_1, e_1 \rangle$. The full set of transitions is given in Fig. 4.7.2. We have introduced an intermediate construct $\texttt{ret}(v^*, e)$ to model the outcome of call invocation, where e denotes the residual code of the call. It is also used to handle local blocks. The forward verification rule for this intermediate construct is given as follows:

$$
\frac{\vdash \{\Delta\}\, e\, \{\Delta_2\} \quad \Delta_1 = (\exists v'^* \cdot \Delta_2)}{\vdash \{\Delta\}\, \texttt{ret}(v^*, e)\, \{\Delta_1\}}\ [\mathbf{FV{-}RET}]
$$

Note that whenever the evaluation yields a value, we assume this value is stored in a special logical variable `res`, although we do not explicitly put `res` in the stack s.

We also have the following postcondition weakening rule:

$$\frac{\vdash \{\Delta\}\, e\, \{\Delta_1\} \quad \Delta_1 \approx\!\gg \Delta_2}{\vdash \{\Delta\}\, e\, \{\Delta_2\}} \; [\textbf{FV-POST-WEAKENING}]$$

where $\Delta_1 \approx\!\gg \Delta_2 \;=_{df}\; \forall s, h \cdot s, h \models \mathit{Post}(\Delta_1) \Longrightarrow s, h \models \mathit{Post}(\Delta_2)$. As discussed earlier, we can view Δ_1 and Δ_2 as binary relations (as far as only program variables are concerned). Therefore, we use $\mathit{Post}(\Delta)$ here to refer to the postcondition(i.e. the set of post-states) specified by Δ. Note also that Δ_1 and Δ_2 share the same set of initial states (in which e start to execute).

Definition 2.7.1 (Poststate)**.** *Given a constraint Δ, $\mathit{Post}(\Delta)$ captures the relation between primed variables of Δ. That is :*

$$\mathit{Post}(\Delta) \;=_{df}\; \rho\,(\exists V \cdot \Delta), \qquad \textit{where}$$
$$V = \{v_1, .., v_n\} \textit{ denotes all unprimed program variables in } \Delta$$
$$\rho = [v_1/v_1', .., v_n/v_n']$$

We now explain the notations used in the operational semantics. We use k to denote a constant, $\perp$ to denote an undefined value, and $()$ to denote the empty expression (program). Note that the runtime stack s is viewed as a 'stackable' mapping, where a variable v may occur several times, and $s(v)$ always refers to the value of the variable v that was popped in most recently.[3] The operation $[v \mapsto \nu] + s$ "pushes" the variable v to s with the value ν, and $([v \mapsto \nu] + s)(v) = \nu$. The operation $s - \{v^*\}$ "pops out" variables v^* from the stack s. The operation $s[v \mapsto \nu]$ changes the value of the most recent v in stack s to ν. The mapping $h[\iota \mapsto r]$ is the same as h except that it maps ι to r. The mapping $h + [\iota \mapsto r]$ extends the domain of h with ι and maps ι to r.

[3] We can give a more formal definition for s, where different occurrences of the same variable can be labeled with different 'frame' numbers. We omit the details here.

$$\frac{\rho{=}[0/\texttt{null}] \quad b{=}(\mathit{XPure}_n(\kappa_1{*}\kappa{\wedge}\pi_1) \implies \exists V{\cdot}\rho\pi_2)}{\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} (\pi_2) * \{\kappa_1{\wedge}\pi_1 \mid b\}} \quad [\textbf{ENT-EMP}]$$

$$\frac{\mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \implies p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \quad \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V) \vdash^{\kappa * p_1::c\langle v_1^*\rangle}_{V-\{v_2^*\}} \rho(\kappa_2{\wedge}\pi_2) * S}{p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} (p_2{::}c\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S} \quad [\textbf{ENT-MATCH}]$$

$$\frac{\mathit{IsPred}(\texttt{c}_2){\wedge}\mathit{IsData}(\texttt{c}_1) \quad \{(\Delta_i,\kappa_i^r,\pi_i^r)\}_{i=1}^n{=}\mathit{fold}^{\kappa}(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1, p_2{::}c_2\langle v_2^*\rangle) \quad \mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \implies p_1{=}p_2 \quad (\pi_i^a,\pi_i^c){=}\mathit{split}_V^{\{v_2^*\}}(\pi_i^r) \quad \Delta_i{\wedge}\pi_i^a \vdash^{\kappa_i^r}_{V} \kappa_2{\wedge}(\pi_2{\wedge}\pi_i^c) * S_i}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \bigcup_{i=1}^n S_i} \quad [\textbf{ENT-FOLD}]$$

$$\frac{\mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1) \implies p_1{=}p_2 \quad \mathit{IsPred}(\texttt{c}_1){\wedge}\mathit{IsData}(\texttt{c}_2) \quad \mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * S} \quad [\textbf{ENT-UNFOLD}]$$

$$\frac{\Delta_1 \vdash^{\kappa}_{V} \Delta_2 * S_1 \quad \Delta_1 \vdash^{\kappa}_{V} \Delta_3 * S_2 \quad S{=}S_1{\cup}S_2}{\Delta_1 \vdash^{\kappa}_{V} (\Delta_2{\vee}\Delta_3) * S} \quad [\textbf{ENT-RHS-OR}]$$

$$\frac{\Delta_1 \vdash^{\kappa}_{V} \Delta_3 * S_1 \quad \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * S_2 \quad S_3{=}\{\Delta_5{\vee}\Delta_6 \mid \Delta_5{\in}S_1, \Delta_6{\in}S_2\}}{\Delta_1{\vee}\Delta_2 \vdash^{\kappa}_{V} \Delta_3 * S_3} \quad [\textbf{ENT-LHS-OR}]$$

$$\frac{\Delta_1 \vdash^{\kappa}_{V\cup\{w\}} ([w/v]\Delta_2) * S_1 \quad \mathit{fresh}\ w \quad S{=}\{\exists\ w \cdot \Delta \mid \Delta{\in}S_1\}}{\Delta_1 \vdash^{\kappa}_{V} (\exists\ v \cdot \Delta_2) * S} \quad [\textbf{ENT-RHS-EX}]$$

$$\frac{[w/v]\Delta_1 \vdash^{\kappa}_{V} \Delta_2 * S \quad \mathit{fresh}\ w}{\exists v \cdot \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * S} \quad [\textbf{ENT-LHS-EX}]$$

$$\frac{c\langle v^*\rangle{\equiv}\Phi\ \texttt{inv}\ \pi\ \in P}{\mathit{unfold}(p{::}c\langle v^*\rangle)\ =_{df}\ [p/\texttt{root}]\Phi} \quad [\textbf{UNFOLDING}]$$

$$\frac{c\langle v^*\rangle{\equiv}\Phi\ \texttt{inv}\ \pi\ \in P \quad W_i{=}V_i{-}\{v^*,p\} \quad \kappa{\wedge}\pi \vdash^{\kappa'}_{\{p,v^*\}} [p/\texttt{root}]\Phi * \{(\Delta_i,\kappa_i,V_i,\pi_i)\}_{i=1}^n}{\mathit{fold}^{\kappa'}(\kappa{\wedge}\pi, p{::}c\langle v^*\rangle) =_{df} \{(\Delta_i,\kappa_i,\exists\, W_i{\cdot}\pi_i)\}_{i=1}^n} \quad [\textbf{FOLDING}]$$

Figure 2.5: Non-Deterministic Separation Constraint Entailment

$$\frac{(c\langle v^*\rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P}{\mathit{Inv}_0(p{::}c\langle v^*\rangle) =_{df} [p/\texttt{root}, 0/\texttt{null}]\pi}$$

$$\frac{(c\langle v^*\rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P \quad n{\geq}1}{\mathit{Inv}_n(p{::}c\langle v^*\rangle) =_{df} [p/\texttt{root}, 0/\texttt{null}]\mathit{XPure}_{n-1}(\Phi)}$$

$$\mathit{XPure}_n(\bigvee(\exists v^*{\cdot}\kappa{\wedge}\pi)^*) =_{df} \bigvee(\exists v^*{\cdot}\mathit{XPure}_n(\kappa){\wedge}[0/\texttt{null}]\pi)^*$$

$$\mathit{XPure}_n(\texttt{emp}) =_{df} \texttt{true}$$

$$\mathit{XPure}_n(\kappa_1 * \kappa_2) =_{df} \mathit{XPure}_n(\kappa_1) \wedge \mathit{XPure}_n(\kappa_2)$$

$$\frac{\mathit{IsData}(c) \quad \mathit{fresh}\ i}{\mathit{XPure}_n(p{::}c\langle v^*\rangle) =_{df} \mathbf{ex}\ i{\cdot}(p{=}i{\wedge}i{>}0)}$$

$$\frac{\begin{array}{c}\mathit{IsPred}(c) \quad \mathit{fresh}\ i^* \\ \mathit{Inv}_n(p{::}c\langle v^*\rangle) = \mathbf{ex}\ j^* \cdot \bigvee(\exists u^*{\cdot}\pi)^*\end{array}}{\mathit{XPure}_n(p{::}c\langle v^*\rangle) =_{df} \mathbf{ex}\ i^* \cdot [i^*/j^*] \bigvee(\exists u^*{\cdot}\pi)^*}$$

Figure 2.6: *XPure* : Translating to Pure Form

$$\langle s,h,v\rangle\hookrightarrow\langle s,h,s(v)\rangle \qquad \langle s,h,k\rangle\hookrightarrow\langle s,h,k\rangle \qquad \langle s,h,v.f\rangle\hookrightarrow\langle s,h,h(s(v))(f)\rangle$$

$$\langle s,h,v{:=}k\rangle\hookrightarrow\langle s[v{\mapsto}k],h,()\rangle \qquad \langle s,h,();e\rangle\hookrightarrow\langle s,h,e\rangle$$

$$\frac{\langle s,h,e_1\rangle\hookrightarrow\langle s_1,h_1,e_3\rangle}{\langle s,h,e_1;e_2\rangle\hookrightarrow\langle s_1,h_1,e_3;e_2\rangle} \qquad \frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,v{:=}e_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_1\rangle} \qquad \frac{s(v){=}\texttt{false}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_2\rangle}$$

$$\langle s,h,\{t\ v;\ e\}\rangle\hookrightarrow\langle[v{\mapsto}\bot]{+}s,h,\texttt{ret}(v,e)\rangle$$

$$\langle s,h,\texttt{ret}(v^*,k)\rangle\hookrightarrow\langle s{-}\{v^*\},h,k\rangle$$

$$\frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,\texttt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\texttt{ret}(v^*,e_1)\rangle}$$

$$\frac{r{=}h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1{=}h[s(v_1){\mapsto}r]}{\langle s,h,v_1.f:=v_2\rangle\hookrightarrow\langle s,h_1,()\rangle}$$

$$\frac{\texttt{data}\ c\ \{t_1\ f_1,..,t_n\ f_n\}{\in}P \quad \iota{\notin}\mathit{dom}(h) \quad r{=}c[f_1{\mapsto}s(v_1),..,f_n{\mapsto}s(v_n)]}{\langle s,h,\texttt{new}\ c(v_{1\cdots n})\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto r],\iota\rangle}$$

$$\frac{s_1{=}[w_i{\mapsto}s(v_i)]_{i=m}^{n}{+}s \quad t_0\ \mathit{mn}((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1},(t_i\ w_i)_{i=m}^{n})\ \{e\}}{\langle s,h,\mathit{mn}(v_{1\cdots n})\rangle\hookrightarrow\langle s_1,h,\texttt{ret}(\{w_i\}_{i=m}^{n},[v_i/w_i]_{i=1}^{m-1}e)\rangle}$$

Figure 2.7: Small-Step Operational Semantics

CHAPTER III

RELATED WORKS SURVEY

Considerable work has been done in analyzing and verifying programs with heap allocated data structures. Next, we provide a summary of the most relevant related works based on several often overlapping categories. For each category, we attempt to position the current thesis in relation to the other works. We start with the general areas of separation logic and shape analysis, followed by more specific work done on tracking size and set/bag properties. Furthermore, we present an overview of other program verifiers, followed by an investigation of the three main research directions pursued by the current thesis: immutability annotations, structures specifications, and static and dynamic specifications.

3.1 Separation Logic

From the moment when separation logic was first proposed [56, 107], its use has been further extended for termination proofs [15], interprocedural shape analysis [45, 18, 109], overlapping structures verification [76, 52], concurrency [119, 120, 47, 46], Java verification [35, 101]. In the search for a decidable fragment of separation logic for automated verification, Berdine et al. [7, 8] support only a limited set of predicates without size properties. Similarly, Jia and Walker [59] postponed the handling of recursive predicates in their work on automated reasoning of pointer programs.

On the inference front, Lee et al. [75] has conducted an intraprocedural analysis for loop invariants using grammar approximation under separation logic. Their analysis can handle a wide range of shape predicates with local sharing but is restricted to predicates with two parameters and without size properties. Separation logic has also shown promising results in interprocedural shape analysis. Several approaches have been formulated, based either on computing procedure summaries [45, 18], or on abstraction of calling context [109].

Following the increasing exploitation of separation logic in verification tools [8, 35, 9, 92], the development of entailment checking procedures for separation logic was tackled in several works, among which we recall those based on SMT solvers [13], and on superposition calculus

[103]. While most of the work on proving properties of mutable data structures using separation logic targeted structures with constrained sharing, there are also works for overlapping data structures. After running multiple sub-analyses for tracking information about non-overlapping components, Lee et al. combine the results of these sub-analysis and derive the safety properties of the entire overlapping structure [76]. A different approach is taken in [52], where Hawkins et al. allow the programmer to specify an overlapping data structure in the style of a relational database, with the goal of generating correct programs.

Sims [114] extends separation logic with fixpoint connectives and postponed substitution to express recursively defined formulae to model the analysis of while-loops. However, it is unclear how to check for entailment in their extended separation logic. While our work does not address the inference/analysis challenge, we have succeeded in providing direct support for automated verification via an expressive specification mechanism.

In this thesis, we make use of the formalism of separation logic for verifying properties of mutable data structures in sequential programs. Our approach requires annotations in the form of pre and postconditions for each method and invariants for each loop. We aim for a sound and terminating formulation of automated verification via separation logic but do not aim for completeness in the expressive fragment that we handle.

3.2 Shape Checking/Analysis

Many formalisms for shape analysis are proposed for checking user programs' intricate manipulations of shapely data structures. One well-known work is the **Pointer Assertion Logic** [89] by Moeller and Schwartzbach, which is a highly expressive mechanism to describe invariants of graph types [63]. The **Pointer Assertion Logic Engine (PALE)** uses Monadic Second-Order Logic over Strings and Trees as the underlying logic and the tool MONA [64] as the prover. PALE invariants are not designed to handle arithmetic, hence it is not possible to encode height-balanced priority queue in PALE. Moreover, PALE is unsound in handling procedure calls [89], whereas we would like to have a sound verifier. Harwood et al. [50] describe a **UTP theory** for objects and sharing in languages like Java or C++. Their work focuses on a denotational model meant to provide a semantical foundation for refinement-based reasoning or Hoare-style axiomatic reasoning. Our work focuses more on practical verification for heap-manipulating programs.

In an object-oriented setting, the **Dafny language** [77] uses dynamic frames (introduced by Kassios [61]) in its specifications. The term frame refers to a set of memory locations, and an expression denoting a frame is dynamic in the sense that as the program executes, the set of locations denoted by the frame can change. A dynamic frame is thus denoted by a set-valued expression (in particular, a set of object references), and this set is idiomatically stored in a field. Methods in Dafny use "modifies" and "reads" clauses, which frame the modifications of methods and dependencies of functions. By comparison, separation logic provides a reasoning logic that hides the explicit representation of dynamic frames.

For shape inference, Sagiv et al. [112] present a parameterized framework, called **TVLA**, using 3-valued logic formulae and abstract interpretation. Based on the expected properties of data structures, programmers must supply a set of predicates to the framework which are then used to analyse that certain shape invariants are maintained.

However, most of these techniques are focused on analysing shape invariants, and do not attempt to track the size and bag properties of complex data structures. One exception is the quantitative shape analysis of Rugina [111] where a data flow analysis is proposed to compute quantitative information for programs with destructive updates. By tracking unique points-to reference and its height property, their algorithm is able to handle AVL-like tree structures. Even then, the author acknowledge the lack of a general specification mechanism for handling arbitrary shape/size properties.

The current thesis aims at handling user-defined shape predicates, which may describe complex data structures as AVL trees, red-black trees, sorted lists, doubly-linked lists. Moreover, the shape predicates may track other properties of the data structures, such as size and bag properties.

3.3 Size Properties

In another direction of research, size properties have been most explored for declarative languages [55, 122, 23] as the immutability property makes their data structures easier to analyse statically. Size analysis was later extended to object-based programs [24] but was restricted to tracking either size-immutable objects that can be aliased and size-mutable objects that are unaliased, with no support for complex shapes. The Applied Type System (ATS) [20] was proposed for combining programs with proofs. In ATS, dependent types for capturing program invariants

are extremely expressive and can capture many program properties with the help of accompanying proofs. Using linear logic, ATS may also handle mutable data structures with sharing in a precise manner. However, users must supply all expected properties, and precisely state where they are to be applied, with ATS playing the role of a proof-checker. In comparison, we use a more limited class of constraint for shape, size and bag analysis but support automated modular verification.

3.4 Set/Bag Properties

Set-based analysis has been proposed to verify data structure consistency properties in [67], where a decision procedure is given for a first order theory that combines set and Presburger arithmetic. This result may be used to build a specialised mixed constraint solver but it currently has high algorithmic complexity. Lahiri and Qadeer [68] reported an intra-procedural reachability analysis for well-founded linked lists using first-order axiomatization. Reachability analysis is related to set/bag property that we capture but implemented by transitive closure at the predicate level.

3.5 Other Verifiers

3.5.1 ESC/Java

Extended Static Checking for Java (ESC/Java) [42], developed at Compaq Systems Research Center, aims for scalability and usability. For that, it forgoes soundness for the potential benefits of more automation and faster verification time. Hence, ESC/Java suffers from both false negatives (programs that pass the check may still contain errors that ESC/Java is designed to handle), and false positives (programs flagged as erroneous are in fact correct programs). On the contrary, our verifier is a sound program verifier as it does not suffer from false negatives: if a program is verified, it is guaranteed to meet its specifications for all possible program executions.

3.5.2 ESC/Java2

The ESC/Java effort is continued with ESC/Java2 [27], which adds support for current versions of Java, and also verifies more JML [71] constructs. One significant addition is the support for model fields and method calls within annotations [25]. Since ESC/Java2 continues to use

Simplify [31] as its underlying theorem prover which does not support transitive closure operations, it may have difficulties in verifying properties of heap-based data structures that require reachability properties, such as collections of values stored in container data structures.

3.5.3 Spec♯/Boogie

Spec♯ [5] is a programming system developed at Microsoft Research. It is an attempt at verifying programs written for the C♯ programming language. It adds constructs tailored to program verification, such as pre- and post-conditions, frame conditions, non-null types, model fields and object invariants. Spec♯ programs are verified by the Boogie verifier [5, 6]. Boogie generates verification conditions that are passed to an SMT solver in order to be discharged. The default SMT solver is Z3 [30]. Spec♯ also supports runtime assertion checking.

Spec♯ supports object invariants but leaves the decision of when to enforce/assume object invariants to the user. In order to verify object invariant modularly, Spec♯ employs an ownership scheme that allows an object o to own its representation – objects that are reachable from o and are part of o's abstract state. The ownership scheme in Spec♯ forces a top-down unpacking of the objects for updates, and a bottom-up packing for re-establishing the object invariant. The packing and unpacking of objects are done explicitly by having programmers writing special commands in method bodies.

In our system, instead of using special fields in method contracts to indicate whether an invariant should be enforced, users directly use predicates. Hence, there is no need for explicitly packing and unpacking the objects in the method body. Consequently, users are shielded from the details of the verification methodology, which are largely irrelevant, from a user's point of view.

3.5.4 Jahob

The main focus of Jahob [65, 66] is on reasoning techniques for data structure verification that combines multiple theorem provers to reason about expressive logical formulas. Jahob uses a subset of the Isabelle/HOL [95] language as its specification language, and works on instantiatable data structures, as opposed to global data structures used in its predecessor, Hob [69]. Like SPEC♯, Jahob supports ghost variables and specification assignments which places onus on programmers to help in the verification process by providing suitable instantiations of these specification variables.

3.5.5 EVE Proofs

EVE Proofs [118] is an automatic verifier for Eiffel [87]. The tool translates Eiffel programs to Boogie [5]. EVE Proofs is integrated in the Eiffel Verification Environment. The authors acknowledge the importance of frame conditions in modular verification. When a routine is called, the verifier is invalidating all knowledge about the locations which may have changed. Therefore it is essential to constrain the effect a routine has on the system to preserve as much information as possible. As Eiffel does not offer a way to specify the frame condition, the authors introduced an automatic extraction of "modifies" clauses. Their approach uses the postcondition to extract a list of locations which constitute the "modifies" clause.

Although the approach uses the dynamic type for the pre- and postcondition of a routine call, it uses the static type for the frame condition. This can lead to unsoundness in the system. As opposed to EVE Proofs, our approach does not have to infer frame conditions, courtesy to the frame rule of separation logic [107]. The crucial power of the frame rule is that it allows a global property to be derived from a local one, without looking at other parts of the program.

Another restriction of EVE Proofs regards the methodology for invariants, which has to take into account that objects can temporarily violate the invariant, but also that an object can call other objects while being in an inconsistent state. As this is not considered at the moment, their current implementation of invariants can introduce unsoundness in the system.

3.5.6 jStar

jStar [35, 34] is an automatic verification tool based on separation logic aiming at object-oriented programs. The tool combines the idea of abstract predicate families [100, 101] and the idea of symbolic execution and abstraction using separation logic [33]. jStar integrates theorem proving and abstract interpretation techniques as loop invariants are synthesized automatically. However, the user must provide abstraction rules used to ensure convergence in the fixed-point computation of loop invariants. While very general structural rules which need to be used with any kind of logic systems are built in, the user must provide the problem specific logical rules used by the theorem prover to decide entailment and other kinds of implications. jStar does not check that user-defined rules are consistent.

By comparison, our system does not require the specification of logical rules. As we do not infer loop invariants, we do not require the abstraction rules either. While the user might provide

lemmas, which state any auxiliary relations between predicate definitions, as opposed to jStar, we do verify that the lemmas provided by the user are sound [93].

3.5.7 SLAyer

SLAyer [9] is a verification system designed to automatically prove the absence of memory safety errors (i.e. dangling pointer dereferences, double frees, memory leaks) of programs written in C. Validity of memory is treated at a per-object granularity, meaning that errors such as buffer overflows are not covered. It has been successfully used for finding bugs in Windows 8 codebase. SLAyer does not intend to check other properties of data structures beyond memory safety.

3.5.8 Thor

Thor [83] is a tool meant to automatically discover memory errors by combining shape analysis and arithmetic reasoning. The implemented shape analysis uses separation logic for describing the memory states and is capable of reasoning about doubly-linked lists. Regarding the arithmetic reasoning, Thor adds support for stack-based integers, integers in the heap, and the lengths of lists by utilizing off-the-shelf arithmetic analysis. If a program can be proven safe by only the shape reasoning, then no further processing is required. However, if the program's safety depends on arithmetic information, then the result of the shape analysis will be translated into an arithmetic program. Consequently, the integer programs produced by the shape analysis phase provide a new source of test programs for the arithmetic analysis tool.

3.5.9 VeriFast

VeriFast [57, 58] is a tool for the specification and verification of safety properties of pointer-manipulating imperative programs. It supports abstract predicates described in the separation logic formalism, such that the memory is represented as a conjunction of points-to assertions and abstract predicate assertions, while data values are represented as first-order logic terms. As opposed to our approach, abstract predicates must be folded and unfolded explicitly using ghost statements. Assertions over data values are delegated to an SMT solver.

3.5.10 Key

Key is a verification tool designed for proving the correctness of programs written in the Java language [1, 121]. The underlying logic used in Key is a dynamic logic [29], an extension of

first-order predicate logic with modal operators. For addressing the aliasing issue, Key makes use of dynamic frames. The dynamic frames allow the explicit specification of the set of memory locations that are relevant for a method or for an abstract variable. "Modifies" clauses, used for listing the memory locations that can be modified, are being encoded in the underlying dynamic logic [37].

3.5.11 Why/Krakatoa/Caduceus/Frama-C

Why/Krakatoa/Caduceus/Frama-C [39, 82] is a set of tools for deductive verification of Java and C source code, where the requirements are specified as annotations in the source. Why provides its own internal language, to which the input languages are compiled. The internal language is a small ML-like language with imperative features (references and arrays), exceptions and annotations. For Java the specifications are given in the Java Modeling Language (JML) and are interpreted by the Krakatoa tool. For C, Why has its own specification language, largely inspired from JML and interpreted by the Caduceus or Frama-C tools. Verification conditions generated by the Why tool can then be discharged by different theorem provers. However, to the best of our knowledge, neither inheritance nor method overriding is supported by their system. Frama-C denotes a suite of tools dedicated to the analysis of the source code written in C, which gathers several static analysis techniques in a single collaborative framework. By using Frama-C, the results output by an analyzer can be used by other analyzers in the framework.

3.5.12 jMoped

jMoped [116, 115] is a checker for Java programs. The tool combines model-checking and testing, as model checking is used to symbolically test many inputs at the same time. Internally, a Java program is translated into a symbolic pushdown system (SPDS), preserving the control flow of the program. JMoped checks for errors such as assertion violations, null pointer exceptions, and array bound violations. Whenever an error is found, jMoped outputs the arguments leading to the error. As opposed to our approach, it requires the user to set bounds for the size of variables and of the heap, in order to perform the modelling only for a finite amount of data.

3.5.13 Remarks

As a comparison, we shall discuss some features in our current verification system that differ from those used in other verifiers. Our use of user-defined predicates, which capture the properties to be analysed, removes the need for model fields. Regarding ghost variables (specification variables), we provide support for automatically instantiating them. Furthermore, we make use of unfold/fold reasoning to handle the properties of recursive data structures. This obviates the need for specifying transitive closure relations that are used by classical verifier, such as Jahob, when tracking recursive properties. Additionally, as separation logic employs local reasoning via a frame rule, our approach does not require a separate "modifies" clause to be prescribed.

3.6 Immutability Annotations

Immutability has been extensively studied in the context of object-oriented programs under the form of object immutability (a given object cannot be mutated through any reference whatsoever), class immutability (no instance of an immutable class can be changed), reference immutability (a given reference is not used to modify its referent) [11, 125, 124, 49, 106]. Our approach works at the specification level, rather than as part of the type system, and can achieve object immutability by marking the object as immutable, and reference immutability by marking the reference as immutable. For field and class immutability, we can provide support for it by adding immutability annotations to field and data (class) declarations, respectively. This makes every instances of such annotated field or data (class) be automatically marked as immutable in our logic.

However, what really distinguishes our work from type-based approaches is that we can apply immutability annotation on user-definable predicate definitions. This can be used to annotate either partial or entire data structures for read-only access, as may be required by the logics of a given program component. This new approach adds considerable expressiveness to immutable specifications, allowing us to support more concise, more precise and more flexible schemes for both analysis and verification of user programs.

Separation logic uses separation and local reasoning as a means to enforce exclusive ownership over each heap cell [107]. This interpretation of the points-to assertion in separation logic is not simply to describe a part of the heap, but can be enhanced with read/write permissions for better analysis and control. This observation has been extensively exploited by works

dealing with permissions [12, 36] for enabling race free sharing of heap storage between concurrent threads. Bornat et al. [12] extended separation logic to allow shared read access using permissions. The authors considered predicates that admit to a weaker permission, namely read-only permission. One problem with the read-only permissions is that $*$ can no longer be used to express disjointness as the same permission can appear twice, i.e. two read permissions on the same cell. Any two read permissions on a field look identical and cannot be distinguished. Hence, permissions need to be disjoint. Parkinson solves this problem by giving names to permissions [99].

Inspired by the use of permissions for the concurrent programming setting [12], we have designed a much simpler mechanism for enabling immutability annotations in the sequential programming setting. In our case, $*$ still expresses disjointness. As our model is focused on the sequential setting, we do not currently support multiple read-only permissions for a given resource. Instead, our goal is to exploit those scenarios in which data structures can be treated as immutable in order to support a more precise verification/analysis. Our solution is tailored towards more flexible alias analysis, supports partial immutability, better cut-points preservation for modular analysis and can support access controls through immutable predicates within both pre and post-conditions. The new specification mechanism allows us to design a more concise and more precise verification/analysis system, with finer controls over accesses to resources (data structures).

3.7 Structured Specifications

Previous works on enhancing pre/post specifications [70, 60] were mainly concerned with improving modularity to allow easier understanding of specifications. With this objective, multiple specifications and redundant representations were advocated as the primary machinery. In the context of shape analysis, Chang and Rival [19] make use of `if` notation for defining inductive checkers. However, the conditional gets approximated to disjunction during the actual analysis. Verification wise, the three structured specification mechanisms that we have proposed are not available in existing tools, such as JML [17], Spec# [5], Dafny [79], JStar [35] and VeriFast [57]. The closest relationships may be summarized, as follows. JML supports specification cases, in the form of multiple pre/post conditions, for better modularity and clarity of specifications. Our case constructs also intend to provide better guidance to the verification process. Spec#/Dafny

supports ghost variables for manual instantiation (by user) of logical variables. In contrast, our early/late instantiation mechanisms provided two solutions to automatic instantiation of logical variables. Overall, little attempt has been made to add specification structures that can help produce a better verification outcome.

On timings, we did not compare with Spec# and Dafny, since our benchmark on heap-manipulating programs is not properly covered by their specification logic. Regarding JStar, it currently uses logics involving only shapes and equalities, it does not support more expressive properties, like set and numeric properties, needed by our benchmark. Lastly, VeriFast requires more user intervention in the form of explicit unfolding and folding of the abstract predicates through ghost statements.

In a distributed systems setting, Seino et al. [113] present a case analysis meant to improve the efficiency of protocol verification, which involves finding appropriate predicates and splitting a case into multiple sub-cases based on the predicates. In order to cover all the possible case splits, they use a special type of matrix. Pientka [104] argues for the need of case analysis in inductive proofs. The potential case splits are selected heuristically, based on the pattern of the theorem. A case split mechanism has been used by Brock et al. [14] to guide case analysis during proving. As opposed to the previous works, our current proposal is to incorporate structured mechanisms within the specification mechanism itself for guiding the case analysis, existential instantiation or staged proving.

3.8 Object Oriented Verification

In support of modular reasoning on properties of object-oriented programs, the notion of behavioral subtyping has been intensively studied in the last two decades, e.g. [80, 2, 81, 32, 86, 41, 90, 99]. The notion of specification inheritance, where an overriding method inherits the specifications of all the overridden methods, was first introduced in Eiffel [86]. As an effort to relate these two notions, [32] presented a modular specification technique which automatically forces behavioral subtyping through specification inheritance. More recently, [73] proposed a formal characterization for behavioral subtyping and modular reasoning. The basic idea of modular reasoning, which the authors call supertype abstraction, is that reasoning about an invocation, say `E.m()`, is based on the specification associated with the static type of the receiver expression `E`. In [73], the authors proved the equivalence between supertype abstraction and

behavioral subtyping. The new formalization is supposed to serve as a semantic foundation for object-oriented specification languages.

Various embodiments of these proposals have been implemented in both static and runtime verification tools and have been applied to rich specification and programming languages such as ESC/Java [42], JML [71], Spec# [5], ESpec [98], Krakatoa [85, 84]. Software model checking frameworks [110, 51] have also been used in the verification of OO programs. Inference mechanisms for loop invariants have been proposed in [94, 102] amongst others, and they can make verification even easier to use. However, most of these works are based primarily on the idea of dynamic specifications. Even when static specifications are added, like code contracts in JML [74, ch 15], they did not enforce an important subtyping relation between a static specification and its dynamic counterpart. Moreover, in comparison with our approach, Spec# is more restrictive in handling overriding as it does not allow any changes in the precondition of the overriding method.

Using the rules of behavioral subtyping, Findler et al. have formalized hierarchy violations and blame assignment for pre and postcondition failures [40, 41]. They identified a problem (related to preservation of class invariants) that arises from synthesizing the specifications of overriding methods through specification inheritance. This problem is caused by specification inheritance's manner of enforcing behavioral subtyping which may wrongly assume that the original specification of an overriding method is too weak. In our proposal, we can avoid this problem by using specification abstraction instead of specification inheritance, if class invariants are to be preserved for the overriding methods. Furthermore, while [40] and [41] focus on checking the correctness of contracts at run-time, we propose a static verification system.

The problem of writing specifications for programs that use various forms of modularity where the internal resources of a module should not be accessed by the module's clients, is tackled in several papers [96, 100, 72]. In [96] the internal resources of a module are hidden from its clients using a so called hypothetical frame rule, whereas in [100] the notion of abstract predicates is introduced. While [96] only supports single instances of the hidden data structure, abstract predicates can deal with dynamic instantiation of a module. Visibility modifiers are taken into consideration in [72] where a set of rules for information hiding in specifications for Java-like languages is given. Moreover, the authors demonstrate their application on the specification language JML. However, some JML tools, including ESC/Java2 [42, 26] ignore

visibility modifiers in specifications.

The emergence of separation logic provided a novel way to handle the challenging aliasing issues for heap-manipulating programs [100, 99, 101].

The closest to our work is the distinction between static and dynamic specifications, proposed by Parkinson and Bierman, in support of modular verification and direct method calls handling [101]. Regarding the modularity issues found in the Java programming language, they address the issue of encapsulation with the concept of an abstract predicate, which is the logical analogue of an abstract datatype, and the issues of inheritance by extending the concept of abstract predicates to abstract predicate families. This extension allows a predicate to have multiple definitions that are indexed by class, which allows subclasses to have a different internal representation while remaining behavioural subtypes. We support encapsulation and inheritance through the use of partial and full views. Similar to their work, we support inheritance and overriding, while avoiding unnecessary re-verifications. We have a marginal emphasis on static specifications over dynamic specifications, as we advocate for the latter to be derived from the former, when needed, using the refinement techniques of specification specialization and abstraction.

CHAPTER IV

IMMUTABILITY ENHANCED SPECIFICATIONS

4.1 Motivation

The importance of immutability information has been discussed in several works [49, 11]. Immutability is essential when we have to guarantee that non-privileged clients are never allowed to modify some given resource (or data structure). Moreover, immutability is useful in the presence of aliasing, where it is challenging [49] to maintain invariants of aliased objects otherwise. In a concurrent programming setting, immutability, seen as a read-only permission, enables safe sharing between different threads without the cost of synchronization [99, 12].

Immutability was extensively investigated in the context of object-oriented programs under several forms, such as *object immutability* (a given object cannot be mutated through any reference whatsoever), *class immutability* (no instance of an immutable class can be changed), *reference immutability* (a given reference is not used to modify its referent) [11, 125, 124]. These proposals were formulated as extensions of the underlying type system.

However, to the best of our knowledge, the concept of immutability has not been applied to a richer specification logic that could be used to specify more concisely and precisely the behaviors of software programs, particularly in the sequential programming setting. In the current thesis, we investigate the benefits of enforcing immutability requirements in the context of an expressive logic, called separation logic [108], that is particularly suited for verifying properties of mutable data structures. In contrast to the previous works, our approach applies at the specification level, rather than as part of the type system.

The current work enhances the approach introduced in the previous chapters of this thesis with the possibility of specifying which parts of the data structures cannot be mutated (are immutable), and by allowing the sharing of aliases in the heap formula through the use of the conjunctive $\wedge$ operator. In several recent verification systems based on separation logic [92, 8, 35], a specification formula is a restricted form of logic that allows only separating conjunction, $*$, in the heap description. The separating conjunction $\Phi_1 * \Phi_2$ denotes a program state with two disjoint heap spaces described by sub-formulae Φ_1 and Φ_2, respectively. Such heap separation

supports precise knowledge about the pointer aliasing, and can be exploited by the frame rule of separation logic [107], that forms the basis for local reasoning.

Our goal is to take advantage of the immutability property, so as to generalize the heap description to allow a greater degree of sharing. Thus, instead of relying on just a spatial formula $\Phi_1 * \Phi_2$ in the specification logic, our new verification system also supports conjunctive formula of the form $\Phi_1 \wedge \Phi_2$ whereby the heap formula Φ_1 is expected to be marked as immutable, whilst Φ_2 is unrestricted. This allows the heap nodes from both Φ_1 and Φ_2 to be possibly aliased with one another, thus supporting a more general specification mechanism for data structures, including the use of overlaid data structures.

4.2 Chapter Overview

In the rest of the chapter we shall focus on the apparatus for writing and verifying specifications with immutability annotations. Sec 4.3 provides examples to motivate the need for immutability annotations for a more concise and precise specification mechanism. Sec 4.4 introduces the specification language. Sec 4.5 and Sec 4.6 formalize the entailment proving for immutability enhanced specifications and the verification rules to generate Hoare triples, respectively. The soundness properties for both the forward verifier and the entailment prover are discussed in Sec 4.7. Sec 4.8 presents our experimental results.

4.3 Examples

Immutable annotations allow us to write stronger specifications that lead to more concise and precise program verification. We shall use simple examples to highlight how these desirable traits for program specifications are being achieved. To help construct simple examples, let us consider the specification for a singly-linked list, already described in Sec 2.2.

$$\texttt{data node}\ \{\ \texttt{int val};\ \texttt{node next}\}$$

$$\begin{array}{l}\texttt{ll}\langle\texttt{n}\rangle \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0\\ \quad \vee\ \texttt{root::node}\langle\_, \texttt{q}\rangle * \texttt{q::ll}\langle\texttt{n}{-}1\rangle\\ \quad \texttt{inv n}{\geq}0;\end{array}$$

4.3.1 Concise Specification

Consider a method that receives a list as its input, before returning the list's length as its result. Typically, such a method would *not* be mutating its input list. Let us first specify this method without any immutability annotation. Note the use of a special variable `res` for denoting the result of the method.

$$
\begin{array}{l}
\texttt{int length(node x)} \\
\quad \texttt{requires x::ll}\langle \texttt{n} \rangle \\
\quad \texttt{ensures x::ll}\langle \texttt{n} \rangle \wedge \texttt{res=n ;} \\
\{\ \texttt{if (x==null) then return 0;} \\
\quad \texttt{else return 1 + length(x.next);}\ \}
\end{array}
$$

The precondition assumes that variable `x` points to a singly-linked list of length `n` using the predicate $\texttt{x::ll}\langle \texttt{n} \rangle$. The same predicate, $\texttt{x::ll}\langle \texttt{n} \rangle$, is also present in the postcondition which suggests that a list of the *same length* is being preserved by the method. However, we would really prefer to express something stronger for this method since it is the same input list (from the method's entry) that is being preserved in the postcondition (at the method's exit).

One solution for capturing total preservation of list is to introduce a stronger `llS` predicate that would capture the entire sequence of elements and their nodes, as shown below. Note that `V` captures the sequence of values, while `L` captures the sequence of pointers to nodes in the linked list.

$$
\begin{array}{l}
\texttt{llS}\langle \texttt{V, L, n} \rangle \equiv \texttt{root=null} \wedge \texttt{n=0} \wedge \texttt{V=L=[]} \\
\quad \vee \texttt{root::node}\langle \texttt{v, q} \rangle * \texttt{q::llS}\langle \texttt{V1, L1, n-1} \rangle \wedge \texttt{V=v:V1} \\
\qquad \wedge \texttt{L=root:L1} \\
\texttt{inv n}\geq \texttt{0;}
\end{array}
$$

With this more informative predicate, we could now capture a more complete specification of `length` method, as follows:

$$
\begin{array}{l}
\texttt{int length(node x)} \\
\quad \texttt{requires x::llS}\langle \texttt{V, L, n} \rangle \\
\quad \texttt{ensures x::llS}\langle \texttt{V, L, n} \rangle \wedge \texttt{res=n ;}
\end{array}
$$

However, this approach makes our verification more complex as we are now required to verify two additional properties, namely the preservation of sequence of values and sequence of addresses, for the input list. Apart from causing more work for our verifier, we must also rely on a more complex prover that is expected to handle proofs involving sequences.

We propose a simpler solution to this problem that uses immutability annotation of the form @I to annotate the specification of each node or each predicate that is *only read* (and not modified) by its method. In the case of the `length` method, we can indicate that the list pointed by `x` is not mutated by its method through the following specification:

```
int length(node x)
  requires x::ll⟨n⟩@I
  ensures res=n ;
```

The precondition states that the linked-list pointed by `x` will only be read by the `length` method. This indirectly ensures the preservation of the input list, which need not be re-proven in the postcondition. The net result is a cleaner specification that more succinctly captures the intended semantics of the `length` method. We said that the new specification is more concise (or shorter) since it has one fewer predicate in the postcondition. It is also more precise (or accurate) since it captures the total preservation of the input list without resorting to the use of a more complex predicate. These improvements are due entirely to the use of an immutable predicate in the precondition.

4.3.2 Flexible Aliasing

Let us now consider a method, called `sum`, that receives two data nodes pointed by `x` and `y`, respectively, and computes the sum of the values stored inside the two nodes.

Under the principle of heap separation, as promoted by separation logic, our specification logic would explicitly state that the two input nodes are either disjoint, as specified using $\texttt{x::node}\langle \texttt{a}, \_\rangle * \texttt{y::node}\langle \texttt{b}, \_\rangle$, or are exact aliases for one another, namely $\texttt{x::node}\langle \texttt{v}_1, \_\rangle \wedge \texttt{x=y}$.

To support both these scenarios, one way for specifying this is to use two pairs of pre and post-conditions, one corresponding to the case when `x` and `y` are disjoint, and another to the case when they are exact aliases, as shown below.

```
int sum(node x, node y)
  requires x::node⟨a, q1⟩∗y::node⟨b, q2⟩
  ensures x::node⟨a, q1⟩∗y::node⟨b, q2⟩∧res=a+b;
  requires x::node⟨a, q⟩∧x=y
  ensures x::node⟨a, q⟩∧res=2 ∗ a;
{return x.val + y.val}
```

With the help of these two specifications, we can now deal with different scenarios, whereby the nodes may either be aliased or otherwise. As an example, consider the two sum calls in the code fragment below.

```
node u = new node(...)
node w = new node(...)
int r1 = sum(u, u)
int r2 = sum(u, w)
```

The first call sum(u, u) would only match with the second pre/post specification, while the second call sum(u, w) where the input nodes are disjoint, would match with the first pre/post specification.

Though this specification is fairly precise, it is unnecessary complicated for the given method. In particular, neither of the two nodes pointed to by x and y are being updated. For such a scenario, it is actually not important if x and y are aliases or not. What is important is to be able to access the values of the two nodes. With the help of immutability annotations, we can actually use a much simpler specification below.

```
int sum(node x, node y)
  requires x::node⟨a, _⟩@I∧y::node⟨b, _⟩@I
  ensures res=a+b;
```

We apply the immutability annotation @I on both the nodes pointed by x and y, respectively. Moreover, we allow the use of $\wedge$ operator, instead of the $*$ operator, to express the fact that x and y may either be aliases or otherwise. As a consequence, the same pre/post specification can now be used for both the sum calls, namely sum(u, u) and sum(u, w), from our earlier code fragment. Yet another improvement is that the two immutable nodes need not re-appear in the

postcondition, thus avoiding the need to re-prove its preservation by the sum method. With this cleaner specification, we are able to claim that the result returned by the method is always a+b, regardless of whether or not x and y are aliases or not.

We refer to this as the flexible aliasing from the use of immutable specification. The knowledge that something is immutable enables us to support arbitrary aliasing in the heap description with the use of the conjunctive $\wedge$ operator. This results in a unified specification that is both concise and precise. Functional programmers are all too aware of this benefit, whereby aliasing was never an issue for understanding (or analysing) purely functional code without any heap mutation. In the present work, we advocate for the use of immutable annotations to help us achieve a similar benefit for the specification of data structures that are only being read, and never modified, in the imperative setting.

4.3.3 Preservation of Cut-Points

Cut-points refer to the intermediate data points that may be encountered prior to a method call. Previous works on program analysis [45] have attempted to preserve cut-points for method calls, where possible. This is typically achieved by keeping track of multiple summaries for each method under analysis, so that more cut-points could be preserved. In this section, we show how cut-points may be preserved with the help of immutability annotations.

Let us re-visit the length example, covered in Sec 4.3.1. Let us also consider the following code fragment:

$$\texttt{node y} \;=\; \texttt{new node}(2, \texttt{null})$$
$$\texttt{node x} \;=\; \texttt{new node}(1, \texttt{y})$$
$$\texttt{int r} \;=\; \texttt{length}(\texttt{x})$$

Prior to the length call, we would have the following heap state, $\texttt{x}::\texttt{node}\langle 1, \texttt{y}\rangle * \texttt{y}::\texttt{node}\langle 2, \texttt{null}\rangle$, that is formed by two assignment statements. This heap state captures two cut-points from variables x and y. If we had used the following mutable specification of length:

$$\texttt{int length}(\texttt{node x})$$
$$\texttt{requires x}::\texttt{ll}\langle \texttt{n}\rangle$$
$$\texttt{ensures x}::\texttt{ll}\langle \texttt{n}\rangle \wedge \texttt{res}=\texttt{n}\ ;$$

We would have matched the current heap state to $\texttt{x}::\texttt{ll}\langle 2\rangle$ before asserting $\texttt{x}::\texttt{ll}\langle 2\rangle \wedge \texttt{r}=2$ as the

post-state of the code fragment. However, this abstracted heap state has lost track of the entire cut-point from variable y, while the node at x is being replaced by a x::ll⟨2⟩ predicate. As a result, we are unable to reason about any information pertaining to y (and to a lesser extend x) after the method call.

To rectify this deficiency, we can use the earlier immutable specification:

$$\begin{array}{l}\texttt{int length(node x)}\\ \quad\texttt{requires x::ll⟨n⟩@I}\\ \quad\texttt{ensures res=n ;}\end{array}$$

During the entailment for each immutable predicate, such as x::ll⟨n⟩@I, the original heap state remains unchanged but instantiation on properties, such as n=2, can still be computed from the current heap state. Thus, at the end of the code fragment given earlier, we expect the following to be captured, x::node⟨1, y⟩∗ y::node⟨2, null⟩∧r=n=2, which preserves the cut-points for both x and y. This difference in the treatments between mutable and immutable predicates by our new entailment procedure, allows considerable more information to be preserved for immutable predicates. We shall formally describe this procedure later in Sec 4.5.2.

4.3.4 Partial Immutability

Another aspect of our immutability enhanced specification is that we could support the annotation of only a segment of linked nodes, rather than a fully linked data structure. Let us first consider a segment of singly-linked nodes, as defined by the following predicate:

$$\begin{array}{l}\texttt{lseg⟨p, n⟩} \equiv \texttt{root=p} \wedge \texttt{n=0}\\ \quad \vee\ \texttt{root::node⟨r, q⟩} * \texttt{q::lseg⟨p, n−1⟩}\\ \quad\texttt{inv n≥0 ;}\end{array}$$

As an example of its use, let us also consider a method to join a linked list to another, as shown below.

$$
\begin{array}{l}
\texttt{void append(node x, node y)} \\
\quad \texttt{requires x::lseg}\langle \texttt{null, n}\rangle \wedge \texttt{x}\neq\texttt{null} \\
\quad \texttt{ensures x::lseg}\langle \texttt{y, n}\rangle \ ; \\
\{\texttt{if (x.next}\neq\texttt{null) then append(x.next, y);} \\
\quad \texttt{else x.next}=\texttt{y;}\}
\end{array}
$$

The first linked-list must be non-empty and moreover it will be mutated by the method to join up with the second list. On closer inspection, it is actually the last node of the first linked list that is being mutated. Thus, to capture the specification of `append` method in a more accurate manner, we can actually use the following pre/post specification instead:

$$
\begin{array}{l}
\texttt{void append(node x, node y)} \\
\quad \texttt{requires x::lseg}\langle \texttt{p, n}\rangle \texttt{@I} * \texttt{p::node}\langle \texttt{v, null}\rangle \\
\quad \texttt{ensures p::node}\langle \texttt{v, y}\rangle \ ;
\end{array}
$$

The entire segment of the first linked list minus the last element, namely $\texttt{x::lseg}\langle \texttt{p, n}\rangle\texttt{@I}$, is now marked as immutable. Only the last node $\texttt{p::node}\langle \texttt{v, null}\rangle$ of the first linked list is left as a mutable node, and suitably specified as modified in the post-condition of the method. This specification is both simpler and more precise. All cut-points pertaining to the initial segment of parameter `x` are preserved by the presence of an immutable `lseg` predicate in the precondition. It can even be used to reason about the formation of cyclic linked list when $\texttt{x} = \texttt{y}$, or even lasso-style circular linked list when `y` points to a node within the first list of nodes.

4.3.5 Read and Write Phases

One of the goals of the current proposal is to relax the heap separation principles specific to separation logic, by allowing two phases in the heap description. The two phases have the following attributes:

- a **read phase**, where all the predicates must be immutable. As mutation is disabled in the read phase, we allow heap sharing between the constituting heap predicates, which can be co-joined by both $*$ and $\wedge$. This corresponds to the initial reading phase in a program, and contains the heap being read before any writing took place.

- a **write phase**, where the predicates can be either mutable or immutable. As this phase

might involve writing, we do not allow heap sharing. Hence, we require structural separation between the heap predicates through $*$.

Our formulation for entailment proving is simple as it comprises only two phases, but is general enough to cover several situations where flexible aliasing may be deployed. We advocate for capturing the precise aliasing only when required, namely after at least one mutation/writing has took place.

For illustration, let us assume a method `call_length`, which calls the `length` method defined in Sec 4.3.1 for two lists received as argument, and records the addition of the two results in a data node, also received as argument. Initially, we leave the `call_length` method unspecified.

$$\begin{array}{l}\texttt{void call\_length(node x, node y, node z)}\\ \{\ \texttt{int l} = \texttt{length(x)}+\texttt{length(y)};\\ \quad \texttt{z.val} = \texttt{l};\ \}\end{array}$$

In the body of the `call_length` method, we can distinguish two distinct phases, namely:

- an initial read phase, where the linked lists pointed by `x` and `y` are being read by the `length` method.
- a subsequent write phase, where the data node pointed by `z` is updated to record the addition of the lengths of the lists pointed by `x` and `y`.

Using our read/write phases approach, we can provide the specification given below, where the heap description precisely captures the code behavior. Semantically, $\Phi_1 \# \Phi_2$ is equivalent to $\Phi_1 \wedge \Phi_2$. Operationally, we interpret $\Phi_1 \# \Phi_2$ as a specification that comprised of a read phase for an immutable formula Φ_1, followed by a write phase for a possibly mutated Φ_2. In this example, the initial read phase is captured by $\texttt{x::ll}\langle\texttt{n}\rangle@I \wedge \texttt{y::ll}\langle\texttt{m}\rangle@I$, while the subsequent write phase is denoted by $\texttt{z::node}\langle\_, \texttt{v}\rangle$. As mentioned before, inside the read phase there is no need to explicitly specify the aliasing between x and y (this is exploited in the specification through the use of the $\wedge$ connector). Moreover, as all the reading takes place before updating the node pointed by z, there is no need to capture the aliasing between x and y and z (this is exploited through the use of the $\#$ connector).

```
void call_length(node x, nodey, node z)
  requires (x::ll⟨n⟩@I∧y::ll⟨m⟩@I)#z::node⟨_, v⟩
  ensures z::node⟨n+m, v⟩;
```

4.3.6 Immutable Postconditions

Apart from improving the analysis of methods, one other application of our immutability enhancement is the construction of immutable (or partially immutable) data structures. For illustration, consider a method for constructing a list of length n. Let us assume that after the construction phase we do *not allow* the list to be mutated.

In order to guarantee that the list cannot be mutated outside this method, we can mark it as immutable in the method's postcondition. Consequently, after a call to the ll_build method, any caller will only be allowed to read the list, and cannot update it in any way.

```
node ll_build(int n)
  requires n≥0
  ensures res::ll⟨n⟩@I;
{ if (n == 0) then null;
  else { int v = ... ;
         new node(v, ll_build(n−1)); } }
```

We can use this mechanism to specify the methods of immutable classes or data structures.

4.4 Specification and Programming Language

In Figure 4.1 we introduce the differences from the programming and specification languages given in Fig 2.1 and Fig 2.2, respectively. Each heap predicate can be annotated with an immutability annotation, $u \in \{I, M\}$, (v::c⟨v*⟩@u). This is to support use-site annotations. In order to support declaration-site annotations, every data type declaration and heap predicate definition can also have attached immutability annotations. If no annotation is present, a data node/heap predicate is considered to be mutable. For illustration, x::node⟨v, y⟩@I corresponds to an immutable node, meaning a node whose both fields cannot be mutated. Note that we support the following subtyping relation between the immutability annotations $M <: I$.

The heap part, κ, is organized according to the code's reading and writing phases:

Data type	*datat*	$::= \texttt{data}\ c[@u]\ \{\ (\textit{field})^*\ \}$
Imm ann.	u	$::= I \mid M$
Shape pred.	*spred*	$::= c\langle v^*\rangle[@u] \equiv \Phi\ \texttt{inv}\ \pi_0$
Heap formula	κ	$::= \kappa_R \# \kappa_W \mid \texttt{emp}$
	κ_R	$::= \kappa_{R_*} \wedge \kappa_R \mid \texttt{v::c}\langle\texttt{v}^*\rangle@I \mid \texttt{emp}$
	κ_{R_*}	$::= \kappa_{R_*} * \kappa_{R_*} \mid \texttt{v::c}\langle\texttt{v}^*\rangle@I \mid \texttt{emp}$
	κ_W	$::= \kappa_W * \kappa_W \mid \texttt{v::c}\langle\texttt{v}^*\rangle[@u] \mid \texttt{emp}$

Figure 4.1: Modifications to the programming and specification languages

- κ_R corresponding to the initial read phase in the code. This phase contains only immutable predicates co-joined by either $*$ or $\wedge$. Note that it is compulsory in this phase for each predicate to be immutable, i.e. to have an immutability annotation $@I$.
- κ_W corresponding to the write phase in the code. This phase can contain both immutable and mutable predicates co-joined by $*$.
- The read and write phases are co-joined by the $\#$ connector, whose semantic is given in Sec. 4.7.2.

During the verification process, the heap state might contain nested read and write phases, $\kappa ::= \kappa_R \#(\kappa_W * \kappa)$. For brevity, we only present the formalization for one read and one write phase.

Our specification is meant to minimize the need to explicitly express aliasing relations. Accordingly, we use the following principles:

1. Aliasing does not need to be considered:
   - inside the read phase as none of the available pointers can be used to mutate the heap.
   - between the read phase and the write phase as, during our entailment checking, once the writing phase is encountered, the read phase is discarded. Basically, we consider

that writing to the heap invalidates all the previous reads from the heap. This will be further described during the entailment proving procedure in Sec 4.5 .

2. Aliasing needs to be considered:

   – inside the write phase

4.5 Entailment Checking

Our goal in the current section is enhancing the entailment proving procedure introduced in Sec 2.4 to handle the current form of the heap formula with conjunction and immutability annotations.

Besides the entailment procedure in Sec 2.4, we also provide an amended form that allows heap sharing between the consequent and the residual states, as long as the shared heap is immutable: $\Delta_A \vdash^{\kappa}_{V} \Delta_C *_i S_R$. If we consider any of the residual states, $\Delta_R \in S_R$, then the semantics of the $*_i$ connector is such that

$$s,h \models \Delta_C *_i \Delta_R \quad \texttt{iff} \quad \exists h_1,h_2,h_3 \cdot h = h_1 \perp h_2 \perp h_3 \text{ and } s,h_1{\cdot}h_3 \models \Delta_C \text{ and } s,h_2{\cdot}h_3 \models \Delta_R$$

Similar to the entailment procedure introduced in Sec 2.4, the purpose of heap entailment is to check that heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and to compute the set of possible residual poststates S_R. The entailment succeeds when S_R is non-empty, otherwise it is deemed to have failed. κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. κ and V are derived. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$.

4.5.1 Splitting the entailment

Our entailment proving is structured as follows:

- split the heap on the RHS according to the rules
  $[\mathbf{ENT{-}SPLIT{-}RHS_1}]$ and $[\mathbf{ENT{-}SPLIT{-}RHS_2}]$ (Fig 4.2)
- split the heap on the LHS according to the rules
  $[\mathbf{ENT{-}SPLIT{-}LHS_1}]$ and $[\mathbf{ENT{-}SPLIT{-}LHS_2}]$ (Fig 4.3)

For convenience, we also provide a lifted variant of the entailment checking procedure, which takes a set of prestates. The entailment succeeds in such a case if any of the prestates gives rise to a successful entailment, that is if at least one of the S_i is non-empty. This variant of the entailment is useful when we break the entailment procedure according to the read and write phases, where each individual phase could potentially give rise to a set of residual states. In order for the entire entailment to succeed, we need it to only succeed for one of the phases.

$$\frac{\forall i{\in}1\ldots n{\cdot}\Phi_i{\vdash}^{\kappa}_{V}\Phi *_i S_i}{\{\Phi_1,\ldots,\Phi_n\}{\vdash}^{\kappa}_{V}\Phi *_i \cup^{n}_{i=1} S_i}$$

For the case of the [$\mathbf{ENT{-}SPLIT{-}RHS_1}$] rule in Fig 4.2, the splitting is performed when encountering a phase split, $\#$. At that point, the heap entailment is divided into sub-phases corresponding to the read and write phases, respectively:

- First the read phase, κ_R, is entailed, obtaining a set of residual states, S. If the heap from the read phase is co-joined through $\wedge$, then the entailment will be further split according to rule [$\mathbf{ENT{-}SPLIT{-}RHS_2}$] in Fig 4.2. To differentiate the entailment of the read phase, we provide the entailment judgement $\Phi_1 {\models_{\mathbf{RD_{RHS}}}}^{\kappa}_{V} \Phi_2 *_i S_R$ (Fig 4.2).

- Secondly, the write phase is entailed with the help of the residual state from the previous entailment. Take note that before entailing the write phase, κ_W, the read phase needs to be dropped as it is no longer safe to use that information (the heap from the read phase might have been mutated by a write in the write phase). The dropping of the read phase is performed by the $dropRP$ function, which is given below.

$$\begin{array}{l} dropRP(S) =_{df} \forall \Phi \in S.dropRP_{\Phi}(\Phi) \\ dropRP_{\Phi}(\bigvee(\exists v^*{\cdot}\kappa{\wedge}\pi)^*) =_{df} \bigvee(\exists v^*{\cdot}dropRP_{\kappa}(\kappa){\wedge}\pi)^* \\ dropRP_{\kappa}(\kappa_R \# \kappa_W) =_{df} \kappa_W \end{array}$$

- Lastly, the pure information from the RHS, π_2, is entailed, generating the final set of residual states, S_2.

For the rule [$\mathbf{ENT{-}SPLIT{-}LHS_1}$] in Fig 4.3, we take advantage of the proof search capability of our system by trying to entail the heap κ from the RHS using both the read phase, κ_R, and write phase, κ_W, from the LHS. The entailment succeeds if either the read or the write phase on the LHS gives rise to a successful entailment, that is if at least one of the residual states S_1

$$\frac{\kappa_1 \wedge \pi_1 \models^{\kappa}_{\mathbf{RD_{RHS}}\,V} \kappa_R *_i S \quad dropRP(S) \vdash^{\kappa}_V \kappa_W *_i S_1 \quad S_1 \vdash^{\kappa}_V \pi_2 *_i S_2}{\kappa_1 \wedge \pi_1 \vdash^{\kappa}_V (\kappa_R \# \kappa_W \wedge \pi_2) *_i S_2}\ [\mathbf{ENT{-}SPLIT{-}RHS_1}]$$

$$\frac{\kappa_1 \wedge \pi_1 \vdash^{\kappa}_V \kappa_{R_*} *_i S \quad S \models^{\kappa}_{\mathbf{RD_{RHS}}\,V} \kappa_R *_i S_1}{\kappa_1 \wedge \pi_1 \models^{\kappa}_{\mathbf{RD_{RHS}}\,V} (\kappa_{R_*} \wedge \kappa_R) *_i S_1}\ [\mathbf{ENT{-}SPLIT{-}RHS_2}]$$

Figure 4.2: Splitting RHS

$$\frac{\kappa_R \models^{\kappa}_{\mathbf{RD_{LHS}}\,V} \kappa *_i S_1 \quad \kappa_W \vdash^{\kappa}_V \kappa *_i S_2}{\kappa_R \# \kappa_W \vdash^{\kappa}_V \kappa *_i (S_1 \cup S_2)}\ [\mathbf{ENT{-}SPLIT{-}LHS_1}]$$

$$\frac{\kappa_{R_*} \vdash^{\kappa}_V \kappa *_i S_1 \quad \kappa_R \models^{\kappa}_{\mathbf{RD_{LHS}}\,V} \kappa *_i S_2}{\kappa_{R_*} \wedge \kappa_R \models^{\kappa}_{\mathbf{RD_{LHS}}\,V} \kappa *_i (S_1 \cup S_2)}\ [\mathbf{ENT{-}SPLIT{-}LHS_2}]$$

Figure 4.3: Splitting LHS

and S_2 is non-empty. When the LHS consists of only the read phase, rule [$\mathbf{ENT{-}SPLIT{-}LHS_2}$] from Fig 4.3 applies. This rule continues splitting the antecedent whenever $\wedge$ is encountered in the heap formula on the LHS. The entailment succeeds if at least one of the sub-entailments succeeds. We provide the entailment judgement $\Phi_1 \models^{\kappa}_{\mathbf{RD_{LHS}}\,V} \Phi_2 *_i S_R$ (Fig 4.3).

4.5.2 Matching

During entailment, each pair of aliased nodes from the antecedent and consequent are matched up, whenever they are proved identical. The formal rules for matching are given in [$\mathbf{ENT{-}MATCH{-}MUT}$] and [$\mathbf{ENT{-}MATCH{-}IMM}$] in Fig 4.5. Rule [$\mathbf{ENT{-}MATCH{-}MUT}$] applies whenever the node to be matched on the RHS is mutable, while [$\mathbf{ENT{-}MATCH{-}IMM}$] applies for immutable nodes on the RHS. For the former rule, the matching node from the LHS is consumed. As the matching process is incremental, we keep the successfully matched nodes from antecedent in κ for better precision. Function XPure_n soundly approximates the heap formula in the antecedent, and it will be described in Sec 4.5.3.

In the case of the rule [$\mathbf{ENT{-}MATCH{-}IMM}$], the matching node on the LHS is not consumed. However, the node needs to be temporarily removed until the entire κ_2 is entailed. This is due to the fact that $\mathtt{p_2}::\mathtt{c}\langle \mathtt{v}_2^* \rangle @I$ and κ_2 must reside in disjoint heaps as they are co-joined by the separating conjunction. Hence, we will extract the matching node from the heap formula

$$
\begin{array}{lll}
SH(S, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) & =_{df} & \forall \Phi \in S.SH_\Phi(\Phi, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) \\
SH_\Phi(\bigvee(\exists v^* \cdot \kappa \wedge \pi)^*, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) & =_{df} & \bigvee(\exists v^* \cdot SH_\kappa(\kappa, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) \wedge \pi)^* \\
SH_\kappa(\kappa_X\ C\ \kappa_Y, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) & =_{df} & SH_\kappa(\kappa_X, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id))\ C \\
 & & \quad SH_\kappa(\kappa_Y, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)), \\
 & & \quad for\ X, Y \in \{R, W\}\ and\ C \in \{\#, *, \wedge\} \\
SH_\kappa(\mathtt{p_2}::\mathtt{c}\langle \mathtt{v_2^*}\rangle@u, (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle, id)) & =_{df} & \mathtt{p_2}::\mathtt{c}\langle \mathtt{v_2^*}\rangle@u \\
SH_\kappa([id_1], (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) & =_{df} & \mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle\ if\ id{=}id_1 \\
SH_\kappa([id_1], (\mathtt{p_1}::\mathtt{c}\langle \mathtt{v_1^*}\rangle@u, id)) & =_{df} & [id_1]\ if\ id{\neq}id_1
\end{array}
$$

Figure 4.4: Function SH

and replace it by a formula hole with a unique identifier ($[id]$). The matching node is substituted back into the formula at the end of the entailment of κ_2 by function SH given in Fig 4.4. This function iterates over the heap formula until reaching the holes. When hitting a hole, it checks the hole identifier and, in the case of a match with the identifier received as argument, it substitutes the corresponding heap predicate back into the formula.

Note that, at the point in the entailment procedure when the matching is reached, the heap formulae in both the antecedent and consequent contain only heap predicates co-joined by $*$. This is due to the fact that the entailment was already split according to the rules in Sec 4.5.1 such that $\wedge$ and $\#$ were eliminated from the heap.

Another feature of the entailment procedure is exemplified by the transfer of the bindings between free variables from the matched node in the consequent and the corresponding variables from the consequent to the antecedent (and subsequently to the residue). In general, when a match occurs (rules [ENT−MATCH−MUT] and [ENT−MATCH−IMM]) and an argument of the heap predicate coming from the consequent is free, the entailment procedure binds the argument to the corresponding variable from the antecedent and moves the equality to the antecedent. In our system, free variables in consequent are variables from method preconditions. These bindings play the role of parameter instantiations during forward reasoning, and can be accumulated

$$[\mathbf{ENT{-}MATCH{-}MUT}]$$
$$\frac{\mathit{XPure}_n(\mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle@u{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \quad \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa * \mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle}_{V-\{v_2^*\}}\rho(\kappa_2{\wedge}\pi_2)*_i S \quad u{<:}M}{\mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle@u{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(\mathtt{p_2{::}c}\langle \mathtt{v}_2^*\rangle@M{*}\kappa_2{\wedge}\pi_2)*_i S}$$

$$[\mathbf{ENT{-}MATCH{-}IMM}]$$
$$\frac{\mathit{XPure}_n(\mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle@u{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \quad [id]{*}\kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa}_{V-\{v_2^*\}}\rho(\kappa_2{\wedge}\pi_2)*_i S \quad u{<:}I}{\mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle@u{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V}(\mathtt{p_2{::}c}\langle \mathtt{v}_2^*\rangle@I{*}\kappa_2{\wedge}\pi_2)*_i SH(S,(\mathtt{p_1{::}c}\langle \mathtt{v}_1^*\rangle,id))}$$

Figure 4.5: Heap Entailment Rules

into the antecedent to allow the subsequent program state (from residual heap) to be aware of their instantiated values. This process is formalized by the function *freeEqn* below, where V is the set of existentially quantified variables:

$$\begin{array}{l}\mathit{freeEqn}([u_i/v_i]_{i=1}^n, V) =_{df}\\ \quad \mathtt{let}\ \pi_i = (\mathtt{if}\ v_i{\in}V\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ v_i{=}u_i)\ \mathtt{in}\ \bigwedge_{i=1}^n \pi_i\end{array}$$

4.5.3 Heap Approximation by a pure formula

As explained in Sec 2.4, in our entailment proof, the entailment between separation formulae is reduced to entailment between pure formulae by successively removing heap nodes from the consequent until only a pure formula remains. When the consequent is pure, the heap formula in the antecedent is approximated by function XPure_n. In the current section, we enhance the definition of the XPure_n function, such that it approximates Φ as a tuple of the form: $(\bigvee(\exists v^*{\cdot}\pi)^*, S)$ where $\bigvee(\exists v^*{\cdot}\pi)^*$ represents a pure formula approximating Φ and S denotes the set of disjoint memory sets in Φ. Each disjoint memory set contains non-aliased symbolic memory addresses. The definition of $\mathit{XPure}_n(\Phi)$ is given in Fig 4.6. The function $\mathit{IsData}(c)$ returns `true` if c is a data node, while $\mathit{IsPred}(c)$ returns `true` if c is a heap predicate. $\cup_{disj}$ and $\cap_{disj}$ are used for computing the set of disjoint memory sets and are defined as follows:

$$S_1 \cup_{disj} S_2 = \{X \cup Y \mid X \in S_1 \wedge Y \in S_2\}$$
$$S_1 \cap_{disj} S_2 = \{X \cap Y \mid X \in S_1 \wedge Y \in S_2\}$$

We illustrate how the approximation functions work by computing

$$\mathit{XPure}_0(\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle@I{*}\mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle@I\#\mathtt{z}{::}\mathtt{ll}\langle\mathtt{v}\rangle \wedge n{\geq}0 \wedge m{\geq}0 \wedge v{\geq}0)$$

Note that, according to the definition provided for the $\mathtt{ll}\langle\mathtt{n}\rangle$ predicate, the pure invariant provided is $\mathtt{n}{\geq}0$.

$$\begin{array}{l}\mathtt{ll}\langle\mathtt{n}\rangle \equiv \mathtt{root}{=}\mathtt{null} \wedge \mathtt{n}{=}0\\ \vee\ \mathtt{root}{::}\mathtt{node}\langle\_,\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle\\ \mathtt{inv}\ \mathtt{n}{\geq}0;\end{array}$$

$$\begin{array}{l}\mathit{XPure}_0(\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle@\mathtt{I}) = (\mathit{Inv}_0(\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle@\mathtt{I}), \{\{\mathtt{x}\}\}) =\\ \quad = (\mathtt{n}{\geq}0, \{\{\mathtt{x}\}\})\ \mathit{given\ that}\ \mathtt{n}{\geq}0\\ \mathit{XPure}_0(\mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle@\mathtt{I}) = (\mathit{Inv}_0(\mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle@\mathtt{I}), \{\{\mathtt{y}\}\}) =\\ \quad = (\mathtt{m}{\geq}0, \{\{\mathtt{y}\}\})\ \mathit{given\ that}\ \mathtt{m}{\geq}0\\ \mathit{XPure}_0(\mathtt{z}{::}\mathtt{ll}\langle\mathtt{v}\rangle) = (\mathit{Inv}_0(\mathtt{z}{::}\mathtt{ll}\langle\mathtt{v}\rangle), \{\{\mathtt{z}\}\}) =\\ \quad = (\mathtt{v}{\geq}0, \{\{\mathtt{z}\}\})\ \mathit{given\ that}\ \mathtt{v}{\geq}0\\ \mathit{XPure}_0(\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle@\mathtt{I}{*}\mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle@\mathtt{I}) =\\ \quad = (\mathtt{n}{\geq}0 \wedge \mathtt{m}{\geq}0, \{\{\mathtt{x}\}\} \cup_{\mathtt{disj}} \{\{\mathtt{y}\}\}) =\\ \quad = (\mathtt{n}{\geq}0 \wedge \mathtt{m}{\geq}0, \{\{\mathtt{x},\mathtt{y}\}\})\\ \mathit{XPure}_0(\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle@\mathtt{I}{*}\mathtt{y}{::}\mathtt{ll}\langle\mathtt{m}\rangle@\mathtt{I}\#\mathtt{z}{::}\mathtt{ll}\langle\mathtt{v}\rangle) =\\ \quad = (\mathtt{n}{\geq}0 \wedge \mathtt{m}{\geq}0 \wedge \mathtt{v}{\geq}0, \{\{\mathtt{x},\mathtt{y}\}\} \cup \{\{\mathtt{z}\}\}) =\\ \quad = (\mathtt{n}{\geq}0 \wedge \mathtt{m}{\geq}0 \wedge \mathtt{v}{\geq}0, \{\{\mathtt{x},\mathtt{y}\},\{\mathtt{z}\}\})\end{array}$$

4.6 Forward Verification

As most of the forward verification rules are the same as those given in Sec 2.3, in Fig 4.7 we only provide new ones for method verification, method call, field read and field update. These rules exploit the modified form of the entailment procedure defined in Sec 4.5 for allowing sharing of immutable heap.

Verification of a method starts with each precondition, and proves that the corresponding postcondition is guaranteed at the end of the method. The verification is formalized in the rule [FV−METH−IMM]. As opposed to the [FV−METH] rule in Sec 2.3, now we make use of the modified form of the entailment procedure through $(\exists W{\cdot}S_1^i) \vdash \Phi_{po}^i *_i S_2^i$.

$$\frac{(\mathtt{c}\langle \mathtt{v}^* \rangle \equiv \Phi\ \mathtt{inv}\ \pi) \in P}{\mathit{Inv}_0(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) =_{df} [p/\mathtt{root}, 0/\mathtt{null}]\pi}$$

$$\frac{(\mathtt{c}\langle \mathtt{v}^* \rangle \equiv \Phi\ \mathtt{inv}\ \pi) \in P \quad n{\geq}1 \quad \mathit{XPure}_{n-1}(\Phi) = (\pi_{n-1}, S_{n-1})}{\mathit{Inv}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) =_{df} [p/\mathtt{root}, 0/\mathtt{null}]\pi_{n-1}}$$

$$\frac{\mathit{XPure}_n(\kappa_i) = (\pi'_i, S)}{\mathit{XPure}_n(\bigvee_i(\exists v_i^* \cdot \kappa_i \wedge \pi_i)^*) =_{df} (\bigvee_i(\exists v_i^* \cdot \pi'_i \wedge [0/\mathtt{null}]\pi_i)^*, \cap_{disj_i} S_i)}$$

$$\mathit{XPure}_n(\mathtt{emp}) =_{df} (\mathtt{true}, \emptyset)$$

$$\frac{\mathit{XPure}_n(\kappa_1) = (\pi_1, S_1) \quad \mathit{XPure}_n(\kappa_2) = (\pi_2, S_2)}{\mathit{XPure}_n(\kappa_1 \wedge \kappa_2) =_{df} (\pi_1 \wedge \pi_2, S_1 \cup S_2)}$$

$$\frac{\mathit{XPure}_n(\kappa_1) = (\pi_1, S_1) \quad \mathit{XPure}_n(\kappa_2) = (\pi_2, S_2)}{\mathit{XPure}_n(\kappa_1 \# \kappa_2) =_{df} (\pi_1 \wedge \pi_2, S_1 \cup S_2)}$$

$$\frac{\mathit{XPure}_n(\kappa_1) = (\pi_1, S_1) \quad \mathit{XPure}_n(\kappa_2) = (\pi_2, S_2)}{\mathit{XPure}_n(\kappa_1 * \kappa_2) =_{df} (\pi_1 \wedge \pi_2, S_1 \cup_{disj} S_2)}$$

$$\frac{\mathit{IsData}(c) \quad \mathit{fresh}\ i}{\mathit{XPure}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) =_{df} (\mathtt{true}, \{\{p\}\})}$$

$$\frac{\mathit{IsPred}(c) \quad p{\neq}\mathtt{null} \quad \mathit{Inv}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) = \bigvee(\exists u^* \cdot \pi)^*}{\mathit{XPure}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) =_{df} (\bigvee(\exists u^* \cdot \pi)^*, \{\{p\}\})}$$

$$\frac{\mathit{IsPred}(c) \quad p{=}\mathtt{null} \quad \mathit{Inv}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) = \bigvee(\exists u^* \cdot \pi)^*}{\mathit{XPure}_n(\mathtt{p}::\mathtt{c}\langle \mathtt{v}^* \rangle @u) =_{df} (\bigvee(\exists u^* \cdot \pi)^*, \emptyset)}$$

Figure 4.6: *XPure* : Translating to Pure Form

$$[\textbf{FV}-\textbf{METH}-\textbf{IMM}]$$

$$\frac{\begin{array}{c} V{=}\{v_m..v_n\} \quad W{=}\mathit{prime}(V) \\ \forall i=1,..,p \cdot (\ \vdash \{\Phi^i_{pr}{\wedge}\mathit{nochange}(V)\}\, e\, \{S^i_1\} \\ (\exists W{\cdot}S^i_1){\vdash}\Phi^i_{po} *_i S^i_2 \qquad S^i_2{\neq}\{\}) \end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\}}$$

- function *prime(V)* returns $\{v' \mid v \in V\}$.
- predicate *nochange(V)* returns $\bigwedge_{v\in V}(v = v')$. If $V = \{\}$, *nochange(V)*=*true*.
- $\exists W \cdot S$ returns $\{\exists W \cdot S_i | S_i \in S\}$.

At a method call, each of the method's precondition is checked, $\Delta{\vdash}\rho\Phi^i_{pr} *_i S_i$, where ρ represents a substitution of v_j by v'_j, for all $j = 1,..,n$. The combination of the residue S_i and the postcondition is added to the poststate (this addition might cause phase nesting). If a precondition is not entailed by the program state Δ, the corresponding residue is not added to the set of states. The test $\texttt{S}{\neq}\{\}$ ensures that at least one precondition is satisfied.

$$[\textbf{FV}-\textbf{CALL}-\textbf{IMM}]$$

$$\frac{\begin{array}{c} t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\} \in P \\ \rho{=}[v'_j/v_j]_{j=m}^{n} \qquad \Delta{\vdash}\rho\Phi^i_{pr} *_i S_i \quad \forall i{=}1,..,p \\ S = \bigcup_{i=1}^{p} \Phi^i_{po} * S_i \qquad S \neq \{\} \end{array}}{\vdash \{\Delta\}\, m(v_1..v_n)\, \{S\}}$$

Whenever there is a field access (read or update), the current state, Δ, must contain the node to be dereferenced, $\texttt{v}'{::}\texttt{c}\langle \texttt{v}_1,..,\texttt{v}_\texttt{n}\rangle$. For [$\textbf{FV}-\textbf{FIELD}-\textbf{READ}$], it is sufficient for the entailed node to be immutable as it will only be read. For [$\textbf{FV}-\textbf{FIELD}-\textbf{UPDATE}$], the node needs to be mutable, as it will be updated. As shown in the matching rules from Sec 4.5.2, for the case of the immutable node $\texttt{v}'{::}\texttt{c}\langle \texttt{v}_1,..,\texttt{v}_\texttt{n}\rangle@I$, the matching of the immutable node on the RHS

$$\frac{\begin{array}{c}[\mathbf{FV{-}FIELD{-}READ}]\\ \Delta \vdash^{\kappa}_{V} \mathtt{v}'::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle @I *_i S_1 \quad \mathit{fresh}\ v_1..v_n \quad S_1 \neq \emptyset \\ S_2 = \exists v_1..v_n \cdot (S_1 \wedge \mathtt{res}{=}v_i)\end{array}}{\vdash \{\Delta\}\, v.f_i\, \{S_2\}}$$

$$\frac{\begin{array}{c}[\mathbf{FV{-}FIELD{-}UPDATE}]\\ \Delta \vdash^{\kappa}_{V} \mathtt{v}'::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle @M * S_1 \quad \mathit{fresh}\ v_1..v_n \quad S_1 \neq \emptyset \\ S_2 = \exists v_1..v_n \cdot (S_1 * [v'_0/v_i]\mathtt{v}'::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle)\end{array}}{\vdash \{\Delta\}\, v.f_i{:=}v_0\, \{S_2\}}$$

Figure 4.7: Forward Verification Rules

with a corresponding node from the LHS will not consume the node from the LHS. Hence, for [**FV−FIELD−READ**] there is no need to add back the node at the end of the entailment. However, when entailing the mutable node $\mathtt{v}'::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle$ in the rule [**FV−FIELD−UPDATE**], the corresponding node on the LHS will be consumed and needs to be added back at the end of the entailment. Moreover, in the entailment $\Delta \vdash^{\kappa}_{V} \mathtt{v}'::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle @I *_i S_1$ from rule [**FV−FIELD−READ**], as the matching node on the LHS is not consumed, there might be heap sharing between the RHS and the residual states. Hence, we use the modified entailment procedure $\Phi_1 \vdash^{\kappa}_{V} \Phi_2 *_i S_R$ introduced in Sec 4.5.

Note that we use the primed notation for denoting the latest value of a variable. Correspondingly, $[v'_0/v_i]$ is a substitution that replaces the value v_i with the latest value of v'_0.

4.7 Soundness

In this section we present the soundness properties for both the forward verifier and the entailment prover. The storage model, semantic model and dynamic semantics are similar to those given in Sec 2.5, 2.6, and 2.7, respectively, with extensions for immutability annotations.

4.7.1 Storage Model

We define our storage model by making use of a domain of heaps, which is equipped with a partial operator for gluing together disjoint heaps. $\mathtt{h_0} \cdot \mathtt{h_1}$ takes the union of partial functions when $\mathtt{h_0}$ and $\mathtt{h_1}$ have disjoint domains of definition, and is undefined when $\mathtt{h_0}(\mathtt{l})$ and $\mathtt{h_1}(\mathtt{l})$ are both defined for at least one location $\mathtt{l} \in \mathit{Loc}$.

To define the model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in$ *Val* denoting `null`, *Var* of variables (program and logical variables), and *ObjVal* of object values stored in the heap, with $c[f_1 \mapsto \nu_1, .., f_n \mapsto \nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. Each object value has attached an immutability annotation from $\{I, M\}$. I means that the corresponding object value cannot be modified, while M allows its mutation.

$$h \in \mathit{Heaps} =_{df} \mathit{Loc} \rightharpoonup_{fin} \mathit{ObjVal} \times \{I, M\}$$
$$s \in \mathit{Stacks} =_{df} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}$$

4.7.2 Semantic Model of the Specification Formula

Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ. Function $dom(f)$ returns the domain of function f. We use $\mapsto$ to denote mappings, not the points-to assertion in separation logic, which has been replaced by $\mathtt{p::c}\langle \mathtt{v}^* \rangle @u$, $u \in \{I, M\}$ in our notation, as mentioned in Sec 4.3. The model relation for separation heap formulae is given in Def 4.7.1. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s. The $addImm$ function in Fig 4.8 propagates the immutability annotation u inside the heap formula Φ.

4.7.3 Dynamic Semantics

This section presents a small-step operational semantics for our language. The rules are given in Fig. 4.9. The machine configuration is represented by $\langle s, h, e \rangle$, where s denotes the current stack, h denotes the current heap, and e denotes the current program code. Note that the operational semantics must consider the mutability assertions recorded in the heap h. Each reduction step is formalized as a transition of the form: $\langle s, h, e \rangle \hookrightarrow \langle s_1, h_1, e_1 \rangle$. We have introduced an intermediate construct $\mathtt{ret}(v^*, e)$ to model the outcome of call invocation, where e denotes the residual code of the call. It is also used to handle local blocks.

Similar to Sec 2.7, we use k to denote a constant, $\perp$ to denote an undefined value, and $()$ to denote the empty expression (program). The operation $[v \mapsto \nu] + s$ "pops in" the variable v to s with the value ν, such that $([v \mapsto \nu] + s)(v) = \nu$. The operation $s - \{v^*\}$ "pops out" variables v^* from the stack s. The operation $s[v \mapsto \nu]$ changes the value of the most recent v in stack s to ν. The mapping $h[\iota \mapsto_u r]$ is the same as h except that it maps ι to r, with u as the attached

Definition 4.7.1 (Model for Specification Formula)**.**

$s,h \models \Phi_1 \vee \Phi_2$ iff $s,h \models \Phi_1$ *or* $s,h \models \Phi_2$

$s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi$ iff $\exists \nu_{1..n} \cdot s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n], h \models \kappa$ *and* $s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n] \models \pi$

$s,h \models \kappa_1 * \kappa_2$ iff $\exists h_1, h_2 \cdot h_1 \perp h_2$ *and* $h = h_1 \cdot h_2$ *and* $s, h_1 \models \kappa_1$ *and* $s, h_2 \models \kappa_2$

$s,h \models \kappa_1 \# \kappa_2$ iff $\exists h_1, h_2, h_3 \cdot h_1 \perp h_2 \perp h_3$ *and* $h = h_1 \cdot h_2 \cdot h_3$ *and* $s, h_1 \cdot h_3 \models \kappa_1$ *and* $s, h_2 \cdot h_3 \models \kappa_2$

$s,h \models \kappa_1 \wedge \kappa_2$ iff $\exists h_1, h_2, h_3 \cdot h_1 \perp h_2 \perp h_3$ *and* $h = h_1 \cdot h_2 \cdot h_3$ *and* $s, h_1 \cdot h_3 \models \kappa_1$ *and* $s, h_2 \cdot h_3 \models \kappa_2$

$s,h \models \texttt{emp}$ iff $dom(h) = \emptyset$

$s,h \models \texttt{p::c}\langle \texttt{v}_{\texttt{1..n}} \rangle @u$ iff `data` $c\,\{t_1\ f_1, .., t_n\ f_n\} \in P$, $h = [s(p) \mapsto_w r]$, *and* $r = c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)]$ *and* $u <: w$

`or` $(c\langle v_{1..n} \rangle \equiv \Phi$ `inv` $\pi) \in P$ *and* $s, h \models [p/\texttt{root}](addImm(\Phi, u))$

immutability annotation, $u \in \{I, M\}$. If no immutability annotation is present, then the mapping $h[\iota \mapsto r]$ maintains the previous immutability annotation. The mapping $h + [\iota \mapsto_u r]$ extends the domain of h with ι and maps ι to r with the immutability annotation u. The operation $h(l)[v_n \mapsto t_n]$ updates the field v_n of the object stored at location l in the heap h with the value t_n. We also make use of the function $type(v)$ to get the run-time type of the variable v.

In order to relate the logic with the storage model, we introduce an auxiliary construct, `assert` Φ, which checks whether the model relation $s,h \models \Phi$ holds. The forward verification rule for this intermediate construct is given below.

$$[\textbf{FV}-\textbf{ASSERT}]$$

$$\frac{\Delta \vdash^{\kappa}_{V} \Phi_1 *_i S_1 \quad S_1 \neq \emptyset}{\vdash \{\Delta\}\ \texttt{assert}\ \Phi_1\ \{\Delta\}}$$

The connection between the immutability information from the specification and the storage

$$addImm(\Phi, u) =_{df} addImm(\bigvee(\exists v^* \cdot \kappa \wedge \pi)^*, u) =_{df} \bigvee(\exists v^* \cdot addImm_\kappa(\kappa, u) \wedge \pi)^*$$

$$addImm_\kappa(\kappa_X\ C\ \kappa_Y, u) =_{df} addImm_\kappa(\kappa_X, u)\ C\ addImm_\kappa(\kappa_Y, u),$$
$$for\ X, Y \in \{R, W\}\ and\ C \in \{\#, *, \wedge\}$$

$$addImm_\kappa(\mathtt{p_2}::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle @u_1, u) =_{df} p_2::c\langle v_1, .., v_n\rangle @u$$

Figure 4.8: Function addImm

model is established at the beginning and end of each method execution, when the `assert` construct is used to check that the stack s and the heap h model the method's precondition and postcondition, respectively. Note that in the rule for `new` $c(v^*)$ we assume that the object newly created is immutable, and, whenever a field update takes place, the immutability annotation is updated to $@M$ (in the rule for field update $v_1.f_i := v_2$ in Fig 4.9).

4.7.4 Soundness of Verification

The soundness of our verification rules is defined with respect to the small-step operational semantics. We need to extract the post-state of a heap constraint by function $Post(\Delta)$ defined in Def 2.7.1.

Theorem 4.7.1 (Preservation)**.** *If*

$$\vdash \{\Delta\}\ e\ \{S_2\} \quad S_2 \neq \{\} \quad s, h \models \mathit{Post}(\Delta) \quad \langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

Then there exists $S_1 \neq \{\}$*, such that* $\forall \Delta_1 \in S_1 \cdot s_1, h_1 \models \mathit{Post}(\Delta_1)$ *and* $\vdash \{\Delta_1\}\ e_1\ \{S_3\} \quad S_3 \subseteq S_2$.

Proof: By structural induction on e. Details can be found in the Appendix.

Theorem 4.7.2 (Progress)**.** *If*

$$\vdash \{\Delta\}\ e\ \{S_1\} \quad S_1 \neq \{\} \quad s, h \models \mathit{Post}(\Delta)$$

then either e *is a value, or there exist* s_1, h_1*, and* e_1*, such that* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$.

Proof: By structural induction on e. Details can be found in the Appendix.

Theorem 4.7.3 (Safety)**.** *Consider a closed term* e *without free variables in which all methods have been successfully verified. Assuming unlimited stack/heap spaces and that* $\vdash \{\mathtt{true}\}\ e\ \{\Delta\}$,

$$\langle s,h,v\rangle\hookrightarrow\langle s,h,s(v)\rangle \qquad \langle s,h,k\rangle\hookrightarrow\langle s,h,k\rangle \qquad \langle s,h,v.f\rangle\hookrightarrow\langle s,h,h(s(v))(f)\rangle$$

$$\langle s,h,v{:=}k\rangle\hookrightarrow\langle s[v{\mapsto}k],h,()\rangle \qquad \langle s,h,();e\rangle\hookrightarrow\langle s,h,e\rangle$$

$$\frac{\langle s,h,e_1\rangle\hookrightarrow\langle s_1,h_1,e_3\rangle}{\langle s,h,e_1;e_2\rangle\hookrightarrow\langle s_1,h_1,e_3;e_2\rangle} \qquad \frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,v{:=}e_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_1\rangle} \qquad \frac{s(v){=}\texttt{false}}{\langle s,h,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,e_2\rangle}$$

$$\langle s,h,\{t\ v;\ e\}\rangle\hookrightarrow\langle[v{\mapsto}\bot]{+}s,h,\texttt{ret}(v,e)\rangle$$

$$\langle s,h,\texttt{ret}(v^*,k)\rangle\hookrightarrow\langle s{-}\{v^*\},h,k\rangle$$

$$\frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,\texttt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\texttt{ret}(v^*,e_1)\rangle}$$

$$\frac{\begin{array}{c}type(v_1)=c\langle w_1,..,w_n\rangle\\ r{=}h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1{=}h[s(v_1){\mapsto_M}r]\end{array}}{\langle s,h,v_1.f_i:=v_2\rangle\hookrightarrow\langle s,h_1,()\rangle}$$

$$\frac{s,h\models\Phi}{\langle s,h,\texttt{assert } \Phi\rangle\hookrightarrow\langle s,h,()\rangle} \qquad \frac{s,h\models_i\Phi}{\langle s,h,\texttt{assert}_{\texttt{imm}}\ \Phi\rangle\hookrightarrow\langle s,h,()\rangle}$$

$$\frac{\texttt{data } c\ \{t_1\ f_1,..,t_n\ f_n\}{\in}P \quad \iota{\notin}dom(h) \quad r{=}c[f_1{\mapsto}s(v_1),..,f_n{\mapsto}s(v_n)]}{\langle s,h,\texttt{new } c(v^*)\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto_I r],\iota\rangle}$$

$$\frac{\begin{array}{c}t_0\ mn((\texttt{ref } t_j\ w_j)_{j=1}^{m-1},(t_j\ w_j)_{j=m}^{n})\ \{\texttt{requires } \Phi_{pr}^i\ \texttt{ ensures } \Phi_{po}^i\}_{i=1}^{p}\ \{e\}\\ s_1{=}[w_j{\mapsto}s(v_j)]_{j=m}^{n}{+}s \qquad \langle s_1,h,(\texttt{assert } \Phi_{pr}^i)_{i=1}^{p}\rangle\hookrightarrow\langle s_1,h,()\rangle\\ \langle s_1,h,\texttt{ret}(\{w_j\}_{j=m}^{n},[v_j/w_j]_{j=1}^{m-1}e;(\texttt{assert } \Phi_{po}^i)_{i=1}^{p})\rangle\hookrightarrow\langle s,h_1,e_1\rangle\end{array}}{\langle s,h,mn(v^*)\rangle\hookrightarrow\langle s,h_1,e_1\rangle}$$

Figure 4.9: Small-Step Operational Semantics

then either $\langle [], [], e\rangle \hookrightarrow^* \langle [], h, v\rangle$ *terminates with a value* v *that is subsumed by the postcondition* Δ*, or it diverges* $\langle [], [], e\rangle \not\hookrightarrow^*$.

Theorem 4.7.4 (Soundness of Entailment)**.** *If entailment check* $\Delta_1 \vdash \Delta_2 *_i S$ *succeeds, we have: for all* $s, h,$ *and* $\Delta \in S$ *, if* $s, h \models \Delta_1$ *then* $s, h \models \Delta_2 *_i \Delta$.

Proof: Details can be found in the Appendix.

4.8 Experimental Evaluation

We have built a prototype system using Objective Caml. The proof obligations generated by our verification are discharged using a number of off-the-shelf constraint solvers (like Omega Calculator [105]) or theorem provers (like Isabelle [95] and Mona [64]).

Preliminary experiments were conducted by testing our system on a suite of examples summarized in Figure 4.10. The examples can handle data structures with sophisticated shape and size properties, such as sorted lists, and balanced trees, in a uniform way. Method `sum` is the one described in Sec 4.3.1. Methods `insert` and `delete` refer to the insertion and deletion of a value into/from the corresponding data structure, respectively. Moreover, we verify a suite of sorting algorithms, which receive as input an unsorted singly-linked list and return a sorted list.

The benchmarks under the category **Big Naturals** consider the representation of a big natural as a list, with each node containing a decimal digit. The order of recording the digits is such that the head of the list contains the least significant digit. We make use of the following heap predicate for describing the singly-linked list containing the big natural number:

$$\begin{array}{l}\texttt{root::bignat}\langle \texttt{v}\rangle \equiv \texttt{root=null} \wedge \texttt{v=0} \\ \quad \vee \texttt{root::node}\langle \texttt{p}, \texttt{q}\rangle * \texttt{q::bignat}\langle \texttt{v}_1\rangle \wedge 0 \leq \texttt{p} \leq 9 \\ \qquad \wedge \texttt{v=10}*\texttt{v}_1+\texttt{p} \wedge \texttt{v}>0 \\ \texttt{inv v} \geq 0;\end{array}$$

The definition asserts that a big natural is either an empty list denoting the value 0 (the base case $\texttt{root=null} \wedge \texttt{v=0}$), or it consists of a head data node ($\texttt{root::node}\langle \texttt{p}, \texttt{q}\rangle$) and a separate tail data structure which is also a big natural ($\texttt{q::bignat}\langle \texttt{v}_1\rangle$). For the latter case, the value of the big natural is computed as $\texttt{v=10}*\texttt{v}_1+\texttt{p}$. To ensure uniqueness in representing the value 0, this predicate also adds the constraint $\texttt{v}>0$ for non-`null` representations of big natural numbers. Using this definition, the methods under the **Big Naturals** category compute the addition, subtraction

Program codes	LOC	Timings [secs]	Heap sharing	Immutability degree [%]	Heap reduction [%]	Aliasing scenarios reduction[%]
Data Nodes (verifies data node value)						
sum	1	0.04	✓	100	50	50
Big Naturals (verifies bignat value)						
addition	29	0.74	✓	62	38	40
subtraction	24	0.76	✓	62	38	40
multiplication	23	0.97	✓	71	36	25
compare	20	0.66	✓	100	50	50
karatsuba_mult	64	3.82	✓	60	60	33
Linked List (verifies shape + length)						
length	6	0.05	χ	100	50	0
append	8	0.08	✓	66	40	50
List Segment (verifies shape + length)						
append	8	0.06	✓	50	33	50
Sorted List (verifies shape + bounds + sortedness)						
insert	17	0.26	χ	50	33	0
insertion_sort	45	0.32	✓	57	36	25
selection_sort	52	0.37	χ	57	36	0
bubble_sort	42	0.45	χ	40	29	0
merge_sort	105	0.51	✓	77	35	20
quick_sort	85	0.68	✓	50	33	20
AVL Tree (verifies shape + height + balance factor)						
insert	169	5.63	χ	44	30	0
delete	287	8.92	✓	69	40	74
Perfect Tree (verifies shape + height + perfectness)						
insert	89	0.34	χ	50	33	0
Binary Search Tree (verifies shape + min + max + sortedness)						
insert	40	0.45	χ	50	33	0
delete	62	0.49	χ	50	33	0
Priority Queue (verifies shape + size + height + max-heap)						
insert	54	0.89	χ	50	33	0
delete_max	140	2.73	✓	38	28	25
Red-Black Tree (verifies shape + size + black-height)						
insert	167	2.65	✓	75	42	75
delete	430	15.16	✓	70	42	93

Figure 4.10: Experimental Results

and multiplication of two big naturals in the corresponding methods, respectively. Under the same category, method `compare` takes as input two big naturals and returns 0 if they are equal, 1 if the first one in bigger, and -1 if the first one is smaller. Lastly, method `karatsuba_mult` uses the fast multiplication Karatsuba algorithm for multiplying two big naturals.

The second column of Figure 4.10 contains the number of lines of code, followed by the timings in the third column. The fourth column records whether or not the specification contains

heap sharing, meaning if there are any usage of $\wedge$ in the heap description. We mark the presence of heap sharing by $\checkmark$, and the absence by χ. The fifth column contains the immutability degree of a specification, denoting the percentage of immutable heap predicates out of the total number of heap predicates in the specification. The sixth and seventh columns record the reduction in the size of the specification when using the immutability enhancements proposed by the current work. The sixth column represents the heap reduction, which is due to the fact that the heap predicates marked as immutable in the precondition are guaranteed to be preserved by the corresponding method, and do not have to be mentioned in the postcondition. The last column registers the reduction in the number of aliasing scenarios that have to be recorded by the specification. This happens when several aliasing scenarios are unified through the use of $\wedge$ in the heap description. In order to compute the timings, immutability degree, heap reduction and aliasing scenarios reduction for each benchmark, we took into consideration all the functions that are being called.

For illustration, let us consider the results recorded for the method `sum` under **Data Nodes**. The specification, which is given below, employs heap sharing due to the presence of $\wedge$ in the precondition. The immutability degree is `100`% (all the heap predicates in the specification are immutable), the reduction of heap predicates is `50`% (the two immutable heap predicates from the precondition do not have to be repeated in the postcondition) and the reduction of aliasing scenarios is `50`% (the scenario when `x` and `y` are aliases is unified with the scenario when they are disjoint).

```
int sum(node x, node y)
  requires x::node⟨a, _⟩@I∧y::node⟨b, _⟩@I
  ensures res=a+b;
```

The results of our preliminary experiments recorded in Fig 4.10 can be interpreted as follows:

- The heap sharing and immutability degree (fourth and fifth columns, respectively) are measures on the applicability of our enhancements. The potential heap sharing in a specification creates the opportunity to use $\wedge$ in the heap description for unifying multiple aliasing scenarios. Fig 4.10 shows that for 15 out of all 24 benchmarks, there is good potential for heap sharing in the specification. Moreover, the immutability degree recorded in the figure confirms the fact that, on average, from all the heap that a method accesses,

62% can be symbolically analyzed for read-only. Hence, only less than half is marked for mutation. The heap that is not being updated can be declared as immutable in the associated specification. These results affirms the applicability of our approach.

- The heap and aliasing scenarios reduction (sixth and seventh columns, respectively) are used to quantify the gains recorded by our current enhancements with respect to the conciseness of the specification. The data shows that, on average, there is a 38% reduction in the number of heap predicates, and a 28% reduction in the number of aliasing scenarios. This heap and aliasing scenarios reduction shows an improvement in the specification conciseness when using the immutability annotations proposed by the current work.

- The immutability degree together with the heap reduction (fifth and sixth columns, respectively) are measures that relate to the improvement on the precision of the specifications, as they denote the cases where cut-point preservations are possible.

When comparing the verification timings for the immutability-enhanced approach with those for the base approach, we found them to be similar. The biggest differences were recorded in the case of the `insert` method for red-black tree (1.2 secs in favor of the base approach), and of the `insert` method for the AVL tree (1.1 secs in favor of the immutability-enhanced approach).

CHAPTER V

CASE STRUCTURED SPECIFICATIONS

5.1 Motivation

Recent developments of the specification mechanisms have focused mostly on expressiveness [8, 5, 17] (to support verification for more properties), abstraction [100, 97] (to support information hiding in specification) and modularity [70, 21, 35] (to support more readable and reusable specifications). To the best of our knowledge, there has been hardly any attempt on the development of specification mechanisms that could support better verifiability (in terms of both efficiency and effectiveness). Most efforts on better verifiability have been confined to the verification technology; an approach that may lead to less portability (as we become more reliant on clever heuristics from the verification tools) and also more complex implementation for the verification tools themselves. In this thesis, we shall propose a novel approach towards better verifiability that focuses on new structures in the specification mechanism instead.

To illustrate the need for an enhanced specification mechanism, we will make use of separation logic, which allows for a precise description of heap-based data structures and their properties. As an example, consider a data node `node2` and a predicate describing an AVL tree that captures the size property via `s` and the height via `h`:

$$\texttt{data node2}\,\{\,\texttt{int val; int height; node2 right; node2 left;}\,\}$$

$$\begin{array}{l}\texttt{avl}\langle \texttt{h},\texttt{s}\rangle \equiv \texttt{root=null} \wedge \texttt{h=0} \wedge \texttt{s=0}\\ \quad \vee\ \texttt{root::node2}\langle \_,\texttt{h},\texttt{r},\texttt{l}\rangle * \texttt{r::avl}\langle \texttt{h}_1,\texttt{s}_1\rangle * \texttt{l::avl}\langle \texttt{h}_2,\texttt{s}_2\rangle \wedge \texttt{h} = \texttt{max}(\texttt{h}_1,\texttt{h}_2)+1\\ \quad \wedge -1 \leq \texttt{h}_1-\texttt{h}_2 \leq 1 \wedge \texttt{s}=\texttt{s}_1+\texttt{s}_2+1\\ \texttt{inv}\ \texttt{h} \geq 0 \wedge \texttt{s} \geq 0;\end{array}$$

The aforementioned definition asserts that an AVL tree is either empty (the base case $\texttt{root=null} \wedge \texttt{h=0} \wedge \texttt{s=0}$), or it consists of a data node ($\texttt{root::node2}\langle \_,\texttt{h},\texttt{r},\texttt{l}\rangle$) and two disjoint subtrees ($\texttt{r::avl}\langle \texttt{h}_1,\texttt{s}_1\rangle * \texttt{l::avl}\langle \texttt{h}_2,\texttt{s}_2\rangle$). Each node is used to store the actual data in the `val` field, and the maximum height of the current subtree in the `height` field. The constraint $-1 \leq \texttt{h}_1-\texttt{h}_2 \leq 1$ states that the tree is balanced, while $\texttt{s}=\texttt{s}_1+\texttt{s}_2+1$ and $\texttt{h}=\texttt{max}(\texttt{h}_1,\texttt{h}_2)+1$ compute the size and

height of the tree pointed by root from the properties s_1, s_2 and h_1, h_2, respectively, that are obtained from the two subtrees. The $*$ connector ensures that the head node and the right and left subtrees reside in disjoint heaps. We also specify a default invariant, $h \geq 0 \wedge s \geq 0$, that holds for all AVL trees.

Next, we specify a method that attempts to retrieve the height information from the root node of the data structure received as argument. In case the argument has the value null, the method returns 0, as captured by res=0. To provide a suitable link between pre- and post-conditions, we use the logical variables v, h, lt, lr that have to be instantiated for each call to the method. As a first try, we capture both the null and non-null scenarios as a composite formula consisting of a disjunction of the two cases, as shown below:

$$\begin{array}{l}\texttt{int get\_height(node2 x)}\\ \quad \texttt{requires x=null} \vee \texttt{x::node2}\langle v, h, lt, lr\rangle\\ \quad \texttt{ensures (x=null} \wedge \texttt{res=0)} \vee (\texttt{x::node2}\langle v, h, lt, lr\rangle \wedge \texttt{res=h});\\ \{\texttt{if (x = null) then 0 else x.height}\}\end{array}$$

This specification introduces disjunctions both in the pre and post-conditions, which would make the verification process perform search over the disjuncts[92]. Basically, each disjunct corresponds to an acceptable scenario of which at least one needs to be proven. However, there are situations when the program state does not contain enough information to determine which of the scenarios applies. For illustration, let us consider that we are interested in retrieving the height information for an AVL tree pointed by x and the program state before the call to the get_height method is $\texttt{x::avl}\langle h_1, s_1\rangle$. We have to verify that the current program state obeys the method's precondition. However, when verifying the null and non-null scenarios separately, both checks fail as the program state $\texttt{x::avl}\langle h_1, s_1\rangle$ does not contain sufficient information to conclude neither that $\texttt{x} \neq \texttt{null}$, nor that x=null. We provide the two failing verification conditions below. As none of the following two entailments succeeds, the verification of the method call fails.

$$\texttt{x::avl}\langle h_1, s_1\rangle \vdash (\texttt{x=null}) * \Phi_{r_1}$$

$$\texttt{x::avl}\langle h_1, s_1\rangle \vdash (\texttt{x::node2}\langle v, h, lt, lr\rangle) * \Phi_{r_2}$$

As a second try, we write the specification in a modular fashion by separating the two scenarios as advocated by past works ([70, 21] and Chapter 2). In [70], Leavens and Baker proposed

for each specification to be decomposed into multiple specifications (where it is called case analysis) to capture different scenarios of usage. Their goal was improving the readability of specifications, as smaller and simpler specifications are easier to understand than larger ones. In Chapter 2 multiple specifications were advocated to help achieve more scalable program verification. By using multiple pre/post conditions, we obtain the following specification:

$$\begin{array}{l}\texttt{int get\_height(node2 x)}\\ \quad\texttt{requires x=null}\\ \quad\texttt{ensures res=0;}\\ \quad\texttt{requires x::node2}\langle\texttt{v,h,lt,rt}\rangle\\ \quad\texttt{ensures x::node2}\langle\texttt{v,h,lt,rt}\rangle\wedge\texttt{res=h;}\end{array}$$

During the verification process, each scenario (denoted by a pre/post-condition pair) is proven separately (Chapter 2). However, neither of the two entailments (for each of the two scenarios) succeeds, causing the verification of the method call to fail.

A possible solution is to perform case analysis on variable x: first assume x=null, then assume x≠null, and try to prove both cases. For soundness, these cases must be disjoint and exhaustively cover all scenarios. Accordingly, the following two provable entailments are obtained, and the verification succeeds:

$$\texttt{x::avl}\langle\texttt{h}_1,\texttt{s}_1\rangle\wedge\texttt{x=null}\vdash(\texttt{x=null})*\Phi_{r_1}$$
$$\texttt{x::avl}\langle\texttt{h}_1,\texttt{s}_1\rangle\wedge\texttt{x}\neq\texttt{null}\vdash(\texttt{x::node2}\langle\texttt{v,h,lt,lr}\rangle)*\Phi_{r_2}$$

However, case analysis is not always available in provers, as it might be tricky to decide on the condition for a case split.

5.2 Chapter Overview

In the rest of the chapter we shall focus on the apparatus for writing and verifying (or checking) structured specifications. Sec 5.3 provides examples to motivate the need for structured specifications, whereas Sec 5.4 formalizes the notion of structured specifications. Sec 5.5 introduces the verification rules to generate Hoare triples, and Sec 5.6 presents the entailment proving for structured specifications. Sec 5.8 presents our experimental results.

5.3 Examples

In the current section we present two more examples that motivate our enhancements to the specification mechanism.

5.3.1 Example 1

Consider a method that receives two AVL trees, `t1` and `t2`, and merges them by recursively inserting all the elements of `t2` into `t1`. By using the case construct introduced in Sec 4.1 we may write a case structured specification, which captures information about the resulting tree size when `t1` is not `null`, and about the resulting size and height, whenever `t1` is `null`:

$$\begin{array}{l}\texttt{case}\{\texttt{t}_1 = \texttt{null} \rightarrow \texttt{requires}\ \texttt{t}_2{::}\texttt{avl}\langle \texttt{s}_2, \texttt{h}_2\rangle \\ \qquad\quad \texttt{ensures}\ \texttt{res}{::}\texttt{avl}\langle \texttt{s}_2, \texttt{h}_2\rangle; \\ \quad \texttt{t}_1 \neq \texttt{null} \rightarrow \texttt{requires}\ \texttt{t}_2{::}\texttt{avl}\langle \texttt{s}_2, \texttt{h}_2\rangle * \texttt{t}_1{::}\texttt{avl}\langle \texttt{s}_1, \_\rangle \\ \qquad\quad \texttt{ensures}\ \texttt{res}{::}\texttt{avl}\langle \texttt{s}_1{+}\texttt{s}_2, \_\rangle\};\end{array}$$

However, let us note that there is a redundancy in this specification, namely the same predicate $\texttt{t}_2{::}\texttt{avl}\langle \texttt{s}_2, \texttt{h}_2\rangle$ appears on both branches of the case construct. After the need for a case construct which was already discussed in Sec 5.1, this is the second deficiency we shall address in our specification mechanism, that is due to a lack of sharing in the logic formula which in turn causes repeated proving of identical sub-formulae. To provide for better sharing of the verification process, we propose to use *staged* formulae of the form $(\Phi_1\ \mathbf{then}\ \Phi_2)$, to allow sub-formula Φ_1 to be proven prior to Φ_2.

Though $(\Phi_1\ \mathbf{then}\ \Phi_2)$ is semantically equivalent to $(\Phi_1 * \Phi_2)$, we stress that the main purpose of adding this new structure is to support more effective verification with the help of specifications with less redundancy. By itself, it is not meant to improve the expressivity of our specification, but rather its effectiveness. Nevertheless, when it is used in combination with the case construct, it could support case analysis of logical variables to ensure successful verification. The same structuring mechanisms can be used by formulae in both predicate definitions and pre/post specifications.

Getting back to the AVL merging example, the redundancy in the specification can be factored out by using a staged formulae, as follows:

$$\texttt{requires } t_2\texttt{::avl}\langle s_2, h_2\rangle \texttt{ then}$$
$$\texttt{case}\{t_1 = \texttt{null} \rightarrow \texttt{ensures res::avl}\langle s_2, h_2\rangle;$$
$$t_1 \neq \texttt{null} \rightarrow \texttt{requires } t_1\texttt{::avl}\langle s_1, \_\rangle$$
$$\texttt{ensures res::avl}\langle s_1 + s_2, \_\rangle\};$$

During the verification process, when reaching a call to the AVL merging method, the current program state must entail the method's precondition. Since the entailment process needs to explore both branches of the specification, the $t_2\texttt{::avl}\langle s_2, h_2\rangle$ node will be proven twice for each method call. By using staged formulae, the second specification will force the common formula to be proved only once. Although the two specifications capture the same information, the second version requires much less proving effort. For this example, there was a 40% reduction in verification time by our system, due solely to the presence of staged formulae.

For the general case, if x denotes the number of heap nodes/predicates that are shared in the consequent formula, and y the number of possible matchings from the antecedent, then the number of redundant matchings that are eliminated is $(x - 1) * y$. An analogy can be made between the use of the staged formula and the use of the binary decision diagram (BDD) as an intermediate representation for SAT formulae to support better sharing of identical sub-formulae [16]. Where applicable, we expect staged formulae to improve the effectiveness of verification.

5.3.2 Example 2

Parameter instantiation is needed primarily for connecting the logical variables between precondition and postcondition of specifications. Traditionally, manual instantiation of ghost variables has played this role. In this thesis, we propose two new mechanisms, early and late instantiations, to support automatic instantiations of logical variables. As an example, consider a data node cell and a predicate cellPred defined as follows:

$$\texttt{data cell } \{ \texttt{ int val}\}$$
$$\texttt{cellPred}\langle i\rangle \equiv \texttt{root=null} \wedge i \leq 3 \vee \texttt{root::cell}\langle \_\rangle \wedge i > 3$$

To highlight the difference between early and late instantiations, we shall consider two separate proof obligations. The first one is given below.

$$p\texttt{::cell}\langle \_\rangle \vdash (p\texttt{::cellPred}\langle j\rangle \wedge j > 2) * \Phi_r$$

At this point, we first need to match a heap predicate $\texttt{p::cellPred}\langle\texttt{j}\rangle$ on the RHS with a data node $\texttt{p::cell}\langle\_\rangle$ on the LHS to obtain an instantiation for the variable $\texttt{j}$. A fundamental question is whether the variable instantiation could occur for just the predicate $\texttt{p::cellPred}\langle\texttt{j}\rangle$ (we refer to this as *early instantiation*), or it has to be for the entire formula $\texttt{p::cellPred}\langle\texttt{j}\rangle \wedge \texttt{j}{>}2$ (known as *late instantiation*). By default, our system uses early (or implicit) instantiation for variables that are not explicitly declared. In this scenario, early instantiation $\texttt{j}{>}3$ is obtained when folding with the predicate $\texttt{p::cellPred}\langle\texttt{j}\rangle$. This instantiation is transferred to the LHS. Consequently, we obtain a successful proof below.

$$\texttt{j}{>}3 \vdash (\texttt{j}{>}2){*}\Phi_r$$

Now, let us consider a second proof obligation that will require late instantiation:

$$\texttt{p}{=}\texttt{null} \vdash (\texttt{p::cellPred}\langle\texttt{j}\rangle{\wedge}\texttt{j}{>}2){*}\Phi_r$$

Similar to the previous case, we will first use a default early instantiation mechanism. After matching $\texttt{p::cellPred}\langle\texttt{j}\rangle$, we obtain the instantiation $\texttt{j}{\leq}3$. However, moving only this binding to the LHS is not enough, causing the proof below to fail.

$$\texttt{p}{=}\texttt{null} \wedge \texttt{j}{\leq}3 \vdash (\texttt{j}{>}2){*}\Phi_r$$

To support late instantiation for variable $\texttt{j}$, we declare it explicitly using $[\texttt{j}]$ below:

$$\texttt{p}{=}\texttt{null} \vdash ([\texttt{j}]\ \texttt{p::cellPred}\langle\texttt{j}\rangle{\wedge}\texttt{j}{>}2){*}\Phi_r$$

This time variable $\texttt{j}$ is kept on the RHS until the end of the entailment. As its proof below succeeds, the instantiation for $\texttt{j}$ will be captured in the residue as $\Phi_r{=}\texttt{j}{\leq}3{\wedge}\texttt{j}{>}2$.

$$\texttt{p}{=}\texttt{null} \vdash (\exists\texttt{j}.\texttt{j}{\leq}3{\wedge}\texttt{j}{>}2){*}\Phi_r$$

Though late instantiation is more general, it may require existential quantifications over a larger formula. Hence, by default, we prefer to use early instantiation where possible, and leave it to the user to manually declare where late instantiation is mandated.

5.4 Specification and Programming Language

We shall now focus on the structured specifications mechanism. Fig 5.1 provides a syntactic description where Z denotes structured (pre/post) specifications, while Q denotes structured

Shape pred.	*spred*	::=	$c\langle v^* \rangle \equiv \mathsf{Q}$ `inv` π	
Pre/Post.	Z	::=	$\exists v_1^* \cdot \mathsf{Y}_1 \ldots \exists v_n^* \cdot \mathsf{Y}_n$	multiple specs
	Y	::=	`case`$\{\pi_1 \Rightarrow \mathsf{Z}_1; \ldots; \pi_n \Rightarrow \mathsf{Z}_n\}$	case construct
			| `requires` $[w^*]$ Φ [`then`] Z	staged spec
			| `ensures` Q	post
Formula	Q	::=	$\bigvee \exists v^* \cdot \mathsf{R}$	multiple disjuncts
	R	::=	`case`$\{\pi_1 \Rightarrow \mathsf{Q}_1; \ldots; \pi_n \Rightarrow \mathsf{Q}_n\}$	case construct
			| $[w^*]$ Φ [**then** Q]	staged formula

Figure 5.1: Structured Specifications

formulae that may be used for pre/post specifications, as well as for predicate definitions. Apart from multiple specifications, our new syntax includes case constructs and staged formulae. Take note that the syntax for the rest of the formulae, namely the pure formula π, the heap formula κ, and the formula Φ remain the same as those given in Fig 2.2.

For structured specification, the `requires` keyword introduces a part of precondition through a staged specification. The postcondition is captured after each `ensures` keyword, which must appear as a terminating branch for the tree-like specification format. We support late instantiation via variables w^*, from `requires` $[w^*]$ Φ Z and $[w^*]$ Φ [**then** Q] at the end of proving Φ. To minimise user annotations, our system automatically determines the other unbound variables (different from those to be late instantiated) as either existential or to be early instantiated.

Our construct to support case analysis is `case`$\{\pi_1 \Rightarrow \mathsf{Z}_1; \ldots; \pi_n \Rightarrow \mathsf{Z}_n\}$ for specification, and `case`$\{\pi_1 \Rightarrow \mathsf{Q}_1; \ldots; \pi_n \Rightarrow \mathsf{Q}_n\}$ for formula. We impose the following three conditions on $\pi_1, \ldots, \pi_n$:

(i) are *restricted* to only pure constraints, without any heap formula.

(ii) are *exclusive*, meaning that $\forall i, j \cdot i \neq j \rightarrow \pi_i \wedge \pi_j = \texttt{false}$.

(iii) are *exhaustive*, meaning that $\pi_1 \vee \ldots \vee \pi_n = \texttt{true}$.

Condition (i) is imposed since pure formula can be freely duplicated. Condition (ii) is imposed to avoid conjunction over the heap-based formula. If absent, each heap state may have

to satisfy multiple case branches. Condition (iii) is needed for soundness of case analysis which requires all scenarios to be considered. To illustrate, consider:

$$[(w:t)^*]\ \Phi\ \texttt{case}\{x{=}\texttt{null}{\Rightarrow}\mathsf{Q}_1;\ x{\neq}\texttt{null}{\Rightarrow}\mathsf{Q}_2\}$$

The first condition holds as the two guards, $x{=}\texttt{null}$ and $x{\neq}\texttt{null}$, are pure. Furthermore, our system checks successfully that the guards are exclusive ($(x{=}\texttt{null}{\wedge}x{\neq}\texttt{null}){=}\texttt{false}$) and exhaustive ($(x{=}\texttt{null} \vee x{\neq}\texttt{null}) = \texttt{true}$).

5.5 Forward Verification

The main goal of structured specification is to support a modular verification process that could be carried out efficiently and precisely. In this section, we propose a set of rules to help generate Hoare-style triples for code verification, together with entailment checking to support proof obligations over the structured formulae domain.

To better support structured specifications and case analysis, we propose a new triple of the form $\{\!|\Phi|\!\}\ e\ \{\!|\mathsf{Z}|\!\}$, with *pre* being an unstructured formula and Z being the structured specification. We use structured specifications in the poststate because our case analysis is guided from the post-states. In contrast, unstructured formulae are used in the prestate since the structured form is unnecessary here. The semantic meaning of this new triple is defined as follows:

Definition 5.5.1. *The validity of* $\{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}|\!\}$ *is defined inductively over the structure of* Z*. That is:*

if $\mathsf{Z} \equiv \texttt{ensures}\ \mathsf{Q}$:
$\models \{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}|\!\} \iff \models \{\Phi\}\mathsf{e}\{\mathsf{Q}\};$

if $\mathsf{Z} \equiv \texttt{requires}\ \Phi_1\ [\texttt{then}]\ \mathsf{Z}_1$:
$\models \{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}|\!\} \iff \models \{\!|\Phi{*}\Phi_1|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}_1|\!\};$

if $\mathsf{Z} \equiv \texttt{case}\{\pi_1{\Rightarrow}\mathsf{Z}_1;\ldots;\pi_n{\Rightarrow}\mathsf{Z}_n\}$:
$\models\{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}|\!\} \iff \forall i{\in}\{1,..,n\}{\cdot}\models\{\!|\Phi{\wedge}\pi_i|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}_i|\!\};$

if $\mathsf{Z} \equiv (\exists v_1^*{\cdot}\mathsf{Y}_1 \ldots \exists v_n^*{\cdot}\mathsf{Y}_n)$:
$\models \{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\mathsf{Z}|\!\} \iff \forall i{\in}\{1,..,n\}{\cdot}\ \models \{\!|\Phi|\!\}\ \mathsf{e}\ \{\!|\exists v_i^*{\cdot}Y_i|\!\})\Box$

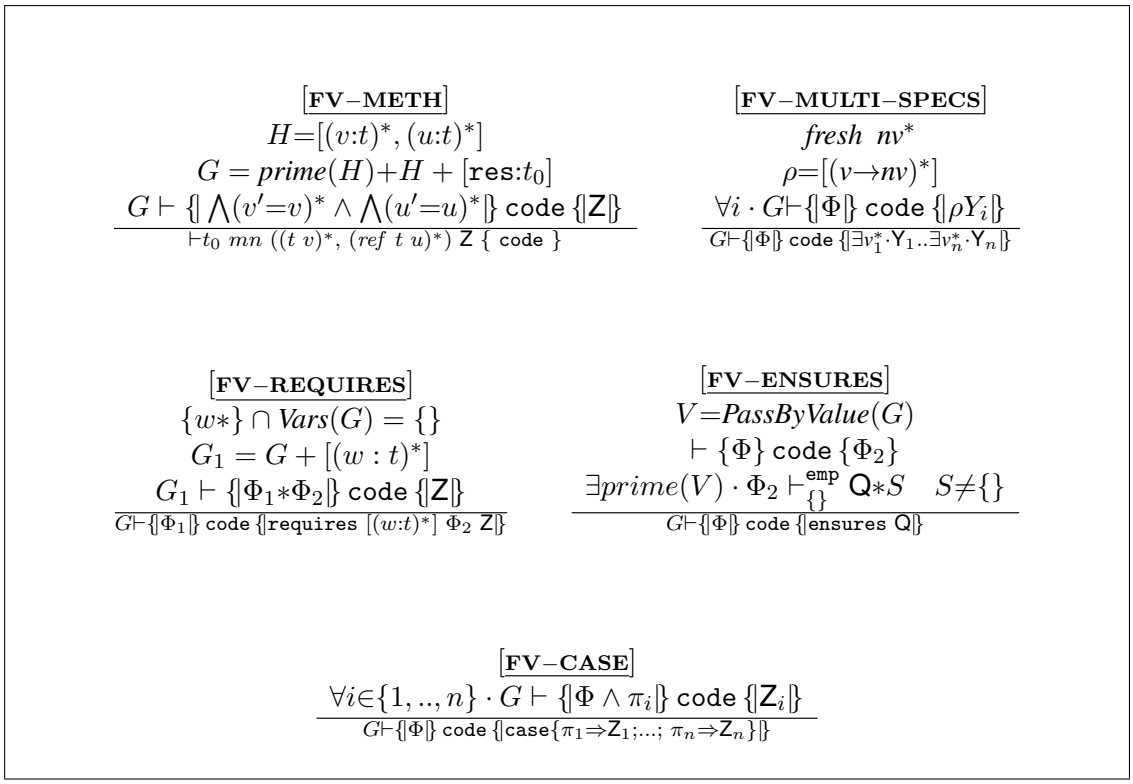

Figure 5.2: Building Verification Rules for Structured Specifications

Our main verification rules are given in Fig. 5.2. Note that G records a list of variables (including `res` as result of the code) visible to the code verifier. Our specification formulae use both primed and unprimed notations, where primed notations represent the latest values of program variables, and unprimed notations denote either logical variables or initial values of program variables.

The verification of method declarations is described by the [FV−METH] rule. It verifies the method body `code` against the specification Z, as indicated by the rule. The function *prime*$(\{v_1, .., v_m\})$ returns the primed version $\{v'_1, .., v'_m\}$. The third line of the premise deals with the verification task $G \vdash \{\!|\bigwedge(v'{=}v)^* \wedge \bigwedge(u'{=}u)^*|\!\}$ `code` $\{\!|\mathsf{Z}|\!\}$, where the precondition indicates that the latest values of program variables are the same as their initial values. The other rules are syntax-directed and rely on the structure of the specification Z.

The rule [FV−MULTI−SPECS] deals with the case where the post-state is a multi-specification. It verifies the code against each of the specifications. Note that the substitution ρ replaces variables v^* with fresh variables nv^*. The rule [FV−REQUIRES] deals with the case where the post-state starts with a `requires` clause. In this case, the formula in the `requires` clause

is added to the pre-state (by separation conjunction) before verifying the code against the remaining part of the specification in the post-state. The variables for late instantiation (w^*) are also attached to the end of the list G. The rule [FV–ENSURES] deals with the case where the post-state starts with an `ensures` clause. It invokes our forward verification rules to derive the strongest postcondition Φ_2 for the normal Hoare triple $\{\Phi\}\texttt{code}\{\Phi_2\}$ and invokes the entailment prover (described in the next section) to check that the derived post-state Φ_2 subsumes the given post-condition Q (The test $S \neq \{\}$ signifies the success of this entailment proof). Note that V denotes the set of pass-by-value parameters that are not modified by the procedure. Hence, their values (denoted by primed variables) are ignored in the postcondition, even if the program code may have updated these parameters. The last rule [FV–CASE] deals with the case where the post-state is a case specification. It verifies in each case the specification Z_i is met when the guard π_i is assumed in the pre-state.

To illustrate the generation of the verification tasks, consider the AVL merging given in Section 5.3.1. By applying the rules from Figure 5.2, two Hoare triples are produced.

$$\vdash \{\mathtt{t_2}{::}\mathtt{avl}\langle \mathtt{s_2}, \mathtt{h_2}\rangle \wedge \mathtt{t_1}{=}\mathtt{null}\}\ \mathtt{code}\ \{\mathtt{res}{::}\mathtt{avl}\langle \mathtt{s_2}, \mathtt{h_2}\rangle\}$$

$$\vdash \{\mathtt{t_1}{::}\mathtt{avl}\langle \mathtt{s_1}, \_\rangle * \mathtt{t_2}{::}\mathtt{avl}\langle \mathtt{s_2}, \mathtt{h_2}\rangle \wedge \mathtt{t_1}{\neq}\mathtt{null}\}\ \mathtt{code}\ \{\mathtt{res}{::}\mathtt{avl}\langle \mathtt{s_1}{+}\mathtt{s_2}, \_\rangle\}$$

5.6 Entailment Checking

In the current section, we enhance the entailment proving procedure to handle structured formulae in the consequent. More specifically, the entailment procedure for structured formulae checks that, given formulae Φ_1 and Q_2, Φ_1 entails Q_2, that is if in all heaps satisfying Φ_1, we can find a subheap satisfying Q_2.

Following from the entailment checking procedure introduced in Sec 2.4, besides determining if the entailment relation holds, the current entailment procedure also infers the residual heap of the entailment, that is a formula Φ_R such that $\Phi_1 \vdash \mathsf{Q}_2 * \Phi_R$, and derives the predicate parameters. The relation is formalized using a judgment of the form $\Phi_1 \vdash^{\kappa}_{V} \mathsf{Q}_2 * \Phi_R$, which is a shorthand for $\Phi_1 * \kappa \vdash \exists V \cdot (\mathsf{Q}_2 * \kappa) * \Phi_R$. Note that κ denotes the consumed heap, while V is a set, $\{v^*, E{:}w^*\}$, containing the existential variables encountered, v^*, together with the variables w^* for late instantiation.

To support proof search, we have also generalised the entailment checking procedure to return a set of residues S_R: $\Phi_1 \vdash^{\kappa}_{V} \mathsf{Q}_2 * S_R$.This entailment succeeds when S_R is non-empty,

otherwise it is deemed to have failed. The multiple residual states captured in S_R signify different search outcomes during proving. The main rules are given in Figure 5.3. Take note that we make use of a method *mark*(V, w^*) , which marks the variables to be late instantiated, w^*, by removing them from the existential variables stored in V and adding them as $E : w^*$:

$$\mathit{mark}(V, w^*) = (V - \{w^*\}) \cup \{(E : w)^*\}$$

The rule [ENT−FORMULA] makes use of the aforementioned marking method in order to mark the fact that variables w^* are to be late instantiated, whereas rule [ENT−EXIST] adds the existentially quantified variables v^* to the set V.

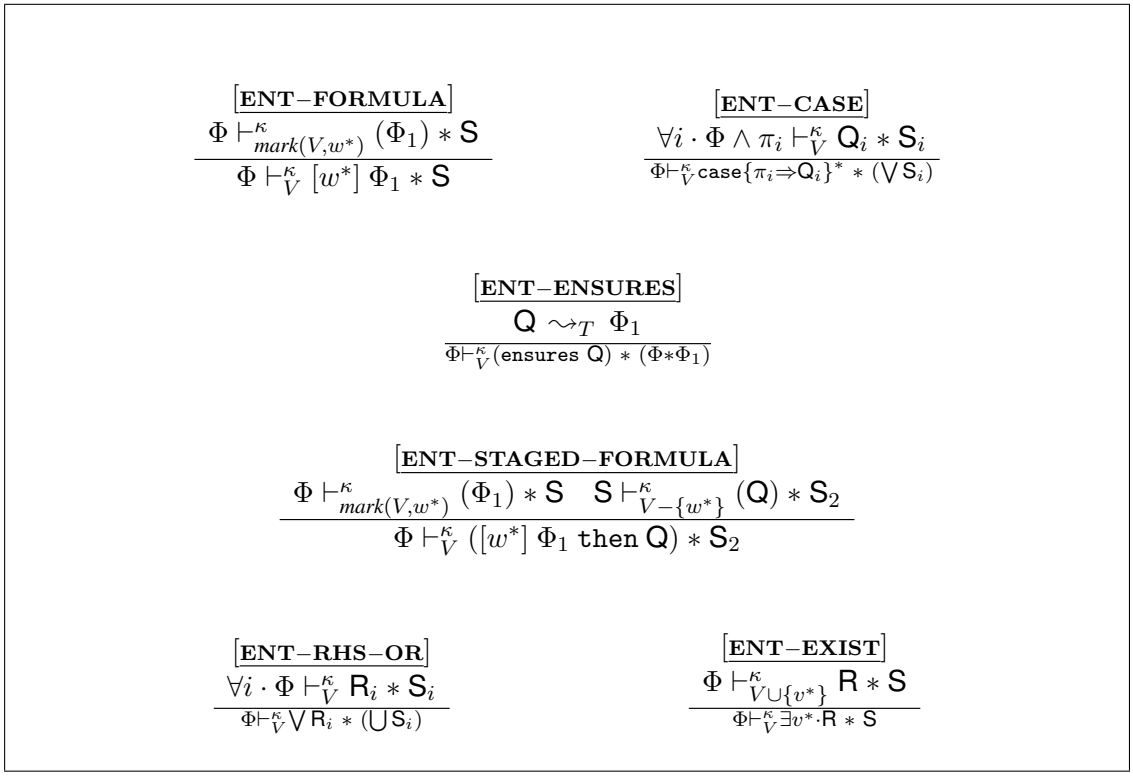

Figure 5.3: Entailment for Structured Formula

In the rule for staged formula, [ENT−STAGED−FORMULA], the instantiation for the variables w^* takes place in the first stage, Φ_1. As instantiation moves the corresponding bindings to the LHS (or antecedent of entailment), the variables w^* must be removed from the set of existentially quantified variables when entailing the rest of the formula, Q. At the end of the entailment proving, the variables that were marked as late-instantiated are existentially quantified in the residue state. The generalised entailment with a set of n formulae in the antecedent is an

abbreviation of the n entailments, as illustrated below:

$$\frac{\forall i \in \{1,..,n\} \cdot \Phi_i \vdash^{\kappa}_{V} (\mathsf{Q}) * \mathsf{S}_i}{\{\Phi_1,..,\Phi_n\} \vdash^{\kappa}_{V} (\mathsf{Q}) * \bigcup_{i=1}^{n} \mathsf{S}_i}$$

The rule [ENT−CASE] adds the pure term π_i to the antecedent. This rule requires a lifted disjunction operation defined as $S_1 \vee S_2 \equiv \{\Phi_1 \vee \Phi_2 | \Phi_1 \in S_1, \Phi_2 \in S_2\}$ when applied to two sets of states, S_1, S_2.

While a successful entailment of one disjunct suffices for the entailment of a disjunctive formula, our entailment rule [ENT−RHS−OR] facilitates a proof search by trying to entail each of the RHS disjuncts separately. Therefore, the residue state must contain the union of all residues corresponding to the proof search from a set of entailments, $\forall i \cdot \Phi \vdash^{\kappa}_{V} \mathsf{R}_i * \mathsf{S}_i$.

Take note that, at each call site, the forward verification procedure ensures that the method's precondition is satisfied and assumes the method's postcondition. This is achieved by entailing a formula denoting a specification of the Z form. As the corresponding entailment rules are similar to those for the entailment of a structured formula given in Figure 5.3, we omit them for brevity. The only unusual rule is ENT−ENSURES that is needed when entailing the actual postcondition `ensures Q`. In this case, the postcondition is added to the residual state in unstructured form, immediately after the translation $\mathsf{Q} \rightsquigarrow_T \Phi_1$ to unstructured form.

5.6.1 Instantiations

Our structured specification mechanism provides three types of instantiations for the logical variables (of consequent) during the entailment proving process:

- **no instantiation** : this technique is used for existential variables and need only apply its substitutions within the consequent itself.
- **early (or implicit) instantiation** : bindings of these variables, obtained from matching or folding predicates, are immediately moved to the antecedent.
- **late (or explicit) instantiation** : bindings of these variables are kept in the consequent, and moved to the antecedent at the end of scope of the logical variables.

If no instantiation is required, we only need to apply direct substitutions of the existential variables during matching or folding of the predicate. This is the simplest mechanism and should be used, where possible.

Parameter instantiation is needed primarily for connecting the logical variables between the precondition and the postcondition of a specification. In this section, we outline the specific mechanisms needed to support both early and late instantiations. We highlight two main rules. The first one occurs when matching a predicate in the antecedent with another predicate (with the same root parameter, x) in the consequent, as follows:

$$\frac{\begin{array}{c}[\textbf{ENT-MATCH}]\\ w^* = \mathit{late\_vars}(V) \quad v^* = V - w^*\\ (\texttt{early\_eq}, \texttt{late\_eq}, \rho) = \mathit{split}(v_1^*, v_2^*, v^*, w^*)\\ \kappa_1 \wedge \pi_1 \wedge \texttt{early\_eq} \vdash_V^{\kappa * \texttt{x::p}\langle \texttt{v}_1^* \rangle} \rho(\kappa_2 \wedge \pi_2 \wedge \texttt{late\_eq}) * \mathsf{S}\end{array}}{\texttt{x::p}\langle \texttt{v}_1^* \rangle * \kappa_1 \wedge \pi_1 \vdash_V^{\kappa} (\texttt{x::p}\langle \texttt{v}_2^* \rangle * \kappa_2 \wedge \pi_2) * \mathsf{S}}$$

We divide the parameter bindings via *split* into three categories:

(i) existential (denoted by v^*),

(ii) to be late instantiated (denoted by w^*),

(iii) to be early instantiated (all other variables of the consequent).

Early binding `early_eq` is immediately moved to the antecedent, while the late binding `late_eq` is kept in the consequent. In contrast, the existential binding ρ is made into a substitution that is directly applied to the consequent. Applying direct substitution into the consequent is simplest since the existential variables are immediately eliminated, where possible, and there is no need to separately identify instantiations to move to the antecedent. For late instantiation, we keep the bindings for the logical variables in the consequent until the end of entailment, in order to obtain a more precise binding, as illustrated by the rule below.

$$\frac{\begin{array}{c}[\textbf{ENT-EMP}]\\ (\mathit{XPure}(\kappa_1 * \kappa) \wedge \pi_1 \Longrightarrow \exists V \cdot \pi_2) \quad w^* = \mathit{late\_vars}(V)\\ v^* = V - w^* \quad \mathcal{I} = (\exists v^* \cdot \pi_2)\end{array}}{\kappa_1 \wedge \pi_1 \vdash_V^{\kappa} (\texttt{emp} \wedge \pi_2) * \{\kappa_1 \wedge (\pi_1 \wedge \mathcal{I})\}}$$

This occurs when the consequent is completely pure and has only an `emp` heap state. We construct an instantiation, named $\mathcal{I}$, for the variables to be late instantiated, w^*, before it is moved to the antecedent via the residue of entailment proving. Note that the function *XPure* will translate a heap-based formula into an approximate heap-independent formula.

5.7 Soundness

The storage model, semantic model and dynamic semantics are similar to those given in Sec 2.5, 2.6, and 2.7, respectively, with extensions for the new structured formulae. Let $s, h \models \mathsf{Q}$ in Fig 5.4 denote the model relation, i.e. the stack s and heap h satisfy the constraint Q, with h, s from the following concrete domains:

$$h \in \mathit{Heaps} =_{df} \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal}$$
$$s \in \mathit{Stacks} =_{df} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}$$

The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

$s, h \models \mathsf{Q}$ iff $\mathsf{Q} = \bigvee_{i=1}^{n} \exists v^* \cdot \mathsf{R}_i$ and $s, h \models \bigvee_{i=1}^{n} \exists v^* \cdot \mathsf{R}_i$

$s, h \models \bigvee_{i=1}^{n} \exists v_{i1..im} \cdot \mathsf{R}_i$ iff $\exists k \in \{1, .., n\} \cdot \exists \alpha_{k1..km} \cdot s[v_{k1} \mapsto \alpha_{k1}, .., v_{km} \mapsto \alpha_{km}], h \models \mathsf{R}_k$

$s, h \models [w_{i=1}^{n}]\Phi$ `then` Q iff $\exists h_1, h_2 \cdot h_1 \perp h_2$ and $h = h_1 \cdot h_2$ and $\exists \alpha_{1..n} \cdot s[w_1 \mapsto \alpha_1, .., w_n \mapsto \alpha_n], h_1 \models \Phi$ *and* $s, h_2 \models \mathsf{Q}$

$s, h \models \mathtt{case}\{(\pi_i \Rightarrow \mathsf{Q}_i)_{i=1}^{n}\}$ iff $\forall k \in \{1, .., n\} \cdot (s, h \models \pi_k \rightarrow s, h \models \mathsf{Q}_k)$

$s, h \models \Phi_1 \vee \Phi_2$ iff $s, h \models \Phi_1$ or $s, h \models \Phi_2$

$s, h \models \exists v_{1..n} \cdot \kappa \wedge \pi$ iff $\exists \alpha_{1..n} \cdot s[v_1 \mapsto \alpha_1, .., v_n \mapsto \alpha_n], h \models \kappa$ and $s[v_1 \mapsto \alpha_1, .., v_n \mapsto \alpha_n] \models \pi$

$s, h \models \kappa_1 * \kappa_2$ iff $\exists h_1, h_2 \cdot h_1 \perp h_2$ and $h = h_1 \cdot h_2$ and $s, h_1 \models \kappa_1$ and $s, h_2 \models \kappa_2$

$s, h \models \mathtt{emp}$ iff $\mathit{dom}(h) = \emptyset$

$s, h \models p::\mathtt{c}\langle \mathtt{v_{1..n}} \rangle$ iff *exists a data type decl.* `data c` $\{t_1\ f_1, .., t_n\ f_n\}$ and $h = [s(p) \mapsto r]$ and $r = \mathtt{c}[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)]$

`or` *exists a pred. def.* $(c\langle v_{1..n} \rangle \equiv \mathsf{Q}$ `inv` $\pi) \in P$ and $s, h \models [p/\mathtt{root}]\mathsf{Q}$

Figure 5.4: Model for Structured Formulae

For the case of a data node, $v::\mathtt{c}\langle v^* \rangle$, h has to be a singleton heap. On the other hand, a shape predicate defined by $c\langle v_{1..n} \rangle \equiv \mathsf{Q}$ may be inductively defined.

With the semantics of the structured formulae in place, we can provide a translation from

$$\frac{\forall i \cdot \mathsf{Q}_i \rightsquigarrow_T \Phi_i}{\texttt{case}\{\pi_i \Rightarrow \mathsf{Q}_i\}^* \rightsquigarrow_T \bigvee(\Phi_i \wedge \pi_i)} \qquad \frac{\mathsf{Q} \rightsquigarrow_T \Phi}{[w^*]\ \Phi_1\ \texttt{then}\ \mathsf{Q} \rightsquigarrow_T \Phi_1 * \Phi}$$

$$\frac{\forall i \cdot \mathsf{R}_i \rightsquigarrow_T \Phi_i}{\bigvee \exists v^* \cdot \mathsf{R}_i \rightsquigarrow_T \bigvee \exists v^* \cdot \Phi_i} \qquad \frac{}{[w^*]\ \Phi \rightsquigarrow_T \Phi}$$

Figure 5.5: Translation from a structured formula to its equivalent unstructured formula

a structured formula to its equivalent unstructured formula. This translation is formalised with $\mathsf{Q} \rightsquigarrow_T \Phi$, as shown in Fig 5.5.

We make use of the semantics for structured formulae Q and for unstructured formula Φ to prove the correctness of the given translation rules.

Theorem 5.7.1 (Correctness of Translation)**.** *Given Q and Φ such that $\mathsf{Q} \rightsquigarrow_T \Phi$: for all s, h, $s, h \models \mathsf{Q}$ if and only if $s, h \models \Phi$.*

Proof: By structural induction on Q.

Theorem 5.7.2 (Safety)**.** *Consider a closed term e without free variables in which all methods have been successfully verified. Assuming unlimited stack/heap spaces and that $\vdash \{\texttt{true}\}\ e\ \{\Delta\}$, then either $\langle [], [], e\rangle \hookrightarrow^* \langle [], h, v\rangle$ terminates with a value v that is subsumed by the postcondition Δ, or it diverges $\langle [], [], e\rangle \not\hookrightarrow^*$.*

Proof: It follows from the safety of our underlying verification system (i.e. the one without structured specifications) [92], the definition 5.5.1, and the soundness of the entailment prover enriched with structured formulae formulated by Theorem 5.7.3.

Theorem 5.7.3 (Soundness of Entailment)**.** *Given Φ, Q such that $s, h \models \Phi$, if $\Phi \vdash^{\kappa}_{V} \mathsf{Q} * \Phi_r$ for some Φ_r, then $s, h \models \mathsf{Q} * \Phi_r$. That is, for all program states in which Φ holds if $\Phi \vdash^{\kappa}_{V} \mathsf{Q} * \Phi_r$ then $\mathsf{Q} * \Phi_r$ holds.*

Proof: By structural induction on Q.

5.8 Experimental Evaluation

We have built a prototype system using Objective Caml. The proof obligations generated by our verification are discharged using some off-the-shelf constraint solvers (like Omega Calculator

[105]) or theorem provers (like MONA [64]). The specification mechanism works with any constraint domain, as long as a corresponding prover for the domain is available. The specific domains that our verifier currently supports, includes linear (Omega Calculator, Z3, CVC-lite) and non-linear arithmetic (Redlog), set (MONA, Isabelle bag tactic) and list properties (a Coq tactic).

We have conducted preliminary experiments by testing our system on a suite of examples summarized in Figure 5.6. These examples are small but can handle data structures with sophisticated shape and size properties such as sorted lists, balanced trees, etc., in a uniform way. Methods "insert" and "delete" refer to the insertion and deletion of a value into/from the corresponding data structure, respectively. Method "delete_first" deletes the node at the head in a circular list. Moreover, we verify a suite of sorting algorithms, which receive as input an unsorted singly-linked list and return a sorted list. Verification time for each function includes the time to verify all functions that it calls. We compare the timings obtained with and without case analysis.

Take note that for each of the verified methods, in order to compare the results obtained with and without case analysis, we provided specifications with the same level of modularity through specifications with multiple pre/post. FAIL for the "without case" means it did not verify functional correctness (including memory safety). This is due the absence of case analysis that would have been provided by the missing case spec.

Preliminary results indicate that case analysis improves both the completeness and the performance of our system. From the completeness point of view, case analysis is important for verifying a number of examples that would *fail* otherwise. For instance, the method implementing the selection sort algorithm over a linked list fails when it is written with multiple specification instead of the case construct. The same scenario is encountered for the method inserting/deleting a node of red black tree, and for the method appending two list segments. The case construct thus helps our system to verify more examples successfully. Regarding the performance, the timings obtained when using case analysis are smaller, taking on average 21% less computation time than those obtained without case analysis. The improvements are due to earlier pruning of `false` contexts with the help of case constructs and optimizations of the case entailment rule.

Program Codes	LOC	Timings (in seconds)		speed gain (%)
		with case	*without* case	
Linked List		verifies length		
delete	20	0.65	0.89	26
append	14	0.30	0.39	23
List Segment		verifies length		
append	11	0.95	*failed*	-
Circular Linked List		verifies length + circularity		
delete_first	15	0.35	0.41	15
insert	10	0.28	0.35	20
Doubly Linked List		verifies length + double links		
insert	18	0.35	0.52	33
delete	29	0.94	1.27	26
Sorted List		verifies bounds + sortedness		
insert	17	0.71	0.96	26
delete	21	0.60	0.68	22
insertion_sort	45	0.92	1.35	32
selection_sort	52	1.24	*failed*	-
bubble_sort	42	1.95	2.92	43
merge_sort	105	2.01	2.53	31
quick_sort	85	1.82	2.47	26
AVL Tree		verifies size + height + balanced		
insert	169	32.27	39.48	19
delete	287	85.1	97.30	13
Perfect Tree		verifies height + perfectness		
insert	89	0.73	0.99	26
Red-Black Tree		verifies size + black-height		
insert	167	5.44	*failed*	-
delete	430	22.43	*failed*	-

Figure 5.6: Verification Times for Case Construct vs Multiple Pre/Post

We also investigated the performance gain that can be attributed to the use of staged formulae. We observed that the timings improved on average by 20%. However, in some cases, such as the AVL-merge example, we obtain a 38% speedup from 5.1 seconds to 3.2 seconds. Other noteworthy examples include the AVL insertion (from 32.27s to 22.93s) and AVL deletion (from 85.1s to 81.6s).

We may conclude from our experiments that structured specifications together with case analysis give better precision to our verification system while also improving its performance, when compared to corresponding unstructured specifications.

CHAPTER VI

STATIC AND DYNAMIC SPECIFICATIONS

6.1 Motivation

Object-based programs are hard to statically analyse mostly because of the need to track object mutations in the presence of aliases. Object-oriented (OO) programs are even harder, as we have to additionally deal with class inheritance and method overriding.

One major issue to consider when verifying OO programs is how to design specification for a method that may be overridden by another method down the class hierarchy, such that it conforms to method subtyping. In addition, it is important to ensure that subtyping is observed for object types in the class hierarchy, including any class invariant that may be imposed. From the point of conformance to OO semantics, most analysis techniques uphold Liskov's Substitutivity Principle [80] on behavioral subtyping. Under this principle, an object of a subclass can always be passed to a location where an object of its superclass is expected, as the object from each subclass must subsume the entire set of behaviors from its superclass. To enforce behavioral subtyping for OO programs, several past works [32, 5, 62] have advocated for class invariants to be inherited by each subclass, and for pre/post specifications of the overriding methods of its subclasses to satisfy a *specification subsumption* (or subtyping) relation with each overridden method of its superclass.

In this chapter, for brevity reasons, in stead of the notation $\{\texttt{requires}\Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}^p_{i=1}$ denoting a method's specification, we will we will use an infix notation $\{\texttt{pre}^\texttt{i} \twoheadrightarrow \texttt{post}^\texttt{i}\}^\texttt{p}_{\texttt{i=1}}$. A basic *specification subsumption* mechanism was originally formulated as follows. Consider a method B.mn in class B with $(\texttt{pre}_\texttt{B} \twoheadrightarrow \texttt{post}_\texttt{B})$ as its pre/post specification, and its overriding method C.mn in subclass C, with a given pre/post specification $(\texttt{pre}_\texttt{C} \twoheadrightarrow \texttt{post}_\texttt{C})$. The specification $(\texttt{pre}_\texttt{C} \twoheadrightarrow \texttt{post}_\texttt{C})$ is said to be a *subtype* of $(\texttt{pre}_\texttt{B} \twoheadrightarrow \texttt{post}_\texttt{B})$ in support of method overriding, if the following subsumption relation holds:

$$\frac{\texttt{pre}_\texttt{B} \wedge \mathit{type}(\texttt{this}) <: \texttt{C} \implies \texttt{pre}_\texttt{C} \qquad \texttt{post}_\texttt{C} \implies \texttt{post}_\texttt{B}}{(\texttt{pre}_\texttt{C} \twoheadrightarrow \texttt{post}_\texttt{C}) <:_\texttt{C} (\texttt{pre}_\texttt{B} \twoheadrightarrow \texttt{post}_\texttt{B})}$$

The two conditions are to ensure contravariance of preconditions, and covariance on post-conditions. They follow directly from the subtyping principle on methods' specifications. As the two specifications are from different classes, we add the subtype constraint $type(\texttt{this})<:\texttt{C}$ to allow the above subsumption relation to be checked for the *same* C subclass. To reflect this, we parameterize the subsumption operator $<:_{\texttt{C}}$ with a C-class as its suffix.

The main purpose of using specification subsumption is to support modular reasoning by avoiding the need to re-verify the code of overriding method C.mn with the specification $(\texttt{pre}_{\texttt{B}} \ast\!\!\rightarrow \texttt{post}_{\texttt{B}})$ of its overridden method B.mn. In the case that specification subsumption does not hold, an alternative way to achieve behavioral subtyping is to use the *specification inheritance* technique of [32] to strengthen the specification of each overriding method with the specification of its overridden method, as follows:

Consider a method B.mn in class B with $(\texttt{pre}_{\texttt{B}} \ast\!\!\rightarrow \texttt{post}_{\texttt{B}})$ as its pre/post specification, and its overriding method C.mn in subclass C, with pre/post specification $(\texttt{pre}_{\texttt{C}} \ast\!\!\rightarrow \texttt{post}_{\texttt{C}})$. To ensure specification subsumption, we can strengthen the specification of the overriding method via *specification inheritance* with the intersection of their specifications, namely:

$$(\texttt{pre}_{\texttt{C}} \ast\!\!\rightarrow \texttt{post}_{\texttt{C}}) \bigwedge (\texttt{pre}_{\texttt{B}} \wedge type(\texttt{this})<:\texttt{C} \ast\!\!\rightarrow \texttt{post}_{\texttt{B}}).$$

Specification inheritance requires the use of multiple specifications (or intersection type) to provide for a more expressive mechanism to describe each method. By inheriting a new specification for the overriding method, this technique uses code re-verification itself to ensure that a behavioral subtyping property would be enforced. We can generalise a definition of the subsumption relation between two multiple specifications, as follows:

Definition 6.1.1 (Multi-Specifications Subsumption). *Given two multiple specifications,* $\bigwedge_{\texttt{j}=1}^{\texttt{n}} \texttt{specB}_{\texttt{j}}$ *(for class* B*) and* $\bigwedge_{\texttt{i}=1}^{\texttt{m}} \texttt{specC}_{\texttt{i}}$ *(for subclass* C*), where each of* $\texttt{specB}_{\texttt{j}}$ *and* $\texttt{specC}_{\texttt{i}}$ *is a pre/post annotation of the form* $\texttt{pre} \ast\!\!\rightarrow \texttt{post}$*. We say that they are in specification subsumption relation,* $(\bigwedge_{\texttt{i}=1}^{\texttt{m}} \texttt{specC}_{\texttt{i}}) <:_{\texttt{C}} (\bigwedge_{\texttt{j}=1}^{\texttt{n}} \texttt{specB}_{\texttt{j}})$*, if the following holds :* $\forall \texttt{j} \in 1..\texttt{n}\ \exists \texttt{i} \in 1..\texttt{m} \cdot \texttt{specC}_{\texttt{i}} <:_{\texttt{C}} \texttt{specB}_{\texttt{j}}$.

While modular reasoning can be supported by the above subsumption relations, the reasoning were originally formulated in the framework of Hoare logic. Recently, separation logic has been proposed as an extension to Hoare logic, providing precise and concise reasoning for pointer-based programs. A key principle followed in separation logic is the use of local reasoning to facilitate modular analysis/reasoning. An early work on applying separation logic to the

OO paradigm was introduced by [100]. In that work, two key concepts were identified. Firstly, an abstract predicate family $\mathtt{p_t}(\mathtt{v_1}, .., \mathtt{v_n})$ was used to capture some program states for objects of class hierarchy with type `t`. Each abstract predicate has a visibility scope and is allowed to have a different number of parameters, depending on the actual type of its root object. Moreover, each predicate family acts as an *extensible* predicate for which incremental specification is given and verified for each class. Secondly, the concept of *specification compatibility* was introduced to capture the subsumption relation soundly, as follows:

A specification $\mathtt{pre_C} \twoheadrightarrow \mathtt{post_C}$ is said to be *compatible* with $\mathtt{pre_B} \twoheadrightarrow \mathtt{post_B}$ under all program contexts, if the following holds:

$$\frac{\forall \mathtt{code} \cdot \{\mathtt{pre_C}\}\mathtt{code}\{\mathtt{post_C}\} \implies \{\mathtt{pre_B}\}\mathtt{code}\{\mathtt{post_B}\}}{(\mathtt{pre_C} \twoheadrightarrow \mathtt{post_C}) <: (\mathtt{pre_B} \twoheadrightarrow \mathtt{post_B})}$$

Specification compatibility can be viewed as a more fundamental way to describe specification subsumption in terms of Hoare logic triples. However, it cannot be *directly* implemented, since its naive definition depends on exploring all possible program codes for compatibility. In this thesis, we provide a *practical* alternative towards automated verification of OO programs that can support better precision and avoid unnecessary code re-verification. We use key principles of separation logic to achieve this.

6.2 Chapter Overview

After introducing the specification language in Sec 6.3, we motivate our work through an example in Sec 6.4. Our proposal is summarized by a few principles for OO verification in Sec 6.5, and more details on the approach towards supporting method inheritance via an enhanced specification subsumption relation are provided in Sec 6.6. Sec 6.7 presents mechanisms for ensuring that method overriding and method inheritance conform to the OO paradigm, whereas Sec 6.8 introduces techniques for deriving dynamic specifications from the static counterparts. Sec 6.9 describes the verification system, while its soundness is discussed in Sec 6.10.

6.3 Specification and Programming Language

We consider a core OO language consisting of a simple sequential language with just the basic features from the OO paradigm. Some omitted features, such as exceptions, static fields and static methods, can be handled in an orthogonal manner and do not cause any difficulty to our

$$
\begin{array}{lll}
\mathit{tdecl} & ::= & \mathit{classt} \mid \mathit{spred} \\
\mathit{classt} & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \texttt{inv}\ \kappa \wedge \pi\ \{\ (t\ v)^*\mathit{meth}^*\ \} \\
\mathit{meth} & ::= & t\ \mathit{mn}\ ((t\ v)^*)\ [\texttt{static}\ \mathit{sp}_1]\ [\texttt{dynamic}\ \mathit{sp}_2]\ \{e\} \\
\mathit{sp} & ::= & \{\Phi^i_{pr} \ast\!\!\rightarrow \Phi^i_{po}\}^p_{i=1} \\
\pi & ::= & \gamma \wedge \phi \wedge \beta \\
\beta & ::= & v{=}c \mid v{<:}c
\end{array}
$$

Figure 6.1: A Core Object-Oriented Language

verification system.

We provide our language in Figure 6.1, and assume that the constructs not covered in Fig 6.1 remain unchanged from the programming and specification languages given in Fig 2.1 and Fig 2.2, respectively. This core language is the target of some preprocessing steps. A program consists of a list of class and view declarations and an expression which corresponds to the `main` method in many languages. We assume that the super class of each class is explicitly declared, except for `Object` at the top of the class hierarchy. We also use `this` as a special variable referring to the receiver object, and `super` to refer to a superclass's method invocation. For simplicity, we assume that variable names declared in each method are all distinct and use the pass-by-value parameter mechanism.

6.4 Examples

The focus on specifications that support method overriding has a potential drawback that these specifications are typically imprecise (or weaker) for methods of superclasses. Such specifications typically have stronger preconditions (which restrict their applicability) and/or weaker postconditions (which lose precision). This drawback can cause imprecision for OO verification which has in turn spurred practical lessons on tips and tricks for specification writers [62]. Furthermore, mechanisms such as specification inheritance may have unnecessary code re-verification, especially when specification subsumption holds.

Let us consider the specification of a simple up-counter class in Figure 1. This `Cnt` class is

accompanied by three possible subclasses:

- `FastCnt` to support a faster `tick` operation,
- `PosCnt` which works only with positive numbers,
- `TwoCnt` which supports an extra backup counter.

Let us first design the specifications for instance methods of class `Cnt` *without* worrying about method overriding. A possible set of pre/post specifications is given below where `this` and `res` are variables denoting the receiver and result of each method.

$$\texttt{void Cnt.tick()} \quad \textbf{static}\ \texttt{this::Cnt}\langle \texttt{n} \rangle \ast\!\!\rightarrow \texttt{this::Cnt}\langle \texttt{n+1} \rangle$$

$$\texttt{void Cnt.set(int x)} \quad \textbf{static}\ \texttt{this::Cnt}\langle \texttt{n} \rangle \ast\!\!\rightarrow \texttt{this::Cnt}\langle \texttt{x} \rangle$$

$$\texttt{int Cnt.get()} \quad \textbf{static}\ \texttt{this::Cnt}\langle \texttt{n} \rangle \ast\!\!\rightarrow \texttt{this::Cnt}\langle \texttt{n} \rangle \wedge \texttt{res=n}$$

We refer to these as *static specifications* and precede them with the **static** keyword. They can be very precise as they were considered statically on a per method basis without concern for method overriding, and can be used whenever the actual type of the receiver is known. The notation $\texttt{y::c}\langle \texttt{v}_1, .., \texttt{v}_\texttt{n} \rangle$ denotes that variable `y` is pointing to an object with the *actual type*[1] of `c`-class and where each field $\texttt{y.f}_\texttt{i}$ is denoted by variable $\texttt{v}_\texttt{i}$. For example, in $\texttt{this::Cnt}\langle \texttt{n} \rangle$, the field `this.val` is denoted by variable `n`. This format for objects is used primarily for static specification. To describe an object type whose type is merely a *subtype* of the `c`-class, we shall use a different notation, namely $\texttt{y::c}\langle \texttt{v}^* \rangle\$$, which implicitly captures an object extension with extra fields from its subclass.

If we take into account the possible overriding of the `tick` method by its corresponding method in the `FastCnt` subclass, we may have to weaken the postcondition of `Cnt.tick`. Furthermore, to guarantee the invariant $\texttt{this.val} \geq 0$ of the `PosCnt` class, we may have to strengthen the preconditions of methods `Cnt.set`, `Cnt.tick` and `Cnt.get`. These weakenings result in the following *dynamic specifications* which are the usual ones being considered for dynamically dispatched methods, where the type of the receiver is a subtype of its current class.

[1] To make our static specifications more reusable through inherited methods, we shall avoid the use of an explicit constraint, like $\texttt{type(this)=c}$, on the actual type of the receiver.

```
class Cnt {  int val;

 Cnt(int v) {this.val:=v}

 void tick() {this.val:=this.val+1}

 int get() {this.val}

 void set(int x) {this.val:=x}
}


class FastCnt extends Cnt {

 FastCnt(int v) {this.val:=v}

 void tick() {this.val:=this.val+2}
}


class PosCnt extends Cnt inv this.val≥0 {

 PosCnt(int v) {this.val:=v}

 void set(int x) {if x≥0 then this.val:=x else error()}
}


class TwoCnt extends Cnt {  int bak;

 TwoCnt(int v, int b) {this.val:=v; this.bak:=b}

 void set(int x) {this.bak:=this.val; this.val:=x}

 void switch(int x)

 {int i:=this.val; this.val:=this.bak; this.bak:=i}
}
```

Figure 6.2: Example: `Cnt` and its subclasses

`void Cnt.tick()` **dynamic** `this::Cnt⟨n⟩$∧n≥0` $*\!\!\rightarrow$ `this::Cnt⟨b⟩$∧n+1≤b≤n+2`

`void Cnt.set(int x)` **dynamic** `this::Cnt⟨n⟩$∧x≥0` $*\!\!\rightarrow$ `this::Cnt⟨x⟩$`

`int Cnt.get()` **dynamic** `this::Cnt⟨n⟩$∧n≥0` $*\!\!\rightarrow$ `this::Cnt⟨n⟩$∧res=n`

Such changes make the specifications of the methods in superclasses less precise, and are carried out to ensure behavioral subtyping. Furthermore, these specifications must also cater to potential modifications that may occur in the extra fields of the subclasses by their overriding methods either directly or indirectly. Due to conflicting requirements, we advocate the co-existence of both static and dynamic specifications. The former is important for precision and shall be used primarily for code verification, while the latter is needed to support method overriding and must be used for dynamically dispatched methods. Formally:

Definition 6.4.1 (**Static Pre/Post).** *A specification is said to be* static *if it is meant to describe a single method declaration, and need not be used for subsequent overriding methods.*

Definition 6.4.2 (**Dynamic Pre/Post).** *A specification is said to be* dynamic *if it is meant for use by a method declaration and its subsequent overriding methods.*

Past works, such as [2, 32, 81, 99, 5, 90], are based primarily on dynamic specifications, though implicit static specifications via `type(this)=c` can also be used in ESC/Java and Spec#, while JML uses *code contract* [74, ch 15] as a form of static specification. However, these proposals for static specifications are somewhat ad-hoc, as they do not impose any relation between static and dynamic specifications. In our approach, we emphasize static specifications over dynamic specifications. Most importantly, we always ensure that the static specification of a method from a given class is always a *subtype* of the dynamic specification of the same method within the same class. This principle is important for modular verification, as we need only verify the code of each method *once* against its static specification. It is unnecessary to verify the corresponding dynamic specification since the latter is a specification supertype. Our proposal uses the following principles for OO verification, achieving both precision and reuse.

6.5 Principles for Enhanced OO Verification

- Static specification is given for each new method declaration, and may be added for inherited methods to support new auxiliary calls and subclasses with new invariants.
- Dynamic specification is either given or derived. Whether given or derived, each dynamic specification must satisfy two subsumption properties. Each must be a :
  - specification supertype of its static counterpart. This helps keep code re-verification to a minimum.

  - specification supertype of the dynamic specification of each overriding method in its subclasses. This helps ensure behavioral subtyping.

- Code verification is only performed for static specifications.

6.6 Our Approach

Our approach to enhancing OO verification is based on separation logic. We shall describe how we adapt separation logic for reasoning about objects from a class hierarchy and how to write precise specifications that avoid unnecessary code re-verification.

6.6.1 Object View and Lossless Casting

For separation logic to work with OO programs, one key problem that we must address is a suitable format to capture the objects of classes. We should preferably also address the problem of performing upcast/downcast operations statically in accordance with the OO class hierarchy, and without loss of information where possible.

Consider two variables, `x` and `y`, which point to objects from `Cnt` class (with a single field) and `TwoCnt` class (with two fields), respectively. Intuitively, we may represent the first object by $\texttt{x::Cnt}\langle\texttt{v}\rangle$ where `v` denotes its field, and the second object by $\texttt{y::TwoCnt}\langle\texttt{v},\texttt{b}\rangle$ where $\texttt{v},\texttt{b}$ denote its two fields. However, a fundamental problem that we must solve is how to cast the object of one class to that of its superclass, and vice-versa when needed. To do this without loss of information, we provide two extra information : (i) a variable to capture the *actual type* of a given object and (ii) a variable to capture the object's *record extension* that contains extra field(s) of its subclass. When a `TwoCnt` object is first created, we may capture its state using the formula :

$$\texttt{y::TwoCnt}\langle\texttt{t},\texttt{v},\texttt{b},\texttt{p}\rangle\wedge\texttt{t=TwoCnt}\wedge\texttt{p=null}$$

The above formula indicates that the actual type of the object is `t=TwoCnt` and that there is no need for any record extension since `p=null`. With this object format, we can now perform an upcast to its parent `Cnt` class by transforming it to:

$$\texttt{y::Cnt}\langle\texttt{t},\texttt{v},\texttt{q}\rangle*\texttt{q::Ext}\langle\texttt{TwoCnt},\texttt{b},\texttt{p}\rangle\wedge\texttt{t=TwoCnt}\wedge\texttt{p=null}$$

Though this cast operation is viewing the object as a member of Cnt class, it is still a TwoCnt object as the type information t=TwoCnt indicates. Furthermore, we have created an extension record q::Ext⟨TwoCnt, b, p⟩ that can capture the extra b field of the TwoCnt subclass. For simplicity, we currently use an implicit pointer q to capture the extension record. This model allows a sequence of upcast operations to be easily captured. Such an upcast operation is *lossless* as we have sufficient information to perform the inverse downcast operation back to the original TwoCnt format. To allow lossless casting between Cnt and TwoCnt, we add an equivalence rule :

$$\texttt{TwoCnt}\langle \texttt{t}, \texttt{v}, \texttt{b}, \texttt{p}\rangle \equiv \texttt{root::Cnt}\langle \texttt{t}, \texttt{v}, \texttt{q}\rangle * \texttt{q::Ext}\langle \texttt{TwoCnt}, \texttt{b}, \texttt{p}\rangle$$

An unfold step (which replaces a term that matches the LHS by RHS) corresponds to an upcast operation, while the fold step (which replaces a term that matches the RHS by LHS) corresponds to a downcast operation. Such a rule can be derived from each superclass-subclass pairing. Formally:

Definition 6.6.1 (Lossless Casting). *Given a class* $\texttt{c}\langle \texttt{v}^*\rangle$ *with fields* $\texttt{v}^*$ *and its immediate subclass* $\texttt{d}\langle \texttt{v}^*, \texttt{w}^*\rangle$ *where* $\texttt{w}^*$ *denotes its extra fields, we shall generate the following casting rule that is coercible in either direction:*

$$\texttt{root::d}\langle \texttt{t}, \texttt{v}^*, \texttt{w}^*, \texttt{p}\rangle \equiv \texttt{root::c}\langle \texttt{t}, \texttt{v}^*, \texttt{q}\rangle * \texttt{q::Ext}\langle \texttt{d}, \texttt{w}^*, \texttt{p}\rangle$$

Note that for any object view $\texttt{d}\langle \texttt{t}, \ldots\rangle$ *it is always the case that the subtype relation* $\texttt{t}<:\texttt{d}$ *holds as its invariant. Furthermore, the default* root *parameter on the LHS may be omitted for brevity.*

Lossless casting is important for establishing the subsumption relation between each static specification and its dynamic counterpart, as the extension record can be preserved, if needed, by each static specification. Lossless casting is also important for the static specification of inherited methods which should preferably be inherited without the need for re-verification. This can be achieved by exploiting local reasoning which allows us to assert that an extension record need not be modified by each inherited method.

There are also occasions when we are required to pass the full object with all its (extended) fields. This occurs for dynamic specifications where subsequent overriding method may change the extra fields of its subclass. To cater to this scenario, we introduce an $\texttt{ExtAll}\langle \texttt{t}_1, \texttt{t}_2\rangle$ view

that can capture all the extension records from a class $\mathtt{t}_1$ for an object with actual type $\mathtt{t}_2$. This scenario occurs for the dynamic specification of `Cnt.set` method, as shown below:

$$\mathtt{this}{::}\mathtt{Cnt}\langle \mathtt{t}, \_, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{ExtAll}\langle \mathtt{Cnt}, \mathtt{t}\rangle \wedge \mathtt{x} \geq 0$$
$$\ast\!\!\rightarrow \mathtt{this}{::}\mathtt{Cnt}\langle \mathtt{t}, \mathtt{x}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{ExtAll}\langle \mathtt{Cnt}, \mathtt{t}\rangle$$

Such a dynamic specification may be used with any subtype of `Cnt`. The entire object view must be passed to support dynamic specifications which are expected to cater to the current method and all subsequent overriding methods. The `ExtAll` predicate itself can be defined as follows:

$$\mathtt{ExtAll}\langle \mathtt{t}_1, \mathtt{t}_2\rangle \equiv \mathtt{t}_1{=}\mathtt{t}_2 \wedge \mathtt{root}{=}\mathtt{null} \vee \mathtt{root}{::}\mathtt{Ext}\langle \mathtt{t}_3, \mathtt{v}^*, \mathtt{q}\rangle$$
$$* \mathtt{q}{::}\mathtt{ExtAll}\langle \mathtt{t}_3, \mathtt{t}_2\rangle \wedge \mathtt{t}_3{<}\mathtt{t}_1 \wedge \mathtt{t}_2{<:}\mathtt{t}_3 \ \mathbf{inv}\ \mathtt{t}_2{<:}\mathtt{t}_1$$

The notation $\mathtt{t}_3{<}\mathtt{t}_1$ denotes a class $\mathtt{t}_3$ and its immediate superclass $\mathtt{t}_1$. The `ExtAll` predicate is used to generate all the extension records from class $\mathtt{t}_1$ to $\mathtt{t}_2$. For example, expression $\mathtt{x}{::}\mathtt{ExtAll}\langle \mathtt{Cnt}, \mathtt{Cnt}\rangle$ yields $\mathtt{x}{=}\mathtt{null}$, and $\mathtt{x}{::}\mathtt{ExtAll}\langle \mathtt{Cnt}, \mathtt{TwoCnt}\rangle$ yields $\mathtt{x}{::}\mathtt{Ext}\langle \mathtt{TwoCnt}, \mathtt{b}, \mathtt{null}\rangle$.

Our format allows two kinds of object views to be supported:

Definition 6.2.2 (Full and Partial Views). *We refer to the use of formula* $\mathtt{x}{::}\mathtt{c}\langle \mathtt{t}, \mathtt{v}^*, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{ExtAll}\langle \mathtt{c}, \mathtt{t}\rangle$ *as providing a* full view *for an object with actual type* `t` *that is being treated as a* `c`*-class object, while* $\mathtt{x}{::}\mathtt{c}\langle \mathtt{t}, \mathtt{v}^*, \mathtt{p}\rangle$ *provides only a* partial view *with no extension record. For brevity, full views are also written as* $\mathtt{x}{::}\mathtt{c}\langle \mathtt{v}^*\rangle\$$*, while partial views are coded using* $\mathtt{x}{::}\mathtt{c}\langle \mathtt{v}^*\rangle$.

This distinction between *partial* and *full* views (for objects) follows directly from our decision to distinguish static from dynamic specifications. Partial views are typically used for the receiver object of static specifications, while full views are used by dynamic specifications. Some readers may contend that lossless casting of an object `x` from $\mathtt{d}\langle \mathtt{v}^*, \mathtt{w}^*\rangle$ to $\mathtt{c}\langle \mathtt{v}^*\rangle$ may also be captured with the help of separating implication by representing the extension record using $\mathtt{x}{::}\mathtt{c}\langle \mathtt{v}^*\rangle \mathrel{-\!\!*} \mathtt{x}{::}\mathtt{d}\langle \mathtt{v}^*, \mathtt{w}^*\rangle$. This approach works well for partial views, but cannot easily handle the `ExtAll` predicate required by full views. Moreover, by omitting separating implication, our current approach to automated verification is easier to build and prove.

6.6.2 Ensuring Class Invariants

Ensuring that class invariants hold can be rather intricate, with the key problems being *how* and *when* to check for the invariants. Based on the simplest assumption, one would expect object

invariants to hold at all times. However, this assumption is impractical for mutable objects. One sensible solution is to expect invariants to hold based on visible state semantics, which is typically aligned to the boundaries of public methods. Even this approach may not be flexible enough. Thus, in Boogie [4, 78], programmers are also allowed to use a specification field, called `valid`, that can indicate if the invariant for an object is being preserved or temporarily broken for mutation. Similarly, in [88], programmers are allowed to indicate invariants that are inconsistent (not preserved) at some method boundary.

We aim for a similar level of flexibility but which still remains easy to use. To achieve this, we introduce the concept of an *invariant-enhanced view* for each class with a non-trivial invariant, as follows:

Definition 6.6.3 (Invariant-Enhanced View). *Consider a class* $\mathtt{c}$ *with a non-trivial invariant* $\delta_{\mathtt{c}}$ ($\neq\mathtt{true}$) *over the fields* $\mathtt{v}^*$ *of the object. We shall define a new view of the form* $\mathtt{c\#I}\langle\mathtt{v}^*\rangle$ *to capture its class invariant, as :* $\mathtt{c\#I}\langle\mathtt{v}^*\rangle \equiv \mathtt{root}::\mathtt{c}\langle\mathtt{v}^*\rangle * \delta_{\mathtt{c}}$. *Furthermore, for each subclass* $\mathtt{d}\langle\mathtt{v}^*, \mathtt{w}^*\rangle$ *with an extra invariant* $\delta_{\mathtt{d}}$ *over the fields* $\mathtt{v}^*, \mathtt{w}^*$*, we expect its invariant to be* $\delta_{\mathtt{c}} * \delta_{\mathtt{d}}$ *and shall provide a corresponding view :* $\mathtt{d\#I}\langle\mathtt{v}^*, \mathtt{w}^*\rangle \equiv \mathtt{root}::\mathtt{d}\langle\mathtt{v}^*, \mathtt{w}^*\rangle * \delta_{\mathtt{c}} * \delta_{\mathtt{d}}$.

The use of separating conjunction to capture the class invariant allows a form of object ownership to be specified for heap objects present in $\delta_{\mathtt{c}}$. Furthermore, invariant-enhanced view can easily and explicitly indicate when an invariant can be enforced and when it can be assumed. If a $\mathtt{c}\langle\mathtt{v}^*\rangle$ is being used, the class invariant is neither enforced nor assumed. If a $\mathtt{c\#I}\langle\mathtt{v}^*\rangle$ is used in the precondition, its invariant must be enforced at each of its call sites, but can be assumed to hold at the beginning of its method declaration. If a $\mathtt{c\#I}\langle\mathtt{v}^*\rangle$ is used in the postcondition, its invariant must be enforced at the end of its method declaration, but can be assumed to hold at the post-state of each of its call sites.

With the help of invariant-enhanced views, we can provide pre/post specifications that guarantee class invariants are always maintained by public methods. This can help ensure that all objects created and manipulated by public methods are guaranteed to satisfy their class invariants. Alternatively, it is also possible to allow some methods (typically private ones) to receive or produce objects *without* the invariant property. This corresponds to situations where the class invariant is temporarily broken. Our invariant-enhanced views can achieve this as they can be selectively and automatically enforced in pre/post annotations.

For example, the invariant-enhanced view of PosCnt is:

$$\texttt{PosCnt\#I}\langle \texttt{t}, \texttt{v}, \texttt{p}\rangle \equiv \texttt{root::PosCnt}\langle \texttt{t}, \texttt{v}, \texttt{p}\rangle * \texttt{v} \geq 0$$

Two methods get and tick are being inherited from the Cnt superclass, while a third method set is re-defined to ensure the class invariant. We may provide new static specifications for these three respective methods, to incorporate the invariant-enhanced view. Figure 6.3 shows how this is done for our running example. It is sufficient to use a weaker precondition of the form $\texttt{this::PosCnt}\langle \texttt{v}\rangle$ for *static-spec*(PosCnt.set) without compromising its postcondition $\texttt{this::PosCnt\#I}\langle \texttt{x}\rangle$. This corresponds to a temporary violation of the class invariant of PosCnt.

6.6.3 Enhanced Specification Subsumption

With our use of more precise static specifications, we can now leverage on a better specification subsumption that can exploit the local reasoning capability of separation logic. In particular, the extended fields of objects that are not used should be preserved by specification subsumption. More formally, we define the enhanced form of specification subsumption, as follows:

Definition 6.6.4 (Enhanced Spec. Subsumption). *A pre/post annotation* $\texttt{preB} * \!\!\rightarrow \texttt{postB}$ *is said to be a subtype of another pre/post annotation* $\texttt{preA} * \!\!\rightarrow \texttt{postA}$ *if the following relation holds:*

$$\frac{\texttt{preA} \vdash \texttt{preB} * \Delta \qquad \texttt{postB} * \Delta \vdash \texttt{postA}}{(\texttt{preB} * \!\!\rightarrow \texttt{postB}) <: (\texttt{preA} * \!\!\rightarrow \texttt{postA})}$$

Note that Δ *captures the* residual heap state *from the contravariance check on preconditions that is carried forward to assist in the covariance check on postconditions.*

As an example of its utility, consider the following specification subsumption that is expected to hold for enhanced OO verification.

$$\textit{static-spec}(\texttt{Cnt.set}) <: \textit{dynamic-spec}(\texttt{Cnt.set})$$

For the above to hold, we must prove:

$$\begin{array}{l} \texttt{this::Cnt}\langle \texttt{t}, \texttt{v}, \texttt{p}\rangle * \!\!\rightarrow \texttt{this::Cnt}\langle \texttt{t}, \texttt{x}, \texttt{p}\rangle \\ \quad <: \texttt{this::Cnt}\langle \texttt{t}, \texttt{v}, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{Cnt}, \texttt{t}\rangle \wedge \texttt{x} \geq 0 * \!\!\rightarrow \\ \qquad \texttt{this::Cnt}\langle \texttt{t}, \texttt{x}, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{Cnt}, \texttt{t}\rangle \end{array}$$

The above subtyping cannot be proven with the basic specification subsumption relation from Sec 1 (without the use of a residual heap state), but succeeds with our enhanced subsumption relation.

We first show the contravariance of the preconditions:

$$\texttt{this}{::}\texttt{Cnt}\langle \texttt{t}, \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{ExtAll}\langle \texttt{Cnt}, \texttt{t}\rangle \wedge \texttt{x} \geq 0$$
$$\vdash \texttt{this}{::}\texttt{Cnt}\langle \texttt{t}, \texttt{v}, \texttt{p}\rangle * \Delta$$

This succeeds with $\Delta \equiv \texttt{p}{::}\texttt{ExtAll}\langle \texttt{Cnt}, \texttt{t}\rangle \wedge \texttt{x} \geq 0$. We then prove covariance on the postconditions using:

$$\texttt{this}{::}\texttt{Cnt}\langle \texttt{t}, \texttt{x}, \texttt{p}\rangle * \Delta \vdash \texttt{this}{::}\texttt{Cnt}\langle \texttt{t}, \texttt{x}, \texttt{q}\rangle * \texttt{q}{::}\texttt{ExtAll}\langle \texttt{Cnt}, \texttt{t}\rangle$$

This is proven with the help of residual heap state Δ (with an extension record) from the entailment of preconditions.

Our preservation of residual heap state is inspired by the needs of static specification. By the use of a new object format (with lossless casting) and a novel specification subsumption mechanism, we can now support a modular verification process in which re-verification is always avoided for dynamic specifications. Our enhanced specification subsumption can also be viewed as a practical algorithm for implementing Parkinson's specification compatibility [99]. This link shall be formally proven later in Lemma 6.1.

6.7 Conformance to the OO Paradigm

We present mechanisms to ensure that method overriding and method inheritance are supported in accordance with the requirements of the OO paradigm.

6.7.1 Behavioral Subtyping with Dynamic Specifications

Dynamic specifications are meant for the methods of a given class and its subclasses. They must conform to the behavioral subtyping principle to support method overriding (and inheritance), as defined by the requirement below:

Definition 6.7.1 (Behavioral Subtyping Requirement). *Given a dynamic specification* $\texttt{preC} \mathrel{*\!\!\rightarrow} \texttt{postC}$ *in a method* mn *in class* C *and another dynamic specification* $\texttt{preD} \mathrel{*\!\!\rightarrow} \texttt{postD}$ *of the corresponding method* mn *in a subclass* D*. We say that the two specifications adhere to the* behavioral subtyping requirement *using* $(\texttt{preD} \mathrel{*\!\!\rightarrow} \texttt{postD}) <:_{\texttt{D}} (\texttt{preC} \mathrel{*\!\!\rightarrow} \texttt{postC})$*, if the following subsumption holds :* $\texttt{preD} \mathrel{*\!\!\rightarrow} \texttt{postD} <: (\texttt{preC} \wedge \mathit{type}(\texttt{this}) <: \texttt{D} \mathrel{*\!\!\rightarrow} \texttt{postC})$.

As shown above, we can use the enhanced specification subsumption relation to check for behavioral subtyping. For an example, consider the dynamic specification of method `Cnt.set` and its overriding method `PosCnt.set`. Assuming that these dynamic specifications are given, the behavioral subtyping requirement can be checked using:

$$\textit{dynamic-spec}(\texttt{PosCnt.set}) <:_{\texttt{PosCnt}} \textit{dynamic-spec}(\texttt{Cnt.set})$$

Hence, we have:

$$\texttt{this::PosCnt}\langle\_\rangle\$\wedge\texttt{x}\geq 0 \mathrel{*\!\!\rightarrow} \texttt{this::PosCnt\#I}\langle\texttt{x}\rangle\$ \;\; <:$$
$$\texttt{this::Cnt}\langle\texttt{v}\rangle\$\wedge\texttt{x}\geq 0\wedge(\textit{type}(\texttt{this})<:\texttt{PosCnt}) \mathrel{*\!\!\rightarrow} \texttt{this::Cnt}\langle\texttt{x}\rangle\$$$

By contravariance of preconditions, we successfully prove:

$$\texttt{this::Cnt}\langle\texttt{v}\rangle\$\wedge\texttt{x}\geq 0\wedge(\textit{type}(\texttt{this})<:\texttt{PosCnt}) \vdash$$
$$\texttt{this::PosCnt}\langle\_\rangle\$\wedge\texttt{x}\geq 0 * \Delta$$

where Δ is derived to be $\texttt{x}\geq 0$. By covariance of postconditions, we can prove:

$$\texttt{this::PosCnt\#I}\langle\texttt{x}\rangle\$ * \Delta \vdash \texttt{this::Cnt}\langle\texttt{x}\rangle\$$$

Hence, the above two dynamic specifications of `Cnt.set` and `PosCnt.set` conform to the behavioral subtyping requirement.

Dynamic specifications may also be given (or derived) for *inherited methods*, especially when their static specifications have been modified. As with method overriding, we continue to expect that the behavioral subtyping requirement holds between a dynamic specification (as supertype) for a method in a class and another dynamic specification (as subtype) for the same inherited method in the subclass. Let us consider `Cnt.tick` and its inherited method `PosCnt.tick`. Though no method overriding is present, we must still ensure $\textit{dynamic-spec}(\texttt{PosCnt.tick}) <:_{\texttt{PosCnt}} \textit{dynamic-spec}(\texttt{Cnt.tick})$.

6.7.2 Statically-Inherited Methods

Under the OO paradigm, it is possible for a method `mn` in a class `C` to be inherited into its subclass `D` without any overriding. Furthermore, the user is free to add a new static/dynamic specification to such an inherited method for each subclass. Such a scenario may occur for a subclass with a strengthened invariant. For each inherited method of this subclass, we anticipate a new static specification possibly using its invariant-enhanced view. An important question to

ask is if there is a need to re-verify this new static specification against the body of the inherited method.

We shall first consider static specification where the receiver is specified using partial view of form $\texttt{this::c}\langle \texttt{t}, \texttt{v}^*, \texttt{p} \rangle$. For this category of static specifications, we are expecting that each method invocation by $\texttt{this.mn}(..)$ does not modify any fields in the extension record and is the same as that in the original method prior to method inheritance. To support the inheritance of static specifications which use partial views for their receivers, without re-verification, we identify a category of inherited methods that is semantically equivalent (modulo the receiver) to the original method in the superclass.

Definition 6.7.2 (**Statically-Inherited Methods).** *Given a method* `mn` *with body* `e` *from class* `A` *that is being inherited into a subclass* `B`*, we say that this method is* statically-inherited*, if the following conditions hold:*

- *it has not been overridden in the* `B` *subclass.*
- *for all auxiliary calls* $\texttt{this.mn2}(..)$ *for which* $\texttt{mn} \neq \texttt{mn2}$*, it must be the case that* `B.mn2` *is* statically-inherited *from* `A.mn2`*.*

We can show that each *statically-inherited* method is *semantically equivalent* to the original method from its superclass. The above conditions ensure this by checking that the inherited method always invokes the same sequence of semantically equivalent method calls, as that when executed with a receiver object from its superclass. With this classification for *statically-inherited* methods, we can check inherited static specifications, as follows:

Definition 6.7.3 (**Checking Inherited Static Specifications).**
Consider a method `mn` *with static specification* `spA` *(using a partial view for its receiver) from class* `A` *that is being inherited into a subclass* `B` *with static specification* `spB`*. If this method has been* statically-inherited *into subclass* `B`*, we only need to check for specification subsumption* $\texttt{spA} <: \texttt{spB}$*. Otherwise, we have to re-verify the method body of* `mn` *with the new static specification* `spB`*.*

As an example, `PosCnt.tick` is statically-inherited from `Cnt.tick` (with a partial view $\texttt{this::Cnt}\langle \texttt{v} \rangle$), and we can conclude that both methods are semantically equivalent modulo the receiver. To avoid the re-verification of the static specification of `PosCnt.tick`, we only need

to check for the following subtyping:

$$\textit{static-spec}(\texttt{Cnt.tick}) <: \textit{static-spec}(\texttt{PosCnt.tick})$$

Some other methods, such as `PosCnt.get`, `FastCnt.get`, `FastCnt.set`, `TwoCnt.tick` and `TwoCnt.get`, are also statically-inherited. For a counterexample that is not statically-inherited, consider:

```
class A {
  int foo { return this.goo() }
  int goo { return 1 }
}

class B extends A {
  int goo { return 2 }
}
```

The `foo` method cannot be statically-inherited in subclass `B`, since it invokes an auxiliary `goo` method that is *not* statically-inherited (in this case overridden). In other words, method `B.foo()` is not semantically equivalent (modulo the receiver) to `A.foo()` since they invoke different sequences of method calls when given the same parameters except for the receiver. As a result, we expect that the static specification (with partial view) for `B.foo` must be re-verified against its inherited method body from `A.foo`.

We have two solutions for handling methods that are not statically-inherited. One solution is to transform each method that is *not* statically-inherited into an overriding method. This is achieved by cloning the method declaration for each such method in its subclass. By doing so, we force code re-verification to be performed for the cloned methods, when inheriting static specifications into such non statically-inherited methods. A second solution is to utilize full views on the receivers of static specifications. By using full views on receivers, we shall be handling each method invocation of the form `this.mn2(...)` by using its corresponding dynamic specification. As a consequence, each such static specification (with full views on receiver) can always be inherited into any subclass without the need for re-verification, regardless of whether the method is statically-inherited or not. However, some loss in precision may occur since dynamic specifications are now used by each receiver during the verification of its method's body.

6.8 Deriving Specifications

While a static specification can give better precision, having to maintain both static and dynamic specifications may seem like more human effort is required by our approach to OO verification. To alleviate this, we provide the following set of derivation techniques that can be used, where needed.

- derive dynamic specifications from static counterparts.
- refine dynamic specifications to meet behavioral subtyping.
- inherit static specifications from method of superclass.

Let us initially assume that none of the dynamic specifications are given for our running example. We first present a simple technique for deriving a dynamic specification from its static counterpart, as follows:

Definition 6.8.1 (From Static to Dynamic Specification). *Given a static specification* `specS` *for class* `C`*, we shall derive its dynamic counterpart* `specD`*, as follows:*

$$\begin{aligned} \texttt{specD} &= \rho_{\texttt{C}}\ \texttt{specS} \qquad \textit{where} \\ \rho_{\texttt{C}} &= [\texttt{this::C}\langle \texttt{v}^* \rangle \mapsto \texttt{this::C}\langle \texttt{v}^* \rangle \$, \\ &\qquad \texttt{this::C\#I}\langle \texttt{v}^* \rangle \mapsto \texttt{this::C\#I}\langle \texttt{v}^* \rangle \$] \end{aligned}$$

Some examples of dynamic specifications that can be automatically derived from their static counterparts are:

$$\begin{aligned}&\textit{dynamic-spec}(\texttt{Cnt.get}) = \rho_{\texttt{Cnt}}\ \textit{static-spec}(\texttt{Cnt.get})\\ &\quad = \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \mathrel{*\!\!\rightarrow} \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \wedge \texttt{res=v}\end{aligned}$$

$$\begin{aligned}&\textit{dynamic-spec}(\texttt{Cnt.tick}) = \rho_{\texttt{Cnt}}\ \textit{static-spec}(\texttt{Cnt.tick})\\ &\quad = \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \mathrel{*\!\!\rightarrow} \texttt{this::Cnt}\langle\texttt{v+1}\rangle\$\end{aligned}$$

$$\begin{aligned}&\textit{dynamic-spec}(\texttt{PosCnt.get}) = \rho_{\texttt{PosCnt}}\ \textit{static-spec}(\texttt{PosCnt.get})\\ &\quad = \texttt{this::PosCnt\#I}\langle\texttt{v}\rangle\$ \mathrel{*\!\!\rightarrow} \texttt{this::PosCnt\#I}\langle\texttt{v}\rangle\$ \wedge \texttt{res=v}\end{aligned}$$

$$\begin{aligned}&\textit{dynamic-spec}(\texttt{FastCnt.tick})\\ &\quad = \rho_{\texttt{FastCnt}}\ \textit{static-spec}(\texttt{FastCnt.tick})\\ &\quad = \texttt{this::FastCnt}\langle\texttt{v}\rangle\$ \mathrel{*\!\!\rightarrow} \texttt{this::FastCnt}\langle\texttt{v+2}\rangle\$\end{aligned}$$

This technique can help us derive dynamic specifications that are almost identical to static specifications, and are especially relevant for methods (e.g. in final classes) where overriding is not possible. However, these automatically derived dynamic specifications may fail to meet the behavioral subtyping requirement. Failure of behavioral subtyping can be due to two possible reasons:

1. Dynamic specification of method in superclass is too strong, or
2. Dynamic specification of method in subclass is too weak.

We propose two refinement techniques for related pairs of dynamic specifications to help them conform to behavioral subtyping. A conventional way is to use *specification inheritance* (or *specialization*) to strengthen the dynamic specification of the overriding method. However, in our approach, this technique of strengthening the dynamic specifications of a method in the subclass may violate a key requirement that the dynamic specification be a supertype of its static counterpart. Thus, prior to using specification specialization, we must either check that each inherited dynamic specification is indeed a supertype of the static specification from the overriding method, or can be made to inherit the static specification from the overridden method, as follows :

Definition 6.8.2 (Specification Specialization). *Given a dynamic specification* $\texttt{preD}_\texttt{A} \ast\!\!\rightarrow \texttt{postD}_\texttt{A}$ *and its static specification* $\texttt{preS}_\texttt{A} \ast\!\!\rightarrow \texttt{postS}_\texttt{A}$ *for a method* `mn` *in class* `A`*, and its overriding method in a subclass* `B` *with static specification* $\texttt{preS}_\texttt{B} \ast\!\!\rightarrow \texttt{postS}_\texttt{B}$*. A dynamic specification* $(\texttt{preD}_\texttt{A} \wedge \mathit{type}(\texttt{this}) <: \texttt{B} \ast\!\!\rightarrow \texttt{postD}_\texttt{A})$ *can be added to the overriding method of the* `B` *subclass if either of the following occurs:*

- $\texttt{preS}_\texttt{B} \ast\!\!\rightarrow \texttt{postS}_\texttt{B} <:_\texttt{B} \texttt{preD}_\texttt{A} \ast\!\!\rightarrow \texttt{postD}_\texttt{A}$ *holds, or*
- $\rho_{\texttt{A}\rightarrow\texttt{B}}(\texttt{preS}_\texttt{A} \ast\!\!\rightarrow \texttt{postS}_\texttt{A})$ *can be inherited into the static specification of* `mn` *in class* `B` *and successfully verified.*

Note that

$$\rho_{\texttt{A}\rightarrow\texttt{B}} = [\texttt{this::A}\langle \texttt{v}^* \rangle \mapsto \texttt{this::B}\langle \texttt{v}^*, \texttt{w}^* \rangle, \texttt{this::A\#I}\langle \texttt{v}^* \rangle \mapsto \texttt{this::B}\langle \texttt{v}^*, \texttt{w}^* \rangle \ast \delta_\texttt{A}]$$

where $\texttt{w}^*$ *are free variables of the extended record, while* $\delta_\texttt{A}$ *captures the invariant of* `A` *class. The refined dynamic specification for the overriding method is obtained via intersection type,*

$$(\texttt{preD}_\texttt{B} \ast\!\!\rightarrow \texttt{postD}_\texttt{B}) \bigwedge (\texttt{preD}_\texttt{A} \wedge \mathit{type}(\texttt{this}) <: \texttt{B} \ast\!\!\rightarrow \texttt{postD}_\texttt{A}).$$

As an example, the pair of dynamic specifications for `Cnt.get` and `PosCnt.get` do not conform to behavioral subtyping. We may therefore attempt to strengthen the dynamic specification of `PosCnt.get` by specification specialization through the following multi-specification:

$$\begin{array}{l} \texttt{this::PosCnt\#I}\langle \texttt{v} \rangle \$ \ast\!\!\rightarrow \texttt{this::PosCnt\#I}\langle \texttt{v} \rangle \$ \wedge \texttt{res=v} \bigwedge \\ \texttt{this::Cnt}\langle \texttt{v} \rangle \$ \wedge \mathit{type}(\texttt{this}) <: \texttt{PosCnt} \ast\!\!\rightarrow \texttt{this::Cnt}\langle \texttt{v} \rangle \$ \wedge \texttt{res=v} \end{array}$$

However, the inherited dynamic specification from `Cnt.get` is *not* a supertype of *static-spec*(`PosCnt.get`). Hence, in order to proceed with this refinement, we must also inherit the static specification of *static-spec*(`Cnt.get`) into `PosCnt.get`, as follows:

$$\begin{array}{l} \texttt{this::PosCnt\#I}\langle \texttt{v} \rangle \ast\!\!\rightarrow \texttt{this::PosCnt\#I}\langle \texttt{v} \rangle \wedge \texttt{res=v} \bigwedge \\ \texttt{this::PosCnt}\langle \texttt{v} \rangle \ast\!\!\rightarrow \texttt{this::PosCnt}\langle \texttt{v} \rangle \wedge \texttt{res=v} \end{array}$$

This strengthened static specification is now a subtype of the correspondingly derived dynamic specification. Furthermore, behavioral subtyping holds between the new dynamic specifications of `Cnt.get` and `PosCnt.get`. A caveat about specification specialization is that the

strengthened static specification of the method in the subclass may *not* always guarantee the invariant property. For example, this::PosCnt#I⟨v⟩ ⋆⇢ this::PosCnt#I⟨v⟩ ∧ res=v guarantees that the class invariant of PosCnt is preserved, but not this::PosCnt⟨v⟩ ⋆⇢ this::PosCnt⟨v⟩∧ res=v. It is thus possible for successfully verified calls of this method to violate the class invariant property, but the above multi-specification is fully aware of when each such violation occurs through the use of different predicates. This violation of a class invariant is one reason why [41] considered specification inheritance to be a potentially 'unsound' derivation technique.

As a complement to specification specialization, we propose a dual mechanism that weakens the specification of the overridden method instead. We refer to this new technique as *specification abstraction*. Instead of an intersection type, we use a *union type* to obtain a weaker dynamic specification for the overridden method. Formally:

Definition 6.8.3 (Specification Abstraction). *Given a dynamic specification* $\texttt{preD}_\texttt{A} \mathbin{⋆⇢} \texttt{postD}_\texttt{A}$ *for a method* mn *in class* A*, and its overriding method in a subclass* B *with dynamic specification* $\texttt{preD}_\texttt{B} \mathbin{⋆⇢} \texttt{postD}_\texttt{B}$*. If behavioral subtyping does not hold between these dynamic specifications, we can generalise the specification of the overridden method using the following union type:*

$$\begin{aligned} \textit{dynamic-spec}(\texttt{A.mn}) = (\texttt{preD}_\texttt{A} \mathbin{⋆⇢} \texttt{postD}_\texttt{A}) \\ \bigvee \rho_{\texttt{B}\to\texttt{A}}(\texttt{preD}_\texttt{B}) \mathbin{⋆⇢} \exists \texttt{w}^* \cdot \rho_{\texttt{B}\to\texttt{A}}(\texttt{postD}_\texttt{B}) \end{aligned}$$

$$\begin{aligned} \rho_{\texttt{B}\to\texttt{A}} = [&\texttt{this::B}\langle \texttt{v}^*, \texttt{w}^* \rangle \$ \mapsto \texttt{this::A}\langle \texttt{v}^* \rangle \$, \\ &\texttt{this::B\#I}\langle \texttt{v}^*, \texttt{w}^* \rangle \$ \mapsto \texttt{this::A\#I}\langle \texttt{v}^* \rangle \$ * \delta_\texttt{B}] \end{aligned}$$

We refer to this process as specification abstraction*. It is a safe operation that weakens the dynamic specification of an overridden method to the point where behavioral subtyping holds.*

As an example, consider the derived dynamic specifications from a pair of methods Cnt.tick and FastCnt.tick where behavioral subtyping does not currently hold. We are unable to apply specification specialization, as the inherited static specification of Cnt.tick cannot be verified by the overriding method of FastCnt.tick. However, with the help of specification abstraction, we can obtain the following union type for *dynamic-spec*(Cnt.tick) instead.

$$\begin{aligned} &\texttt{this::Cnt}\langle \texttt{v} \rangle \$ \mathbin{⋆⇢} \texttt{this::Cnt}\langle \texttt{v+1} \rangle \$ \bigvee \\ &\texttt{this::Cnt}\langle \texttt{v} \rangle \$ \mathbin{⋆⇢} \texttt{this::Cnt}\langle \texttt{v+2} \rangle \$ \end{aligned}$$

Our current separation logic prover is able to directly handle intersection types but *not* union types for its multi-specifications. We propose to handle union type by the following translation instead:

$$(\mathtt{pre}_1 \mathrel{*\!\!\rightarrow} \mathtt{post}_1) \bigvee (\mathtt{pre}_2 \mathrel{*\!\!\rightarrow} \mathtt{post}_2)$$
$$\Longrightarrow (\mathtt{pre}_1 \wedge \mathtt{pre}_2) \mathrel{*\!\!\rightarrow} (\mathtt{post}_1 \vee \mathtt{post}_2)$$

For brevity, we shall omit the formal details of how normalization (of separation logic formulae) is carried out for the above translation. In the case of `Cnt.tick`, we can perform normalization to obtain the following weakened dynamic specification:

$$\mathtt{this::Cnt}\langle \mathtt{v} \rangle\$ \mathrel{*\!\!\rightarrow} \mathtt{this::Cnt}\langle \mathtt{w} \rangle\$ \wedge (\mathtt{w}{=}\mathtt{v}{+}1 \vee \mathtt{w}{=}\mathtt{v}{+}2)$$

It would appear that the use of specification abstraction loses modularity, due to its dependence on the dynamic specifications of overriding methods. However, this is not true. Firstly, the purpose of specification abstraction is to derive dynamic specifications which need not be re-verified. Secondly, we maintain modularity as each static specification is verified once, but need only be re-verified when the specifications it depends on change. Though changes may occur for a dynamic specification that a method depends on; the necessity for re-verification is analogous to a modular compilation system which re-compiles a module whenever the type interface it depends on changes.

While our approach can theoretically derive all dynamic specifications, we shall also allow the option for users to directly specify dynamic specifications, where required. This option is especially helpful in supporting modular open-ended classes that could be further extended with new subclasses. Our overall procedure for selectively but automatically deriving dynamic specifications shall be as follows:

Definition 6.8.4 (Deriving Dynamic Specifications). *We derive and refine dynamic specifications, as follows:*

- *If the dynamic specifications of both overridden and overriding methods are given, check for the behavioral subtyping requirement.*

- *If only the dynamic specification of an overridden method is given, derive the dynamic specification of the overriding method and then use* specification specialization *to refine*

it.

- *If only the dynamic specification of an overriding method is given, derive the dynamic specification of the overridden method and then use* specification abstraction *to refine it.*

- *Otherwise, derive both dynamic specifications and then use* specification abstraction *to refine the dynamic specification of the overridden method in the superclass.*

Note that the procedure is geared towards the preservation of class invariants, where possible, as it favours specification abstraction over specification specialization.

Lastly, it may also be possible for static specifications to be omitted for some statically-inherited methods. We propose a way to derive static specifications for such methods, as follows:

Definition 6.8.5 (Deriving Static Specifications). *Given a method* `mn` *from class* `A` *with static specification* `spA`*, and a subclass* `B` *where the same method has been* statically-inherited*. If no static specification is given for* `B.mn`*, we can derive a static specification for it, as follows :*

$$\textit{static-spec}(\texttt{B.mn}) = [\texttt{this::A}\langle \texttt{v}^* \rangle \mapsto \texttt{this::B}\langle \texttt{v}^*, \texttt{w}^* \rangle]\ \texttt{spA}$$

The extra fields, $\texttt{w}^*$*, in the subclass are never modified by each statically-inherited method.*

The above substitution is only applicable for partial views, and it is not needed for full views which will remain unchanged when deriving static specifications.

Though specification derivation techniques are important aids that make it easier for users to adopt our OO verification methodology, they are not fundamental in the current work. In the rest of this chapter, we shall assume that all required dynamic and static specifications are available, and proceed to describe core components of our enhanced OO verification system.

6.9 Forward Verification

Our verification system for OO programs is implemented in a modular fashion. It processes the class declarations in a top-down manner whereby the methods of superclasses are verified before those of the subclasses. We shall assume that static specifications are given, and that dynamic specifications are already given (or automatically derived). Also, for each method that

is *not* statically inherited in a subclass, we shall clone the method for that subclass. There are three major subsystems present, namely: (i) View Generator, (ii) Inheritance Checker, and (iii) Code Verifier. These are elaborated next.

6.9.1 View Generator

For each subclass in the class hierarchy, we must generate a lossless upcasting rule in accordance with Defn 6.6.1. However, the format $\texttt{Ext}\langle \texttt{c}, \texttt{v}^*, \texttt{p}\rangle$ actually denotes a family of record extensions that is distinct for each subclass c. To distinguish them clearly in our implementation, we provide a set of specialised record extensions of the form $\texttt{Ext}_\texttt{c}\langle \texttt{v}^*, \texttt{p}\rangle$ instead. With this change, we can generate the following casting rules for our running example:

$$\texttt{PosCnt}\langle \texttt{t},\texttt{n},\texttt{p}\rangle \equiv \texttt{root::Cnt}\langle \texttt{t},\texttt{n},\texttt{q}\rangle * \texttt{q::Ext}_{\texttt{PosCnt}}\langle \texttt{p}\rangle$$

$$\texttt{FastCnt}\langle \texttt{t},\texttt{n},\texttt{p}\rangle \equiv \texttt{root::Cnt}\langle \texttt{t},\texttt{n},\texttt{q}\rangle * \texttt{q::Ext}_{\texttt{FastCnt}}\langle \texttt{p}\rangle$$

$$\texttt{TwoCnt}\langle \texttt{t},\texttt{n},\texttt{b},\texttt{p}\rangle \equiv \texttt{root::Cnt}\langle \texttt{t},\texttt{n},\texttt{q}\rangle * \texttt{q::Ext}_{\texttt{TwoCnt}}\langle \texttt{b},\texttt{p}\rangle$$

Correspondingly, we may also provide an ExtAll view for the class hierarchy. In the case of our running example, we can generate the following definition for the ExtAll view:

$$\begin{aligned}\texttt{ExtAll}\langle \texttt{t}_1,\texttt{t}_2\rangle \equiv\ & (\texttt{t}_1{=}\texttt{t}_2)\wedge\texttt{root}{=}\texttt{null}\\ \vee\ & \texttt{root::Ext}_{\texttt{PosCnt}}\langle \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{PosCnt},\texttt{t}_2\rangle\\ & \wedge\texttt{PosCnt}{<}\texttt{t}_1\wedge\texttt{t}_2{<}\texttt{:PosCnt}\\ \vee\ & \texttt{root::Ext}_{\texttt{FastCnt}}\langle \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{FastCnt},\texttt{t}_2\rangle\\ & \wedge\texttt{FastCnt}{<}\texttt{t}_1\wedge\texttt{t}_2{<}\texttt{:FastCnt}\\ \vee\ & \texttt{root::Ext}_{\texttt{TwoCnt}}\langle \texttt{b},\texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{TwoCnt},\texttt{t}_2\rangle\\ & \wedge\texttt{TwoCnt}{<}\texttt{t}_1\wedge\texttt{t}_2{<}\texttt{:TwoCnt}\end{aligned}$$

Lastly, for each subclass with a non-trivial invariant, we also generate two invariant-enhanced views for this subclass. For our running example, only the subclass PosCnt has an invariant. Hence, our generator will provide the following:

$$\texttt{PosCnt\#I}\langle \texttt{t},\texttt{n},\texttt{p}\rangle \equiv \texttt{root::PosCnt}\langle \texttt{t},\texttt{n},\texttt{q}\rangle\wedge\texttt{n}{\geq}0$$

In summary, the above shows how we explicitly generate predicate views for casting and class invariants. In practice, our prototype verification system creates these views on demand during entailment checking itself.

6.9.2 Inheritance Checker

This subsystem ensures that specifications of added methods are consistent with class inheritance and method overriding requirements. Whenever a new subclass `B` is added, we expect a set of new overriding methods and another set of statically-inherited methods. We propose to check for consistency, as follows:

- Firstly, we check that each static specification is a subtype of the dynamic specification.
- For each new overriding method `B.mn`, we identify the nearest overridden method in a superclass of `B`. We then check that each given dynamic specification is a subtype of the given dynamic specification of its overridden method in its superclass.
- For each statically-inherited method `B.mn`, we check that its given static specification is a supertype of the corresponding static specification in its superclass. If a dynamic specification is also given, we check that it is a subtype of the given dynamic specification in its superclass.

Some of the static and dynamic specifications may have been automatically derived. As these derived specifications are correct by construction, we shall not be checking for the specification subsumption relation amongst them.

6.9.3 Code Verifier

There are four features in our core language that are peculiar to the OO paradigm, namely (i) the object constructors, (ii) the cast constructs, (iii) instance method invocations and (iv) `super` calls. Let us discuss how they are handled.

For the object constructor, we use the following rule where each primed variable $\mathtt{v}'_\mathtt{i}$ captures the latest value of variable $\mathtt{v}_\mathtt{i}$:

$$\frac{S_1{=}\{\Delta * \mathtt{res}{::}\mathtt{c}\langle \mathtt{c}, v'_1, .., v'_n, \mathtt{null}\rangle\}}{\vdash \{\Delta\}\ \mathtt{new}\ \mathtt{c}(v_1, .., v_n)\ \{S_1\}}$$

This rule produces an object of actual type `c` using partial view.

Consider a cast construct $(\mathtt{c})(\mathtt{v}{:}\mathtt{c}_1)$ where $\mathtt{v}{:}\mathtt{c}_1$ captures the compile-time type of v inserted before verification. We shall treat it as being equivalent to a primitive call of the form:

$$\mathtt{c}\ \mathtt{cast}_{\mathtt{c}}\ (\mathtt{c}_1\ \mathtt{v})\ \mathbf{static}\quad \mathtt{v}{::}\mathtt{c}_1\langle \mathtt{t},..\rangle \ast\!\!\rightarrow \mathtt{v}{::}\mathtt{c}_1\langle \mathtt{t},..\rangle \wedge \mathtt{t}{<:}\mathtt{c}$$
$$\wedge\ \mathtt{true} \ast\!\!\rightarrow \mathtt{true}$$

The above declaration allows the cast construct to possibly fail at runtime. If casting succeeds, we may expect that the actual type of the object to be a subtype of c, as captured by the first pre/post annotation. The second pre/post annotation is added for completeness, and may be used if we are unable to establish the heap state of v.

Another important feature to consider is instance method call of the form $(\mathtt{v}{:}\mathtt{c}).\mathtt{mn}(\mathtt{v}_1..\mathtt{v}_n)$. We first identify the best possible type of v using $\beta{=}\mathtt{findtype}(\Delta, \mathtt{v}{:}\mathtt{c})$. The result β will tell us if we have the actual type $\mathtt{t}{=}\mathtt{c}_1$ or the best static type $\mathtt{t}{<:}\mathtt{c}_1$ where $\mathtt{t}{=}\mathit{type}(\mathtt{v})$ and $\mathtt{c}_1{<:}\mathtt{c}$. Note that $\mathtt{c}_1$ can be more precise than the compile-time type c due to our use of flow- and path-sensitive reasoning. If the actual type is known, we choose the static specification of method mn from class $\mathtt{c}_1$. Otherwise, we choose its dynamic specification instead. This decision is captured by $\mathtt{spec}{=}\mathtt{findspec}(\mathtt{P}, \beta, \mathtt{mn})$ where P denotes the entire OO program. The overall rule is:

$$\frac{\begin{array}{c}\beta{=}\mathtt{findtype}(\Delta, v{:}c) \qquad \rho = [v_1{\mapsto}v_1', ..., v_n{\mapsto}v_n'] \\ \mathtt{findspec}(\mathtt{P}, \beta, \mathtt{mn}) = \{\Phi^i_{pr} \ast\!\!\rightarrow \Phi^i_{po}\}_{i=1}^{p} \\ \rho{=}[v_j'/v_j]_{j=1}^{n} \qquad \Delta{\vdash}\rho\Phi^i_{pr} * S_i \quad \forall i{=}1,..,p \\ S = \bigcup_{i=1}^{p} S_i * \Phi^i_{po}\end{array}}{\vdash \{\Delta\}\ (v{:}c).\mathtt{mn}(v_1..v_n)\ \{S\}}$$

If Δ is a disjunctive formula with different types for v, we can use findtype/findspec operations in the entailment procedure, so that the best specification is selected for either the actual or the static type of the object at v for each disjunct. For multi-specifications, we choose the first specification whose precondition holds. We assume that these multiple specifications are ordered to yield a more precise result ahead of the less precise ones.

We can easily deal with the invocation of super methods. This feature can be used in place of the receiver this parameter to refer to the overridden method. It can be easily handled by our approach since super method calls are essentially *static* calls that can be precisely captured by static specifications. Consider an overridden method mn in a superclass A and a call super.mn(..) being used in an overriding method in subclass B. We can handle this super call by re-writing it to this.A.mn(..). In this case, our verification process will select the *static* specification of the

overridden method in class A to use. Past works, such as [100, 62], do not handle super method calls for verification well, as there is an inherent mismatch between super method calls (which are static calls) and the mechanism based on dynamic specifications.

6.10 Soundness

The semantics of our constraints is that introduced in Sec 2.6. There are several soundness results that are needed to show the overall safety of our verification system. The following lemma highlights a key result showing that our use of specification subsumption relation is sound for avoiding re-verification, as follows:

Lemma 6.10.1 (Soundness of Enhanced Spec. Subsumption).

Given that method body e *has been successfully verified using* $\texttt{preB} \mathrel{*\!\!\rightarrow} \texttt{postB}$. *If specification subsumption* $\texttt{preB} \mathrel{*\!\!\rightarrow} \texttt{postB} <: \texttt{preA} \mathrel{*\!\!\rightarrow} \texttt{postA}$ *holds, then its specification supertype* $\texttt{preA} \mathrel{*\!\!\rightarrow} \texttt{postA}$ *is guaranteed to verify successfully against the same method body.*

Proof: From the premise of specification subsumption (Defn 6.6.4), we can obtain: $\texttt{preA} \vdash \texttt{preB} * \Delta$ and $\texttt{postB} * \Delta \vdash \texttt{postA}$. In our context, preconditions preA, preB and their entailment's residual Δ do not contain any primed variables, while only primed variables are modified indirectly by our program. Hence, adding a formula with only unprimed variables, such as Δ, to both pre/post always satisfies the side condition of the frame rule. Let e denote the method body which has been preprocessed to a form where pass-by-value parameters are never modified. Let $\{\texttt{v}_1, .., \texttt{v}_\texttt{n}\}$ denote the set of free variables in e, and let $\texttt{N} = \bigwedge_{\texttt{i}=1}^{\texttt{n}} (\texttt{v}'_\texttt{i} = \texttt{v}_\texttt{i})$. From the premise that $\texttt{preB} \mathrel{*\!\!\rightarrow} \texttt{postB}$ is a verified specification for the code, we have $\vdash \{\texttt{preB} \wedge \texttt{N}\}\ \texttt{e}\ \{\texttt{postB}\}$. In order to show that its specification supertype $\texttt{preA} \mathrel{*\!\!\rightarrow} \texttt{postA}$ is also verifiable for the same code, we need to derive $\vdash \{\texttt{preA} \wedge \texttt{N}\}\ \texttt{e}\ \{\texttt{postA}\}$. We conclude based on the following steps:

$\vdash \{\texttt{preB} \wedge \texttt{N}\}\ \texttt{e}\ \{\texttt{postB}\}$	premise
$\vdash \{\texttt{preB} \wedge \texttt{N} * \Delta\}\ \texttt{e}\ \{\texttt{postB} * \Delta\}$	frame rule
$\vdash \{\texttt{preA} \wedge \texttt{N}\}\ \texttt{e}\ \{\texttt{postB} * \Delta\}$	precondition strengthening
$\vdash \{\texttt{preA} \wedge \texttt{N}\}\ \texttt{e}\ \{\texttt{postA}\}$	postcondition weakening □

The above proof uses the following Consequence Lemma stating the soundness of precondition strengthening and postcondition weakening:

Lemma 6.10.2 (**Consequence Rule).** *The following verification holds:*

$$\frac{P' \vdash P \quad \vdash \{P\}\,\mathtt{e}\,\{Q\} \quad Q \vdash Q'}{\vdash \{P'\}\,\mathtt{e}\,\{Q'\}}$$

Proof Sketch: Based on the premise, we have a set of s, h such that $\langle s, h, \mathtt{e}\rangle \hookrightarrow^* \langle s_1, h_1, v\rangle$ and $s, h \models P \wedge s_1{+}[\mathtt{res}{\mapsto}v], h_1 \models \mathit{Post}(Q)$. By Galois connection, we have $s_1{+}[\mathtt{res}{\mapsto}v], h_1 \models \mathit{Post}(Q')$. Thus, for all $s, h \models P'$, we have $\vdash \{P'\}\,\mathtt{e}\,\{Q'\}$. □

We extract the post-state of a heap constraint by:

Definition 6.10.1 (**Poststate).** *Given a constraint* Δ*,* $\mathit{Post}(\Delta)$ *captures the relation between primed variables of* Δ*. That is :*

$$\begin{array}{l}
\mathit{Post}(\Delta) \;=_{df}\; \rho\,(\exists V{\cdot}\Delta), \qquad \textit{where}\\
V = \{v_1, .., v_n\} \textit{ denotes all unprimed program variables in } \Delta\\
\rho = [v_1'{\mapsto}v_1, .., v_n'{\mapsto}v_n]
\end{array}$$

The next two lemmas state some results on statically-inherited methods for which re-verification is proven not to be needed.

Lemma 6.10.3 (**Equivalence of Statically-Inherited Methods).**

Consider a method mn *from class* A *that satisfies the conditions of being statically-inherited into a* B *subclass. Assuming that*

$$\bigwedge \begin{array}{l}
\langle s, h_1, \mathtt{o.mn(p^*)}\rangle \hookrightarrow^* \langle s_1, h_3, v\rangle\\
h_1{=}h{+}[s(\mathtt{o}){\mapsto}\mathtt{A}(v_1..v_n)]\\
h_3{=}h'{+}[s(\mathtt{o}){\mapsto}\mathtt{A}(w_1..w_n)]
\end{array}$$

then

$$\bigwedge \begin{array}{l}
\langle s, h_2, \mathtt{o.mn(p^*)}\rangle \hookrightarrow^* \langle s_1, h_4, v\rangle\\
h_2{=}h{+}[s(\mathtt{o}){\mapsto}\mathtt{B}(v_1..v_n, v_{n+1}..v_m)]\\
h_4{=}h'{+}[s(\mathtt{o}){\mapsto}\mathtt{B}(w_1..w_n, v_{n+1}..v_m)]
\end{array}$$

Proof Sketch : Using the conditions of Defn 6.7.2, we can prove the above by an induction on the dynamic semantics (see Fig. 2.7) over execution of the body of statically-inherited methods. □

Lemma 6.10.4 (**Soundness of Statically-Inherited Specifications).** *Consider a method* mn *from class A that has been successfully verified against its static specification* $\mathtt{preS_A} \ast\!\!\rightarrow \mathtt{postS_A}$*,*

and a subclass B that statically-inherits mn *with static specification* $\texttt{preS}_\texttt{B} \ast\!\!\rightarrow \texttt{postS}_\texttt{B}$. *Assuming that a specification subsumption relation of the form* $\texttt{preS}_\texttt{A} \ast\!\!\rightarrow \texttt{postS}_\texttt{A} <: \texttt{preS}_\texttt{B} \ast\!\!\rightarrow \texttt{postS}_\texttt{B}$ *holds, then* B.mn *is guaranteed to verify successfully against its specification* $\texttt{preS}_\texttt{B} \ast\!\!\rightarrow \texttt{postS}_\texttt{B}$.

Proof Sketch : Follows from Lemmas 6.10.3 and 6.10.1. □

We shall now show a result regarding behavioral subtyping.

Lemma 6.10.5 (**Soundness of Behavioral Subtyping).** *Consider a method* mn *from class* A *with dynamic specification* $\texttt{preD}_\texttt{A} \ast\!\!\rightarrow \texttt{postD}_\texttt{A}$ *and that*

$$\bigwedge \begin{array}{l} s, h_1 \models \texttt{preD}_\texttt{A} \\ h_1{=}h{+}[s(\texttt{o}){\mapsto}\texttt{A}(v^*)] \\ \langle s, h_1, \texttt{o.mn}(\texttt{p}^*)\rangle{\hookrightarrow}^*\langle s_3, h_3, v\rangle \\ s_3 + [\texttt{res}{\mapsto}v], h_3 \models \mathit{Post}(\texttt{postD}_\texttt{A}) \end{array}$$

If we assume a similar object from a subclass B *such that*

$$\bigwedge \begin{array}{l} s, h_2 \models \texttt{preD}_\texttt{A} \\ h_2{=}h{+}[s(\texttt{o}){\mapsto}\texttt{B}(v^*, w^*)] \end{array}$$

and we call the overriding method, then we obtain :

$$\bigwedge \begin{array}{l} \langle s, h_2, \texttt{o.mn}(\texttt{p}^*)\rangle{\hookrightarrow}^*\langle s_4, h_4, v\rangle \\ s_4 + [\texttt{res}{\mapsto}v], h_4 \models \mathit{Post}(\texttt{postD}_\texttt{A}) \end{array}$$

Proof Sketch : Follows from Defn 6.7.1 of the behavioral subtyping requirement and Lemma 6.10.1. □

Lastly, we prove the soundness of our verification system using preservation and progress lemmas.

Lemma 6.10.6 (**Preservation).** *If*

$$\vdash \{\Delta\}\ e\ \{\Delta_2\} \quad s, h \models \mathit{Post}(\Delta) \quad \langle s, h, e\rangle{\hookrightarrow}\langle s_1, h_1, e_1\rangle$$

Then there exists Δ_1 *such that* $s_1, h_1 \models \mathit{Post}(\Delta_1)$ *and* $\vdash \{\Delta_1\}\ e_1\ \{\Delta_2\}$.

Proof Sketch: By induction on e. □

Lemma 6.10.7 (**Progress).** *If* $\vdash \{\Delta\}\ e\ \{\Delta_1\}$, *and* $s, h \models \mathit{Post}(\Delta)$, *then either* e *is a value, or there exist* s_1, h_1, *and* e_1, *such that*

$$\langle s, h, e\rangle{\hookrightarrow}\langle s_1, h_1, e_1\rangle.$$

Proof Sketch: By induction on e. □

Theorem 6.10.8 (**Soundness of Verification).** *Consider a closed term e without free variables in which all methods have been successfully verified. Assuming that $\vdash \{\texttt{true}\}\ e\ \{\Delta\}$, then either $\langle [], [], e\rangle \hookrightarrow^* \langle [], h, v\rangle$ terminates with a value v such that the following $([\texttt{res}\mapsto v], h) \models \Delta$ holds, or it diverges $\langle [], [], e\rangle \not\hookrightarrow^*$.*

Proof Sketch: Follows from Lemma 6.10.6 and Lemma 6.10.7. □

```
class Cnt { int val;
 Cnt(int v) static true ⋆→ res::Cnt⟨v⟩
 {this.val:=v}
 void tick() static this::Cnt⟨v⟩ ⋆→ this::Cnt⟨v+1⟩;
    dynamic this::Cnt⟨v⟩$∧v≥0 ⋆→ this::Cnt⟨w⟩$∧v+1≤w≤v+2
 {this.val:=this.val+1}
 int get() static this::Cnt⟨v⟩ ⋆→ this::Cnt⟨v⟩∧res=v
    dynamic this::Cnt⟨v⟩$∧v≥0 ⋆→ this::Cnt⟨v⟩$
 {this.val}
 void set(int x) static this::Cnt⟨v⟩ ⋆→ this::Cnt⟨x⟩;
    dynamic this::Cnt⟨v⟩$∧x≥0 ⋆→ this::Cnt⟨x⟩$
 {this.val:=x}
}

class FastCnt extends Cnt {
 FastCnt(int v) static true ⋆→ res::FastCnt⟨v⟩
 {this.val:=v}
 void tick() static this::FastCnt⟨v⟩ ⋆→
    this::FastCnt⟨v+2⟩  {this.val:=this.val+2}
}

class PosCnt extends Cnt
    inv this.val≥0 {
 PosCnt(int v) static v≥0 ⋆→ res::PosCnt#I⟨v⟩
 {this.val:=v}
 void tick() static this::PosCnt#I⟨v⟩ ⋆→ this::PosCnt#I⟨v+1⟩
    dynamic this::PosCnt#I⟨v⟩$ ⋆→ this::PosCnt#I⟨v+1⟩$
 int get() static this::PosCnt#I⟨v⟩ ⋆→ this::PosCnt#I⟨v⟩∧res=v
 void set(int x) static this::PosCnt⟨v⟩∧x≥0 ⋆→ this::PosCnt#I⟨x⟩
    dynamic this::PosCnt⟨v⟩$∧x≥0 ⋆→ this::PosCnt#I⟨x⟩$
 {if x≥0 then this.val:=x else error()}
}

class TwoCnt extends Cnt { int bak;
 TwoCnt(int v, int b) static true ⋆→ res::TwoCnt⟨v, b⟩
 {this.val:=v; this.bak:=b}
 void set(int x) static this::TwoCnt⟨v, _⟩ ⋆→ this::TwoCnt⟨x, v⟩
 {this.bak:=this.val; this.val:=x}
 void switch(int x) static this::TwoCnt⟨v, b⟩ ⋆→ this::TwoCnt⟨b, v⟩
 {int i:=this.val; this.val:=this.bak; this.bak:=i}
}
```

Figure 6.3: Static and Dynamic Specifications given for `Cnt` and its Subes

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

In this thesis, we aimed at enhancing the specification apparatus in order to achieve more precise and concise specifications that also assist in improving the precision and efficiency of the verification process. More specifically, we proposed three new specification mechanisms:

- **Immutability annotations :** We show new application of immutability annotation, for verifying and reasoning of sequential programs. Our solution is tailored towards more flexible alias analysis, supports partial immutability, better cut-points preservation for modular analysis through immutable predicates within both pre and post-conditions. The new specification mechanism enables the design of a more concise and more precise verification/analysis system, with finer controls over accesses to resources (data structures).

  In order to promote immutability as part of the verification methodology, this thesis shows three main advantages of its use in a sequential setting through data from a set of small experiments:

  - more concise specifications

    (38% reduction in shape predicates, and 28% reduction in aliasing scenarios)

  - more precise analysis/verification

    (62% immutability annotations that increase precision by helping to preserve cut-points across method boundary)

  - capability for finer-grained immutability.

- **Structuring constructs :** Some existing theorem provers use tactics as a way to automate or semi-automate proofs, and verification systems can take advantage of them through lower-level proofs. However, for Hoare-style specification and verification, we have chosen to design a structured specification (rather than another tactic language) for the following reasons:

  - It can be provided at a higher-level that users can understand more easily, since it is closer to specification mechanism rather than the (harder) verification process.
  - It is more portable, as specification is tied to program codes, while tactic language tend to be prover-specific, requiring the invoked prover to understand the relevant commands. Our approach basically breaks down larger proofs into smaller (simpler) proofs that provers could more easily and more effectively handle, as confirmed by our experiments.

  The current thesis tackles two key problems of verification, namely better modularity and better completeness through a new form of structured specification. Our proposal has been formalized and implemented with a promising set of experimental results.

- **Static and dynamic specifications :** We present an enhanced approach to OO verification based on the co-existence of both static and dynamic specifications, together with a principle that each static specification be a subtype of its corresponding dynamic specification. Our approach attempts to track the actual type of each object, where possible, to allow static specifications to be preferably used. We have designed a new object format that allows each object to assume the form of its superclass via lossless casting. Another useful feature of our proposal is a new specification subsumption relation for pre/post specifications that is novel in using the residual heap state from precondition checking to assist in postcondition checking.

  By making each static specification be a subtype of its dynamic specification, we can limit code verification to only static specifications.

7.1 Future Work

We conclude this thesis by outlining a series of possible future works.

7.1.1 Declaration-Site vs. Use-Site Immutability Annotations

In Chapter 4, we made use of predicate (or points-to fact) instances that have been annotated for immutability which corresponds to use-site annotation. It is also possible to annotate the entire data and predicate definition for immutability, so that all instances of the particular points-to fact or predicate are automatically marked as immutable. This corresponds to declaration-site

annotation. This immutability annotation scheme may be suitable if a functional programming discipline is to be adopted. Two examples of declaration-site annotation are illustrated below:

$$\texttt{data node@I} \{ \texttt{int val; node next}\}$$

$$\begin{aligned}\texttt{ll}\langle\texttt{n}\rangle\texttt{@I} \equiv\ & \texttt{root=null} \wedge \texttt{n=0} \\ \vee\ & \texttt{root::node}\langle\_, \texttt{q}\rangle * \texttt{q::ll}\langle\texttt{q}, \texttt{n-1}\rangle \\ & \texttt{inv n}\geq 0;\end{aligned}$$

Note the presence of @I annotation immediately after the name of the data or predicate definitions. This signifies that all instances of this data or predicate definition will always be marked as being immutable.

7.1.2 Selective Immutability

Throughout Chapter 4, the immutability annotations can scope over an entire data node/shape predicate. We are interested in supporting more selective immutability through the use of field annotations at either the point of declaration or the point of use. For example, at the point of a specific node declaration, we could mark its next field as being immutable, as shown below:

$$\texttt{data node} \{ \texttt{int val; node@I next}\}$$

This means that all next field instances of node type are always immutable after each node instances have been constructed. This manner of marking just the pointer field as immutable has the effect of always preserving the original shape, but not the contents of the data structures. For this example, the val field remains mutable, whereas the shape formed through pointer fields has been frozen. Such partial immutability can be used to enforce shape immutability, whereby the shape of linked-structures (as well as properties that are derived from such shapes) are being marked as immutable. Once a linked structure has been constructed, its existing shape remains unchanged but its data contents may still be modified. A corresponding predicate definition for shape immutable is given below:

$$\begin{aligned}\texttt{ll}\langle\texttt{n}\rangle \equiv\ & \texttt{root=null} \wedge \texttt{n=0} \\ \vee\ & \texttt{root} \mapsto \texttt{node}\langle\_, \texttt{q@I}\rangle * \texttt{q::ll}\langle\texttt{n-1}\rangle \\ & \texttt{inv n}\geq 0;\end{aligned}$$

where the next field of nodes captured by any ll predicate is not allowed to be modified.

7.1.3 Inferring Immutability Enhanced Specifications

As a future direction for the work presented in Chapter 4, we intend to offer the user the possibility of automatically inferring immutability enhanced specifications. This capability would be particularly useful for legacy code. Given a specification without immutability annotations, this inference process will consist of two steps:

- inferring the immutability annotations for each data node and shape predicate in the specification.
- relaxing the specifications such that they allow heap sharing whenever aliasing information is not critical.

7.1.4 Inferring Structured Specifications

With respect to the work on structured specifications presented in Chapter 5, one future work is heuristically inferring structured specifications from unstructured counterparts. Though the inferred specifications may not be as good as those provided by expert users, they can nevertheless be used to handle most of the straightforward cases for legacy code, while leaving the harder unverified examples to be handled by users.

.1 Proofs

Theorem. *4.7.1(Preservation)*

If

$$\vdash \{\Delta\}\ e\ \{S_2\} \quad S_2{\neq}\{\} \quad s,h \models \mathit{Post}(\Delta) \quad \langle s,h,e\rangle{\hookrightarrow}\langle s_1,h_1,e_1\rangle$$

Then there exists $S_1{\neq}\{\}$*, such that* $\forall\Delta_1 \in S_1 \cdot s_1,h_1 \models \mathit{Post}(\Delta_1)$ *and* $\vdash \{\Delta_1\}\ e_1\ \{S_3\} \quad S_3{\subseteq}S_2$.

Proof: By structural induction on e.

- Case $v := e_3$. There are two cases according to the dynamic semantics:

  - e_3 is not a value. Then, from [FV–ASSIGN] in Sec 2.3, if $\vdash \{\Delta\}e_3\{S_3\}$, then $S_2{=}\exists\texttt{res}\cdot S_3 \wedge_{\{v\}} v'{=}\texttt{res}$. From dynamic rules, if $\langle s,h,e_3\rangle{\hookrightarrow}\langle s_4,h_4,e_4\rangle$, then $\langle s,h,v{:=}e_3\rangle{\hookrightarrow}\langle s_4,h_4,v{:=}e_4\rangle$.

    From $\langle s,h,e_3\rangle{\hookrightarrow}\langle s_4,h_4,e_4\rangle$ and $\vdash \{\Delta\}e_3\{S_3\}$, by induction hypothesis, we have that there exists S_4 such that:

    $$\forall\Delta_4 \in S_4 \cdot s_4,h_4 \models Post(\Delta_4)$$

    and

    $$\vdash \{\Delta_4\}e_1\{S_3\}.$$

    Then, from verification rule [FV–ASSIGN] in Sec 2.3:

    $$\forall\Delta_4 \in S_4 \cdot \vdash \{\Delta_4\}v{:=}e_1\{S_2\}.$$

    The proof is complete for $h_1{=}h_4$, $s_1{=}s_4$, and $S_1{=}S_4$.

  - e_3 is a value. Let:

    * $e_1{=}()$,
    * $S_1{=}S_2{=}\{(\Delta \wedge v'{=}e_3)\}$,
    * $h_1{=}h_2{=}h$, and $s_1{=}s_2{=}s[v \mapsto e_3]$.

    Straightforward.

- Case $v_1.f := v_2$. Take:

  - $e_1{=}()$,
  - $S_1{=}S_2$,

– $h_1{=}h_2{=}h[s(v_1){\mapsto_M} r]$, where $r{=}h(s(v_1))[f{\mapsto}s(v_2)]$,

– $s_1{=}s_2{=}s$.

Hence, it is straightforward that:

$$\forall \Delta_1 \in S_1{\cdot}\{\Delta_1\}\ e_1\ \{S_2\}.$$

Now, we must show that:

$$\forall \Delta_1 \in S_1{\cdot}s_1, h_1 \models \mathit{Post}(\Delta_1).$$

From the rule [FV–FIELD–UPDATE] in Sec 4.6, we have $\Delta \vdash^{\kappa}_{V} \mathtt{v}'_1{::}\mathtt{c}\langle \mathtt{w_1}, .., \mathtt{w_f}, .., \mathtt{w_n}\rangle @M * S_3$, $S_3{\neq}\emptyset$ and $S_1 = \exists w_1..w_n{\cdot}(S_3 * [v'_2/w_f]\mathtt{v}'_1{::}\mathtt{c}\langle \mathtt{w_1}, .., \mathtt{w_n}\rangle)$ (*).

From the hypothesis, we know that $s, h \models \mathit{Post}(\Delta)$ (**).

From (*) and (**), by Theorem 4.7.4, $\forall\ \Delta_3 \in S_3.s, h \models \mathit{Post}(\Delta_3)$.

The conclusion follows from $s{=}s_1$ and $h_1{=}h[s(v_1){\mapsto}\ @Mr]$, where $r{=}h(s(v_1))[f{\mapsto}s(v_2)]$.

- Case $\mathtt{new}\ c(v^*)$. From verification rule [FV–NEW] in Sec 2.3, we have $\vdash \{\Delta\}\mathtt{new}\ c(v^*)\{S_2\}$, where $S_2 = \{\Delta{*}\mathtt{res}{::}c\langle v'_1, .., v'_n\rangle @M\}$. Let $S_1 = S_2$. From the dynamic semantics, we have:

$$\langle s, h, \mathtt{new}\ c(v^*)\rangle {\hookrightarrow} \langle s, h{+}[\iota{\mapsto}\ @Ir], \iota\rangle,$$

where $\iota \notin \mathit{dom}(h)$. From $s, h \models \mathit{Post}(\Delta)$, we have:

$$\forall \Delta_1 \in S_1.s, h{+}[\iota \mapsto_I r] \models \mathit{Post}(\Delta_1).$$

Moreover, $\vdash \{S_1\}\iota\{S_2\}$.

- Case $e_1; e_2$. We consider the case where e_1 is not a value (otherwise it is straightforward). From the dynamic semantics, we have $\langle s, h, e_1\rangle {\hookrightarrow} \langle s_1, h_1, e_3\rangle$. From verification rule [FV–SEQ] in Sec 2.3, we have $\vdash \{\Delta\}e_1\{S_3\}$. By induction hypothesis, there exists S_1 s.t. $\forall \Delta_1 \in S_1.s_1, h_1 \models \mathit{Post}(\Delta_1)$ and $\vdash \{S_1\}e_3\{S_3\}$. By rule [FV–SEQ] in Sec 2.3, we have $\vdash \{S_1\}e_3; e_2\{S_2\}$.

- Case $\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$. There are two possibilities in the dynamic semantics:

  – $s(v){=}\mathtt{true}$. We have $\langle s, h, \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle {\hookrightarrow} \langle s, h, e_1\rangle$. Let $S_1 = \{\Delta{\wedge}v'\}$. It is obvious that $\forall\ \Delta_1 \in S_1.s, h \models \mathit{Post}(\Delta_1)$. By the rule [FV–IF] in Sec 2.3, we

have $\vdash \{\Delta \wedge v'\}\, e_1\, \{S^1\}$. From the same rule $S_2 = S^1 \vee S^2$, where $\vdash \{\Delta \wedge \neg v'\}\, e_2\, \{S^2\}$. By weakening the postcondition (rule FV–POST–WEAKENING in Sec 2.7), we get $\vdash \{\Delta \wedge \neg v'\}\, e_1\, \{S_2\}$, which is $\vdash \{S_1\}\, e_1\, \{S_2\}$.

- $s(v) = \texttt{false}$. Analogous to the above.

- Case $t\ v;\ e$. Let $S_1 = \{\Delta\}$. From the dynamic rules in fig 4.9, we have $e_1 = \texttt{ret}(v, e)$, $s_1 = [v \mapsto \bot] + s$, $h_1 = h$. We conclude immediately from the assumption and the rules [FV–LOCAL] and [FV–RET] in Sec 2.3 and Sec 2.7, respectively.

- Case $mn(v_{1..n})$. From the dynamic rule for method call in Fig 4.9:

$$s_1 = [w_j \mapsto s(v_j)]_{j=m}^{n} + s,$$

and

$$h_1 = h$$

From rule [FV–CALL–IMM] in Sec 4.6, we know:

$$\Delta \vdash \rho\, \Phi^i_{pr} *_i S_i.$$

Take:

$$S_1 = \rho\, \Phi^i_{pr} *_i S_i.$$

From the hypothesis and Theorem 4.7.4, we have:

$$\forall \Delta_1 \in S_1 . s_1, h_1 \models \mathit{Post}(\Delta_1).$$

From rule [FV–CALL–IMM], we have:

$$S_2 = \bigcup_{i=1}^{p} S_i * \Phi^i_{po},$$

while from the dynamic semantics we have:

$$e_1 = \texttt{ret}(\{w_j\}_{j=m}^{n}, [v_j / w_j]_{j=1}^{m-1} e).$$

From rule [FV–METH–IMM] in Sec 4.6, we have

$$\vdash \{\rho\, \Phi^i_{pr} * S_i\}\, e_1\, \{S_i * \Phi^i_{po}\},$$

which concludes.

- Case $\texttt{ret}(v^*, e)$. There are two cases:
  - e is a value k. Let $S_1 = \{\exists v'^* \cdot \Delta\}$. It concludes immediately.
  - e is not a value. $\langle s, h, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(v^*, e_1)\rangle$. By [FV–RET] and induction hypothesis, there exists S_1 s.t. $\forall \Delta_1 \in S_1 . s_1, h_1 \models \mathit{Post}(\Delta_1)$ and $\vdash \{S_1\}\, e_1\, \{S_3\}$, and $S_2 = \exists v'^* \cdot S_3$. By rule [FV–RET] again, we have $\vdash \{S_1\}\, \texttt{ret}(v^*, e_1)\, \{S_2\}$.
- Case $\texttt{null} \mid k \mid v \mid v.f \mid \texttt{assert}\ \Phi$. Straightforward.

Theorem. *4.7.2(Progress)*

If

$$\vdash \{\Delta\}\ e\ \{S_1\} \quad S_1 \neq \{\} \quad s, h \models \mathit{Post}(\Delta)$$

then either e is a value, or there exist s_1, h_1, and e_1, such that $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$.

Proof: By structural induction on e.

- Case $v := e$. There are two cases:
  - e is a value k. We conclude.
  - e is not a value. By [FV–ASSIGN] in Fig 2.3, we have $\vdash \{\Delta\}\, e\, \{S_2\}$. By induction hypothesis, there exist s_1, h_1, e_1, such that $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$. We conclude immediately from the dynamic semantics.
- Case $v_1.f := v_2$. for fresh $v_1, .., v_n$. Follows from:
  - $e_1 = ()$,
  - $s_1 = s$,
  - $h_1 = h[s(v_1) \mapsto_M r]$, where $r = h(s(v_1))[f \mapsto s(v_2)]$.
- Case $\texttt{new}\ c(v_{1\ldots n})$. Let ι be a fresh location, r denotes the object value

$$c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)].$$

  Take:
  - $s_1 = s$,
  - $h_1 = h + [\iota \mapsto_I r]$,

– $e_1 = \iota$.

We conclude.

- Case $e_1; e_2$. If e_1 is a value $()$, we conclude immediately by taking $s_1 = s, h_1 = h$. Otherwise, by induction hypothesis, there exist s_1, h_1, e_3 s.t. $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. We then have $\langle s, h, e_1; e_2\rangle \hookrightarrow \langle s_1, h_1, e_3; e_2\rangle$ from the dynamic semantics.

- Case $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. It concludes immediately from a case analysis (based on value of v) and the induction hypothesis.

- Case $t\ v;\ e$. Let:

  – $s_1 = [v \mapsto \bot] + s$,

  – $h_1 = h$,

  – $e_1 = \texttt{ret}(v, e)$.

  We conclude immediately.

- Case $mn(v_{1..n})$. Suppose $v_1, .., v_m$ are pass-by-reference, while others are not. Take:

  – $s_1 = [w_j \mapsto s(v_j)]_{j=m}^{n} + s$,

  – $h_1 = h$,

  – $e_1 = \texttt{ret}(\{w_j\}_{j=m}^{n}, [v_j/w_j]_{j=1}^{m-1} e)$, where w_j are from method specification

  $t_0\ mn((\texttt{ref}\ t_j\ w_j)_{j=1}^{m-1}, (t_j\ w_j)_{j=m}^{n})\ \texttt{where}\ \{\texttt{requires}\ \Phi_{pr}^{i}\ \texttt{ensures}\ \Phi_{po}^{i}\}_{i=1}^{p}\ \{e\}$.

  From hypothesis, we have:

$$s, h \models \mathit{Post}(\Delta).$$

  From [FV–CALL–IMM] rule in Sec 4.6, we know:

$$\Delta \vdash \rho \Phi_{pr}^{i} *_i S_i.$$

  By Theorem 4.7.4, we have:

$$s, h \models \Phi_{pr}^{i}.$$

  From the dynamic semantics, [FV–METH–IMM] rule in Sec 4.6 and Theorem 4.7.4, we have:

$$s_1, h_1 \models \Phi_{po}^{i}.$$

We conclude by the dynamic semantics.

- Case $\texttt{ret}(v^*, e)$. If e is a value k, let:
  - $s_1 = s - \{v^*\}$,
  - $h_1 = h$,
  - $e_1 = k$.

  We conclude. Otherwise, by induction hypothesis, there exist s_1, h_1, e_1 such that:

  $$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle.$$

  We then have:

  $$\langle s, h, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(v^*, e_1)\rangle.$$

- Case $\texttt{null} \mid k \mid v \mid v.f \mid \texttt{assert}\ \Phi$. Straightforward.

Theorem. *4.7.3(Safety)*
Consider a closed term e without free variables in which all methods have been successfully verified. Assuming unlimited stack/heap spaces and that $\vdash \{\texttt{true}\}\ e\ \{\Delta\}$, then either $\langle [], [], e\rangle \hookrightarrow^* \langle [], h, v\rangle$ terminates with a value v that is subsumed by the postcondition Δ, or it diverges $\langle [], [], e\rangle \not\hookrightarrow^*$.

Before we present the proof for Theorem 4.7.3, we state and prove the following lemma:

Lemma .1.1. *For any s, h, e, if $\langle s, h, e\rangle \hookrightarrow^* \langle \hat{s}, \hat{h}, \nu\rangle$ for some $\hat{s}, \hat{h}, \nu$, where ν is a value, and all free variables of e are already in the domain of the stack s, i.e. free-vars$(e) \subseteq$dom(s), then dom$(\hat{s})$ = dom(s).*

Proof: By structural induction over e.

Base cases: e is

- $\texttt{null}$
- k
- v
- $v.f$
- $v.f = v_1$

- `assert` Φ

The conclusion is obvious as the stack remains unchanged during the evaluation of e.

Inductive cases:

- e is $v := e_1$. By the operational semantics, we know that

$$\langle s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, \nu_1\rangle$$

for some s_1, h_1, ν_1, and

$$\langle s_1, h_1, v := \nu_1\rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu\rangle.$$

Note that

$$\textit{free-vars}(e_1) \subseteq \textit{free-vars}(e) \subseteq \textit{dom}(s),$$

by induction hypothesis, we have $\textit{dom}(s_1) = \textit{dom}(s)$. The conclusion follows since

$$\textit{dom}(\hat{s}) = \textit{dom}(s_1).$$

- e is $e_1; e_2$. By the operational semantics, there are s_1, h_1 such that

$$\langle s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, ()\rangle,$$
$$\langle s_1, h_1, (); e_2\rangle \hookrightarrow \langle s_1, h_1, e_2\rangle,$$
$$\langle s_1, h_1, e_2\rangle \hookrightarrow^* \langle \hat{s}, \hat{h}, \nu\rangle.$$

Note that, for $i{=}1, 2$, we have

$$\textit{free-vars}(e_i) \subseteq \textit{free-vars}(e) \subseteq \textit{dom}(s).$$

By induction hypothesis, we have

$$\textit{dom}(\hat{s}) = \textit{dom}(s_1) = \textit{dom}(s).$$

- e is $t\ v; e_1$. By the operational semantics, we have

$$\langle s, h, e\rangle \hookrightarrow \langle [v \mapsto \_] + s, h, \texttt{ret}(v, e_1)\rangle,$$

and

$$\langle [v \mapsto \_] + s, h, e_1\rangle \hookrightarrow^* \langle s_1, h_1, \nu\rangle$$

for some s_1, h_1, and

$$\langle s_1, h_1, \texttt{ret}(v, \nu)\rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu\rangle,$$

where $\hat{s} = s_1 - \{v\}$. Note that

$$\textit{free-vars}(e_1) \subseteq \textit{dom}([v \mapsto \_] + s),$$

by induction hypothesis, we have

$$\textit{dom}(s_1) = \textit{dom}([v \mapsto \_] + s).$$

So $\textit{dom}(\hat{s}) = \textit{dom}(s_1) - \{v\} = \textit{dom}([v \mapsto \_] + s) - \{v\} = \textit{dom}(s)$.

- e is $mn(u^*; v^*)$, where v^* are arguments for call-by-value parameters w^*. By the operational semantics, we have

$$(1) \langle s, h, e \rangle \hookrightarrow \langle [w^* \mapsto v^*] + s, h, \texttt{ret}(w^*, e_{mn}) \rangle,$$

where e_{mn} is the body of the method mn, and

$$(2) \langle [w^* \mapsto v^*] + s, h, e_{mn} \rangle \hookrightarrow^* \langle s_1, h_1, \nu \rangle$$

for some s_1, h_1, and

$$(3) \langle s_1, h_1, \texttt{ret}(w^*, \nu) \rangle \hookrightarrow \langle \hat{s}, \hat{h}, \nu \rangle,$$

where $\hat{s} = s_1 - \{w^*\}$. Note also that we have

$$\textit{free-vars}(e_{mn}) \subseteq \textit{dom}([w^* \mapsto v^*] + s),$$

by induction hypothesis, we have

$$\textit{dom}(s_1) = \textit{dom}([w^* \mapsto v^*] + s).$$

So $\textit{dom}(\hat{s}) = \textit{dom}(s_1) - \{w^*\} = \textit{dom}(s)$. □

Proof of Theorem 4.7.3: If the evaluation of `e` does not diverge (is not infinite), it will terminate in a finite number of steps (say n): $\langle [], [], e \rangle \hookrightarrow \langle s_1, h_1, e_1 \rangle \hookrightarrow \cdots \hookrightarrow \langle s_n, h_n, e_n \rangle$, and there are no further reductions possible. By Theorem 4.7.1, there exist $\Delta_1, .., \Delta_n$ such that, $s_i, h_i \models \textit{Post}(\Delta_i)$, and $\vdash \{\Delta_i\}\ e_i\ \{\Delta\}$.

By Theorem 4.7.2, The final result e_n must be some value v (or it will make another reduction). The conclusion that the stack s_n in the final state is empty is drawn from Lemma .1.1 in the above. □

Theorem. *4.7.4(Soundness of Heap Entailment)*

If entailment check $\Delta_1 \vdash \Delta_2 *_i S$ *succeeds, we have: for all* $s, h,$ *and* $\Delta \in S$ *, if* $s, h \models \Delta_1$ *then* $s, h \models \Delta_2 *_i \Delta$.

Proof: Note that the entailment rules [ENT−MATCH−MUT] and [ENT−MATCH−IMM] in Fig. 4.5 denote a match of two nodes/shape predicates between the antecedent and the consequent. We apply induction on the number of such matches for each path in the entailment search tree for $\mathcal{E}_0$.

Base case.

The entailment search succeeds requiring no matches, meaning that the consequent consists of only a pure formula. It can only be the case where rule [ENT−EMP] in Fig. 2.5 is applied. It is straightforward to conclude.

Inductive case.

Suppose a sequence of transitions $\mathcal{E}_0 \rightarrow \cdots \rightarrow \mathcal{E}_n$ where no match transitions (due to rules [ENT−MATCH−MUT] and [ENT−MATCH−IMM]) are involved in this sequence but $\mathcal{E}_n$ will perform a match transition. These transitions can only be generated by the following rules: [ENT−UNFOLD], [ENT−FOLD], [ENT−LHS−OR], [ENT−RHS−OR], [ENT−LHS−EX], [ENT−RHS−EX], [ENT−SPLIT−RHS$_1$], [ENT−SPLIT−RHS$_2$], [ENT−SPLIT−LHS$_1$], [ENT−SPLIT−LHS$_2$]. A case analysis on these rules shows that the following properties hold:

$$\begin{array}{l} s, h \models \mathit{LHS}(\mathcal{E}_i) \implies s, h \models \mathit{LHS}(\mathcal{E}_{i+1}) \\ \quad (LHS\ is\ always\ weakened\ by\ the\ entailment\ rules) \\ s, h \models \mathit{RHS}(\mathcal{E}_{i+1}) \implies s, h \models \mathit{RHS}(\mathcal{E}_i) \\ \quad (RHS\ is\ always\ strengthened\ by\ the\ entailment\ rules) \end{array} \qquad (\dagger)$$

Now suppose a match between the antecedent and the consequent. There can be two situations, which we discuss below:

- Suppose the match node for $\mathcal{E}_n \equiv \Delta_a \vdash^{\kappa}_{V} \Delta_c *_i S_r$ is $p{::}c\langle v^* \rangle @I$, and $\mathcal{E}_n$ becomes:

$$\Delta'_a \vdash^{\kappa * p::c\langle v^* \rangle}_{V} \Delta'_c *_i S'_r$$

  where:

  - $\Delta_a = SH(\Delta'_a, (p{::}c\langle v^* \rangle @I, h_a))$, for some h_a and Δ'_a,
  - $\Delta_c = p{::}c\langle v^* \rangle * \Delta'_c$, for some Δ'_c,

– $\forall \Delta_r \in S_r . \exists \Delta'_r \in S'_r . \Delta_r = SH(\Delta'_r, (p::c\langle v^* \rangle @I, h_r))$, for some h_r and S'_r,

By induction, we have

$$\forall s, h \cdot s, h \models \Delta'_a \implies s, h \models \Delta'_c *_i S'_r \qquad (\ddagger)$$

Suppose $s, h \models \Delta_a$, then there exist h_0, h_1, h_2, such that:

– $h_0 \perp h_1 \perp h_3$,

– $h = h_0 \cdot h_1 \cdot h_3$,

– $s, h_0 \cdot h_3 \models p::c\langle v^* \rangle @I$,

– $s, h_1 \cdot h_3 \models \Delta'_a$.

From $(\ddagger)$, we have:

$$s, h_1 \cdot h_3 \models \Delta'_c *_i S'_r,$$

which yields:

$$s, h \models \Delta_c * S_r.$$

We then conclude from $(\dagger)$.

- Suppose the match node for $\mathcal{E}_n \equiv \Delta_a \vdash^{\kappa}_{V} \Delta_c *_i S_r$ is $p::c\langle v^* \rangle$@M, and $\mathcal{E}_n$ becomes:

$$\Delta'_a \vdash^{\kappa * p::c\langle v^* \rangle}_{V} \Delta'_c *_i S_r$$

for some Δ'_a, Δ'_c. By induction, we have:

$$\forall s, h \cdot s, h \models \Delta'_a \implies s, h \models \Delta'_c *_i S_r \qquad (\ddagger)$$

From the entailment process, we have:

$$\Delta_a = p::c\langle v^* \rangle * \Delta'_a,$$

and

$$\Delta_c = p::c\langle v^* \rangle * \Delta'_c.$$

Suppose $s, h \models \Delta_a$, then there exist h_0, h_1, such that $h = h_0 * h_1$, $s, h_0 \models p::c\langle v^* \rangle$, and $s, h_1 \models \Delta'_a$. From $(\ddagger)$, we have:

$$s, h_1 \models \Delta'_c *_i S_r,$$

which immediately yields:

$$s, h \models \Delta_c *_i S_r.$$

We then conclude from (†).

BIBLIOGRAPHY

[1] AHRENDT, W., BECKERT, B., HÄHNLE, R., and SCHMITT, P. H., "Key: A formal method for object-oriented systems," in *FMOODS*, pp. 32–43, 2007.

[2] AMERICA, P., "Designing an object-oriented programming language with behavioural subtyping," in *the REX School/Worshop on Foundations of Object-Oriented Languages*, pp. 60–90, 1991.

[3] BACON, D. F. and SWEENEY, P. F., "Fast static analysis of C++ virtual function calls.," in *SIGPLAN Object-Oriented Programming Systems, Languages and Applications*, pp. 324–341, 1996.

[4] BARNET, M., DELINE, R., FAHNDRICH, M., LEINO, K., and SCHULTE, W., "Verification of object-oriented programs with invariants," *Journal of Object Technology*, vol. 3, no. 6, pp. 27–56, 2004.

[5] BARNETT, M., LEINO, K. R. M., and SCHULTE, W., "The Spec# programming system: An overview," in *Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[6] BARNETT, M., CHANG, B.-Y. E., DELINE, R., 0002, B. J., and LEINO, K. R. M., "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, pp. 364–387, 2005.

[7] BERDINE, J., CALCAGNO, C., and O'HEARN, P. W., "Symbolic Execution with Separation Logic," in *APLAS*, Springer-Verlag, Nov. 2005.

[8] BERDINE, J., CALCAGNO, C., and O'HEARN, P. W., "Smallfoot: Modular automatic assertion checking with separation logic," in *4th International Symposium on Formal Methods for Components and Objects (FMCO05)*, vol. 4111 of *Springer LNCS*, 2006.

[9] BERDINE, J., COOK, B., and ISHTIAQ, S., "Slayer: Memory safety for systems-level code," in *CAV*, pp. 178–183, 2011.

[10] BINGHAM, J. and RAKAMARIC, Z., "A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs," in *Intl Conf. on Verification, Model Checking and Abstract Interpretation*, Springer LNCS 3855, (Charleston, U.S.A), pp. 207–221, Jan. 2006.

[11] BIRKA, A. and ERNST, M. D., "A practical type system and language for reference immutability," in *OOPSLA* (VLISSIDES, J. M. and SCHMIDT, D. C., eds.), pp. 35–49, ACM, 2004.

[12] BORNAT, R., CALCAGNO, C., O'HEARN, P. W., and PARKINSON, M. J., "Permission accounting in separation logic," in *POPL*, pp. 259–270, 2005.

[13] BOTINCAN, M., PARKINSON, M. J., and SCHULTE, W., "Separation logic verification of c programs with an smt solver," *Electr. Notes Theor. Comput. Sci.*, vol. 254, pp. 5–23, 2009.

[14] BROCK, B., KAUFMANN, M., and MOORE, J. S., "ACL2 Theorems About Commercial Microprocessors," in *FMCAD*, pp. 275–293, 1996.

[15] BROTHERSTON, J., BORNAT, R., and CALCAGNO, C., "Cyclic proofs of program termination in separation logic," in *POPL*, pp. 101–112, 2008.

[16] BRYANT, R. E., "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, 1986.

[17] BURDY, L., CHEON, Y., COK, D. R., ERNST, M. D., KINIRY, J. R., LEAVENS, G. T., LEINO, K. R. M., and POLL, E., "An overview of JML tools and applications," *Software Tools for Technology Transfer*, 2005.

[18] CALCAGNO, C., DISTEFANO, D., O'HEARN, P. W., and YANG, H., "Compositional shape analysis by means of bi-abduction," in *POPL*, pp. 289–300, 2009.

[19] CHANG, B.-Y. E. and RIVAL, X., "Relational inductive shape analysis," in *POPL*, pp. 247–260, 2008.

[20] CHEN, C. and XI, H., "Combining Programming with Theorem Proving," in *ACM SIGPLAN ICFP*, (Tallinn, Estonia), Sept. 2005.

[21] CHIN, W.-N., DAVID, C., NGUYEN, H. H., and QIN, S., "Multiple pre/post specifications for heap-manipulating methods," in *HASE*, 2007.

[22] CHIN, W.-N., DAVID, C., NGUYEN, H. H., and QIN, S., "Enhancing modular oo verification with separation logic," in *POPL*, 2008.

[23] CHIN, W. and KHOO, S., "Calculating sized types," in *ACM SIGPLAN PEPM*, (Boston, United States), pp. 62–72, Jan. 2000.

[24] CHIN, W., KHOO, S., QIN, S., POPEEA, C., and NGUYEN, H., "Verifying Safety Policies with Size Properties and Alias Controls," in *IEEE/ACM Intl. Conf. on Software Engineering*, (St. Louis, Missouri), May 2005.

[25] COK, D. R., "Reasoning with specifications containing method calls and model fields," *Journal of Object Technology*, vol. 4, no. 8, pp. 77–103, 2005.

[26] COK, D. R. and KINIRY, J., "ESC/Java2: Uniting ESC/Java and JML.," in *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 108–128, 2004.

[27] COK, D. R. and KINIRY, J. R., "ESC/Java2: Uniting ESC/Java and JML," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362 of *Lecture Notes in Computer Science*, pp. 108–128, Springer, 2004.

[28] DAVID, C. and CHIN, W.-N., "Immutable specifications for more concise and precise verification," in *SPLASH/OOPSLA*, 2011.

[29] DAVID HAREL, D. K. and TIURYN, J., *Dynamic Logic*. MIT Press, 2000.

[30] DE MOURA, L. M. and BJØRNER, N., "Efficient E-Matching for SMT Solvers," in *International Conference on Automated Deduction*, vol. 4603 of *Lecture Notes in Computer Science*, pp. 183–198, Springer, 2007.

[31] DETLEFS, D., NELSON, G., and SAXE, J. B., "Simplify: A theorem prover for program checking," Tech. Rep. HPL-2003-148, HP Laboratories Palo Alto, 2003.

[32] DHARA, K. K. and LEAVENS, G. T., "Forcing behavioral subtyping through specification inheritance," in *IEEE/ACM Intl. Conf. on Software Engineering*, pp. 258–267, 1996.

[33] DISTEFANO, D., O'HEARN, P. W., and YANG, H., "A Local Shape Analysis based on Separation Logic," in *TACAS*, Springer-Verlag, Mar. 2006.

[34] DISTEFANO, D., DODDS, M., and PARKINSON, M. J., "How to verify java program with jstar: a tutorial." University of Cambridge, UK, 2011.

[35] DISTEFANO, D. and PARKINSON, M. J., "jstar: towards practical verification for java," in *OOPSLA*, pp. 213–226, 2008.

[36] DOCKINS, R., HOBOR, A., and APPEL, A. W., "A fresh look at separation algebras and share accounting," in *APLAS*, pp. 161–177, 2009.

[37] ENGEL, C., ROTH, A., SCHMITT, P. H., and WEIS, B., "Verification of modifies clauses in dynamic logic with non-rigid functions," Tech. Rep. 2009-9, Department of Computer Science, University of Karlsruhe, 2009.

[38] ERNST, M. D., "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27, 2003.

[39] FILLIÂTRE, J.-C. and MARCHÉ, C., "The why/krakatoa/caduceus platform for deductive program verification," in *CAV*, pp. 173–177, 2007.

[40] FINDLER, R. B. and FELLEISEN, M., "Contract soundness for object-oriented languages," in *SIGPLAN Object-Oriented Programming Systems, Languages and Applications*, pp. 1–15, 2001.

[41] FINDLER, R. B., LATENDRESSE, M., and FELLEISEN, M., "Behavioral contracts and behavioral subtyping.," in *ESEC/SIGSOFT Foundations of Software Engr.*, 2001.

[42] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., and STATA, R., "Extended Static Checking for Java," in *ACM PLDI*, June 2002.

[43] FLOYD, R. W., "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.

[44] GHERGHINA, C., DAVID, C., QIN, S., and CHIN, W.-N., "Structured specifications for better verification of heap-manipulating programs," in *FM*, 2011.

[45] GOTSMAN, A., BERDINE, J., and COOK, B., "Interprocedural Shape Analysis with Separated Heap Abstractions," in *SAS*, Springer LNCS, (Seoul, Korea), August 2006.

[46] GOTSMAN, A., BERDINE, J., COOK, B., and SAGIV, M., "Thread-modular shape analysis," in *PLDI*, pp. 266–277, 2007.

[47] GOTSMAN, A., COOK, B., PARKINSON, M. J., and VAFEIADIS, V., "Proving that non-blocking algorithms don't block," in *POPL*, pp. 16–28, 2009.

[48] GUTTAG, J. V. and HORNING, J. J., eds., *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[49] HAACK, C., POLL, E., SCHÄFER, J., and SCHUBERT, A., "Immutable objects for a java-like language," in *ESOP*, pp. 347–362, 2007.

[50] HARWOOD, W., CAVALCANTI, A., and WOODCOCK, J., "A Theory of Pointers for the UTP," in *International Colloquium on Theoretical Aspects of Computing*, vol. 5160 of *Lecture Notes in Computer Science*, pp. 141–155, Springer, 2008.

[51] HATCLIFF, J., DENG, X., DWYER, M. B., JUNG, G., and RANGANATH, V. P., "Cadena: An integrated development, analysis, and verification environment for component-based systems.," in *IEEE/ACM Intl. Conf. on Software Engineering*, 2003.

[52] HAWKINS, P., AIKEN, A., FISHER, K., RINARD, M. C., and SAGIV, M., "Data structure fusion," in *APLAS* (UEDA, K., ed.), vol. 6461 of *Lecture Notes in Computer Science*, pp. 204–221, Springer, 2010.

[53] HOARE, C. A. R., "An axiomatic basis for computer programming," *COMMUNICATIONS OF THE ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[54] HOARE, T., "Verified software: Theories, tools, experiments," in *International Conference on Engineering of Complex Computer Systems*, 2008.

[55] HUGHES, J., PARETO, L., and SABRY, A., "Proving the correctness of reactive systems using sized types," in *ACM POPL*, pp. 410–423, ACM Press, Jan. 1996.

[56] ISHTIAQ, S. and O'HEARN, P. W., "BI as an Assertion Language for Mutable Data Structures," in *ACM POPL*, (London), Jan. 2001.

[57] JACOBS, B., SMANS, J., and PIESSENS, F., "A Quick Tour of the VeriFast Program Verifier," in *APLAS*, pp. 304–311, 2010.

[58] JACOBS, B., SMANS, J., and PIESSENS, F., "Verification of unloadable modules," in *FM*, pp. 402–416, 2011.

[59] JIA, L. and WALKER, D., "ILC: A foundation for automated reasoning about pointer programs," in *15th ESOP*, Mar. 2006.

[60] JONKERS, H. B. M., "Upgrading the pre- and postcondition technique," in *VDM*, (London, UK), pp. 428–456, Springer-Verlag, 1991.

[61] KASSIOS, I. T., "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *International Symposium on Formal Methods*, vol. 4085 of *Lecture Notes in Computer Science*, pp. 268–283, Springer, 2006.

[62] KINIRY, J., POLL, E., and COK, D., "Design by contract and automatic verification for Java with JML and ESC/Java2. ETAPS tutorial," 2005.

[63] KLARLUND, N. and SCHWARTZBACH, M. I., "Graph Types," in *ACM POPL*, (Charleston, South Carolina), Jan. 1993.

[64] KLARLUND, N. and MLLER, A., "Mona version 1.4 - user manual."

[65] KUNCAK, V., *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.

[66] KUNCAK, V., LAM, P., ZEE, K., and RINARD, M., "Modular pluggable analyses for data structure consistency," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 988–1005, 2006.

[67] KUNCAK, V., NGUYEN, H. H., and RINARD, M., "An algorithm for deciding bapa: Boolean algebra with presburger arithmetic," in *20th International Conference on Automated Deduction (CADE-20)*, (Tallinn, Estonia), Jul 2005.

[68] LAHIRI, S. and QADEER, S., "Verifying Properties of Well-Founded Linked Lists," in *ACM POPL*, (South Carolina), Jan. 2006.

[69] LAM, P., *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, 2007.

[70] LEAVENS, G. T. and BAKER, A. L., "Enhancing the Pre- and Postcondition Technique for More Expressive Specifications," in *World Congress on Formal Methods (FM '99)* (WING, J. M., WOODCOCK, J., and DAVIES, J., eds.), Sept. 1999.

[71] LEAVENS, G. T., BAKER, A., and RUBY, C., "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[72] LEAVENS, G. T. and MULLER, P., "Information hiding and visibility in interface specifications," in *IEEE/ACM Intl. Conf. on Software Engineering*, (Washington, DC, USA), pp. 385–395, IEEE Computer Society, 2007.

[73] LEAVENS, G. T. and NAUMANN, D. A., "Behavioral subtyping is equivalent to modular reasoning for object-oriented programs," Tech. Rep. 06-36, Department of Computer Science, Iowa State University, 2006.

[74] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MLLER, P., and KINIRY, J., " JML Reference Manual (DRAFT)," Feb. 2007.

[75] LEE, O., YANG, H., and YI, K., "Automatic verification of pointer programs using grammar-based shape analysis," in *ESOP*, Springer Verlag, Apr. 2005.

[76] LEE, O., YANG, H., and PETERSEN, R., "Program analysis for overlaid data structures," in *CAV*, pp. 592–608, 2011.

[77] LEINO, K. R. M., "Specification and verification of object-oriented software. lecture notes. Marktoberdorf international summer school," 2008.

[78] LEINO, K. R. M. and MÜLLER, P., "Object invariants in dynamic contexts.," in *ECOOP*, pp. 491–516, 2004.

[79] LEINO, K. R. M., "Dafny: An automatic program verifier for functional correctness," in *LPAR (Dakar)*, pp. 348–370, 2010.

[80] LISKOV, B. H., "Data abstraction and hierarchy," *ACM SIGPLAN Notices*, vol. 23, pp. 17–34, May 1988. Revised version of the keynote address given at OOPSLA'87.

[81] LISKOV, B. H. and WING, J. M., "A behavioral notion of subtyping," *ACM Trans. on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, 1994.

[82] LOIC CORRENSON, PASCAL CUOQ, A. P. and SIGNOLES, J., "Frama-c user manual," 2011.

[83] MAGILL, S., TSAI, M.-H., LEE, P., and TSAY, Y.-K., "Thor: A tool for reasoning about shape and arithmetic," in *CAV*, pp. 428–432, 2008.

[84] MARCHÉ, C. and PAULIN-MOHRING, C., "Reasoning about Java programs with aliasing and frame conditions," in *18th Int'l Conf. on Theorem Proving in Higher Order Logics*, Springer, LNCS, Aug. 2005.

[85] MARCHÉ, C., PAULIN-MOHRING, C., and URBAIN, X., "The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML," *Journal of Logic and Algebraic Programming*, vol. 58, no. 1–2, pp. 89–106, 2004.

[86] MEYER, B., *Object-oriented Software Construction*. Prentice Hall. Second Edition., 1997.

[87] MEYER, B., *Eiffel: the language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[88] MIDDELKOOP, R., HUIZING, C., KUIPER, R., and LUIT, E. J., "Invariants for non-hierarchical object structures," in *Proceedings of the 9th Brazilian Symposium on Formal Methods (SBMF'06)* (RIBEIRO, L. and MOREIRA, A. M., eds.), (Natal, Brazil), 2006.

[89] MOELLER, A. and SCHWARTZBACH, M. I., "The Pointer Assertion Logic Engine," in *ACM PLDI*, June 2001.

[90] MULLER, P., *Modular specification and verification of object-oriented programs*. New York, NY, USA: Springer, 2002.

[91] NELSON, C. G., *Techniques for program verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[92] NGUYEN, H. H., DAVID, C., QIN, S., and CHIN, W., "Automated Verification of Shape And Size Properties via Separation Logic," in *Intl Conf. on Verification, Model Checking and Abstract Interpretation*, (Nice, France), Jan. 2007.

[93] NGUYEN, H. H. and CHIN, W.-N., "Enhancing program verification with lemmas," in *CAV*, pp. 355–369, 2008.

[94] NIMMER, J. W. and ERNST, M. D., "Invariant inference for static checking.," in *ESEC/SIGSOFT Foundations of Software Engr.*, pp. 11–20, 2002.

[95] NIPKOW, T., PAULSON, L. C., and WENZEL, M., *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[96] O'HEARN, P. W., YANG, H., and REYNOLDS, J. C., "Separation and Information Hiding," in *ACM POPL*, (Venice, Italy), Jan. 2004.

[97] O'HEARN, P., YANG, H., and REYNOLDS, J., "Separation and Information Hiding," in *ACM POPL*, (Venice, Italy), Jan. 2004.

[98] OSTROFF, J., WANG, C., KERFOOT, E., and TORSHIZI, F. A., "Automated model-based verification of object-oriented code," Tech. Rep. CS-2006-05, York University, Canada, May 2006.

[99] PARKINSON, M. J., *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[100] PARKINSON, M. J. and BIERMAN, G. M., "Separation logic and abstraction," in *ACM POPL*, pp. 247–258, 2005.

[101] PARKINSON, M. J. and BIERMAN, G. M., "Separation logic, abstraction and inheritance," in *ACM POPL*, 2008.

[102] PASAREANU, C. and VISSER, W., "Verification of Java programs using symbolic execution and invariant generation," in *SPIN Workshop*, Apr. 2004.

[103] PÉREZ, J. A. N. and RYBALCHENKO, A., "Separation logic + superposition calculus = heap theorem prover," in *PLDI*, pp. 556–566, 2011.

[104] PIENTKA, B., "A heuristic for case analysis," tech. rep., 1995.

[105] PUGH, W., "The Omega Test: A fast practical integer programming algorithm for dependence analysis," *Communications of the ACM*, vol. 8, pp. 102–114, 1992.

[106] QUINONEZ, J., TSCHANTZ, M. S., and ERNST, M. D., "Inference of reference immutability," in *ECOOP*, pp. 616–641, 2008.

[107] REYNOLDS, J., "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE Logic in Computer Science*, (Copenhagen, Denmark), July 2002.

[108] REYNOLDS, J., "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE Logic in Computer Science*, (Copenhagen, Denmark), July 2002.

[109] RIVAL, X. and CHANG, B.-Y. E., "Calling context abstraction with shapes," in *POPL*, pp. 173–186, 2011.

[110] ROBBY, DWYER, M. B., and HATCLIFF, J., "Bogor: an extensible and highly-modular software model checking framework," in *ESEC/SIGSOFT Foundations of Software Engr.*, pp. 267–276, 2003.

[111] RUGINA, R., "Quantitative Shape Analysis," in *SAS*, Springer LNCS, (Verona, Italy), Aug. 2004.

[112] SAGIV, S., REPS, T., and WILHELM, R., "Parametric shape analysis via 3-valued logic," *ACM Trans. on Programming Languages and Systems*, vol. 24, May 2002.

[113] SEINO, T., OGATO, K., and FUTATSUGI, K., "Mechanically supporting case analysis for verification of distributed systems," *IJPCC*, 2005.

[114] SIMS, E.-J., "Extending separation logic with fixpoints and postponed substitution," *Theoretical Computer Science*, vol. 351, no. 2, pp. 258–275, 2006.

[115] SUWIMONTEERABUTH, D., BERGER, F., SCHWOON, S., and ESPARZA, J., "jmoped: A test environment for java programs," in *CAV*, pp. 164–167, 2007.

[116] SUWIMONTEERABUTH, D., SCHWOON, S., and ESPARZA, J., "jmoped: A java bytecode checker based on moped," in *TACAS*, pp. 541–545, 2005.

[117] TONY HOARE, J. M., "Verified software: Theories, tools, experiments - vision of a grand challenge project," 2005.

[118] TSCHANNEN, J., "Automatic Verification of Eiffel Programs," Master's thesis, ETH Zurich, 2009.

[119] VAFEIADIS, V., "Automatically proving linearizability," in *CAV*, pp. 450–464, 2010.

[120] VAFEIADIS, V. and PARKINSON, M. J., "A marriage of rely/guarantee and separation logic," in *CONCUR*, pp. 256–271, 2007.

[121] WEIS, B., *Deductive Verification of Object-Oriented Software*. PhD thesis, University of Karlsruhe, 2010.

[122] XI, H., *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[123] ZEE, K., KUNCAK, V., and RINARD, M. C., "An integrated proof language for imperative programs," in *PLDI*, (New York, NY, USA), pp. 338–351, ACM, 2009.

[124] ZIBIN, Y., POTANIN, A., ALI, M., ARTZI, S., KIEZUN, A., and ERNST, M. D., "Object and reference immutability using java generics," in *ESEC/SIGSOFT FSE* (CRNKOVIC, I. and BERTOLINO, A., eds.), pp. 75–84, ACM, 2007.

[125] ZIBIN, Y., POTANIN, A., LI, P., ALI, M., and ERNST, M. D., "Ownership and immutability in generic java," in *OOPSLA* (COOK, W. R., CLARKE, S., and RINARD, M. C., eds.), pp. 598–617, ACM, 2010.