Reasoning About Frame Properties in Object-oriented Programs

2017

Yuyan Bao  
*University of Central Florida*

Find similar works at: [https://stars.library.ucf.edu/etd](https://stars.library.ucf.edu/etd)

University of Central Florida Libraries [http://library.ucf.edu](http://library.ucf.edu)

Part of the [Computer Sciences Commons](https://stars.library.ucf.edu/etd)

**STARS Citation**

[https://stars.library.ucf.edu/etd/6052](https://stars.library.ucf.edu/etd/6052)

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
REASONING ABOUT FRAME PROPERTIES IN OBJECT-ORIENTED PROGRAMS

by

YUYAN BAO
M.S. Software Engineering, Beihang University, 2007
B.S. Computer Science, Beijing University of Technology, 2003

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering & Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2017

Major Professor: Gary T. Leavens
ABSTRACT

Framing is important for specification and verification of object-oriented programs. This dissertation develops the local reasoning approach for framing in the presence of data structures with unrestricted sharing and subtyping. It can verify shared data structures specified in a concise way by unifying fine-grained region logic and separation logic. Then the fine-grained region logic is extended to reason about subtyping.

First, fine-grained region logic is adapted from region logic to express regions at the granularity of individual fields. Conditional region expressions are introduced; not only does this allow one to specify more precise frame conditions, it also has the ability to express footprints of separation logic assertions.

Second, fine-grained region logic is generalized to a new logic called unified fine-grained region logic by allowing the logic to restrict the heap in which a program runs. This feature allows one to express specifications in separation logic.

Third, both fine-grained region logic and separation logic can be encoded to unified fine-grained region logic. This result allows the proof system to reason about programs specified in both styles.

Finally, fine-grained region logic is extended to reason about a programming language that is similar to Java. To reason about inheritance locally, a frame condition for behavioral subtyping is defined and proved sound.
ACKNOWLEDGMENTS

Firstly, I would like to thank my adviser, Gary T. Leavens, for his patient guidance and encouragement for the past 7 years. He introduced me to this research area and inspired me with the spirit of adventures in research. His valuable suggestions had a great impact on my work. I am also grateful for him sending me to various summer schools, where I met various other scholars and researchers; all such chances enhanced my background and broadened my view.

Secondly, I thank Gidon Ernst for introducing me the KIV theorem prover and for discussing the semantics of separation logic and its encoding in the KIV. Also, I thank David A. Naumann for discussion about region logic and for providing feedback on the FTfJP’15 paper and on an early draft of this work. His work has great influence on this work as well. I thank Rustan Leino for teaching me Dafny and answering related questions. I also thank the anonymous referees for the feedback of this work.

Thirdly, I would like to thank my colleagues in the formal method lab for providing a friendly environment to work in.

Last, but not least, I would like to thank my parents and sister for their constant love. Also, I think my friends for their friendship. My special thanks go to my boyfriend, Xin Li, for his understanding, ongoing support and encouragement.


TABLE OF CONTENTS

LIST OF FIGURES ......................................................... x

LIST OF TABLES ......................................................... xiv

CHAPTER 1: INTRODUCTION ............................................. 1

1.1 Background ......................................................... 2

1.1.1 Separation Logic ................................................. 2

1.1.2 Region Logic ..................................................... 3

1.1.3 Supertype Abstraction ........................................... 4

1.2 Problems .......................................................... 5

1.3 Contributions ....................................................... 5

1.4 Overview .......................................................... 6

1.5 Related Work ....................................................... 6

1.5.1 Non-local Reasoning Approaches to Framing ...................... 7

Ownership-based Model: ................................................ 7

Linear Logic: .......................................................... 9

Linear Type: .......................................................... 10
CHAPTER 4: FINE-GRAINED REGION LOGIC

4.1 Axioms and Inference Rules

4.1.1 The Sequence Rules

4.1.2 The Loop Rule

4.2 Soundness

CHAPTER 5: UNIFIED FINE-GRAINED REGION LOGIC

5.1 Axioms and Inference Rules

5.1.1 The Sequence Rules

5.2 Soundness

CHAPTER 6: INTEROPERABILITY

6.1 FRL - An Instance of UFRL

6.2 Encoding Separation Logic

6.2.1 Separation Logic Review

6.2.2 Supported Separation Logic

6.2.3 Encoding SSL Assertions

6.2.4 SSL Proofs Review and Approach

6.2.5 Translating SSL Proofs into UFRL
6.3 Extending the UFRL (FRL) Proof System with Separating Conjunction . . . . . . . 79

   6.3.1 Extending the Syntax and the Semantics ........................................ 80
   6.3.2 Proof Rules ..................................................................................... 82
   6.3.3 Encoding SSL specifications: ............................................................. 86
   6.3.4 Summary ......................................................................................... 86

CHAPTER 7: RECURSIVE PREDICATES ............................................................. 87

   7.1 Recursive Predicated in UFRL (FRL) ..................................................... 87
   7.2 Inductive Definition in SSL ................................................................. 89
   7.3 Encoding ............................................................................................. 91

CHAPTER 8: REASONING ABOUT SUBTYPING ............................................. 94

   8.1 Programming Language Extended with Inheritance .............................. 94
   8.2 Semantics ............................................................................................ 98
   8.3 Effects ................................................................................................. 101
     8.3.1 The read effect of a class ............................................................... 102
   8.4 Supertype Abstraction and Local Reasoning ....................................... 102
     8.4.1 Problem ....................................................................................... 103
     8.4.2 Encapsulation ............................................................................. 105
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Taxonomy of methodologies for framing</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>The syntax of the programming language</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>The definition of the function $MV$</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>The semantics of expressions</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>The semantics of statements</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>The syntax of assertions</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>The semantics of assertions</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>The grammar of effects</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>The sub-effect rules</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>The read effects of expressions, region expressions and atomic assertions</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>The inference rules for the framing judgment</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>The definition of separator</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>The correctness axioms and proof rules for statements in FRL</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>The structural rules in FRL (1)</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>The structural rules in FRL (2)</td>
<td>42</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.1</td>
<td>The correctness axioms and proof rules for statements in UFRL</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>The structural rules in UFRL (1)</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>The structural rules in UFRL (2)</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>A summary of results on encoding assertions</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>The axioms and proof rules for statements in SSL [77]</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>A linked-list example written in UFRL with separating conjunction</td>
<td>84</td>
</tr>
<tr>
<td>7.1</td>
<td>Translation of inductive definition in SSL to recursive predicates in UFRL</td>
<td>92</td>
</tr>
<tr>
<td>7.2</td>
<td>The encoding of the predicate Eq. (7.3)</td>
<td>92</td>
</tr>
<tr>
<td>8.1</td>
<td>The extended syntax with OO features</td>
<td>94</td>
</tr>
<tr>
<td>8.2</td>
<td>An example of framing invariant</td>
<td>103</td>
</tr>
<tr>
<td>8.3</td>
<td>Classes Cell, ReCell and FCell</td>
<td>106</td>
</tr>
<tr>
<td>8.4</td>
<td>Encapsulation example</td>
<td>107</td>
</tr>
<tr>
<td>8.5</td>
<td>The specification of the class ECell</td>
<td>109</td>
</tr>
<tr>
<td>8.6</td>
<td>Argument exposure example</td>
<td>110</td>
</tr>
<tr>
<td>8.7</td>
<td>The specification of class Cell</td>
<td>119</td>
</tr>
<tr>
<td>8.8</td>
<td>The revised specification of class ECell</td>
<td>121</td>
</tr>
</tbody>
</table>
9.1 The class $\text{NumberS}$ specified in the style of SSL ................. 125
9.2 The class $\text{NumberR}$ specified in the style of FRL ................. 125
9.3 Translating method specifications in the class $\text{NumberS}$ and $\text{NumberR}$ ...... 126
9.4 The specification of marking a DAG ........................................ 128
9.5 A client code of a coffee shop .................................................. 135
9.6 The specification of a generic linked-list written in a mixed style .......... 137
9.7 The class $\text{ListIterator}$ specified in the style of FRL .................. 138
9.8 The specification of a generic dictionary written in the style of SSL ...... 140
9.9 The class $\text{Order}$ ............................................................. 141
9.10 The specification of an application program written in a mixed style (part 1) . 142
9.11 The specification of an application program written in a mixed style (part 2) . 143
9.12 The specification of the class $\text{DCell}$ ...................................... 147
9.13 The specification of the class $\text{ReCell}$ .................................... 150
9.14 The specification of the class $\text{TCell}$ ...................................... 151
9.15 The specification of the class $\text{OCell}$ .................................... 152
A.1 The typing rules for pure expressions and region expressions ............. 156
A.2 The typing rules for statements ............................................... 157
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3</td>
<td>The typing rules for assertions</td>
<td>158</td>
</tr>
<tr>
<td>H.1</td>
<td>The derivation of rule $T_R \llbracket ALLOC_s \rrbracket$</td>
<td>206</td>
</tr>
<tr>
<td>H.2</td>
<td>The derivation of rule $T_R \llbracket ACC_s \rrbracket$</td>
<td>207</td>
</tr>
<tr>
<td>H.3</td>
<td>The derivation of rule $T_R \llbracket UPD_s \rrbracket$</td>
<td>207</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Features of the programming language that are not formalized in this dissertation</td>
<td>19</td>
</tr>
<tr>
<td>8.1</td>
<td>Auxiliary functions used in the semantics</td>
<td>95</td>
</tr>
<tr>
<td>9.1</td>
<td>Selected specifications for the class Node&lt;T&gt;</td>
<td>139</td>
</tr>
<tr>
<td>9.2</td>
<td>The predicates that are used by clients</td>
<td>144</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION

As software are widely used in our daily life, its quality is drawing concern. Although software testing can improve software quality by reducing defects, it cannot guarantee the absence of defects. On the contrary, software verification can prove that the software has no defects and behaves exactly as the specifications describe.

In software engineering, modularization allows large projects to be decomposed into smaller components. Each component performs distinguished functionality, and can be developed independently. Thus, specifications that document functionality and verification that checks implementations’ functionality against their specifications should be carried out modularly as well. This modularity poses challenges for local reasoning about object-oriented programs. In particular, local reasoning about mutable data structures with unrestricted sharing is complex and requires onerous annotations. And reasoning about subtypes lacks a modular treatment of framing.

In software specification and verification, the frame property is used to achieve local reasoning [20]. Local reasoning means that specifications only mentions what matters to the program under verification. The classical Hoare logic [36, 38] with an added frame property provides proof axioms and rules that are used to reason about imperative programs containing, for example, assignment, sequence statements, conditional statements and loop statements. The logic uses formulas of the form \( \{P\}S\{Q\}[X] \), for partial correctness, where \( P \) and \( Q \) are assertions, \( S \) is a program statement, and \( X \) is a set of variables that specifies the frame property of \( S \), which allows \( S \) to only modify variables in the set \( X \). The state of a program is the program’s information characterized by predicates, such as \( P \) and \( Q \), at a given time. The validity of a Hoare-formula \( \{P\}S\{Q\}[X] \) means that if a program, \( S \), executes from an initial state satisfying \( P \), if \( S \) does not cause an error

\(^1\)Part of the content in this chapter was presented at FTfJP ’15 [7] and is submitted to Formal Aspects of Computing.
and terminates, then the final state satisfies $Q$, and $S$ can only modify the variables in $X$. As defined, such a formula deals with *partial correctness*; total correctness additionally means that the statement will terminate.

In object-oriented (OO) programs, frame properties are difficult to specify because of their use of complex data structures and abstractions. These data structures may consist of recursively structured and shared objects, such as directed acyclic graphs. Moreover, OO features, such as aliasing, encapsulation and dynamic dispatch, raise additional challenges to modularly specifying and verifying frame properties. For example, specifying the frame conditions of the methods who may be overridden in subclasses, and the overriding method may modify additional states introduced in the subclasses [50].

### 1.1 Background

A summary of the relevant background is provided in this section. However, knowledge about OO programming [25, 61], first-order logic and Hoare logic [36, 38] is assumed. Firstly, two approaches to local reasoning, separation logic and region logic, are discussed. Then dynamic dispatch and an approach to reason about it, i.e., *supertype abstraction* are discussed.

#### 1.1.1 Separation Logic

In separation logic (SL) [40, 79], the introduction of separating conjunction leads to its frame rule:

\[
\frac{\Gamma \vdash a \{a\} S \{a'\}}{\Gamma \vdash \{a \rightarrow c\} S \{a' \rightarrow c\}} \quad \text{where } \text{MV}(S) \cap \text{FV}(c) = \emptyset
\]
where \( \text{FV}(c) \) returns the set of free variables in \( c \) and \( \text{MV}(S) \) returns the set of variables that may be modified by \( S \). Local reasoning is achieved by this frame rule, since the specification in the hypothesis (above the horizontal line) can solely describe the partial state that program \( S \) uses. Assertions, such as \( c \), depending on other disjoint parts of the states that are untouched, can be preserved by applying the frame rule. The side-condition is needed since separating conjunction does not describe separation in the store, but only in the heap.

However, the frame rule in SL cannot be directly used when verifying data structures with unrestricted sharing [39] because of the use of regular (i.e., non-separating) conjunctions, e.g., the following definition of the predicate \( \text{dag} \):

\[
\text{dag}(d) \overset{\text{def}}{=} d \neq \text{null} \Rightarrow \exists i, j, k. (d.\text{mark} \leftrightarrow i \ast d.l \leftrightarrow j \ast d.r \leftrightarrow k \ast (\text{dag}(j) \land \text{dag}(k)))
\]

where the assertions of the form \( x.f \leftrightarrow e \) mean that the location \( x.f \) stores the value of \( e \). The use of the conjunction (instead of separating conjunction) indicates that sub-Dags may share some locations. Thus, changes in the left descendants may affect the value of the right descendants, and hence the validity of assertions that describe the values of the right descendants.

### 1.1.2 Region Logic

Region logic [2, 4] (RL) supports local reasoning by the means of effects. The effects may be variables in stores or locations in heaps; locations are expressed in terms of sets of objects and their fields [4]. RL’s frame rule uses effects to distinguish what is preserved, shown as follows:

\[
\frac{\vdash_{rl} \{P\} S\{P'\}[\varepsilon] \quad P \vdash_{rl} \delta \text{ frm } Q}{\vdash_{rl} \{P \&\& Q\} S\{P' \&\& Q\}[\varepsilon]} \quad \text{where } P \&\& Q \Rightarrow \delta/\varepsilon
\]

The formula \( \varepsilon \) is a write effect that denotes the set of variables and locations that may be modified by \( S \). The formula \( \delta \) is a read effect that denotes the set of variables and locations that the assertion
$Q$ relies on. The formula $\delta/\varepsilon$ denotes the disjointness of the two sets of variables and locations. The frame rule says that to preserve the validity of the assertion $Q$ after executing the statement $S$, one has to prove that the variables and locations in $S$’s write effects are disjoint with those that $Q$ depends on. In the conclusion of the frame rule, $P$ is connected with $Q$ by the conjunction. Thus, this rule allows one to use the frame rule directly when reasoning about data structures with sharing.

### 1.1.3 Supertype Abstraction

OO programs allow a subclass, $S$, to inherit from a superclass, $T$, by either adding or modifying fields or methods of its superclass. In this case, the type $S$ is a subtype of $T$, and the type $T$ is the type $S$’s supertype. A variable may have two types: a static type and a dynamic type. The static type is the declared one, and the dynamic type is the most specific type of the object that the variable denotes at runtime. Moreover, any instance of a subtype can be used in place of its supertype. A method call $x.m()$ runs code that is determined by the dynamic type of the receiver $x$, not $x$’s static type. This is known as dynamic dispatch.

A standard form of modular static verification uses the method’s specification of its receiver’s static type, as its exact dynamic type may be unknown. In other words, verification may use a supertype’s specification to reason about an overriding method call that may dynamically dispatch to its subtypes. This kind of reasoning is known as supertype abstraction [48]. Then, one can use its supertype’s specification to soundly describe the method’s behavior and the method’s receiver may be its subtype. Validity of supertype abstraction is ensured by behavioral subtyping [1, 46, 60], in which each overridden method obeys the specifications declared in its supertypes.

Leavens and Naumann [47] have shown that behavioral subtyping is necessary and sufficient for the validity of supertype abstraction. They define specification refinement in terms of preconditions
and postconditions. However, to apply their result to the framework of local reasoning, frame conditions are needed.

1.2 Problems

The overall problems of this dissertation are to unify methodologies for reasoning about frame properties, i.e., separation logic and region logic, and to modularly reason about frame properties in object-oriented programs. That is the second problem is to find a sound frame condition for reasoning about dynamic dispatch.

1.3 Contributions

My work has two major contributions. One is that it combines two successful logics for framing: a commonly used subset of SL and a fine-grained variant of region logic, FRL. The combined logic, unified fine-grained region logic (UFRL), is enriched by features of both SL and FRL: separating conjunction can be expressed along with explicit write and read effects specified by region expressions. Specifications written in these two styles thus can interoperate with each other as they can both be encoded into UFRL. Therefore, specifying and verifying one module can use other modules’ specifications written in different styles. The FRL and SL assertion languages have been formalized in the KIV theorem prover [30]. Lemmas and theorems that are not formally proved in the dissertation have been proved in KIV. These machine-checked proofs have been exported and are available online [5, 6].

Another contribution is that my work defines a frame condition for behavioral subtyping, which is proved sound for supertype abstraction. Framing for subtyping handles extended state in subtype objects through a novel notion of encapsulation.
1.4 Overview

Chapter 2 defines an object-based programming language, formalizes its type system and its denotational semantics. Chapter 3 introduces effects, separator and framing that serve as foundations for FRL and UFRL. Chapter 4 defines the notion of validity for FRL Hoare-formula and introduces the proof axioms and rules for FRL. Chapter 5 extends FRL to UFRL. It defines the validity for UFRL Hoare-formula and introduces the proof axioms and rules for UFRL. Chapter 6 presents the formal connections between FRL and UFRL and between SL and UFRL. Chapter 7 extends these results with SSL inductive predicates. Chapter 8 extends FRL to reason about Object-Oriented programs. Chapter 9 presents potential applications of UFRL. For example, it introduces a scheme that interprets different styles of specifications in a single mechanism. It also shows more examples of behavioral subtyping. Section 10 concludes the dissertation and discusses future work.

1.5 Related Work

There are several approaches to framing that have been described in the formal methods literature. Historically specification languages, such as VDM [42] and interface specification languages in the Larch family [34], specify frames for procedures by writing a clause in the specification that names the variables and locations that are allowed to be changed during the procedure’s execution; all other locations must be unchanged. However, such an approach does not easily generalize to layered structures of mutable objects. For example, it is difficult to specify dynamically-allocated objects where locations are generated through underlying data structures at run time.

Fig. 1.1 lists related methodologies for framing with a view towards classifying different methodologies as a guide for this section. One could imagine different classifications depending on the problems one is concerned about. Fig. 1.1 categorizes each methodology as either supporting local
reasoning or not. In general, local reasoning approaches use “small axioms” [71] that focus on the part of a program state that is being written (or read) by the statement in question [4], and use frame rules to derive the property of global states by means of explicit locations that the program may write or read. Non-local reasoning approaches do not focus only on such local state, and commonly combine an aliasing control mechanism with the technique of object invariant. For example, updating an object’s field cannot unnoticeably violate other objects invariant. Section 1.5.1 summarizes those works that use non-local reasoning. Section 1.5.2 and Section 1.5.3 discuss local reasoning approaches in detail, as the approaches presented in this dissertation belong to this category. Section 1.5.4 discusses the related work on behavioral subtyping.

Non-local reasoning approaches and the work in this dissertation are orthogonal. None of the local reasoning approaches relates and encodes other approaches, except the work of Parkinson and Summers [78] which connects one variant of SL with another.

1.5.1 Non-local Reasoning Approaches to Framing

Ownership-based Model:

The ownership-based model [70] is one approach that works with object structures. It only allows a designated owner object to mutate the objects that make up part of a complex object structure. Consider the data structure of a linked list. Its representation is a list of nodes, where each node contains its value and a reference to the next node in the list. In an ownership model, the list is the owner of all the nodes that it contains. Modifying any node in the list has to go through the methods of the list object. However, in the approaches that support local reasoning, the precondition for an update statement (i.e., \( n.\text{val} := e; \)) is just \( n \neq \text{null} \) in region logic, or \( \exists x. n.\text{val} \rightarrow x \) in separation logic, no matter whether \( n \) is a node in a list or not.
Figure 1.1: Taxonomy of methodologies for framing

The universe type system [63] combines type checking and some dynamic checks to enforce the ownership property; the universe type system also gives a semantics to specifications of frames in a way that allows modular verification of frame conditions and invariants [66]. However, the universe type system and other approaches based on ownership have difficulties in specifying and verifying some shared data structures, for example, the subject-observer design pattern has sharing that is not compatible with ownership. However, SL and RL do not have difficulty in reasoning about this pattern [4, 74].

The Boogie methodology [9] and its variants [11, 56] encode the ownership model by specification-only fields. The Boogie methodology [9] introduces a field modifier `rep` that identifies an object’s
representation; a `rep` field means that the field is owned by the enclosing object of its declaring class. An object may have different owners. Although the Boogie methodology eases the problem of dealing with shared mutable data structures, it introduces a fair amount of overhead and complexity in writing specifications. For example, each object is instrumented by specification-only fields `inv` and `committed` that denote the states of an object and are specified in method specifications. And their values can only be modified through the use of the two specification statements: `pack` and `unpack`, which can be used in a method body.

Leino and Müller [56] improve the Boogie methodology by allowing dynamic contexts and ownership transfer. Their work introduces another field `owner` storing the owning object. Its value can be changed through the statement `pack` and `unpack` as well.

Barnett and Naumann [68] introduce an explicit friend declaration, e.g., `friends T reads f`, which denotes the declaring class grants the type `T` a read access to its field `f`. A flexible protocol, called “friendship”, is established between the two types, where invariant can be expressed over their shared data structures, e.g., the subject-observer design pattern.

**Linear Logic:**

Bierhoff and Aldrich’s work [18] combines linear logic [33] with access permissions. Specifications of a statement `S` are defined by linear implication: `P \rightarrow_o Q`, where `P` is a precondition and `Q` is a postcondition. It means that `S` consumes `P` and yields `Q`; that has some similarities to the meaning of specifications in SL. Sharing in Bierhoff and Aldrich’s work can be expressed by permission predicates. However, it does not make footprints explicit, i.e., locations that an object’s invariant depends on. Therefore, it lacks a way to tell whether updating an object’s field may influence other objects. It uses the `pack` and `unpack` statements to transfer from a state when the invariant holds to a state where it may not hold.
Nistor et al. [69]’s approach also builds on top of linear logic. Similar to Bierhoff and Aldrich’s work, sharing is expressed by means of permissions. An assignment to a field of an object $t$ needs to know the state of the current $\texttt{this}$.

*Linear Type:*

Linear type systems restrict aliasing. An object with linear type is not aliased; an object with non-linear type may be aliased. To safely allow a nonlinear type to have linear components, Fähndrich and Deline [31] introduces two operations: *adoption* and *focus*. Adoption, $\texttt{adopt} \ e_1 \ \texttt{by} \ e_2$, allows the object that $e_2$ denotes to reference the object that $e_1$ denotes. Thus, it allows a linear type to transfer to a non-linear one where aliasing is allowed. Focus allows a non-linear type to be treated as a linear one when all aliases become invalid. To do so, the type system needs to track aliases’ lifetimes. It is not local as updating a field of an object needs to be aware of all references to the field. The dynamic frames approach does not prevent aliasing, but prevents harmful updating by various proof obligations (which can be decided by set-theoretic judgment, e.g., the location where a change may happen to be disjoint with the locations that intended invariant depends on).

1.5.2 Dynamic Frames Approaches

The dynamic frames approach [43, 44] dynamically tracks sets of locations (regions) stored in specification variables (or computed by functions); these regions are used in the specification of frames. The resulting flexibility allows the specification of shared data structures, but reasoning about dynamic frame uses second-order logic, which makes automation difficult.
Our work is partly based on the work of Banerjee et al. on region logic (RL) [4]. However, there are several key differences between FRL (and UFRL) and this work on RL:

1. In RL, regions are sets of references, possibly containing null [4]. For example, \{x\} is a region containing a singleton object \(x\). In RL, image expressions (like \(x^f\)) denote a region only if the field referenced \(f\) has type \texttt{rgn} or \texttt{Object}. By contrast, in FRL regions are sets of locations, which makes it convenient to form unions of sets of locations, something that is more difficult to express in RL. This difference also makes it more convenient in FRL to express footprints of SL assertions, which are used in the encoding. Using sets of locations also matches specification languages in which frames are specified using such sets, like JML [22].

2. In RL, the footprints of region expressions are larger than the corresponding footprints in FRL. For example, in RL the footprint of the region expression \(\{x\}^f\) is \texttt{rd} \(\{x\}^f, x\), meaning that the value of this region expression depends on \(\{x\}^f\) itself, since \(f\) may not be a field declared in \(x\)'s class. In FRL the region expression, \texttt{region}\{\(x.f\}\}, only depends on the variable, \(x\), as FRL’s type system makes sure that \(f\) is a declared field name.

3. Finally, RL does not have conditional region expressions, which makes FRL more convenient for specifying the frames of SL assertions that involve implication.

However, FRL (UFRL) and RL also share lots of similarities, due to FRL and UFRL being adapted from RL:

1. Both use ghost fields with type regions to express frame conditions, i.e., read effects, write effects and fresh effects. The effects are stateful, which follows the work of dynamic frames.
2. RL’s read effects have the same granularity as FRL (and UFRL). The formula $\text{rd} G^f$ allows one to read the field of objects in $G$ [4, p.22]; e.g., the RL read effect $\text{rd} x^f$ is equivalent to the FRL read effect $\text{reads region}\{x.f\}$.

3. The read effects of the points-to predicate are consistent in RL and FRL (and UFRL). In RL, the read effects of the points-to predicate, which are called “footprints” in their work [4], are defined by $\text{ftpt}(x.f = E) = \text{rd} x, x.f, \text{ftpt}(E)$, where $\text{rd}$ is the keyword for read effects (this work uses $\text{reads}$ instead). The form $\text{rd} x.f$ abbreviates the form $\text{rd} \{x\}^f$ [4, p.23]. Although $x^f$ may not be the same as the region expression $\text{region}\{x.f\}$ as explained previously, $\text{rd} x^f$ is semantically equivalent to $\text{reads region}\{x.f\}$ in FRL and UFRL.

4. RL and FRL (and UFRL) have similar definitions of agreement, frame validity, separator, immunity, and Hoare-formula. Therefore, their proof rules are quite similar as well. In particular, the frame conditions for the proof axioms are semantically equivalent.

Rosenberg’s work [80] implements a semi-decision procedure for RL as a plugin inside the satisfiability modulo theories (SMT) solver Z3. Similarly, FRL and UFRL expressions could also be encoded into SMT, but such an encoding is beyond the scope of this dissertation.

**Dafny:**

Leino’s Dafny language [51, 53, 54] is based on dynamic frames, in which frames are specified using sets of objects stored in ghost fields. Our work has adopted several programming and specification language features from Dafny, i.e., the style of programming and specification languages. However, unlike FRL, Dafny does not make it easy to specify frames at the level of locations, so instead one must strengthen postconditions by using $\text{old}$ expressions to specify which fields of
threatened objects must not change. Consider the following example adapted from Leino’s work [54].

```java
class C {
    var x : int;  var y : int;

    method Update()
        modifies {this};
        ensures x = old(x) + 1;
        ensures y = old(y);
        { x := x + 1; }
}
```

The `modifies` clause indicates the objects that may be modified by a method. In the above code, all the fields of the object `this` are allowed to be modified by the method `Update`. Although the implementation only updates the field `x` and leaves the field `y` unchanged, the second postcondition `y = old(y)` is needed, otherwise, the caller would lose the value about the field `y`.

*The work of Smans et al.:*

The dynamic frames approach used by Smans et al. [84], however, does use sets of locations. These sets can be computed by pure functions. This use of pure functions supports data abstraction and information hiding. Data abstraction and information hiding are considered to be orthogonal to the problems discussed in this dissertation, as standard solutions can be applied [2, 3, 50, 58]. While their language has much of the power of FRL, they do not formally connect SL with their language, and they do not address the problem of allowing specifications in both SL and RL to interoperate.
The KeY Tool:

The KeY tool [13, 86] extends JML with dynamic frames. It introduces a type `\texttt{locset}` that stands for sets of memory locations. Recently, Mostowski and Ulbrich [62] replace ghost fields with model methods that allow method contracts to dynamically dispatch through abstract predicates. However, neither KeY nor JML addresses the problem of connecting SL to RL and mixing specification styles.

1.5.3 Related work on Separation Logic

This dissertation also draws on work in separation logic (SL) [40, 79]. It supports local reasoning and its frame conditions are implicit from preconditions where a program’s read and write locations are requested. The introduction of separating conjunction and the frame rule allows a program to be verified on partial heaps, which has been explained in Section 1.1.1. There has been much work on automating SL using first-order tools [14, 15, 16, 19, 24, 29, 78]. Our results show another way of encoding SL into first order logic, via UFRL. However, these work do not show connection with dynamic frames approaches.

Implicit Dynamic Frames

Implicit dynamic frames [83] is a variant of separation logic [82]. It introduces a predicate `\texttt{acc}` that specifies locations that are requested and returned by preconditions and postconditions. The upper bound of the requested locations by a method’s precondition are considered as the method’s implicit frame conditions. Similar to SL, it does not separate the locations that may be written from those that may be read.
An inspiration for this dissertation was the work of Parkinson and Summers [78], who showed a relationship between SL and the methodology of Chalice [57] that combines the core of implicit dynamic frames [83] with fractional permissions and concurrency. They encode a separation logic fragment (similar to the subset of separation logic that are encoded in this dissertation) into the language of implicit dynamic frames by defining a total heap semantics of SL, which agrees with the weakest precondition semantics of the implicit dynamic frames language. While their work did not connect SL and RL, the results in this dissertation go further than the analogous results in their paper. In this thesis, a translation of axioms and proof rules for a SL Hoare logic is formalized and proved to be sound.

### 1.5.4 Related Work on Behavioral Subtyping

Behavioral subtyping [1, 46, 60] constrains the behavior of each method that overrides the one in its supertype, such that one can use the supertype’s specifications to reason about the clients that may invoke the subtype’s methods at run time. This is known as supertype abstraction [48]. Leavens and Naumann [46] have shown that behavioral subtyping is necessary and sufficient for the soundness of supertype abstraction. They define specification refinement in terms of preconditions and postconditions, but give no explicit treatment of frame conditions. Thus, their results are difficult to apply to the framework of local reasoning.

Müller’s work [63] achieves behavioral subtyping by specification inheritance [28] in the Universe Type System, which is an ownership model for flexible alias control. His approach provides explicit frame conditions and the extended state are allowed to be modified in subtypes, as the encapsulation is proved by universes [63]. However, the approach in this dissertation does not have the concept of universes. Instead of controlling the aliasing, my work uses regions to frame an assertion, and uses the frame rule to derive assertions whose values are preserved, i.e., the assertions
who depends on disjoint regions from a program’s write effects. Thus, an unsound conclusion due to aliasing is prevented. Therefore, this dissertation defines a new definition of encapsulation in the terms of region expressions.

Barnett et al.’s Spec7 specification language [10] encodes an ownership-based model with ghost variables. In Spec7, an overriding method can enhance supertype’s postconditions, and has to preserve supertype’s precondition. The extended states are allowed to be modified by subtype’s methods given that each component is owned by a unique owner at a time. Such mechanism provides encapsulations. However, the work in this dissertation is not based on any ownership models. A new mechanism of encapsulation is needed, which is one of the problems that is solved here.

DeLine and Fähndrich’s work [27] handles aliasing by a linear type system. Their work follows the notion of behavioral subtyping of Liskov and Wing [60]. Their uses abstract predicates in pre- and postconditions. The extended states of a subclass are specified by either enhancing typestates that are defined in its superclass or by adding new typestates. However, specifications in their work are transitions of typestates together with aliasing information. There are no explicit frame conditions. Thus, their work does not help solve the problem of specifying frame conditions for subtypes.

Parkinson and Bierman [76] handle different types of inheritance by introducing an abstract predicate family based on the formalism of second-order separation logic. Each method has a static specification and a dynamic specification. Dynamic specifications follow the behavioral subtyping criteria defined in Leavens and Naumann’s work [46]. Encapsulation is implicit in SL in the sense that \( \alpha * c \) implies that all the predicates belong to the predicate family \( \alpha \) separate-conjuncts from the assertion \( c \). However, the work in this dissertation needs methodology to express encapsulation.
CHAPTER 2: PROGRAMMING LANGUAGE

This chapter presents the programming language for which the programming logic is formalized.

2.1 Syntax

Fig. 2.1 defines the syntax of sequential object-based programs. Over-lines indicate possible empty sequences. Square brackets mean optional elements. There are distinguished variable names. The variable \texttt{this} is the receiver object; the variable \texttt{ret} stores the return value of a method if the method has one; the variable \texttt{alloc} stores the domain of the heap. In the syntax, the notation \texttt{n} is a numeral, \texttt{x} is a variable name (or a pseudo-variable, such as \texttt{alloc}), and \texttt{f} is a field name.

\begin{verbatim}
Prog ::= Class S
Class ::= class C { Member }
Member ::= Field | Method
Field ::= var f:T;
Method ::= method m(x:T)[:T'] { S }
T ::= int | bool | region | C | C<T>
E ::= n | x | null | E_1 ⊕ E_2
RE ::= x | region{} | region{x.f} | region{x.*} | E ? RE_1 : RE_2
    | filter{RE,T,f} | filter{RE,T} | RE_1 ⊗ RE_2
G ::= E | RE
S ::= skip; | var x:T; | x:=G; | x_1:=x_2.f; | x.f:=G;
    | x:=new T; | if E then {S_1} else {S_2} | while E {S} | S_1S_2
⊕ ::= = | + | - | * | ≤ ...
⊗ ::= + | - | *
\end{verbatim}

Figure 2.1: The syntax of the programming language

\footnote{Subsets of the presented language in this chapter was presented at FTfJP ’15 [7] and is submitted to Formal Aspects of Computing.}
A class consists of fields and methods. A field is declared with type integer, a user-defined datatype, or \texttt{region}. A method is declared in a class. A constructor is the method whose name is the same as the class name. Each class must define its constructor that must be invoked after an object of the class is allocated. In a method implementation, there are local variable declarations, update statements, condition statements, and loop statements. The statement for garbage collection or deallocation are excluded in our statements. Fig. 2.2 shows the definition of the function \( \text{MV} \) that returns a set of variables that may be modified by a given statement.

The syntactic category \( E \) describes expressions, \( RE \) describes region expressions, and \( S \) describes statements. Expressions and region expressions are pure, so cannot cause errors. There is a type \texttt{region}, which is a set of locations. The region expression \texttt{region}{} denotes the empty region. The region expression of the form \texttt{region}\{\texttt{x.f}\} denotes a singleton set that stores the location of field \( f \) in the object that is the value of \( x \). The region expression of the form \texttt{region}\{\texttt{x.}\*\} denotes a set that contains the abstract locations represented by the reference \( x \) and all its fields. \footnote{Since FRL does not have subclassing or subtyping, the fields in \texttt{region}\{\texttt{x.}\*\} are based on the static type of the reference denoted by \( x \), which will also be its dynamic type.}

The conditional region expression, \( E \ ? \ RE_1 : RE_2 \), is stateful; it denotes that if \( E \) is true, then the region is \( RE_1 \), otherwise the region is \( RE_2 \). A region expression of the form \texttt{filter}\{\texttt{RE}, \texttt{T}, \texttt{f}\} denotes the set of locations of form \( (o, f) \) in \( RE \), where each object reference, \( o \), has the type \( T \). A region expression of the form \texttt{filter}\{\texttt{RE}, \texttt{T}\} denotes the subset of \( RE \) with references of type \( T \). For example, let \( RE = \{o_1.f_1, o_1.f_2, o_2.f\} \), where only \( o_1 \) has type \( T \), then \texttt{filter}\{\texttt{RE}, \texttt{T}\} =

\begin{align*}
\text{MV(} \textit{skip}; \text{)} &= \emptyset \\
\text{MV(} \texttt{var} \ x : T; \text{)} &= \emptyset \\
\text{MV(} x := \texttt{new} \ T; \text{)} &= \{ x \} \\
\text{MV(} x := G; \text{)} &= \{ x \} \\
\text{MV(} x.f := G; \text{)} &= \emptyset \\
\text{MV(} x := x'.f; \text{)} &= \{ x \} \\
\text{MV(} \texttt{if} \ E \ \texttt{then} \ \{S_1\} \ \texttt{else} \ \{S_2\} \text{)} &= \text{MV}(S_1) \cup \text{MV}(S_2) \\
\text{MV(} \texttt{while} \ E \ \{S\} \text{)} &= \text{MV}(S) \\
\text{MV}(S_1S_2) &= \text{MV}(S_1) \cup \text{MV}(S_2)
\end{align*}
Table 2.1: Features of the programming language that are not formalized in this dissertation.

<table>
<thead>
<tr>
<th>seq&lt;T&gt;</th>
<th>sequence type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>the length of the sequence s</td>
</tr>
<tr>
<td></td>
<td>the element at index i of the sequence s if 0 ≤ i and i &lt;</td>
</tr>
<tr>
<td>s[i..]</td>
<td>generate a new sequence that has the same elements in the same order as s, but the first one, if 0 ≤ i and i &lt;</td>
</tr>
<tr>
<td>s[i..j]</td>
<td>generate a new sequence that has j − i elements, and elements in the same order as s but starting with the element s[i], if 0 ≤ i and i &lt;</td>
</tr>
<tr>
<td>s1 + s2</td>
<td>sequence concatenation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>map&lt;K,V&gt;</th>
<th>map type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>the value of a given key k in a map m, if k is in the domain of m</td>
</tr>
<tr>
<td>m[k]</td>
<td>test whether the key k is in the domain of the map m</td>
</tr>
<tr>
<td>k in m</td>
<td>test whether the key k is not in the domain of the map m</td>
</tr>
<tr>
<td>m[k := v]</td>
<td>generate a new map that adds k to the domain of the map m, and associates the key k with the value v, if k is not in the domain of m, otherwise it is overridden in the new map</td>
</tr>
<tr>
<td>map i</td>
<td>i in m &amp; &amp; i ≠ k :: m[i]</td>
</tr>
</tbody>
</table>

{o1.f1, o1.f2}. The operators +, −, and * denote union, difference and intersection respectively.

In addition to the language that has been formalized, Table 2.1 shows notations for a generic mathematical type seq adopted from Dafny [51, 53]. It is used in examples, and is not formalized here for simplicity.

For simplicity, functions and pure methods are not formalized, but rely on the formalization in Banerjee et al.’s work [3]. Functions are just pure methods. In examples, a predicate declaration

```
predicate p(z:T)  reads r { P }
```

is syntax sugar for the declaration

```
method p(z:T)  reads r { ret := P; }
```

Predicates are used for the purpose of specification, and cannot be invoked by programs.
There is a type environment, $\Gamma$, which maps variables to types:

$$\Gamma \in TypeEnv = var \rightarrow T.$$  

A type environment, $\Gamma$, is *well-formed* if it is a partial function, i.e., for all $x \in \text{dom}(\Gamma)$, $\Gamma(x)$ is unique. Typing rules for expressions, region expressions and statements are not surprising, thus, are defined in Appendix A.

### 2.2 Semantics

In order to define the semantics of the programming language, the definition of some common semantic functions are given. A program state is a pair of a store and a heap. A store, $\sigma$, is a partial function that maps a variable to its value. A heap, $h$ or $H$, is a finite partial map from $\text{Loc}$ to values. The set $\text{Loc}$ represents locations in a heap. A location is denoted by a pair of an allocated reference, $o$, and its field name, $f$. We call a set of locations a *region*, written $R$. Heaps and regions are manipulated using the following operations.

**Definition 1** (Heap and Region Operations). *Lookup in a heap*, written $H[o, f]$, *is defined when* $(o, f) \in \text{dom}(H)$. $H[o, f]$ *is the value that $H$ associates to* $(o, f)$.

$H_1$ is extended by $H_2$, written $H_1 \subseteq H_2$, *means*: $\forall (o, f) \in \text{dom}(H_1) : (o, f) \in \text{dom}(H_2) \land H_1[o, f] = H_2[o, f]$.

$H_1$ is disjoint from $H_2$, written $H_1 \perp H_2$, *means* $\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$.

*The combination of two heaps* written $H_1 \cdot H_2$, *is defined when* $H_1 \perp H_2$ *holds, and is the partial heap such that*: $\text{dom}(H_1 \cdot H_2) = \text{dom}(H_1) \cup \text{dom}(H_2)$, *and for all* $(o, f) \in \text{dom}(H_1 \cdot H_2)$ :

$$\begin{align*}
(H_1 \cdot H_2)[o, f] &= \begin{cases} 
H_1[o, f], & \text{if } (o, f) \in \text{dom}(H_1), \\
H_2[o, f], & \text{if } (o, f) \in \text{dom}(H_2).
\end{cases}
\end{align*}$$
Let $H$ be a heap and $R$ be a region. The restriction of $H$ to $R$, written $H \upharpoonright R$ is defined by:
\[
dom(H \upharpoonright R) = \dom(H) \cap R \text{ and } \forall (o,f) \in (H \upharpoonright R) :: (H \upharpoonright R)[o,f] = H[o,f].
\]
We use $\text{err}$ to denote an error region or an error heap; the restriction of a heap $H$ to an error region is defined by: $H \upharpoonright \text{err} = \text{err}.$

Notations: Let $f$ and $g$ be two partial functions. Then $f \preceq g$ means that $\dom(f) \subseteq \dom(g)$ and $\forall x \in \dom(f) :: f(x) = g(x)$. And $f \ast g$ means that $\dom(f) \cap \dom(g) = \emptyset$. The notation $f \cdot g$ means disjoint union, i.e., $\dom(f \cdot g) = \dom(f) \cup \dom(g)$, $\forall x \in \dom(f) :: f(x) = (f \cdot g)(x)$ and $\forall x \in \dom(g) :: g(x) = (f \cdot g)(x)$. Let $f : X \mapsto Y$ and $g : Y \mapsto Z$. Then $g \circ f : X \mapsto Z$, i.e., $\forall x \in X :: (g \circ f)(x) = g(f(x))$.

Fig. 2.3 on the next page shows the semantics of properly typed programming language, where $\mathcal{N}$ is the standard meaning function for numeric literals. The function $\text{MO}$ gives the semantics of operators. A $\text{Value}$ is either a Boolean, an object reference (which may be $\text{null}$), an integer or a set of locations: $\text{Value} = \text{Boolean} + \text{Object} + \text{Int} + \text{PowerSet}(\text{Loc})$. The auxiliary function $\text{fields}(T)$ takes a reference type $T$ and returns a list of its declared field names and their types.

The function $\text{type}(o)$ takes a reference $o$ and returns its type. Pure expressions evaluate to $\text{Values}$; thus the semantics of $E_1 = E_2$ and $E_1 \neq E_2$ have no need to check for errors. Region expressions evaluate to regions, i.e., sets of locations, and also cannot produce errors. For example, the region expression $\text{region}\{x.f\}$ is evaluated to an empty set when $x = \text{null}$. The pair $(\text{null}, f)$ is not allowed in the regions of our language’s semantics.

We consider the form $\text{skip}; S$ to be identical to $S$. In examples, $\textbf{if } E \textbf{ then } \{S\}$ is syntax sugar for $\textbf{if } E \textbf{ then } \{S\} \textbf{ else } \{\text{skip;}\}$.

A semantic function, $\mathcal{MS} : \text{Statement Typing Judgment} \to (\text{State} \to \text{State}_\perp)$, maps an input state to an output state, an error state, $\text{err}$, or $\perp$ (in case of infinite loops). The function, $\text{default}$,
Figure 2.3: The semantics of expressions

takes a type and returns its default value. The allocate function takes the heap and the class name as parameters, and returns a location and a new heap. An error happens, for example, when statements attempt to access a location not in the domain of the heap. The semantics does not have garbage collection and there is no explicit deallocation. The underlined lambda (λ) denotes a strict function that cannot recover from a nonterminating computation [81]. The semantics of statements are standard, and are defined in Fig. 2.4 on the following page.

The following lemma states that extending a type environment does not change the computation. The proof can be done by induction on the structure of the statement, and is easy. Thus, it is omitted. This lemma is used in proving Lemma 26 in Chapter 8.

**Lemma 1.** Let Γ and Γ’ be two well-formed type environments. Let S be a statement, such that Γ ⊢ S : ok(Γ’). Let Γ” be a well-formed type environment, such that dom(Γ’) ∩ dom(Γ”) = ∅ and
\[\mathcal{MS} : \text{Statement Typing Judgment} \to \text{State} \to \text{State}_\perp\]
\[\mathcal{MS}[\Gamma \vdash \text{skip} : \text{ok}(\Gamma)](\sigma, H) = (\sigma, H)\]
\[\mathcal{MS}[\Gamma \vdash \text{var } x : T ; \text{ok}(\Gamma, x : T)](\sigma, H) = (\sigma[x \mapsto \text{default}(T)], H)\]
\[\mathcal{MS}[\Gamma \vdash x := G ; \text{ok}(\Gamma)](\sigma, H) = (\sigma[x \mapsto \mathcal{E}[\Gamma \vdash G : T](\sigma)], H)\]
\[\mathcal{MS}[\Gamma \vdash x.f := G ; \text{ok}(\Gamma)](\sigma, H) =\]
\[\text{if } \sigma(x) \neq \text{null} \text{ then } (\sigma, H[(\sigma(x), f) \mapsto \mathcal{E}[\Gamma \vdash G : T](\sigma)]) \text{ else err}\]
\[\mathcal{MS}[\Gamma \vdash x := \text{new } T ; \text{ok}(\Gamma)](\sigma, H) = \text{let } (l, H') = \text{allocate}(T, H) \text{ in}\]
\[\text{let } (f_1, \ldots, f_n) = \text{fields}(T) \text{ in}\]
\[\sigma' = \sigma[x \mapsto l] \text{ in}\]
\[\sigma', H'[\sigma'(x), f_1] \mapsto \text{default}(T_1), \ldots, (\sigma'(x), f_n) \mapsto \text{default}(T_n)]\]
\[\mathcal{MS}[\Gamma \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 ; \text{ok}(\Gamma)](\sigma, H) =\]
\[\text{if } \mathcal{E}[\Gamma \vdash E : \text{bool}](\sigma) \text{ then } \mathcal{MS}[\Gamma \vdash S_1 : \text{ok}(\Gamma_1)](\sigma, H)\]
\[\text{else } \mathcal{MS}[\Gamma \vdash S_2 : \text{ok}(\Gamma_2)](\sigma, H)\]
\[\mathcal{MS}[\Gamma \vdash \text{while } E \{S\} ; \text{ok}(\Gamma)](\sigma, H) =\]
\[\text{fix}(\lambda g . \lambda s . \text{let } v = \mathcal{E}[\Gamma \vdash E : \text{bool}](\sigma) \text{ in}\]
\[\text{if } v \neq 0 \text{ then } \text{let } s' = \mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')](s) \text{ in } g \circ s'\]
\[\text{else } \text{if } v = 0 \text{ then } s \text{ else err}(\sigma, H)\]
\[\mathcal{MS}[\Gamma \vdash S_1 S_2 : \text{ok}(\Gamma_1)](\sigma, H) = \text{let } s' = \mathcal{MS}[\Gamma \vdash S_1 : \text{ok}(\Gamma'')](\sigma, H) \text{ in}\]
\[\text{if } s' \neq \text{err} \text{ then } \mathcal{MS}[\Gamma'' \vdash S_2 : \text{ok}(\Gamma'')](s') \text{ else err}\]

Figure 2.4: The semantics of statements

\[\text{dom}(\Gamma') \cap \text{dom}(\Gamma'') = \emptyset. \text{ Then}\]

1. if \(\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, h) \neq \text{err}\), then \(\mathcal{MS}[\Gamma, \Gamma'' \vdash S : \text{ok}(\Gamma', \Gamma'')](\sigma, h) \neq \text{err}\).

2. if \(\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, h) = (\sigma', h')\), then \(\mathcal{MS}[\Gamma, \Gamma'' \vdash S : \text{ok}(\Gamma', \Gamma'')](\sigma, h) = (\sigma', h')\).
CHAPTER 3: ASSERTION LANGUAGES AND FRAMING

This chapter formalizes the assertion language and effects using region expressions. And a framing judgment is defined in term of effects.

3.1 Syntax and Semantics of Assertions

The syntax of assertions is defined in Fig. 3.1. The first three are called atomic assertions. Quantification is restricted in the syntax. Quantified variables may denote an int, or a location drawn from a region.

\[
\begin{align*}
B & ::= E_1 = E_2 \mid E_1 \neq E_2 \\
P & ::= B \mid x.f = E \mid RE_1 \leq RE_2 \mid P_1 \& \& P_2 \mid P_1 \| P_2 \mid \neg P \\
& \mid \forall x:\text{int}::P \mid \forall x:T:\text{region}\{x.f\} \leq RE:P \mid \exists x:\text{int}::P \\
& \mid \exists x:T:\text{region}\{x.f\} \leq RE:P
\end{align*}
\]

Figure 3.1: The syntax of assertions

The typing rules for assertions are in Fig. A.3. The semantics of assertions is shown in Fig. 3.2 on the following page. The assertion \(RE_1 \leq RE_2\) checks that \(RE_1\) is a subregion of \(RE_2\). The assertion \(RE_1 \not\leq RE_2\) checks that \(RE_1\) and \(RE_2\) are disjoint. The semantics of assertions identifies errors (err) with false, and is two-valued. For example, \(x.f = 5\) is false if \(x.f\) is err. This design follows the works of Banerjee et al. [4].

The following lemma states that the value of a well-typed assertion in a given state is preserved

---

1The content in this chapter was partially presented at FTfJP ´15 [7]. And part of the content is submitted to Formal Aspects of Computing.
\[\sigma, H \vdash^\Gamma E_1 = E_2 \iff \mathcal{E}[\Gamma \vdash E_1 : T_1](\sigma) = \mathcal{E}[\Gamma \vdash E_2 : T_2](\sigma)\]

\[\sigma, H \vdash^\Gamma E_1 \neq E_2 \iff \mathcal{E}[\Gamma \vdash E_1 : T_1](\sigma) \neq \mathcal{E}[\Gamma \vdash E_2 : T_2](\sigma)\]

\[\sigma, H \vdash^\Gamma x.f = E \iff (\sigma(x), f) \in \text{dom}(H) \text{ and } H[\sigma(x), f] = \mathcal{E}[\Gamma \vdash E : T](\sigma)\]

\[\sigma, H \vdash^\Gamma \text{RE}_1 \leq \text{RE}_2 \iff \mathcal{E}[\Gamma \vdash \text{region}](\sigma) \subseteq \mathcal{E}[\Gamma \vdash \text{region}](\sigma)\]

\[\sigma, H \vdash^\Gamma \text{RE}_1 \not= \text{RE}_2 \iff \mathcal{E}[\Gamma \vdash \text{region}](\sigma) \cap \mathcal{E}[\Gamma \vdash \text{region}](\sigma) = \emptyset\]

\[\sigma, H \vdash^\Gamma P_1 \& \& P_2 \iff \sigma, H \vdash^\Gamma P_1 \text{ and } \sigma, H \vdash^\Gamma P_2\]

\[\sigma, H \vdash^\Gamma P_1 \mid P_2 \iff \sigma, H \vdash^\Gamma P_1 \text{ or } \sigma, H \vdash^\Gamma P_2\]

\[\sigma, H \vdash^\Gamma \neg P \iff \sigma, H \not\models P\]

\[\sigma, H \vdash^\Gamma \forall x : \text{int} :: P \iff \text{for all } v :: \sigma[x \mapsto v], H \models^\Gamma, x : \text{int} P\]

\[\sigma, H \vdash^\Gamma \forall x : T :: \text{region}(x,f) \leq \text{RE} : P \iff \text{for all } o : (o,f) \in \mathcal{E}[\Gamma \vdash \text{RE} : \text{region}](\sigma) \text{ and } \text{type}(o) = T : \sigma[x \mapsto o], H \models^\Gamma, x : T P\]

\[\sigma, H \vdash^\Gamma \exists x : \text{int} :: P \iff \text{exists } v :: \sigma[x \mapsto v], H \models^\Gamma, x : \text{int} P\]

\[\sigma, H \vdash^\Gamma \exists x : T :: \text{region}(x,f) \leq \text{RE} : P \iff \text{exists } o : (o,f) \in \mathcal{E}[\Gamma \vdash \text{RE} : \text{region}](\sigma) \text{ and } \text{type}(o) = T : (\sigma[x \mapsto o], H) \models^\Gamma, x : T P\]

Figure 3.2: The semantics of assertions

under type extension. This lemma is used in proving Lemma 26 in Chapter 8. The proof is done by induction on the assertion’s structure, and is omitted as it is intuitive.

**Lemma 2.** Let \(\Gamma\) and \(\Gamma'\) be two well-formed type environments. Let \((\sigma, H)\) be a \(\Gamma\)-state. Then \(\sigma, H \vdash^\Gamma P\) implies \((\sigma, H) \models^{\Gamma, \Gamma'} P\).

### 3.2 Effects

FRL uses effects to specify frame conditions and to frame formulas. The grammar for effects is given in Fig. 3.3 on the next page. The latter five forms are called atomic effects. The keyword **modifies** specifies write effects and **reads** specifies read effects. The keyword, **modifies** or **reads**, is omitted when the context is obvious, or when listing the same type effects, e.g.,

\((\text{modifies } x, \text{region}\{y.f\})\) is short for \((\text{modifies } x, \text{modifies region}\{y.f\})\). The effect **fresh**(RE) means all the locations in RE did not exist (were not allocated) in the pre-state. To
\[ \varepsilon, \delta ::= \emptyset \mid \varepsilon_1, \varepsilon_2 \mid E \ ? \ \varepsilon_1 : \varepsilon_2 \mid \text{reads } RE \mid \text{reads } x \mid \text{modifies } RE \]
\[ \mid \text{modifies } x \mid \text{fresh } (RE) \]

Figure 3.3: The grammar of effects

To avoid ambiguity, the notation \text{reads } x \downarrow means that reading the locations that are in \( x \), where \( x \) has type \text{region}; the notation \text{reads } x means that reading the variable \( x \). The write effects are similar.

Effects must be well-formed (wf) for the well-formed type environment \( \Gamma \); for example, \text{reads } x is meaningless if \( x \) is not in the domain of \( \Gamma \).

**Definition 2 (Well-formed Effects).** Let \( \Gamma \) be a well-formed type environment, and \( \delta \) be an effect. The effect \( \delta \) is well-formed in \( \Gamma \) if,

1. for all \( (M \ x) \in \delta :: x \in \text{dom}(\Gamma) \),
2. for all \( (M \ \text{region}\{x.f\}) \in \delta :: x \in \text{dom}(\Gamma) \), and
3. for all \( (M \ \text{region}\{x.*\}) \in \delta :: x \in \text{dom}(\Gamma) \),

where \( M \) is either \text{reads}, \text{modifies}, or \text{fresh}.  

A correct method must have an actual write effect that is a sub-effect of its specified effect.\(^\text{2}\) A set of subeffect rules is defined in Fig. 3.4 on the following page to reason about such cases; it encodes the standard properties of sets.

To streamline explanations, the following functions on effects are defined.

\(^\text{2}\)The sub-effect rules are also applicable for read effects.
\[ \vdash \gamma \varepsilon \leq \varepsilon \] 
\[ \vdash \gamma \varepsilon, \varepsilon' \leq \varepsilon', \varepsilon \]
\[ \varepsilon' \text{ is a write or read effect} \]
\[ \vdash \gamma \text{fresh } RE, \varepsilon \leq \varepsilon \]

\[ \text{false} \vdash \gamma \varepsilon \leq \varepsilon' \]
\[ P \vdash \gamma \varepsilon_1 \leq \varepsilon_2 \quad P \vdash \gamma \varepsilon_2 \leq \varepsilon_3 \]
\[ P' \Rightarrow P \quad P \vdash \gamma \varepsilon_1 \leq \varepsilon_2 \]
\[ P \vdash \gamma \varepsilon_1 \leq \varepsilon_2 \]
\[ P \vdash \gamma \varepsilon_1, \varepsilon \leq \varepsilon_2, \varepsilon \]
\[ \vdash \gamma \text{modifies } RE_1, RE_2 \leq \text{modifies } RE_1 + RE_2 \]
\[ \vdash \gamma \text{modifies filter}\{RE, T, f\} \leq \text{modifies } RE \]
\[ \vdash \gamma \text{modifies filter}\{RE, T\} \leq \text{modifies } RE \]
\[ RE_1 \leq RE_2 \vdash \gamma \text{modifies } RE_1 \leq \text{modifies } RE_2 \]
\[ RE_1 \leq RE_2 \vdash \gamma \text{reads } RE_1 \leq \text{reads } RE_2 \]
\[ \vdash \gamma E?\varepsilon_1 : \varepsilon_2 \leq \varepsilon_1, \varepsilon_2 \]
\[ P \&\& E_1 \&\& E_2 \vdash \gamma \varepsilon_1 \leq \varepsilon_3 \]
\[ P \&\& E_1 \&\& \neg E_2 \vdash \gamma \varepsilon_1 \leq \varepsilon_4 \]
\[ P \&\& \neg E_1 \&\& E_2 \vdash \gamma \varepsilon_2 \leq \varepsilon_3 \]
\[ P \&\& \neg E_1 \&\& \neg E_2 \vdash \gamma \varepsilon_2 \leq \varepsilon_4 \]
\[ P \vdash \gamma E_1?\varepsilon_1 : \varepsilon_2 \leq E_2?\varepsilon_3 : \varepsilon_4 \]

Figure 3.4: The sub-effect rules

- **writeR** discards all but region expressions in write effects; for example, \(\text{writeR(reads } x,\text{ modifies } y, \text{ modifies region}\{x.f\}\) is equal to \(\text{region}\{x.f\}\).

- **readR** discards all but region expressions in read effects; e.g., \(\text{readR(reads } x, \text{ reads region}\{x.f\}\) = \(\text{region}\{x.f\}\).

- **freshR** discards all but region expressions in fresh effects; e.g., \(\text{freshR(reads } x, \text{ modifies region}\{x.f\}, \text{ fresh region}\{y.*\}\) = \(\text{region}\{y.*\}\).

- **readVar** discards all but variables in read effects; for example, \(\text{readVar(reads } x, \text{ reads }\)
region\{x.f\}) = x.

- \textit{regRW} unites all the region expressions in both read and write effects; for example,
  \textit{regRW}(\textit{reads } x, \textit{modifies region\{x.f\}}, \textit{reads region\{y.f\}}) = \textit{region\{x.f\}} + \textit{region\{y.f\}}.

Write effects and fresh effects make sense for two consecutive states, written \((\sigma, h) \leftrightarrow (\sigma', h')\). The following defines the semantics of write effects and fresh effects. It allows changes for variables and in the heaps of the pre-state. However, fresh effects are evaluated in the post-state. Regions specified in a fresh effect do not exist in the pre-state.

\textbf{Definition 3} (Changes allowed by write and fresh effects). Let \(\Gamma\) be a well-formed type environment. Let \(\varepsilon\) be well-formed effects in \(\Gamma\), and \((\sigma, h)\) and \((\sigma', h')\) be \(\Gamma\)-states. The effect \(\varepsilon\) allows change from \((\sigma, h)\) to \((\sigma', h')\), written \((\sigma, h) \rightarrow (\sigma', h') \models\varepsilon\) if and only if \((\sigma, h) \leftrightarrow (\sigma', h')\) and the following holds:

1. for all \(x \in \text{dom}(\Gamma)\), either \(\sigma(x) = \sigma'(x)\) or \textit{modifies } \(x\) is in \(\varepsilon\);

2. for all \((o, f) \in \sigma(\text{alloc})\), either \(h[o, f] = h'[o, f]\) or there is \(\text{RE}\) such that \textit{modifies } \(\text{RE}\) is in \(\varepsilon\), such that \((o, f) \in \mathcal{E}[\Gamma \vdash \text{region}](\sigma)\);

3. for all \(\text{RE}\) such that \textit{fresh } \(\text{RE}\) is in \(\varepsilon\), it must be true that \(\mathcal{E}[\Gamma \vdash \text{region}](\sigma') \subseteq (\sigma'(\text{alloc}) - \sigma(\text{alloc}))\).

Def. 4 says that if two states agree on a read effect, \(\delta\), then the values that depend on \(\delta\) are identical. As the programming language defined so far does not have subclassing or subtyping, a variable’s static type is also its dynamic type. There is no need to state that types are congruent in the two states. This definition is generalized in Chapter 8, where the language is extended with inheritance.
Definition 4 (Agreement on Read Effects). Let $\Gamma$ be a well-formed type environment. Let $\delta$ be a well-formed effect in $\Gamma$. Let $\Gamma' \geq \Gamma$ and $\Gamma'' \geq \Gamma$. Let $(\sigma', h')$ and $(\sigma'', h'')$ be a $\Gamma'$-state and a $\Gamma''$-state respectively. Then it is said that $(\sigma', h')$ and $(\sigma'', h'')$ agree on $\delta$, written $(\sigma', h') \equiv (\sigma'', h'')$, when the following holds:

1. for all $\textbf{reads } x \in \delta : \sigma'(x) = \sigma''(x)$

2. for all $\textbf{reads RE} \in \delta : (o, f) \in E[\Gamma' \vdash RE : \textbf{region}](\sigma') : h'[o, f] = h''[o, f]$.

3.3 Framing

Let $R$ be the region that the frame condition of a method, $m$, specifies in a given state; these locations may be modified by $m$. The locations that are preserved are the complement of $R$, written $\bar{R}$. Let $R'$ be locations that may be used in evaluating an assertion, $P$, written $\textbf{reads } R' \textbf{ frm } P$. If $R' \subseteq \bar{R}$, i.e., $R' !\subseteq R$, then $P$’s validity is preserved after $m$ is called. The function $efs(-)$ shown in Fig. 3.5 inductively defines $R'$ for expressions, region expressions, and atomic assertions. For example, $efs(x.f = y) = \textbf{reads } x, \textbf{region}\{x.f\}, y$.

\[
\begin{align*}
efs(x) = \textbf{reads } x & \quad \text{efs}(n) = \emptyset & \quad \text{efs}(null) = \emptyset \\
efs(E_1 + E_2) = efs(E_1), efs(E_2) & \quad \text{efs}(\textbf{region}\{\}) = \emptyset \\
efs(\textbf{region}\{x.f\}) = \textbf{reads } x & \quad \text{efs}(\textbf{region}\{x.*\}) = \textbf{reads } x \\
efs(E ? RE_1 : RE_2) = efs(E), E ? efs(RE_1) : efs(RE_2) & \\
efs(\textbf{filter}\{RE, f\}) = efs(RE) & \quad \text{efs}(\textbf{filter}\{RE, T, f\}) = efs(RE) \\
efs(RE_1 \odot RE_2) = efs(RE_1), efs(RE_2) & \quad efs(E_1 = E_2) = efs(E_1), efs(E_2) \\
efs(E_1 \neq E_2) = efs(E_1), efs(E_2) & \quad efs(x.f = E) = \textbf{reads } x, \textbf{region}\{x.f\}, efs(E) \\
efs(RE_1 \leq RE_2) = efs(RE_1), efs(RE_2) & \quad efs(RE_1 ?! RE_2) = efs(RE_1), efs(RE_2)
\end{align*}
\]

Figure 3.5: The read effects of expressions, region expressions and atomic assertions
The framing judgment, $P \vdash \Gamma \delta \text{frm} Q$, means that read effects, $\delta$, contains the variables and locations that are needed to evaluate $Q$ in a $\Gamma$-state that satisfies $P$. Fig. 3.6 shows the judgment for assertions.

$$
\text{(FrmFipt)}
\begin{align*}
\text{true} & \vdash \Gamma \text{efs}(P) \text{frm} P \\
\text{where } P & \text{ is atomic}
\end{align*}
$$

$$
\text{(FrmNeg)}
\begin{align*}
\text{true} & \vdash \Gamma \text{efs}(P) \text{frm} \neg P \\
\text{where } P & \text{ is atomic}
\end{align*}
$$

$$
\text{(FrmSub)}
\begin{align*}
R & \vdash \Gamma \delta_1 \text{frm} Q \\
& \implies \\
Q & \vdash \Gamma \delta_2 \leq \delta_1
\end{align*}
$$

$$
\text{where } P \Rightarrow R
$$

$$
\text{(FrmConj)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} Q_1 \\
& \implies \\
& \\
P & \vdash \Gamma \delta \text{frm} Q_2 \\
\text{where } Q_1 & \iff Q_2
\end{align*}
$$

$$
\text{(FrmDisj)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} Q_1 \\
& \implies \\
& \\
P & \vdash \Gamma \delta \text{frm} Q_2 \\
\end{align*}
$$

$$
\text{where } Q_1 \lor Q_2
$$

$$
\text{(FrmEq)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} Q_1 \\
& \implies \\
& \\
P & \vdash \Gamma \delta \text{frm} Q_2 \\
\text{where } Q_1 & \iff Q_2
\end{align*}
$$

$$
\text{(FrmProjCtx)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} Q \\
& \implies \\
\text{reads } x & \vdash \Gamma \delta \text{frm} Q \\
\text{reads } x & \vdash \Gamma \delta \text{frm} \forall x : \text{int} :: Q
\end{align*}
$$

$$
\text{(FrmEq\_1)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} Q \\
& \implies \\
\text{reads } x & \vdash \Gamma \delta \text{frm} \forall x : \text{int} :: Q
\end{align*}
$$

$$
\text{(FrmEq\_2)}
\begin{align*}
P & \vdash \Gamma \text{efs}(RE) \leq \delta \\
& \implies \\
P & \vdash \Gamma \text{efs}(\text{region}\{x.f\}) \leq \text{region}\{x.f\} \leq \text{region}\{x.f\} \\
& \leq \text{efs}(\text{region}\{x.f\}) \\
& \leq \text{region}\{x.f\} \\
& \leq \text{efs}(\text{region}\{x.f\})
\end{align*}
$$

$$
\text{(FrmEq\_3\_1)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} \exists x : \text{int} :: Q \\
& \implies \\
\text{reads } x & \vdash \Gamma \delta \text{frm} \exists x : \text{int} :: Q
\end{align*}
$$

$$
\text{(FrmEq\_3\_2)}
\begin{align*}
P & \vdash \Gamma \text{efs}(RE) \leq \delta \\
& \implies \\
P & \vdash \Gamma \text{efs}(\text{region}\{x.f\}) \leq \text{region}\{x.f\} \\
& \leq \text{efs}(\text{region}\{x.f\}) \\
& \leq \text{region}\{x.f\} \\
& \leq \text{efs}(\text{region}\{x.f\})
\end{align*}
$$

$$
\text{(FrmEq\_3\_3)}
\begin{align*}
P & \vdash \Gamma \delta \text{frm} \exists x : \text{int} :: Q \\
& \implies \\
\text{reads } x & \vdash \Gamma \delta \text{frm} \exists x : \text{int} :: Q
\end{align*}
$$

Figure 3.6: The inference rules for the framing judgment

The following defines the meaning of a framing judgment.

**Definition 5** (Frame Validity). Let $\Gamma$ be a well-formed type environment. Let $P$ and $Q$ be asser-
tions, and \( \delta \) be a read effect. The framing judgment \( P \vdash^\Gamma \delta \text{ frm } Q \) is valid, written \( P \vdash^\Gamma \delta \text{ frm } Q \), if and only if for all \( \Gamma \)-states \( (\sigma, h) \) and \( (\sigma', h') \), if \( (\sigma, h) \vdash^\delta (\sigma', h') \) and \( \sigma, h \vdash^\Gamma P \land \& \& Q \), then \( \sigma', h' \vdash^\Gamma Q \).

The framing judgment is stateful. For example, the judgment \( x = y \vdash^\Gamma (\text{reads } y, \text{region } \{x.f\}) \text{ frm } (x.f = 5) \) is valid, but \( \vdash^\Gamma (\text{reads } y, \text{region } \{x.f\}) \text{ frm } (x.f = 5) \) may not.

**Lemma 3** (Framing Soundness for Expressions). Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) and \( (\sigma', h') \) be two \( \Gamma \)-states. Let \( G \) be an expression. If \( (\sigma, h) \vdash^G (\sigma', h') \), then it must be true that \( E[\Gamma \vdash G : T](\sigma) = E[\Gamma \vdash G : T](\sigma') \).

**Proof.** The proof is straightforward by structural induction on expressions, thus is omitted. \( \square \)

**Lemma 4** (Framing Soundness for Assertions). Every derivable framing judgment is valid.

**Proof.** By induction on a derivation of a framing judgment \( P \vdash^\Gamma \delta \text{ frm } Q \). \( \square \)

### 3.4 Separator and Immune

The notation, \( / \), is used to define the disjointness on effects in Fig. 3.7 on the following page, where \( \delta \) is a read effect and \( \varepsilon \) is a write effect. \( \delta / \varepsilon \) means that the read effects in \( \delta \) are disjoint with the write effects in \( \varepsilon \). The effect, \( \text{reads } \delta \), where \( \delta \) is not a conditional effect, is treated as \( \text{reads if } \text{true then } \delta \text{ else } \emptyset \). For example, let \( RE \) be \( \text{if } x.f = 0 \text{ then region } \{y.f\} \text{ else region } \{\} \). Suppose \( x \neq y \) and \( x.f \neq 0 \). The separation of \( \text{reads region } \{y.f\} \) and \( \text{modifies } RE \) can be derived to \( \text{reads region } \{y.f\} / \text{modifies region } \{\} \) by the rule ConMask introduced in the next section.
\begin{align*}
\delta \vdash \emptyset &= \text{true} \\
\emptyset \vdash \varepsilon &= \text{true} \\
\text{reads } y \vdash \text{modifies } x &= y \neq x \\
\text{reads } y \vdash \text{modifies } RE &= \text{true} \\
\text{reads } RE_1 \vdash \text{modifies } x &= \text{true} \\
\text{reads } RE_1 \vdash \text{modifies } RE_2 &= RE_1 \neq ! \! RE_2 \\
\delta \vdash (\varepsilon, \varepsilon') &= (\delta \vdash \varepsilon) \land (\delta' \vdash \varepsilon') \\
(\delta, \delta') \vdash \varepsilon &= (\delta \vdash \varepsilon) \land (\delta' \vdash \varepsilon) \\
\delta \vdash (E \ ? \ \varepsilon_1 : \varepsilon_2) &= \begin{cases} 
\delta \vdash \varepsilon_1 & \text{if } E \\
\delta \vdash \varepsilon_2 & \text{if } \neg E 
\end{cases} \\
(E \ ? \ \delta_1 : \delta_2) \vdash \varepsilon &= \begin{cases} 
\delta_1 \vdash \varepsilon & \text{if } E \\
\delta_2 \vdash \varepsilon & \text{if } \neg E 
\end{cases}
\end{align*}

Figure 3.7: The definition of separator

The following lemma says if read effects, \(\delta\), and write effects, \(\varepsilon\) are separate, then the values on \(\delta\) are preserved.

**Lemma 5** (Separator Agreement). Let \(\Gamma\) be a well-formed type environment. Let \(\varepsilon\) and \(\delta\) be effects in \(\Gamma\). Let \((\sigma, h)\) and \((\sigma', h')\) be two \(\Gamma\)-states. If \((\sigma, h) \rightarrow (\sigma', h') \models^\Gamma \varepsilon\) and \((\sigma, h) \models^\Gamma \delta \vdash \varepsilon\), then \((\sigma, h) \equiv (\sigma', h')\).

**Proof.** According to the definition of agreement on read effects (Def. 4), there are two cases.

1. Let \textbf{reads } \(x\) in \(\delta\) be arbitrary. Since \((\sigma, H) \models^\Gamma \delta \vdash \varepsilon\), \textbf{modifies } \(x \neq \varepsilon\). By the assumption \((\sigma, h) \rightarrow (\sigma', h') \models^\Gamma \varepsilon\) and the definition of changes allowed by write and fresh effects (Def. 3), it must be true that \(\sigma (x) = \sigma' (x)\).

2. Let \textbf{reads } \(RE\) in \(\delta\) be arbitrary. It is to show that for all \((o, f) \in E[\Gamma \vdash RE : \text{region}] (\sigma), h[o, f] = h'[o, f]\). By the definition of separator (Fig. 3.7), for any \textbf{modifies } \(RE'\) in \(\varepsilon\), it must be true that \(\sigma, h \models^\Gamma \delta \vdash \varepsilon\). By the assumption that \(\sigma, h \models^\Gamma \delta \vdash \varepsilon\), it must be
true that $(o, f) \notin \mathcal{E}[\Gamma \vdash RE' : \texttt{region}](\sigma, h)$. So by the definition of changes allowed by write and fresh effects (Def. 3), it must be true that $h[o, f] = h'[o, f]$.

To prevent interference of the effects of two sequential statements, immunity of two effects under certain condition is introduced. Consider the statement: $x := y; x.f := 5$. The write effect of the first statement is \texttt{modifies} $x$, and that of the second statement is \texttt{region}\{x, f\}. The effect of their composition is not necessarily \texttt{modifies} $(x, \texttt{region}\{x, f\})$, as \texttt{region}\{x.f\} may denote different locations after $x$ is assigned to the value of variable $y$. To reason about this example, a rule of state-dependent effect subsumption is used, ascribing to $x.f := 5$ the effect \texttt{modifies region}\{y, f\} which is sound owing to the postcondition of $x := y$, which is $x = y$. The effect \texttt{modifies region}\{y, f\} is immune from updating $x$. Immunity is used in the proof of Theorem 1 on page 49.

\textbf{Definition 6 (Immune).} Let $RE$ be a region expression, $P$ be an assertion, and $\varepsilon$ and $\delta$ be two effects. Then $RE$ is immune from $\varepsilon$ under $P$, written $RE$ is $P/\varepsilon$-immune, if and only if $P$ implies $\texttt{efs}(RE)/\varepsilon$.

Effect $\delta$ is immune from $\varepsilon$ under $P$, if and only if for all \texttt{modifies} $RE$ in $\delta :: RE$ is $P/\varepsilon$-immune.

This notion is used to prevent naive accumulation of write effects. To explain this, let $\varepsilon_1$ and $\varepsilon_2$ be the two write effects of two sequential statements. Intuitively, if the variables and regions that $\varepsilon_1$ contains overlap with the variables and regions that $\varepsilon_2$ depends on, then $\varepsilon_2$ is not $\varepsilon_1$-immune.

Consider the example $xf := x; xf := x;$. Assume the precondition of the first update statement is $x \neq \texttt{null}$. The write effects of both update statements, $\varepsilon_1$ and $\varepsilon_2$, are \texttt{modifies region}\{x, f\}.
The proof obligation is to show that $\varepsilon_2$ is $x \neq \text{null}/\varepsilon_1$-immune. Informally, the write effect $\varepsilon_2$ relies on the variable $x$. But, the write effect $\varepsilon_1$ does not contain $\text{modifies } x$. Therefore, $\text{modifies region}\{x.f\}$ is $x \neq \text{null}/\text{modifies region}\{x.f\}$-immune. A proof of this is calculated as follows.

\[
\text{modifies region}\{x.f\} \text{ is } x \neq \text{null} / \text{modifies region}\{x.f\} \text{-immune}
\]

\[
\iff \langle \text{by the definition of Immune (Def. 6)} \rangle
\]

\[
\text{for all} \text{modifies } RE \text{ in } \text{modifies region}\{x.f\} :: RE \text{ is }
\]

\[
x \neq \text{null} / \text{modifies region}\{x.f\} \text{-immune}
\]

\[
\iff \langle \text{by } RE \text{ is } \text{region}\{x.f\} \rangle
\]

\[
\text{region}\{x.f\} \text{ is } x \neq \text{null} / \text{modifies region}\{x.f\} \text{-immune}
\]

\[
\iff \langle \text{by the definition of Immune (Def. 6)} \rangle
\]

\[
x \neq \text{null} \text{ implies } \text{efs} (\text{region}\{x.f\}) / \text{modifies region}\{x.f\}
\]

\[
\iff \langle \text{by the definition of read effects (Fig. 3.5)} \rangle
\]

\[
x \neq \text{null} \text{ implies } \text{reads } x / \text{modifies region}\{x.f\}
\]

\[
\iff \langle \text{by the definition of separator (Fig. 3.7)} \rangle
\]

\[
\text{true}
\]

However, note that if the first statement were $x := y$, then the effect $\text{modifies region}\{x.f\}$ would not be $x \neq \text{null}/\text{modifies } x$-immune.
To make a comparison, consider another example $x.f.g := x; x.f := x$. (This is not syntactically correct, but one can desugar it to $z := x.f; z.g := x; x.f := x$, where $z$ is fresh.) Assume the precondition of the first update statement is $x \neq \text{null} \&\& x.f \neq \text{null}$. In this case, $\varepsilon_1$ is modifies region$\{x.f.g\}$, and $\varepsilon_2$ is modifies region$\{x.f\}$. The following shows that modifies region$\{x.f.g\}$ is $x \neq \text{null}$/modifies region$\{x.f\}$-immune is false.

\[
\text{modifies region}\{x.f.g\} \text{ is } x \neq \text{null}/\text{modifies region}\{x.f\}-\text{immune}
\]

iff \( \langle \text{by the definition of Immune (Def. 6)} \rangle \)

for all modifies\ RE \in\ \text{modifies region}\{x.f.g\} ::\ \text{RE} \text{ is}

\( x \neq \text{null}/\text{modifies region}\{x.f\}-\text{immune} \)

iff \( \langle \text{by } \text{RE} \text{ is } \text{region}\{x.f.g\} \rangle \)

\( \text{region}\{x.f.g\} \text{ is } x \neq \text{null}/\text{modifies region}\{x.f\}-\text{immune} \)

iff \( \langle \text{by the definition of Immune (Def. 6)} \rangle \)

\( x \neq \text{null} \text{ implies efs}(\text{region}\{x.f.g\})/\text{modifies region}\{x.f\} \)

iff \( \langle \text{by the definition of read effects (Fig. 3.5 on page 29)} \rangle \)

\( x \neq \text{null} \text{ implies } efs(x, \text{region}\{x.f\})/\text{modifies region}\{x.f\} \)

iff \( \langle \text{by the definition of separator (Fig. 3.7 on page 32)} \rangle \)

false

**Lemma 6.** Let $\Gamma$ be a well-formed type environment. Let $\varepsilon$ be an effect, RE be a region expression, and $P$ be an assertion, such that $RE$ is $P/\varepsilon$-immune. Then for all $\Gamma$-states, $(\sigma, h)$ and $(\sigma', h')$, such that $(\sigma, h) \rightarrow (\sigma', h') \models_{\Gamma} \varepsilon$, if $\sigma, h \models_{\Gamma} P$, then $E[\Gamma \vdash \text{RE} : \text{region}] (\sigma) = E[\Gamma \vdash \text{RE} : \text{region}] (\sigma')$.

**Proof.** By the assumption that $RE$ is $P/\varepsilon$-immune, and by the definition of immunity (Def. 6), it must be true that $\sigma, h \models_{\Gamma} P \Rightarrow efs(RE)/\varepsilon$. By the assumption that $(\sigma, h) \rightarrow (\sigma', h') \models_{\Gamma} \varepsilon$ and by the separator agreement (Lemma 5), it must be true that $(\sigma, h)^{efs(RE)} \equiv (\sigma', h')$. Then by the
soundness of framing for expressions (Lemma 3), it must be true that $E[\Gamma \vdash RE : \text{region}](\sigma) = E[\Gamma \vdash RE : \text{region}](\sigma')$. 

The effects of a sequence statement $S_1S_2$ are gained from the effects of the two constituent statements, $S_1$ and $S_2$, where the write effect of $S_2$ may contain regions that are allocated by $S_1$. Then such write effect can be dropped by the effect of $S_1S_2$. This idea is described in the following lemma, which is used in proving the soundness theorem (Theorem 1).

**Lemma 7 (Effect Transfer).** Let $\Gamma_0$, $\Gamma_1$ and $\Gamma_2$ be well-formed type environments. Let $(\sigma_0, h_0)$, $(\sigma_1, h_1)$, $(\sigma_2, h_2)$ be $\Gamma_0$, $\Gamma_1$ and $\Gamma_2$-states respectively. Let $\varepsilon_1$ and $\varepsilon_2$ be two effects, and $P$ and $P'$ be two assertions. If the following hold:

1. $\sigma_0, h_0 \vdash_{\Gamma_0} P$ and $\sigma_1, h_1 \vdash_{\Gamma_1} P'$;

2. $(\sigma_0, h_0) \rightarrow (\sigma_1, h_1) \vdash_{\Gamma_0} \varepsilon_1$;

3. $(\sigma_1, h_1) \rightarrow (\sigma_2, h_2) \vdash_{\Gamma_1} \varepsilon_2, \text{modifies } RE$;

4. $\varepsilon_2$ is $P/\varepsilon_1$-immune;

5. for all fresh(RE) $\in \varepsilon_1 :: RE$ is $P/(\varepsilon_2, \text{modifies } RE)$-immune;

6. $E[\Gamma_1 \vdash RE : \text{region}](\sigma_1, h_1) \cap \sigma_0(\text{alloc}) = \emptyset$.

Then $(\sigma_0, h_0) \rightarrow (\sigma_2, h_2) \vdash_{\Gamma_0} \varepsilon_1, \varepsilon_2$.

**Proof.** To prove the conclusion, it needs to show that all the conditions defined in Def. 3 hold.

For condition (1) in Def. 3, let $x$ be a variable, such that $\sigma_0(x) \neq \sigma_2(x)$. It is the case that either $\sigma_0(x) \neq \sigma_1(x)$ or $\sigma_1(x) \neq \sigma_2(x)$ or both. By the assumption 2 and 3, modifies $x$ is either in $\varepsilon_1$ or in $\varepsilon_2$ or both.
For condition (2) in Def. 3, let \((o, f) \in \sigma_0\texttt{alloc}\), such that \(h_0[o, f] \neq h_2[o, f]\). There are two cases:

1. \(h_0[o, f] \neq h_1[o, f]\): By assumption 2, there is a region expression \(RE_0\), such that \texttt{modifies} \(RE_0\) in \(\varepsilon_1\) and \((o, f) \in \mathcal{E}[\Gamma_0 \vdash RE_0 \text{ region}](\sigma_0, h_0)\). Thus, \texttt{modifies} \(RE_0\) in \((\varepsilon_1, \varepsilon_2)\).

2. \(h_1[o, f] \neq h_2[o, f]\): By assumption 2, there are two cases.

   - there is a region expression \(RE'\) in \(\varepsilon_2\) and \((o, f) \in \mathcal{E}[\Gamma_1 \vdash RE'](\sigma_1, h_1)\). By assumption (4) and Lemma 6, it is true that \(\mathcal{E}[\Gamma_0 \vdash RE'](\sigma_0, h_0) = \mathcal{E}[\Gamma_1 \vdash RE'](\sigma_1, h_1)\). Thus, \texttt{modifies} \(RE'\) in \((\varepsilon_1, \varepsilon_2)\).

   - \((o, f) \in \mathcal{E}[\Gamma_1 \vdash RE](\sigma_1, h_1)\). By assumption (6), \(\mathcal{E}[\Gamma_1 \vdash RE](\sigma_1, h_1) \cap \sigma_0(\texttt{alloc}) = \emptyset\). That implies \((o, f) \notin \sigma_0(\texttt{alloc})\), which contradicts the assumption that \((o, f) \in \sigma_0(\texttt{alloc})\).

For condition (3) in Definition 3, let \(r_1 = \mathcal{E}[\Gamma_1 \vdash RE](\sigma_1, h_1)\) and \(r_2 = \mathcal{E}[\Gamma_2 \vdash RE](\sigma_2, h_2)\). There are two cases:

1. Suppose \texttt{fresh(}RE\texttt{)} \(\in \varepsilon_1\). By assumption 2, it is true that \(r_1 \subseteq (\sigma_1(\texttt{alloc}) - \sigma_0(\texttt{alloc}))\) and \((\sigma_1(\texttt{alloc}) \subseteq \sigma_2(\texttt{alloc}))\). By assumption 3, \(r_1 = r_2\). Thus, it is true that \(r_2 \subseteq (r_2 - \sigma_0(\texttt{alloc}))\). So \((\sigma_0, h_0) \rightarrow (\sigma_1, h_1) \models_{\Gamma_0} \varepsilon_1, \varepsilon_2\).

2. Suppose \texttt{fresh(}RE\texttt{)} \(\in \varepsilon_2\). By assumption 2, it is true that \(r_2 \subseteq (\sigma_2(\texttt{alloc}) - \sigma_1(\texttt{alloc}))\), and \(\sigma_1(\texttt{alloc}) \subseteq \sigma_0(\texttt{alloc})\). Thus, it is true that \(r_2 \subseteq (r_2 - \sigma_0(\texttt{alloc}))\). So \((\sigma_0, h_0) \rightarrow (\sigma_2, h_2) \models_{\Gamma_0} \varepsilon_1, \varepsilon_2\).

\[\square\]
CHAPTER 4: FINE-GRAINED REGION LOGIC

This chapter defines the correctness judgment in FRL, and presents the proof axioms and rules for statements and structural rules.

The correctness judgment of FRL, a Hoare-formula of form \( \{P_1\} S \{P_2\}[\varepsilon] \), means that \( S \) is partially correct, its write effects are contained in \( \varepsilon \), and the locations specified to be fresh in \( \varepsilon \) are newly allocated. Following the work on RL [2, 4] a statement \( S \) is partially correct if it cannot encounter an error when started in a pre-state satisfying the specified precondition, however \( S \) may still loop forever.

**Definition 7** (Valid FRL Hoare-Formula). Let \( \Gamma \) be a well-formed type environment. Let \( S \) be a statement, let \( P_1 \) and \( P_2 \) be assertions, let \( \varepsilon \) be effects, and let \( (\sigma, H) \) be a \( \Gamma \)-state. Then \( \{P_1\} S \{P_2\}[\varepsilon] \) is valid in \( (\sigma, H) \), written \( \sigma, H \models_{\Gamma} \{P_1\} S \{P_2\}[\varepsilon] \), if and only if whenever \( \sigma, H \models_{\Gamma} P_1 \), then

1. \( \mathcal{M} S[\Gamma] \models S : ok(\Gamma') \] (\( \sigma, H \) \( \neq \) err);

2. if \( (\sigma', H') = \mathcal{M} S[\Gamma] \models S : ok(\Gamma') \] (\( \sigma, H \)), then

   \( \sigma', H' \models_{\Gamma'} P_2 \)

   - for all \( x \in \text{dom}(\sigma) : \sigma'(x) \neq \sigma(x) \) : modifies \( x \in \varepsilon \)

   - for all \( (o, f) \in \text{dom}(H) : H'[o, f] \neq H[o, f] \) :

     \((o, f) \in \mathcal{E}[\Gamma] \models \text{writeR}(\varepsilon) : \text{region}] \] (\( \sigma \)), and

   - for all \( (o, f) \in \mathcal{E}[\Gamma'] \models \text{freshR}(\varepsilon) : \text{region}] \] (\( \sigma' \)): (\( o, f \) \( \in \) (\( \text{dom}(H') \) \( - \) \( \text{dom}(H) \)) \( . \)

\(^1\)The content in this chapter is submitted to Formal Aspects of Computing.
A Hoare-formula \( \{ P_1 \} S \{ P_2 \}[\varepsilon] \) is valid, written \( \vdash^r_\Gamma \{ P_1 \} S \{ P_2 \}[\varepsilon] \), if and only if for all \( \Gamma \)-states \( (\sigma, H) :: \sigma, H \vdash^r_\Gamma \{ P_1 \} S \{ P_2 \}[\varepsilon] \).

Note that the region expressions in the write effects are evaluated in the pre-state, since frame conditions only specify changes to pre-existing locations, not changes to freshly allocated ones. On the other hand, the region expressions in the fresh effects are evaluated in the post-state. Note that write effects are permissions to change locations, as write effects may leave the values in locations unchanged, but specified fresh effects are indeed obligations.

Write effects in FRL can specify both variables (in stores) and heap regions. Write effects do not restrict a statement’s access to the heap, since in FRL statements can implicitly access all of the program’s heap, whose domain is written \texttt{alloc}.

### 4.1 Axioms and Inference Rules

Fig. 4.1 on the following page shows the axioms and inference rules for statements. The predicate \texttt{true} is syntactic sugar for \( 1 = 1 \). The axioms for variable assignment, field access, field update and allocation are “small” [71] in the sense that the union of write effects and read effects describe the least upper bound of variables and locations that \( S \) accesses, and the write effects describe the least upper bound of the variables and locations that \( S \) may modify. The fresh effects in the rule of the new statement accounts to a newly allocated object. Fig. 4.2 and Fig. 4.3 show the structural rules. In the rules, the notation, \texttt{new}(\( T, x \)), means that \( x.f_1 = \text{default}(T_1) \& \& \ldots \& \& x.f_n = \text{default}(T_n) \), where the \( f_i : T_i \) are defined by \( (f_1 : T_1, \ldots, f_n : T_n) = \text{fields}(T) \).
(\textit{SKIP}_r)
\[ \vdash^r_r \{ \text{true} \} \text{skip}; \{ \text{true} \}[\varnothing] \]

(\textit{VAR}_r)
\[ \vdash^r_r \{ \text{true} \} \text{var } x : T; \{ x = \text{default}(T) \}[\varnothing] \]

(\textit{ALLOC}_r)
\[ \vdash^r_r \{ \text{true} \} x := \text{new } T; \{ \text{new}(T, x) \} \]
\[ \quad \text{modifies } x, \text{modifies alloc} \text{. fresh(region\{x.*\})} \]

(\textit{ASGN}_r)
\[ \vdash^r_r \{ \text{true} \} x := G; \{ x = G \} \text{[modifies } x \text{] where } x \notin \text{FV}(G) \]

(\textit{ACC}_r)
\[ \vdash^r_r \{ x' \neq \text{null} \} x := x'.f; \{ x = x'.f \} \text{[modifies } x \text{] where } x \neq x' \]

(\textit{UPD}_r)
\[ \vdash^r_r \{ x \neq \text{null} \} x.f := G; \{ x.f = G \} \text{[modifies region\{x.f\}] \]

(\textit{SEQ}_r)
\[ \begin{array}{c}
\vdash^r_r \{ P \} \text{S}_1 \{ P_1 \}[\varepsilon_1, \text{fresh(RE)}] \quad \vdash^r_r \{ P_1 \} \text{S}_2 \{ P' \}[\varepsilon_2, \text{modifies RE}_1] \\
\hline
\vdash^r_r \{ P \} \text{S}_1 \text{S}_2 \{ P' \}[\varepsilon_1, \varepsilon_2, \text{fresh(RE)}] \\
\end{array} \]
\[ \text{where } S_1 \neq \text{var } x : T; \varepsilon_1 \text{ is fresh-free, } \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune,} \]
\[ \text{RE is } P_1/(\varepsilon_2, \text{modifies } RE_1)\text{-immune and } P_1 \Rightarrow RE_1 \leq RE \]

(\textit{SEQ2}_r)
\[ \begin{array}{c}
\vdash^r_r x : T \{ P \& \& x = \text{default}(T) \} S \{ Q \}[\text{modifies } x, \varepsilon] \\
\hline
\vdash^r_r \{ P \} \text{var } x : T; S \{ Q \}[\varepsilon] \\
\end{array} \]

(\textit{IF}_r)
\[ \begin{array}{c}
\vdash^r_r \{ P \& \& E \} \text{S}_1 \{ P' \}[\varepsilon] \quad \vdash^r_r \{ P \& \& \neg E \} \text{S}_2 \{ P' \}[\varepsilon] \\
\hline
\vdash^r_r \{ P \} \text{if } E \text{ then } \{ S_1 \} \text{ else } \{ S_2 \} \{ P' \}[\varepsilon] \\
\end{array} \]

(\textit{WHILE}_r)
\[ \begin{array}{c}
\vdash^r_r \{ P \& \& E \} \text{S} \{ P \}[\varepsilon, \text{modifies RE}] \\
\hline
\vdash^r_r \{ P \& \& r = \text{alloc} \} \text{while } E \{ S \} \{ P \& \& \neg E \}[\varepsilon] , \\
\end{array} \]
\[ \text{where } P \Rightarrow RE \uparrow r, \varepsilon \text{ is fresh-free, } \varepsilon \text{ is } P/\varepsilon\text{-immune, and } \text{modifies } r \notin \varepsilon \]

Figure 4.1: The correctness axioms and proof rules for statements in FRL
(FRM r)
\[
\frac{\Gamma \vdash \forall r \delta \text{ frm } Q}{\Gamma \vdash \forall r \{P \&\& Q\}[\varepsilon]} \]
\text{where } P \&\& Q \Rightarrow \delta / \varepsilon

(SUBEFF r)
\[
\frac{\Gamma \vdash \forall r \{P\}[\varepsilon]}{\Gamma \vdash \forall r \{P\}[\varepsilon']} \]
\text{where } P \Rightarrow P' \text{ and } \Gamma \vdash \forall r \{P\}[\varepsilon']

(CONSEQ r)
\[
\frac{\Gamma \vdash \forall r \{P_1\}[\varepsilon]}{\Gamma \vdash \forall r \{P_2\}[\varepsilon]} \]
\text{where } P_2 \Rightarrow P_1 \text{ and } P'_1 \Rightarrow P'_2

(ConEff r)
\[
\frac{\Gamma \vdash \forall r \{P \&\& E\}[\varepsilon_1]}{\Gamma \vdash \forall r \{P \&\& \neg E\}[\varepsilon_2]} \]
\[
\frac{\Gamma \vdash \forall r \{P\}[\varepsilon]}{\Gamma \vdash \forall r \{P\}[\varepsilon_1 : \varepsilon_2]} \]
\text{where } P \Rightarrow E \text{ and modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)

(ConMask1 r)
\[
\frac{\Gamma \vdash \forall r \{P\}[\varepsilon, \text{modifies } (E ? \varepsilon_1 : \varepsilon_2)]}{\Gamma \vdash \forall r \{P\}[\varepsilon_1]} \]
\text{where } P \Rightarrow \neg E \text{ and modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)

(ConMask2 r)
\[
\frac{\Gamma \vdash \forall r \{P\}[\varepsilon, \text{modifies } (E ? \varepsilon_1 : \varepsilon_2)]}{\Gamma \vdash \forall r \{P\}[\varepsilon_2]} \]
\text{where } P \Rightarrow \neg E \text{ and modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)

(PostToFr r)
\[
\frac{\Gamma \vdash \forall r \{P\}[\varepsilon]}{\Gamma \vdash \forall r \{P\}[\varepsilon, \text{fresh } (RE_1) : \text{fresh } (RE_2)]} \]
\text{where } P \Rightarrow r = \text{alloc}, P \Rightarrow (E \&\& \text{fresh } RE_1 ! ! r), P \Rightarrow (\neg E \&\& \text{fresh } RE_2 ! ! r) \text{ and modifies } r \notin \varepsilon

Figure 4.2: The structural rules in FRL (1)
(FrToPost_r)
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ fresh(RE_1) : fresh(RE_2)]
\]
\[
\Gamma \vdash [E \ ? \ fresh(RE_1) : fresh(RE_2)]
\]
where \( P \Rightarrow b = E, P \Rightarrow b, P \Rightarrow r = alloc, \ modifies b \notin \varepsilon \) and \( \text{modifies} r \notin \varepsilon \)

(VarMask1_r)
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ modifies x, \varepsilon_1 : \varepsilon_2]
\]
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ \varepsilon_1 : \varepsilon_2]
\]
where \( P \Rightarrow E, P \parallel P' \Rightarrow x = y \) and \( P \&\& E \Rightarrow \text{reads} y/(x, \varepsilon) \)

(VarMask2_r)
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ \varepsilon_1 : (\text{modifies} x, \varepsilon_2)]
\]
\[
\Gamma \vdash P \{ P \} \{ P' \}[\text{if} E \text{ then } \varepsilon_1 \text{ else } \varepsilon_2]
\]
where \( P \Rightarrow \neg E, P \parallel P' \Rightarrow x = y \) and \( P \&\& \neg E \Rightarrow \text{reads} y/(x, \varepsilon) \)

(FieldMask1_r)
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ (\text{modifies region}\{ x.f \}, \varepsilon_1) : \varepsilon_2]
\]
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ \varepsilon_1 : \varepsilon_2]
\]
where \( P \Rightarrow E, P \parallel P' \Rightarrow x.f = y, P' \&\& E \Rightarrow \text{reads} x/\text{modifies} \varepsilon, P' \&\& \neg E \Rightarrow \text{reads} y/\text{modifies} \varepsilon \)

(FieldMask2_r)
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ \varepsilon_1 : \text{modifies region}\{ x.f \}, \varepsilon_2]
\]
\[
\Gamma \vdash P \{ P \} \{ P' \}[E \ ? \ \varepsilon_1 : \varepsilon_2]
\]
where \( P \Rightarrow \neg E, P \parallel P' \Rightarrow x.f = y, P' \&\& \neg E \Rightarrow \text{reads} x/\text{modifies} \varepsilon \)
and \( P' \&\& \neg E \Rightarrow \text{reads} y/\text{modifies} \varepsilon \)

Figure 4.3: The structural rules in FRL (2)

4.1.1 The Sequence Rules

This subsection explains the use of the two sequence rules with examples. The rule \( SEQ_r \) may look complicated. However, the complication arises from the side conditions that handle how the effects of \( S_1 S_2 \) are collected from those of \( S_1 \) and \( S_2 \). To understand \( SEQ1_r \), it may be helpful to
consider two cases:

1. $S_1$ allocates some new objects, which are updated by $S_2$. This is the case where the freshly allocated region $RE$ is not empty. Then the write effects of $S_1S_2$ can drop $RE$ from the write effects of $S_2$. For example, consider the sequence: $x := \text{new } T; x.f := 5;$. Assume $f$ is the only field of the reference type $T$ for simplicity. Using the axioms $ALLOC_r$ and $UPD_r$, Eq. (4.1) must be true, which is

$$
\Gamma_r \models \begin{cases}
{\text{true} \times := \text{new } T}; \
\{\text{new}(T, x)\}
\end{cases}
[\text{modifies } x, \text{modifies alloc, fresh(region}\{x.\})]$$

(4.1)

$$
\Gamma_r \models \{x \neq \text{null}\}x.f := 5; \{x.f = 5\}[\text{modifies region}\{x.f\}]$$

(4.2)

Then, the $SubEff_r$ rule is used to loosen the write effect of Eq. (4.2), and get

$$
\Gamma_r \models \{x \neq \text{null}\}x.f := 5; \{x.f = 5\}[\text{modifies region}\{x.\}]$$

(4.3)

Then, using the $CONSEQ_r$ rule on Eq. (4.1), the following is derived

$$
\Gamma_r \models \begin{cases}
{\text{true} \times := \text{new } T}; \
\{x \neq \text{null}\}
\end{cases}
[\text{modifies } x, \text{modifies alloc, fresh(region}\{x.\})]$$

(4.4)

In order to use the rule $SEQ1_r$ on Eq. (4.4) and Eq. (4.3), the rule $SEQ1_r$ is instantiated with $RE := \text{region}\{x.\}, RE_1 := \text{region}\{x.\}, \varepsilon_1 := \text{modifies } x, \text{modifies alloc}$ and $\varepsilon_2 := \emptyset$. Then, the immune side conditions has to be true, which are:

$$
\text{modifies } x \text{ is } true/\text{modifies } x, \text{modifies alloc}-\text{immune}
$$

(4.5)

and

$$
\emptyset \text{ is } true/\text{modifies } x, \text{modifies alloc}-\text{immune}
$$

(4.6)
Eq. (4.6) is obviously true. By the definition of immune (Def. 6 on page 33), to prove Eq. (4.5) is to show

\[ \text{for all modifies } \text{RE} \in (\text{modifies } x) :: \text{RE} \text{ is} \]

\[ \text{true/modified } x, \text{modifies alloc}-\text{immune} \quad (4.7) \]

Eq. (4.7) is vacuously true, since no region expression RE can be a variable x. Now, using the rule SEQ1, the following is derived

\[ \vdash^r \{ \text{true} \} x := \text{new } T; x.f := 5; \{ x.f = 5 \} \]

\[ \left[ \text{modifies } x, \text{modifies alloc, fresh(region}\{x.\ast\}\} \right] \]

In this case, the write effect of the second statement, modifies region\{x.\ast\}, is dropped in that of the sequence statement, as the fresh effect of the first statement become the fresh effect of the sequence.

2. \( S_1 \) does not allocate any new objects. Then the sequence rule can be simplified as:

\[ \vdash^r \{ P \} S_1 \{ P_1 \}[\varepsilon_1] \quad \vdash^r \{ P_1 \} S_2 \{ P' \}[\varepsilon_2] \]

\[ \vdash^r \{ P \} S_1 S_2 \{ P' \}[\varepsilon_1, \varepsilon_2] \]

where \( \varepsilon_1 \) is fresh-free and \( \varepsilon_2 \) is \( P/\varepsilon_1 \)-immune

The two side conditions on immunity are to prevent interference of the effects of two sequential statements. For the write effect, variables and regions that \( \varepsilon_1 \) contains have to be disjoint with those that \( \varepsilon_2 \) depends on. Examples have been given in Section 3.4 in Chapter 3. Similarly, for the read effect, variables and regions that \( \varepsilon_1 \) contains have to be disjoint with those that \( \delta_2 \) depends on. Consider the statements: \( y := z; x := y.f; \). The read and write effects of the first statement are reads \( z \) and modifies \( y \) respectively, and the read effect of the second statement is reads \( y \), reads region\{y.f\}. The read effects of their composition may not be (reads \( z \), reads \( y \), reads region\{y.f\}), as region\{y.f\} may denote a different location
after $y$ is assigned to the value of $z$. To reason about this example, a rule of state-dependent effect subsumption is used, ascribing to $x := y.f$; the read effect reads region$(z.f)$, which is immune from updating $y$.

Consider again the example in Section 3.4, $x.f := x$; $x.f := x$; There, it has been proved that $\varepsilon_2$ is $P/\varepsilon_1$-immune, where $\varepsilon_1$ and $\varepsilon_2$ are both modifies region$(x.f)$, and $P$ is $x \neq null$. Here shows that $\delta_2$ is $P/\varepsilon_1$-immune as follows, where $\delta_2$ is reads $x$.

reads $x$ is $x \neq null$/modifies region$(x.f)$-immune
ifff (by the definition of Immune (Def. 6))
for all reads $RE \in$ reads $x :: RE$ is $x \neq null$/modifies region$(x.f)$-immune
ifff (by there does not exist such $RE$)
true

The following example shows the use of the rule $SEQ2_r$. Consider the program var $y :$ int; $y := 5$; After using the axiom $VAR_r$, the following is derived

$$\Gamma \vdash_r \{true\} \text{var } y : \text{int}; \{y = 0\}[\emptyset] \quad (4.8)$$

After using the axiom $ASGN_r$, the following is derived

$$\Gamma \vdash_r \{true\} y := 5; \{y = 5\} [\text{modifies } y] \quad (4.9)$$

By the rule $CONSEQ_r$ on Eq. (4.9), the following is derived

$$\Gamma \vdash_r \{y = 0\} y := 5; \{y = 5\} [\text{modifies } y] \quad (4.10)$$

Using the rule $SEQ2_r$ on Eq. (4.8) and Eq. (4.10), the following is derived $\Gamma \vdash_r \{true\} \text{var } y : \text{int}; y := 5; \{y = 5\} [\emptyset]$
4.1.2 The Loop Rule

For the rule WHILE\(_r\), \(P\) is the loop invariant and \(r\) stores the locations in the pre-state of the loop. The side condition \(P \Rightarrow RE ! r\) indicates that \(RE\) specifies the locations that may be allocated by the loop body. An example shows how to instantiate \(r\) in the rule WHILE\(_r\). Consider the following program in program context \(\Gamma = alloc: region, f: region, y: int, x: C:\)

\[
B \triangleq x := \text{new} T; f := f + \text{region}\{x.\}; y := y - 1;
S \triangleq f := \text{region}\{\}; y := 5; \text{while} y \in B
\]

The proof obligation is to show that

\[
\Gamma \vdash \{true\} S\{y = 0\}
\]

\[
[\text{modifies } f, \text{modifies } alloc, \text{modifies } x, \text{modifies } y, \text{fresh}(f)]
\]

(4.11)

After using the axiom ASGN\(_r\), once for each of the following, the following is derived

\[
\Gamma \vdash \{true\} f := \text{region}\{\}; \{f = \text{region}\{\}\}[\text{modifies } f]
\]

(4.12)

\[
\Gamma \vdash \{true\} y := 5; \{y = 5\}[\text{modifies } y]
\]

(4.13)

After using the rule FRM\(_r\) on Eq. (4.13), the following is derived

\[
\Gamma \vdash \{f = \text{region}\{\}\} y := 5; \{f = \text{region}\{\} \& \& y = 5\}[\text{modifies } y]
\]

(4.14)

From Eq. (4.12) and Eq. (4.14), the rule SEQ1\(_r\) is instantiated with \(RE := \text{region}\{\}\). As the immunity conditions are vacuously true, the following is derived

\[
\Gamma \vdash \{true\} f := \text{region}\{\}; y := 5; \{f = \text{region}\{\} \& \& y = 5\}[\text{modifies } f, \text{modifies } y]
\]

(4.15)
Now, consider the loop. Let variable \( g \) be fresh; \( g \) is used to snapshot the initial value of \( \text{alloc} \).

For the loop body \( B \), the proof obligation is to derive

\[
\Gamma \vdash g, f \rightarrow B \{g, f\} [\text{modifies } f, \text{modifies } x, \text{modifies } y, \text{modifies } \text{alloc}] \quad (4.16)
\]

From Eq. (4.16), the rule \( \text{WHILE}_r \) is instantiated with \( r := g \) and \( RE = \text{region}\{\} \). Because the immunity conditions are vacuously true, the following is derived.

\[
\Gamma \vdash g, f \rightarrow \text{while } y \{B\} \{g, f \& \& y = 0\} [\text{modifies } x, \text{modifies } y, \text{modifies } f, \text{modifies } \text{alloc}]
\quad (4.17)
\]

The rule \( \text{PostToFr}_r \) is instantiated with \( r := g \) and \( RE := f \). The following is derived.

\[
\Gamma \vdash g, f \rightarrow \text{while } y \{B\} \{g, f \& \& y = 0\} [\text{modifies } x, \text{modifies } y, \text{modifies } f, \text{modifies } \text{alloc}, \text{fresh}(f)]
\quad (4.18)
\]

After using the rule \( \text{CONSEQ}_r \) from the above, the following is derived.

\[
\Gamma \vdash g, f \rightarrow \text{while } y \{B\} \{y = 0\} [\text{modifies } x, \text{modifies } y, \text{modifies } f, \text{modifies } \text{alloc}, \text{fresh}(f)]
\quad (4.19)
\]

The postcondition of Eq. (4.15) implies the precondition of Eq. (4.19). After using the rule \( \text{CONSEQ}_r \) on Eq. (4.15), the following is derived.

\[
\Gamma \vdash \text{true} [\text{modifies } f, \text{modifies } y]
\quad (4.20)
\]

From Eq. (4.20) and Eq. (4.19), the rule \( \text{SEQ1}_r \) is instantiated with \( RE = \text{region}\{\} \). As the immunity conditions are vacuously true, Eq. (4.11) is derived.

Here shows the proof of Eq. (4.16). After using the axiom \( \text{ALLOC}_r \), the following is derived.

\[
\Gamma \vdash \text{true} [\text{modifies } x, \text{modifies } \text{alloc}, \text{fresh}(\text{region}\{x.\})]
\quad (4.21)
\]
Then by the rule $FRM_r$, from the above, the following is derived.

\[ \Gamma \vdash_r \{ g \land f \} \ x := \text{new} \ T; \ \text{new}(T, x) \land g \land f \ [\text{modifies} \ x, \text{modifies alloc}, \text{fresh}(\text{region}(x.*))] \]  
\hspace{1cm} (4.22)

The rule $FrToPost_r$ is instantiated with $r := g$ and $RE = \text{region}(x.*)$. And reads $g$ (modifies $x$, modifies alloc) is true. After applying the rule, the following is derived.

\[ \Gamma \vdash_r \{ g \land f \} \ x := \text{new} \ T; \ \text{new}(T, x) \land g \land f \land \text{region}(x.*) \ [\text{modifies} \ x, \text{modifies alloc}, \text{fresh}(\text{region}(x.*))] \]  
\hspace{1cm} (4.23)

Let $f'$ be a fresh variable and is used to snapshot the initial value of $f$. Then the assignment statement is written as $f := f' + \text{region}(x.*)$. After using the rule $ASGN_r$, the following is derived.

\[ \Gamma \vdash_r \{ \text{true} \} \ f := f' + \text{region}(x.*); \ \{ f = f' + \text{region}(x.*) \} [\text{modifies} \ f] \]  
\hspace{1cm} (4.24)

After using the rule $FRM_r$, the following is derived.

\[ \{ \text{new}(T, x) \land g \land f \land \text{region}(x.*) \} \]

\[ \Gamma \vdash_r \ f := f' + \text{region}(x.*); \ \{ f = f' + \text{region}(x.*) \} \land \text{new}(T, x) \land g \land f \land \text{region}(x.*) \ [\text{modifies} \ f] \]  
\hspace{1cm} (4.25)

From Eq. (4.23) and Eq. (4.25), the rule $SEQ1_r$ is instantiated with $RE = \text{region}()$. As the immunity conditions are vacuously true, the following is derived.

\[ \{ g \land f \} \]

\[ \Gamma \vdash_r \ x := \text{new} \ T; \ f := f' + \text{region}(x.*); \ \{ f = f' + \text{region}(x.*) \} \land \text{new}(T, x) \land g \land f \land \text{region}(x.*) \ [\text{modifies} \ f, \text{modifies} \ x, \text{modifies alloc}, \text{fresh}(\text{region}(x.*))] \]  
\hspace{1cm} (4.26)
Then by the rules $CONSEQ_r$ and $SubEff_r$, the following is derived.

\[
\Gamma \vdash \{g \uparrow f\} x := \text{new} T; f := f' + \text{region}\{x.\} \{g \uparrow f\} \\
\text{[modifies } f, \text{modifies } x, \text{modifies } \text{alloc]}\tag{4.27}
\]

Let $y'$ be a fresh variable and is used to snapshot the initial value of $y$. Then the assignment is written as $y := y' - 1;$. Then, using the axiom $ASGN_r$, the following is derived. \[
\Gamma \vdash \{\text{true}\} y := y' - 1; \{y = y' - 1\} [\text{modifies } f]\]
Then, by the rules $FRM_r$, $SEQ1_r$ and $CONSEQ_r$, Eq. (4.16) is derived.

4.2 Soundness

Theorem 1. The judgment $\Gamma \vdash \{P\}S\{Q\}[\varepsilon]$ that is derivable by the axioms and inference rules in Fig. 4.1 and the structural rules in Fig. 4.2 and Fig. 4.3, is valid.

The proof is done by induction on the derivation and by cases on the last rule used. In each axiom, it is shown that the judgment is valid according to the statement’s semantics. In each inference rule, it is shown that the proof rule derives valid conclusions from valid premises when its side conditions is satisfied. The proof can be found in Appendix B.
CHAPTER 5: UNIFIED FINE-GRAINED REGION LOGIC

This chapter generalizes FRL to UFRL. It defines the correctness judgment in UFRL, and presents the proof axioms and rules for statements and structural rules.

Unified Fine-Grained Region Logic (UFRL) was created to enable using FRL and SL together. UFRL has explicit read and write effects. It is a generalization of FRL; thus UFRL’s assertion and programming languages (Chapter 2 and Chapter 3) are the same as those in FRL.

However, Hoare-formulas in UFRL are different. The correctness judgment in UFRL has the form $[\delta]P_1S\{P_2\}[\varepsilon]$, where $\delta$ are read effects (on the heap) and $\varepsilon$ are write effects; thus $(\varepsilon, \delta)$ contains all the heap locations that $S$ may access. Note that $\delta$ and $\varepsilon$ may have locations in common.

Validity of UFRL Hoare-formulas uses the same notion of partial correctness as in FRL: statements must not encounter an error when started in a pre-state satisfying the specified precondition, but may still loop forever.

Definition 8 (Validity of UFRL Hoare-formula). Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement. Let $P_1$ and $P_2$ be assertions. Let $\varepsilon$ be effects and $\delta$ be read effects, let $(\sigma, H)$ be a $\Gamma$-state. Then $[\delta]P_1S\{P_2\}[\varepsilon]$ is valid in $(\sigma, H)$, written $\sigma, H \models_{\Gamma}^u [\delta]P_1S\{P_2\}[\varepsilon]$, if and only if whenever $\sigma, H \models_{\Gamma} P_1$, then:

1. $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')][\sigma, H]\mathcal{E}[\Gamma \vdash regRW(\varepsilon, \delta) : \text{region}](\sigma) \neq \text{err}$, and

2. if $(\sigma', H') = \mathcal{MS}[\Gamma \vdash S : ok(\Gamma')][\sigma, H]\mathcal{E}[\Gamma \vdash regRW(\varepsilon, \delta) : \text{region}](\sigma)$, then the following all hold:

   • $\sigma', H' \models_{\Gamma'} P_2$.

---

1 The content in this chapter is submitted to Formal Aspects of Computing.
\begin{itemize}
\item for all $x \in \text{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : \text{modifies } x \in \varepsilon,$
\item for all $(o, f) \in \text{dom}(H) : H'[o, f] \neq H[o, f] :$
\hspace{1cm} (o, f) \in \mathcal{E}_{\Sigma} \vdash \text{write}(\varepsilon) : \text{region}(\sigma), \text{ and}
\item for all $(o, f) \in \mathcal{E}_{\Sigma} \vdash \text{fresh}(\varepsilon) : \text{region}(\sigma') : (o, f) \in (\text{dom}(H') - \text{dom}(H)).$
\end{itemize}

A UFRL Hoare-formula $[\delta]\{P_1\}S\{P_2\}[\varepsilon]$ is valid, written $\models_{\text{uf}}^\Gamma [\delta]\{P_1\}S\{P_2\}[\varepsilon], \text{ if and only if for all states } (\sigma, H) :: \sigma, H \models_{\text{uf}}^\Gamma [\delta]\{P_1\}S\{P_2\}[\varepsilon].$

The above definition limits the heap that a statement can access. Consider the following formula

$$[\text{reads region}\{x.f\}]\{x \neq \text{null}\}y := x.f; \{y = x.f\}[\text{modifies } y]. \quad (5.1)$$

Eq. (5.1) is a valid UFRL Hoare-formula, because $\text{reg}_{\text{UF}}(\text{reads region}\{x.f\}, \text{modifies } y) = \text{region}\{x.f\}$. The region $\text{region}\{x.f\}$ is the least set of locations that the statement needs to make sure that its execution does not cause an error. On the contrary, the formula $[\emptyset]\{x \neq \text{null}\}y := x.f; \{y = x.f\}[\text{modifies } y]$ is not a valid UFRL Hoare-formula, as $\text{reg}_{\text{UF}}(\text{reads } \emptyset, \text{modifies } y) = \text{region}\{}$. As another example, consider the following formula:

$$[\emptyset]\{x \neq \text{null}\}x.f := y; \{x.f = y\}[\text{modifies region}\{x.f\}]. \quad (5.2)$$

Eq. (5.2) is a valid UFRL Hoare-formula, because $\text{reg}_{\text{UF}}(\emptyset, \text{modifies region}\{x.f\}) = \text{region}\{x.f\}$.

For the purpose of framing, which is the focus of this work, there is no need to track read effects, although the above definition does limit to the heap which the statement can access to. However, read effects (on the heap) are needed for future work; e.g., for framing of specifications with pure method calls [3].

51
5.1 Axioms and Inference Rules

This section shows the axioms and proof rules for proving statements correct in UFRL. Fig. 5.1 shows the axioms and proof rules. Fig. 5.2 and Fig. 5.3 show the structural rules. These are based on FRL, but with read effects ($\delta$ and $\eta$) specified.

The axioms for variable declaration, variable assignment, field access, field update and allocation are “small” [71] in the sense that the union of write effects and read effects describe the least upper bound of variables and locations that $S$ accesses, and the write effects describe the least upper bound of the set of variables and locations that $S$ may modify. The proof system does not split the store, as variables are discarded by $regRW$ (Def. 8).

The structural rules are shown in Fig. 5.2 and Fig. 5.3. The rule $FRM_u$ follows the $FRM_r$ rule. The rule $SubEff_u$ allows approximations of effects; it can be used to match up the effects for the rule $IF_u$, where different branches may have different effects. The rule $SubEff_u$ also allows a correctness proof to switch from a smaller to a larger heap. The rule $CONSEQ_u$ is the standard consequence rule. The rule $FrToPost_u$ and $PostToFr_u$ are dual; the first allows one to add fresh effects and the second allows one to eliminate fresh effects. To make the $PostToFr_u$ rule clear, the following from the rule $FrToPost_u$ is derived.

\[
\Gamma \vdash_u [\delta] \{P\} \ S \ \{P'\}[\varepsilon, fresh(\mathit{RE})] \quad \text{where } P \Rightarrow r = alloc
\]

\[
\Gamma \vdash_u [\delta] \{P\} \ S \ \{P' \land \land \ r \land \land \ RE\}[\varepsilon]
\]

This uses the subeffect rule, because $regRW(\delta, fresh(\mathit{RE}), \varepsilon) \leq regRW(\delta, \varepsilon)$, and $regRW$ ignores fresh effects.
\[
(\text{SKIP}_u) \quad \vdash \Delta \varnothing \{ \text{true} \} \text{skip} ; \{ \text{true} \} \varnothing \\
(\text{VAR}_u) \quad \vdash \Delta \varnothing \{ \text{true} \} \text{var} : T; \{ x = \text{default}(T) \} \varnothing \\
(\text{ALLOC}_u) \quad \vdash \Delta \{ \text{true} \} \text{x := new T; \{ new(T, x) \}} \quad \text{modifies x, modifies alloc, fresh(region\{x.x\})} \\
(\text{ASGN}_u) \quad \vdash \Delta \varnothing \{ \text{true} \} \text{x := G; \{ x = G \}} \{ \text{modifies x} \} \text{where x \notin FV(G) and } \eta = \text{efs}(G) \\
(\text{UPD}_u) \quad \vdash \Delta \varnothing \{ x, \eta \} \{ x \neq \text{null} \} \text{x.f := G; \{ x.f = G \}} \{ \text{modifies region\{x.f\}} \} \text{where } \eta = \text{efs}(G) \\
(\text{ACC}_u) \quad \vdash \Delta \varnothing \{ x', \eta \} \{ x' \neq \text{null} \} \text{x := x'.f; \{ x = x'.f \}} \{ \text{modifies x} \}, \text{where } x \neq x' \text{ and } \eta = \text{efs}(x'.f) \\
(\text{IF}_u) \quad \vdash \Delta \{ \delta \} \{ P \land \text{true} \} \{ S_1 \} \{ \varepsilon \} \quad \vdash \Delta \{ \delta \} \{ P \land \neg \text{true} \} \{ S_2 \} \{ \varepsilon \} \\
\vdash \Delta \{ \delta, \delta_E \} \{ P \} \text{if } E \{ S_1 \} \text{else } \{ S_2 \} \{ \varepsilon \} \quad \text{where } \delta_E = \text{efs}(E) \\
(\text{SEQ}_u) \quad \vdash \Delta \{ \delta_1 \} \{ P \} \{ P_1 \} \{ \varepsilon_1 \} \text{fresh(RE)} \\
\vdash \Delta \{ \delta_2 \} \{ \text{reads } \text{RE}_1 \} \{ P_1 \} \{ P' \} \{ \varepsilon_2 \} \text{modifies } \text{RE}_2 \\
\vdash \Delta \{ \delta_1, \delta_2 \} \{ P \} \{ S_1 S_2 \} \{ P' \} \{ \varepsilon_1, \varepsilon_2 \} \{ \text{fresh(RE)} \} \\
\text{where } S_1 \neq \text{var} x : T; \varepsilon_1 \text{ is fresh-free, } \delta_2 \text{ is P/} \varepsilon_1 \text{-immune, } \varepsilon_2 \text{ is P/} \varepsilon_1 \text{-immune, } \text{RE is P/} \{ \text{modifies } \text{RE}_2, \varepsilon_2 \}-\text{immune, } \text{RE}_1 \leq \text{RE} \text{ and } \text{RE}_2 \leq \text{RE} \\
(\text{SEQ}^2_u) \quad \vdash \Delta \{ \delta_1 \} \{ \text{reads } \text{x} \text{=} \text{default(T)} \} \{ S \} \{ \varepsilon \} \text{modifies } \text{x, } \varepsilon \\
\vdash \Delta \{ \delta \} \{ P \} \text{var } x : T; \{ S \} \{ \varepsilon \} \\
(\text{WHILE}_u) \quad \vdash \Delta \{ \delta \} \{ P \land \text{true} \} \{ S \} \{ \varepsilon \} \text{modifies } \text{RE} \\
\vdash \Delta \{ \delta, \delta_E \} \{ P \land \text{r := alloc} \} \text{while } E \{ S \} \{ P \land \neg \text{E} \} \{ \varepsilon \} \\
\text{where } \delta_E = \text{efs}(E), P \Rightarrow \text{RE} \not\Rightarrow r, \varepsilon \text{ is fresh-free, modifies } r \neq \varepsilon, \delta \text{ is P/} \varepsilon \text{-immune and } \varepsilon \text{ is P/} \varepsilon \text{-immune} \\
\]

Figure 5.1: The correctness axioms and proof rules for statements in UFRL
\[(FRM_u)\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon], \quad P \vdash \eta \text{ frm } Q\]
\[\Gamma \vdash \delta \{P & & Q\} S \{P' & & Q\}[\varepsilon]\]
\[\text{where } P & & Q \Rightarrow \eta / \varepsilon\]

\[(SubEff_u)\]
\[\Gamma \vdash \delta \{P_1\} S \{P_2\}[\varepsilon], \quad \varepsilon \leq \varepsilon'\]
\[\Gamma \vdash \delta' \{P_1\} S \{P_2\}[\varepsilon']\]
\[\text{where } P_1 \Rightarrow \text{regRW}(\varepsilon, \delta) \leq \text{regRW}(\varepsilon', \delta')\]

\[(CONSEQ_u)\]
\[\Gamma \vdash \delta \{P_1\} S \{P_1'\}[\varepsilon]\]
\[\Gamma \vdash \delta \{P_2\} S \{P_2'\}[\varepsilon]\]
\[\text{where } P_2 \Rightarrow P_1 \text{ and } P_1' \Rightarrow P_2'\]

\[(ConEff_u)\]
\[\Gamma \vdash \delta \{P & & E\} S \{P'\}[\varepsilon_1], \quad \Gamma \vdash \delta \{P & & \lnot E\} S \{P'\}[\varepsilon_2]\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[E ? \varepsilon_1 : \varepsilon_2]\]

\[(ConMask1_u)\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, \varepsilon_1]\]
\[\text{where } P \Rightarrow E\]

\[(ConMask2_u)\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, \varepsilon_2]\]
\[\text{where } P \Rightarrow \lnot E\]

\[(PostToFr_u)\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon]\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, E ? \text{fresh}(RE_1) : \text{fresh}(RE_2)]\]
\[\text{where } P \Rightarrow (E & & \lnot \text{RE}_1 \text{ alloc}) \text{ and } P \Rightarrow (\lnot E \land \text{RE}_2 \text{ alloc})\]

\[(FrToPost_u)\]
\[\Gamma \vdash \delta \{P\} S \{P'\}[\varepsilon, E ? \text{fresh}(RE_1) : \text{fresh}(RE_2)]\]
\[\Gamma \vdash \delta \{P\} S \{P' \land (b \Rightarrow \text{RE}_1 \text{ alloc}) \land (\lnot b \Rightarrow \text{RE}_2 \text{ alloc})\}[\varepsilon, E ? \text{fresh}(RE_1) : \text{fresh}(RE_2)]\]
\[\text{where } P \Rightarrow b = E, P \Rightarrow r = \text{alloc}, P \Rightarrow E, \text{modifies } b \notin \varepsilon \text{ and modifies } r \notin \varepsilon\]

Figure 5.2: The structural rules in UFRL (1)
The complication arising from read effects is discussed. Consider the case where $S_1$ allocates some new objects, which are read by $S_2$. This is the case where the freshly allocated region $RE$ is not empty. Then the read effects of $S_1S_2$ can drop $RE$ from the read effects of $S_2$. For example, consider the sequence: $x := \texttt{new} T; y := x.f$, where $x \neq y$. Assume that $f$ is the only field of
reference type $T$ for simplicity. Using the rules $ALLOC_u$ and $ACC_u$, the following must be true:

\[
\{ \varnothing \} \\
\vdash \Gamma_u \{ \text{true} \} x := \text{new } T; \{ \text{new}(T, x) \} \\
\begin{array}{c}
\text{modifies } x, \text{modifies alloc, fresh(region\{x.*\})}
\end{array}
\]  
\hspace{1cm} (5.3)

\[
\vdash \Gamma_u \{ \text{reads } x, \text{region\{x.f\}} \} \{ x \neq \text{null} \} y := x.f; \{ y = x.f \} \text{modifies } y
\]  
\hspace{1cm} (5.4)

Then, after using the $SubEff_u$ rule to loosen the read effect of Eq. (5.4), the following is derived:

\[
\vdash \Gamma_u \{ \text{reads } x, \text{region\{x.*\}} \} \{ x \neq \text{null} \} y := x.f; \{ y = x.f \} \text{modifies } y
\]  
\hspace{1cm} (5.5)

Then, after using the $CONSEQ_u$ rule on Eq. (5.3), the following is derived:

\[
\{ \varnothing \} \\
\vdash \Gamma_u \{ \text{true} \} x := \text{new } T; \{ x \neq \text{null} \} \\
\begin{array}{c}
\text{modifies } x, \text{modifies alloc, fresh(region\{x.*\})}
\end{array}
\]  
\hspace{1cm} (5.6)

In order to use the $SEQ1_u$ rule on Eq. (5.6) and Eq. (5.5), it is instantiated with $RE := \text{region\{x.*\}}, RE_1 := \text{region\{x.*\}}, RE_2 := \text{region\{\}}, \varepsilon_1 := \text{modifies } x, \text{modifies alloc}$ and $\varepsilon_2 := \text{modifies } y$. Then, the proof obligation is to check the immune side conditions, which are:

\[
\text{reads } x \text{ is true}/(\text{modifies } x, \text{modifies alloc})\text{-immune}
\]  
\hspace{1cm} (5.7)

and

\[
\text{modifies } y \text{ is true}/(\text{modifies } x, \text{modifies alloc})\text{-immune}
\]  
\hspace{1cm} (5.8)

By the definition of immune (Def. 6 on page 33), to prove Eq. (5.7) and Eq. (5.8) is to show

\[
\text{for all reads } RE \in \text{(reads } x) :: RE \text{ is}
\]  
\hspace{1cm} (5.9)
and

\[
\text{for all }	ext{modifies } \text{RE} \in (\text{modifies } y) :: \text{RE} \text{ is }
\]
\[
\text{true/(modifies } x, \text{modifies alloc)-immune} \quad (5.10)
\]

Eq. (5.9) and Eq. (5.10) are vacuously true. Now, using the rule \( \text{SEQ1}_u \), the following is derived

\[
\begin{align*}
\text{[reads } x] \\
\Gamma \vdash \text{reads } x \\
\Gamma \vdash \text{true} x := \text{new } T; \ y := x.f; \ \{ y = x.f \}
\end{align*}
\]

\[
\text{[modifies } x, \text{modifies alloc, modifies } y, \text{fresh(region } \{ x.* \})]
\]

In this case, the \( \text{region } \{ x.* \} \) of the read effect in the second statement is dropped in that of the sequence statement, as the fresh effects of the first statement become the fresh effect of the sequence.

5.2 Soundness

**Theorem 2.** The judgment \( \Gamma \vdash \text{reads } x \) \( \vdash \text{true} x := \text{new } T; \ y := x.f; \ \{ y = x.f \} \)

\[
\text{[modifies } x, \text{modifies alloc, modifies } y, \text{fresh(region } \{ x.* \})]
\]

is derivable by the axioms and inference rules in Fig. 5.1, and the structural rules in Fig. 5.2 and Fig. 5.3 is valid.

**Proof.** Using the result of Theorem 1 on page 49, the proof only needs to check the read effects. Let \( S \) be a statement and \( (\sigma, h) \) be \( \Gamma \)-state. Assume \( \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \) and \( \sigma, h \models \Gamma P \). Then it needs to be true that \( \mathcal{M}S[[\Gamma \vdash S : \text{ok}(\Gamma')]](\sigma, h)[\mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma)) \neq \text{err}. \)

1. \( \text{(SKIP}_u) \) In this case, \( S \) is \( \text{skip} \), \( P \) is \( \text{true} \), and \( \delta \) and \( \varepsilon \) are both \( \emptyset \). As it is known that \( h \models \mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \emptyset \), by the program semantics Fig. 2.4, it must be true that \( \mathcal{M}S[[\Gamma \vdash \text{skip} : \text{ok}(\Gamma')]](\sigma, \emptyset) \neq \text{err}. \)
2. (\text{VAR}_u) In this case, $S$ is $\text{var} \ x : T;$, $P$ is $\text{true}$, and $\delta$ and $\varepsilon$ are both $\emptyset$. As it is known that $h \models \mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \emptyset$, by the program semantics Fig. 2.4, it must be true that $\mathcal{M}S[[\Gamma \vdash \text{var} \ x : T; \ ok(\Gamma, x : T)]](\sigma, \emptyset) \neq \text{err}$.

3. (\text{ALLOC}_u) In this case, $S$ is $x := \text{new} \ T;$, $P$ is $\text{true}$ and $\varepsilon$ and $\delta$ are both $\emptyset$. As it is known that $h \models \mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \emptyset$, by the program semantics Fig. 2.4, it must be true that $\mathcal{M}S[[\Gamma \vdash x := \text{new} \ T; \ ok(\Gamma)]](\sigma, \emptyset) \neq \text{err}$.

4. (\text{ASSGN}_u) In this case, $S$ is $x := G;$, $P$ is $x = x'$ and $\delta$ is $\text{efs}(G)$ and $\varepsilon$ is $\text{modifies} \ x$, where $x \notin \text{FV}(G)$. Since it is known that $h \models \mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \emptyset$, by the program semantics Fig. 2.4 on page 23, it must be true that $\mathcal{M}S[[\Gamma \vdash x := G; \ ok(\Gamma)]](\sigma, \emptyset) \neq \text{err}$.

5. (\text{UPD}_u) In this case, $S$ is $x.f := G;$, $P$ is $x \neq \text{null}$, $\delta$ is $\{\text{reads} \ x, \text{efs}(G)\}$ and $\varepsilon$ is $\text{modifies} \ \text{region}\{x.f\}$. By the precondition, it is known that $\sigma(x) \neq \text{null}$. Since it is known that $\mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \{\sigma(x, f)\}$, by the program semantics Fig. 2.4, it must be true that $\mathcal{M}S[[\Gamma \vdash x := G; \ ok(\Gamma)]](\sigma, h\{\sigma(x, f)\}) \neq \text{err}$.

6. (\text{ACC}_u) In this case, $S$ is $x := x'.f;$, $P$ is $x' \neq \text{null}$, $\delta$ is $\{\text{reads} \ x', \text{region}\{x'.f\}\}$ and $\varepsilon$ is $\text{modifies} \ x$, where $x \neq x'$. The precondition implies that $\sigma(x') \neq \text{null}$. As $\mathcal{E}[[\Gamma \vdash \text{regRW}(\varepsilon, \delta) : \text{region}]](\sigma) = \{\sigma(x', f)\}$, by the program semantics shown in Fig. 2.4, it must be true that $\mathcal{M}S[[\Gamma \vdash x := x'.f; \ ok(\Gamma)]](\sigma, h\{\sigma(x', f)\}) \neq \text{err}$.

Other inductive cases follow inductive hypotheses.

\qed
CHAPTER 6: Interoperability

This chapter shows the connections of UFRL, FRL and SL. Section 6.1 shows that FRL is just an instance of UFRL. Section 6.2 shows how to encode SL to UFRL. Section 6.3 presents another way to allow SL style assertions to appear in UFRL (or FRL itself) without using the somewhat verbose encoding of separating conjunction.

6.1 FRL - An Instance of UFRL

The section shows that FRL Hoare formulas can be translated into UFRL by using the read effect \texttt{reads alloc}.

Lemma 8. Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement, and let $P_1$ and $P_2$ be assertions. Let $\varepsilon$ be effects, and let $(\sigma, H)$ be a $\Gamma$-state. Then

$$\sigma, H \vdash_{\Gamma} \{P_1\} S \{P_2\}[\varepsilon] \text{ iff } \sigma, H \vdash_{\Gamma} \{\texttt{reads alloc}\}_\Gamma \{P_1\} S \{P_2\}[\varepsilon].$$

Proof. The lemma is proved as follows, starting from the left side.

$$\sigma, H \vdash_{\Gamma} \{P_1\} S \{P_2\}[\varepsilon]$$

iff (by the definition of FRL valid Hoare-formula (Def. 7).)

---

1The content in this chapter is submitted to Formal Aspect of Computing.
σ, H ⊨ Γ P_1 implies MS[Γ ⊢ S : ok(Γ')]((σ, H) ≠ err and
if (σ', H') = MS[Γ ⊢ S : ok(Γ')]((σ, H), then σ', H' ⊨ Γ P_2 and
(for all x ∈ dom(σ) : σ'(x) ≠ σ(x) implies modifies x ∈ ε) and
(for all (o, f) ∈ dom(H) : (H'[o, f] ≠ H[o, f] implies
(o, f) ∈ E[Γ ⊢ writeR(ε) : region](σ)) and
(for all (o, f) ∈ E[Γ' ⊢ freshR(ε) : region](σ') :: (o, f) ∈ (dom(H') - dom(H))
iff (by the definition of UFRL valid Hoare-formula (Def. 8))
σ, H ⊨ Γ P_1 implies MS[Γ ⊢ S : ok(Γ')]((σ, H) ≠ err and if
(σ', H') = MS[Γ ⊢ S : ok(Γ')]((σ, H) ⊨ Γ[regRW(ε, alloc) : region](σ)),
then (σ', H' ⊨ Γ P_2) and (for all x ∈ dom(σ) : σ'(x) ≠ σ(x) implies modifies x ∈ ε) and
(for all (o, f) ∈ dom(H) : (H'[o, f] ≠ H[o, f] implies
(o, f) ∈ E[Γ ⊢ writeR(ε) : region](σ)) and
(for all (o, f) ∈ E[Γ' ⊢ freshR(ε) : region](σ') :: (o, f) ∈ (dom(H') - dom(H))
iff (by the definition of UFRL valid Hoare-formula (Def. 8))
σ, H ⊨_u {P_1} S {P_2}[ε][reads alloc]

Corollary 1. Let Γ be a well-formed type environment. Let S be a statement, and let P_1 and P_2 be assertions. Let ε be effects, and η be read effects. Then

σ, H ⊨^Γ_u [η]{P_1}S{P_2}[ε] implies σ, H ⊨^Γ_r {P_1}S{P_2}[ε].

Def. 9 shows a syntactic mapping from the axioms and rules of FRL to those of UFRL. Recall that the assertions in FRL and URL have the same syntax.

Definition 9 (Syntactic Mapping from FRL to UFRL). Let Γ be a well-formed type environment. Let P_1 and P_2 be assertions in FRL. Let ε be effects. A syntactic mapping TR{[r]} from FRL rules
to those of UFRL is defined below:

For the FRL axioms: \[ TR[\Gamma \vdash \{P_1\} S \{P_2\}[\varepsilon]] = \vdash^f_u \text{[reads alloc]} \{P_1\} S \{P_2\}[\varepsilon]. \]

For the FRL rules, let \(h_1, \ldots, h_n\) be hypotheses and \(c\) be a conclusion; then the syntactic mapping from a FRL rule to a UFRL rule is defined as follows:

\[
\begin{align*}
TR_R[\vdash^f_r h_1 \ldots \vdash^f_r h_n] &\quad \Rightarrow \quad TR_R[\vdash^f_r c] \\
TR_R[\vdash^f_r h_1] \ldots TR_R[\vdash^f_r h_n] &\quad \Rightarrow \quad TR_R[\vdash^f_r c]
\end{align*}
\]

\[ \blacksquare \]

**Theorem 3.** Let \(\Gamma\) be a well-formed type environment. Let \(S\) be a statement. Let \(P_1\) and \(P_2\) be assertions. Let \(\varepsilon\) be effects. Then

\[ \vdash^f_r \{P_1\} S \{P_2\}[\varepsilon] \iff \vdash^f_u \text{[reads alloc]} \{P_1\} S \{P_2\}[\varepsilon] \]

where \(P_1 \Rightarrow r = \text{alloc}\) and \(\text{modifies} r \notin \varepsilon\)

The proof is found in Appendix C.

**Corollary 2.** The meaning of a FRL judgment is preserved by the syntactic mapping.

**Corollary 3.** Let \(\Gamma\) be a well-formed type environment. Let \(S\) be a statement. Let \(P_1\) and \(P_2\) be assertions. Let \(\varepsilon\) be effects and \(\eta\) be read effects. Then

\[ \vdash^f_u [\eta] \{P_1\} S \{P_2\}[\varepsilon] \text{ implies } \vdash^f_r \{P_1\} S \{P_2\}[\varepsilon]. \]

The proof uses the subeffect rule to convert \(\eta\) into \(\text{reads} r\), where \(P_1 \Rightarrow r = \text{alloc}\), and then applies Theorem 3.
6.2 Encoding Separation Logic

6.2.1 Separation Logic Review

To understand the relationship between SL and UFRL, their semantics are connected by defining the semantics of SL in terms of a heap and a region. This section is inspired by Parkinson and Summers’ work [78]. They connect the semantics of separation logic and implicit dynamic frames [83] by a “total heap semantics” [78]. However, our heap is a partial function.

Separation logic introduces separating conjunction and magic wand (separating implication). The separating conjunction, \( a_1 \ast a_2 \), denotes that assertions \( a_1 \) and \( a_2 \) hold in separate parts of the current heap. The separating implication, \( a_1 \rightarrow a_2 \), denotes that if assertion \( a_1 \) holds in an extra part of the heap, then \( a_2 \) will hold in a heap that is a combination of the extra heap and the current heap.

**Definition 10** (Separation Logic Assertions (SL)). Let \( x \) be variables and \( f \) be field names. The syntax of assertions in separation logic is as follows:

\[
e ::= x | \text{null} | n
\]

\[
a ::= e = e | x.f \rightarrow e | a \ast a | a \rightarrow a | a \land a | a \lor a | a \Rightarrow a | \exists x.a
\]

The semantics given below assumes that expressions and assertions are properly typed. Expressions are pure, meaning that they are independent of the heap. Intuitionistic separation logic [40, 77] is considered in this dissertation. Recall that its semantics [40, 77] is as follows.

**Definition 11** (Separation Logic Semantics). Assuming that \( N \) is the standard meaning function
for numeric literals and \((\sigma, h)\) is a state, then the semantics of expressions in separation logic is:

\[
\begin{align*}
\mathcal{E}_s : & \text{Typing Judgment} \to e \to \text{State} \to \text{Value} \\
\mathcal{E}_s[\Gamma \vdash x : T](\sigma, h) = \sigma(x) & \quad \mathcal{E}_s[\Gamma \vdash \textbf{int}](\sigma, h) = \mathbb{N}\[n] \\
\mathcal{E}_s[\Gamma \vdash \textbf{null} : T](\sigma, h) = \text{null}
\end{align*}
\]

And the validity of assertions in separation logic is defined by:

\[
\begin{align*}
\sigma, h \models^\Gamma e = e' & \iff \mathcal{E}_s[\Gamma \vdash e : T](\sigma) = \mathcal{E}_s[\Gamma \vdash e' : T](\sigma) \\
\sigma, h \models^\Gamma x.f \mapsto e & \iff (\sigma(x), f) \in \text{dom}(h) \text{ and } h[\sigma(x), f] = \mathcal{E}_s[\Gamma \vdash e : T](\sigma) \\
\sigma, h \models^\Gamma a_1 \ast a_2 & \iff \exists h_1, h_2, (h_1 \perp h_2 \text{ and } h = h_1 \cdot h_2 \text{ and } \sigma, h_1 \models^\Gamma a_1 \text{ and } \\
& \quad \sigma, h_2 \models^\Gamma a_2) \\
\sigma, h \models^\Gamma a_1 \rightarrow a_2 & \iff \text{for all } h'. (h' \perp h \text{ and } \sigma, h' \models^\Gamma a_1 \text{ implies } \sigma, h' \models^\Gamma a_2) \\
\sigma, h \models^\Gamma a_1 \land a_2 & \iff \sigma, h \models^\Gamma a_1 \text{ and } \sigma, h \models^\Gamma a_2 \\
\sigma, h \models^\Gamma a_1 \lor a_2 & \iff \sigma, h \models^\Gamma a_1 \text{ or } \sigma, h \models^\Gamma a_2 \\
\sigma, h \models^\Gamma a_1 \Rightarrow a_2 & \iff \text{for all } h'. (\sigma, h \cdot h' \models^\Gamma a_1 \text{ implies } \sigma, h \cdot h' \models^\Gamma a_2) \\
\sigma, h \models^\Gamma \exists x.a & \iff \exists v. (\sigma[x \mapsto v], h \models^\Gamma a)
\end{align*}
\]

The points-to assertion specifies the least segment of the current heap that makes it true. Magic wand and logical implication both involve all possible extensions of the current heap.

Given a fixed program state, assertions in UFRL are all evaluated by the same heap. However, in SL nested sub-assertions of an assertion may be evaluated by a subheap, and the heap can be split and recombined during the evaluation process. This splitting and recombining of heaps can be modeled in the semantics using a heap \(H\), various regions, and region operators along with the heap restriction operator (\(\lceil\cdot\rceil\)) from Def. 1. Indeed the definitions of the semantics of separation logic and validity of assertions can be given using this idea. That is, when \(r \subseteq \text{dom}(H)\), define
\( \sigma, H \models_{\text{sl}} a \) if and only if \( \sigma, (H \models_r) \models_{\text{sl}} a \), however, for clarity, the following definitions of validity for separating conjunction and implication are used.

Let \( r \) be a region such that \( r \subseteq \text{dom}(H) \). The semantics for the separating conjunction expresses the required splitting of partial heaps by restricting the heap to the split regions.

\[
\sigma, H \models_{\text{sl}} a_1 \land a_2 \iff \exists r_1, r_2 : (r_1 \cap r_2 = \emptyset \text{ and } r = r_1 \cup r_2 \text{ and } \sigma, H \models_{\text{sl}} a_1 \text{ and } \sigma, H \models_{\text{sl}} a_2)
\]

The semantics for the magic wand and logical implication consider all possible extensions of the partial heap \( H \models_r \). The extensions are not necessarily disjoint with the heap \( H \), but must be disjoint with the subheap \( H \models_r \), so that the extended heap \( h' \) satisfies \( \text{dom}(h') \cap r = \emptyset \).

\[
\sigma, H \models_{\text{sl}} a_1 \rightarrow a_2 \iff \text{for all } h', r' : (\text{dom}(h') \cap r = \emptyset \text{ and } \sigma, h' \models_{\text{sl}} a_1 \text{ implies } \sigma, (H \cup h') \models_{\text{sl}} r' \models_{\text{sl}} a_2)
\]

\[
\sigma, H \models_{\text{sl}} a_1 \Rightarrow a_2 \iff \text{for all } h', r' : (\text{dom}(h') \cap r = \emptyset \text{ and } \sigma, (H \cup h') \models_{\text{sl}} r' \models_{\text{sl}} a_1 \text{ implies } \sigma, (H \cup h') \models_{\text{sl}} a_2)
\]

The following theorem is used to justify a semantic of SL in terms of a heap and a region.

**Theorem 4.** Let \( \Gamma \) be a well-formed type environment. Let \( \sigma \) be a store, \( h \) and \( H \) be heaps, and \( r \) be a region, such that \( r \subseteq \text{dom}(H) \) and \( h = H \models_r \), then \( \sigma, h \models_{\text{sl}} a \iff \sigma, H \models_{\text{sl}} a \).

The above theorem chooses \( \text{dom}(h) \) to be \( r \), but this requires the user of the theorem to know exactly the heap that a SL assertion talks about in order to encode it. However, the intuitionistic semantics of SL do not precisely prescribe a unique solution to \( h \), thus it is difficult to use Theorem 4. Therefore, in the next section, another candidate for \( r \), which is more constructive, is found.
This section shows that the semantic footprint is another candidate for the region $r$ needed in Theorem 4. Moreover, this section establishes the relationship between semantic footprints and supported separation logic (SSL), which is a fragment of SL where all assertions are supported [73].

The semantics of the points-to assertion, $x.f \mapsto e$ in a state $(\sigma, h)$ indicates that there is a collection of heaps that make it true and those are all supersets of the heap with the singleton cell $\{(\sigma(x), f) \mapsto E \models e : T(\sigma)\}$. Since intuitionistic SL is used, this heap is the greatest lower bound (glb) of the heaps in which the assertion holds. The semantic footprint for SL assertions is defined, which capture this glb. Validities are congruent on the heaps ranging from the glb to $h$. It is said that validity is closed under heap extension as are the semantics of the semantic footprint, as any extension to the glb will preserve validity. But some assertions in SL do not have a semantic footprint, because the glb does not exist.

The semantic footprint of a SL assertion $a$ is the glb of (heap) locations on which $a$ depends. The notion of the glb is formalized by the intersection of the regions of the given heap on which the given assertion $a$ is true:

$$\text{MinReg}(a, \sigma, h) = \bigcap\{r \mid r \subseteq \text{dom}(h) \text{ and } (\sigma, h \models^r \Gamma a) \implies (h \upharpoonright r \models^r \Gamma a)\},$$

where $(\sigma, h)$ is a state. However, $(\sigma, (h \upharpoonright \text{MinReg}(a, \sigma, h)) \models^r \Gamma a$ is not always true. For example, consider $(x.f \mapsto 5) \lor (y.g \mapsto 6)$ in a state where both disjuncts are true; note that the intersection of regions whose domains are $\{((\sigma(x), f))\}$ and $\{((\sigma(y), g))\}$ is an empty set. But $(\sigma, (h \upharpoonright \emptyset) \models^r \Gamma (x.f \mapsto 5) \lor (y.g \mapsto 6)$ is false. So, some assertions containing disjunction do not have a semantic footprint. Semantic footprints are defined as follows.

**Definition 12** (Semantic Footprint). Let $\Gamma$ be a well-formed type environment. Let $a$ be an assertion
In SL, and $(\sigma, h)$ be a $\Gamma$-state. Then $\text{MinReg}(a, \sigma, h)$ is the semantic footprint of $a$ if and only if $\sigma, (h\mid\text{MinReg}(a, \sigma, h)) \models^\Gamma_s a$. In this case, it is said $a$ has a semantic footprint.

In general, formulas that use disjunction do not have a semantic footprint, neither do formulas that use negation, due to DeMorgan’s law for conjunctions. Similarly, general existential assertions do not always have a semantic footprint. Eliminating these types of assertions leaves a fragment of separation logic, which includes just the supported assertions in the work of O’Hearn et al. [73]. This fragment is called supported separation logic (SSL). This is the biggest subset of the syntax in Def. 10, where all assertions are necessarily supported. This syntax is the core fragment of separation logic that contains or corresponds with the SL syntax used by automated reasoning or analysis work [14, 15, 16, 19, 24, 29, 78].

To avoid introducing new notations, the syntax of separation logic (Def. 10) is reused. From now on, those notations mean supported separation logic.

**Definition 13 (Supported Separation Logic).** The syntax of supported separation logic has expressions ($e$), Boolean expressions ($b$) and assertions ($a$) defined as follows:

\[
\begin{align*}
  e & ::= x \mid \textbf{null} \mid n \\
  b & ::= e_1 = e_2 \mid e_1 \neq e_2 \\
  a & ::= b \mid x.f \mapsto e \mid a_1 \ast a_2 \mid a_1 \wedge a_2 \mid b \Rightarrow a \mid \exists x. \ (y.f \mapsto x \ast a)
\end{align*}
\]

The first semantic lemma below states that the truth of assertions is closed under heap extension. That means if an assertion $a$ is true in a heap $h$, then it is also true in an extension of $h$. The proof of encoding separating conjunction, $a_1 \ast a_2$, needs this property. Given the truth of $a_1 \ast a_2$ on heap $h$, where $a_1$ and $a_2$ hold on partitions of $h$, $h_1$ and $h_2$ respectively, the evaluation of the encoded expression is on each partition’s extension to $h$. However, the witnesses for $h_1$ and $h_2$, regions $r_1$

\footnote{For the work with classical separation logic, the \textit{emp} predicate is needed.}
and \( r_2 \), must satisfy \( r_1 \cup r_2 = \text{dom}(h) \), which is required by its semantics. Picking \( h \uparrow r_1 \) as the witness for \( h_1 \) pushes the proof to take \( h \uparrow (\text{dom}(h) - r_1) \) as the witness for \( h_2 \). Lemma 9 below can be applied in this scenario as \( h \uparrow r_2 \subseteq h \uparrow (\text{dom}(h) - r_1) \), as \( r_1 \subseteq \text{dom}(h) \) and \( r_2 \subseteq \text{dom}(h) \).

**Lemma 9.** Let \( \Gamma \) be a well-formed type environment. Let \( a \) be an SSL assertion, and \( (\sigma, h) \) be a \( \Gamma \)-state. Let \( h' \) be a heap, such that \( h \subseteq h' \). Then \( \sigma, h \models_\sigma a \Rightarrow \sigma, h' \models_\sigma a \).

The semantic footprints for assertions in SSL are derived in Lemma 10 based on the SL semantics in terms of a heap and a region.

**Lemma 10.** Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) be a state, and let \( e, b \) and \( a \) be an SSL expression, a Boolean expression, and an assertion. Then:

1. \( \text{MinReg}(b, \sigma, h) = \emptyset \).
2. if \( \sigma, h \models_\sigma x. f \mapsto e \), then \( \text{MinReg}(x. f \mapsto e, \sigma, h) = \{ (\sigma(x), f) \} \).
3. if \( \sigma, h \models_\sigma a_1 \ast a_2 \), then \( \text{MinReg}(a_1 \ast a_2, \sigma, h) = \text{MinReg}(a_1, \sigma, h) \cup \text{MinReg}(a_2, \sigma, h) \).
4. if \( \sigma, h \models_\sigma a_1 \land a_2 \), then \( \text{MinReg}(a_1 \land a_2, \sigma, h) = \text{MinReg}(a_1, \sigma, h) \cup \text{MinReg}(a_2, \sigma, h) \).
5. if \( \sigma, h \models_\sigma b \Rightarrow a \) and \( \sigma, h \not\models_\sigma b \), then \( \text{MinReg}(b \Rightarrow a, \sigma, h) = \text{MinReg}(a, \sigma, h) \).
6. if \( \sigma, h \models_\sigma b \Rightarrow a \) and \( \sigma, h \not\models_\sigma b \), then \( \text{MinReg}(b \Rightarrow a, \sigma, h) = \emptyset \).
7. if \( \sigma, h \models_\sigma \exists x. (y.f = x \ast a) \), then \( \text{MinReg}(\exists x. (y.f = x \ast a), \sigma, h) = \text{MinReg}(y.f \mapsto x \ast a, \sigma[x \mapsto h[\text{Env}][\Gamma \vdash y : T](\sigma), f]], h) \);

Moreover, \( \sigma, h \models_\sigma a \) iff \( (h \uparrow \text{MinReg}(a, \sigma, h)) \models_\sigma a \).
The proof from the left to the right of the above equivalences can be proved by cases on the structure of \( a \), which is the seven cases in Lemma 10 on the previous page, and the converse can be proved using Lemma 9 on the preceding page.

O’Hearn et al. [73] note that for the soundness of proofs under hypothesis, assertions used in preconditions and resource invariants need to be supported (Theorem 26 [73, p. 11:44]). Thus to reason about programs using specifications of other modules specified by SL, only supported assertions should be considered. This section establishes the connection between supported assertions and assertions in SSL.

The following recalls the definition of supported and intuitionistic assertions in the work of O’Hearn et al. [73].

**Definition 14 (Supported).** Let \( \Gamma \) be a well-formed type environment. An assertion \( a \) is supported if and only if for all \( \Gamma \)-states \( (\sigma, h) \), when \( h \) has a subheap \( h_0 \subseteq h \) such that \( \sigma, h_0 \models^\Gamma_s a \), then there is at least subheap \( h_a \subseteq h \) with \( \sigma, h_a \models^\Gamma_s a \) such that for all subheaps \( h' \subseteq h \), if \( \sigma, h' \models^\Gamma_s a \), then \( h_a \subseteq h' \).

The definition means that, given a state \( (\sigma, h) \) and an assertion \( a \), for any pair of \( h \)'s sub heaps, \( h_1 \) and \( h_2 \), such that \( \sigma, h_1 \models^\Gamma_s a \) and \( \sigma, h_2 \models^\Gamma_s a \), if \( h_a = h_1 \cap h_2 \) and \( \sigma, h_a \models^\Gamma_s a \), then \( a \) is supported. In other words, \( a \) has a greatest lower bound heap that makes it \( a \) true, then \( a \) is supported.

The definition of semantic footprint can be interpreted in a similar way. Consider a given state \( (\sigma, h) \), and any pair of regions \( r_1 \) and \( r_2 \) where \( r_1 \subseteq dom(h) \) and \( r_2 \subseteq dom(h) \), and a separation logic assertion \( a \), such that \( \sigma, (h \upharpoonright r_1) \models^\Gamma_s a \) and \( \sigma, (h \upharpoonright r_2) \models^\Gamma_s a \). Let \( r \) be the glb of \( r_1 \) and \( r_2 \), such that such that \( r \subseteq r_1 \cap r_2 \). If \( \sigma, (h \upharpoonright r) \models^\Gamma_s a \), then \( a \) has a semantic footprint. The following theorem summaries this. The proof is found in Appendix D.

**Theorem 5.** An assertion in SL is supported if and only if it has semantic footprint.
SSL assertions are supported by Theorem 5. This property provides the soundness of the hypothetical frame rule for Hoare triple judgment under certain hypothesis [72, 73].

### 6.2.3 Encoding SSL Assertions

This section constructs region expressions that can syntactically denote semantic footprints for SSL assertions, and shows the translation from SSL to UFRL (or FRL). The footprint of an implication \( b \Rightarrow a \) technically should include the footprint of \( b \). However, since \( b \)'s footprint is \( \text{region}\{\} \), the definition ignores it.

**Definition 15** (Semantic Footprint Function for SSL). *Let \( e, b \) and \( a \) be an SSL expression, a Boolean expression, and an assertion. Then the semantic footprint function for each SSL assertion is defined as follows.*

\[
\begin{align*}
\text{fpt}_s(b) & = \text{region}\{\} \\
\text{fpt}_s(x.f \leftrightarrow e) & = \text{region}\{x.f\} \\
\text{fpt}_s(b \Rightarrow a) & = b \oplus \text{fpt}_s(a) : \text{region}\{\} \\
\text{fpt}_s(a_1 * a_2) & = \text{fpt}_s(a_1) + \text{fpt}_s(a_2) \\
\text{fpt}_s(a_1 \land a_2) & = \text{fpt}_s(a_1) + \text{fpt}_s(a_2) \\
\text{fpt}_s(\exists x.(y.f = x * a)) & = \text{region}\{y.f\} + \text{fpt}_s(a)[y.f/x]
\end{align*}
\]

However, the defining clause for implication is technically suspect, because the SSL Boolean expression \( b \) is technically not an UFRL expression. However, it is obvious that the identity map is a semantics-preserving translation of pure Boolean expressions as shown below.

**Definition 16** (Mapping from SSL to UFRL (or FRL) assertions). *A function, \( TR \), syntactically*
maps from SSL to UFRL as follows:

\[
\begin{align*}
\text{TR}[x] &= x, & \text{TR}[n] &= n, & \text{TR}[\text{null}] &= \text{null} \\
\text{TR}[e_1 = e_2] &= \text{TR}[e_1] = \text{TR}[e_2] \\
\text{TR}[e_1 \neq e_2] &= \text{TR}[e_1] \neq \text{TR}[e_2] \\
\text{TR}[x \mapsto e] &= \text{TR}[x].f = \text{TR}[e] \\
\text{TR}[a_1 \ast a_2] &= \text{TR}[a_1] \& \& \text{TR}[a_2] \& \& (\text{fpt}_s(a_1) \; \Rightarrow \; \text{fpt}_s(a_2)) \\
\text{TR}[a_1 \wedge a_2] &= \text{TR}[a_1] \& \& \text{TR}[a_2] \\
\text{TR}[b \Rightarrow a] &= \text{TR}[b] \Rightarrow \text{TR}[a] \\
\text{TR}[\exists x.(y.f \mapsto x \ast a)] &= \exists x.(\text{TR}[y.f \mapsto x] \& \& \text{TR}[a] \& \& (\text{region}(y.f) \; \Rightarrow \; \text{fpt}_s(a)))
\end{align*}
\]

Lemma 11 and Lemma 12 state that the meaning of pure expressions and pure Boolean assertions are preserved in this translation, and are preserved under heap extension. Hence, \(e\) and \(\text{TR}[e]\), as well as \(b\) and \(\text{TR}[b]\) can be used interchangeably.

**Lemma 11.** Let \(\Gamma\) be a well-formed type environment. Let \(\sigma\) be a store. Let \(e\) be an expression in SSL. Then \(E_s(\Gamma \vdash e : T)(\sigma) = E(\Gamma \vdash \text{TR}[e] : T)(\sigma)\).

**Lemma 12.** Let \(\Gamma\) be a well-formed type environment. Let \((\sigma, h)\) be a \(\Gamma\)-state, and \(H\) be a heap such that \(h \subseteq H\). Let \(b\) be a pure assertion in SSL. Then \(\sigma, h \models^\sigma b\; \Leftrightarrow \; \sigma, h \models^\Gamma \text{TR}[b] \; \Leftrightarrow \; H \models^\Gamma \text{TR}[b]\).

The following theorem shows that the semantics of the semantic footprint function, \(\text{fpt}_s(a)\), is its semantic footprint in a given state, where \(a\) is true. Its proof can be done by induction on the structure of assertions. With this theorem, henceforth, a semantic footprint is just called the “footprint”.

**Theorem 6.** Let \(\Gamma\) be a well-formed type environment. Let \(a\) be an assertion in SSL, and let \((\sigma, h)\) be a \(\Gamma\)-state. If \(\sigma, h \models^\sigma a\), then \(a\) has a semantic footprint in \((\sigma, h)\), and this semantic footprint is \(\text{MinReg}(a, \sigma, h) = E(\Gamma \vdash \text{fpt}_s(a) : \text{region})(\sigma)\).
The following corollary show that given a state where \( a \) is true, \( a \)'s semantic footprint is a subset of the domain of the heap. This property is essential for the proof of the encoding for separating conjunction in Theorem 7.

**Corollary 4.** Let \( \Gamma \) be a well-formed type environment. Let \( a \) be an assertion in SSL. Let \( (\sigma, h) \) be a \( \Gamma \)-state. If \( \sigma, h \models^\Gamma_a \), then \( \mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma) \subseteq \text{dom}(h) \).

The following corollary shows that \( \mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma) \) is another candidate for the region \( r \) needed in Theorem 4. As \( fpt_s(a) \) gives the semantic footprint for each \( a \), the corollary can be proved by Lemma 10 and Theorem 6.

**Corollary 5.** Let \( \Gamma \) be a well-formed type environment. Let \( a \) be an assertion in SSL. Let \( (\sigma, h) \) be a \( \Gamma \)-state. Then \( \sigma, h \models^\Gamma_a \) iff \( \sigma, h \models_\Gamma (\mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma)) \).

The following theorem shows that \( \text{TR} \) is an isomorphism of SSL assertions into UFRL in the sense that the translation preserves validity. The proof about separating conjunction is the most interesting one as it partitions heaps. The translated expression consists of two conjunctions. The first one checks the value of the two assertions. The second one says that their footprints are disjoint. The proof for this separating conjunction case is found in appendix E. The proof for the existential case needs the substitution laws for assertions that are not surprising and are found in the KIV formal proof [6], and thus are omitted.

**Theorem 7.** Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) be a \( \Gamma \)-state. Let \( a \) be an assertion in SSL. Then \( \sigma, h \models^\Gamma_a \) iff \( \sigma, h \models^\Gamma \text{TR}[^\Gamma a] \).

Fig. 6.1 summarizes the previous results, where \( h = H \upharpoonright (\mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma)) \). The \( r \) is found for the SL's semantics for Theorem 4, which is \( \mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma) \); since it has been proved that \( \sigma, h \models^\Gamma_a \) if and only if \( \sigma, H \upharpoonright (\mathcal{E}[^\Gamma \vdash fpt_s(a) : \text{region}]](\sigma)) \models^\Gamma_a \), it
must be that \( h = H \upharpoonright (E \models \Gamma \vdash fpt_s(a) : \text{region})(\sigma) \). In addition, by Corollary 5, it must be true that \( \sigma, h \models^\Gamma_s a \iff \sigma, h \models (E \models \Gamma \vdash fpt_s(a) : \text{region})(\sigma) \models^\Gamma_s a \). Furthermore, by Theorem 7 on the preceding page twice, it must be true that \( \sigma, h \models (E \models \Gamma \vdash fpt_s(a) : \text{region})(\sigma) \models^\Gamma_s a \iff \sigma, h \models^\Gamma \text{TR}[a] \). Therefore, by transitivity, it must be true that \( \sigma, h \models^\Gamma \text{TR}[a] \iff \sigma, H \models (E \models \Gamma \vdash fpt_s(a) : \text{region})(\sigma) \models^\Gamma \text{TR}[a] \).

Figure 6.1: A summary of results on encoding assertions

6.2.4 SSL Proofs Review and Approach

This section encodes SSL’s axioms and rules into those in UFRL, and shows that encoded SSL axioms are derivable and that the encoding translates proofs in SSL into proofs in UFRL.

The correctness judgment of SSL, a Hoare-formula \( \{a\} S \{a'\} \), means that \( S \) is partially correct, and \( S \) can only access the regions that are guaranteed by \( a \). Consider the region guaranteed by \( a \) as its implicit frame. Thus, the proof obligation is to show that the following encoding into UFRL is valid (in Section 6.2.5):

\[
\vdash^\Gamma_s \{a\} S \{a'\} \iff \\
\vdash^\Gamma_u \left[ \text{reads}_{fpt_s(a)} \right] \{\text{TR}[a]\} S \{\text{TR}[a']\} \left[ \text{modifies}(\text{MV}(S), fpt_s(a)), \text{fresh}(fpt_s(a') \setminus r) \right]
\]

where \( r \) is a region variable, such that \( \text{TR}[a] \Rightarrow r = fpt_s(a) \) and \( r \notin \text{MV}(S) \)

(6.1)

where \( \text{MV}(S) \) is the set of variables that \( S \) may modify, and \( r \) snapshots the set of locations of \( fpt_s(a) \) in the pre-state. This translation is not the only way to establish the equivalence, e.g., the
read effects can be anything from $\emptyset$ to $fpt_s(a)$. This encoding corresponds to the definition of validity for Hoare-formula in SSL, which is presented next.

The definition of validity for SL Hoare-formulas uses the notion of partial correctness that are used for FRL and UFRL: statements are not permitted to encounter errors in states that satisfy the precondition, but may still loop forever.

**Definition 17** (Validity of SSL Hoare-formula). Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement. Let $a$ and $a'$ be assertions in SSL. Let $(\sigma, H)$ be a $\Gamma$-state. Then $\{a\}S\{a'\}$ is valid in $(\sigma, H)$, written $\sigma, H \vdash_{ss}^\Gamma \{a\}S\{a'\}$, if and only if whenever $\sigma, H \vdash_{ss}^\Gamma a$, then $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \neq err$ and if $(\sigma', H') = \mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H)$, then $\sigma', H' \vdash_{ss}^{\Gamma'} a'$.

A SSL Hoare-formula $\{a\}S\{a'\}$ is valid, written $\vdash_{ss}^\Gamma \{a\}S\{a'\}$, if and only if for all states $(\sigma, H) :: \sigma, H \vdash_{ss}^\Gamma \{a\}S\{a'\}$.

The locality properties [73, 87] of SSL Hoare-formula are:

1. **Safety Monotonicity**: for all states $(\sigma, H)$ and heaps $H'$, such that $H \bot H'$, if $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \neq err$, then $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H \cdot H') \neq err$.

2. **Termination Monotonicity**: for all states $(\sigma, H)$ and heaps $H'$, such that $H \bot H'$, if $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H)$ terminates normally, then $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H \cdot H')$ terminates normally.

3. **Frame Property**: for all states $(\sigma, H_0)$ and heaps $H_1$, such that $H_0 \bot H_1$, if $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H_0) \neq err$ and $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H_0 \cdot H_1) = (\sigma', H')$, then there is a subheap $H'_0 \subseteq H'$ such that $H'_0 \bot H_1, H'_0 \cdot H_1 = H'$, and $\mathcal{MS}[\Gamma \vdash S : ok(\Gamma')](\sigma, H_0) = (\sigma', H'_0)$.

Hoare-style proof rules for SSL are found in Fig. 6.2, following Parkinson’s work [77]. In the figure, the shorthand $new_s(T, x)$ means $x.f_1 \mapsto \text{default}(T_1) \ast \cdots \ast x.f_n \mapsto \text{default}(T_n)$, where
the \( f_i : T_i \) are defined by \((f_1 : T_1, \ldots, f_n : T_n) = \text{fields}(T)\). SSL expressions \((e)\) are used in the syntax of the statements, instead of FRL expressions \((E)\), although the statements of SSL are those of FRL, the expressions have the same syntax and meaning, by Lemma 11.

The following lemma states the frame property of SL Hoare-formulas semantically. It is used in the proof of Lemma 8 later. The proof is found in Appendix F.

**Lemma 13.** Let \( \Gamma \) be a well-formed type environment. Let \( a \) and \( a' \) be assertions and \( S \) be a statement, such that \( \Gamma \models_s \{a\} S \{a'\} \). Let \( (\sigma, H) \) be a \( \Gamma \)-state. If \( \sigma, H \models_s a \) and \( \mathcal{M} \mathcal{S}[(\Gamma \models S : \text{ok}(\Gamma'))](\sigma, H) = (\sigma', H') \), then:

1. for all \( x \in \text{dom}(\sigma) \), if \( \sigma'(x) \neq \sigma(x) \), then \( x \in \text{MV}(S) \).
2. for all \( (o, f) \in \text{dom}(H) \), if \( H'[o, f] \neq H[o, f] \), then \( (o, f) \in \mathcal{E}[(\Gamma \models \text{fpt}_s(a) : \text{region})](\sigma) \).
3. for all \( (o, f) \in (\mathcal{E}[(\Gamma' \models \text{fpt}_s(a') : \text{region})](\sigma') - \mathcal{E}[(\Gamma \models \text{fpt}_s(a) : \text{region})](\sigma)) \), it is that \( (o, f) \in (\text{dom}(H') - \text{dom}(H)) \).

There are several lemmas connecting FRL and UFRL separation operator (\( \backslash / \)) to SL’s separating conjunction operator (\( * \)). These lemmas are used to prove the frame rule case of the Theorem that the translation between SSL and UFRL preserves provability (Theorem 9 in Section 6.2.5).

The following lemma says that the footprints of assertions in a separating conjunction are also separated in the sense of FRL’s separation operator.

**Lemma 14.** Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) be a \( \Gamma \)-state. Let \( a_1 \) and \( a_2 \) be assertions in SSL. Then

\[
\sigma, h \models_s a_1 * a_2 \text{ implies } \sigma, h \models_s \text{efs}(\text{TR}[a_2]) \mid \text{modifies fpt}_s(a_1)
\]

Informally, the proof goes as follows. By the semantics of separating conjunction, it is known that \( a_1 \) and \( a_2 \) hold on disjoint heaps, say \( h_1 \) and \( h_2 \), respectively. By Corollary 4, it must be true that
\[(SKIP_s) \vdash_s ^\Gamma \{true\} \text{skip}; \{true\}\]

\[(VAR_s) \vdash_s ^\Gamma \{true\} \text{var } x : T; \{x = \text{default}(T)\}\]

\[(ALLOC_s) \vdash_s ^\Gamma \{a\} x := \text{new } T; \{a * \text{new}_s(T, x)\}, \text{where } x \notin \text{FV}(a)\]

\[(ASGN_s) \vdash_s ^\Gamma \{true\} x := e; \{x = e\}, \text{where } x \notin \text{FV}(e)\]

\[(UPD_s) \vdash_s ^\Gamma \{x.f \mapsto \_\} x.f := e; \{x.f \mapsto e\}\]

\[(ACC_s) \vdash_s ^\Gamma \{x'.f \mapsto z\} x := x'.f; \{x = z * x'.f \mapsto z\}, \text{where } x \neq x', x' \neq z \text{ and } x \neq z\]

\[(IF_s) \quad \quad \vdash_s ^\Gamma \{a \land e\} S_1 \{a'\}, \quad \vdash_s ^\Gamma \{a \land \neg e\} S_2 \{a'\} \quad \quad \vdash_s ^\Gamma \{a\} \text{if } S_1 \text{else } S_2 \{a'\}\]

\[(WHILE_s) \quad \quad \vdash_s ^\Gamma \{I \land e\} S \{I\} \quad \quad \vdash_s ^\Gamma \{I\} \text{while } e \{S\} \{I \land \neg e\}\]

\[(SEQ_s) \quad \quad \vdash_s ^\Gamma \{a\} S_1 \{b\}, \quad \vdash_s ^\Gamma \{b\} S_2 \{a'\} \quad \quad \vdash_s ^\Gamma \{a\} S_1 S_2 \{a'\}\]

\[
\begin{align*}
\text{FV}(x) &= \{x\} & \text{FV}([null]) &= \emptyset & \text{FV}(n) &= \emptyset \\
\text{FV}(e_1 = e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2) & \text{FV}(e_1 \neq e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(x.f \mapsto e) &= \{x\} \cup \text{FV}(e) & \text{FV}(a_1 * a_2) &= \text{FV}(a_1) \cup \text{FV}(a_2) \\
\text{FV}(a_1 \land a_2) &= \text{FV}(a_1) \cup \text{FV}(a_2) & \text{FV}(b \Rightarrow a) &= \text{FV}(b) \cup \text{FV}(a) \\
\text{FV}(\exists x.y.f = x * a) &= (\{y\} \cup \text{FV}(a)) - \{x\}
\end{align*}
\]

Figure 6.2: The axioms and proof rules for statements in SSL [77]
\[ \mathcal{E}[\Gamma \vdash fpt_s(a_1) : \text{region}](\sigma) \subseteq \text{dom}(h_1). \] So the following holds.

\[ \text{for all reads } RE \leq \text{efs}(\text{TR}[a_2]) :: RE \equiv fpt_s(a_1). \] (6.2)

In addition, by definition of separator (Fig. 3.7), the following must be true

\[ \text{for all reads } X \leq \text{efs}(\text{TR}[a_2]) :: \text{reads } X/\text{modifies} fpt_s(a_1). \] (6.3)

Using Eq. (6.3) together with Eq. (6.2) and the definition of separator (Fig. 3.7), proves that \( \text{efs}(\text{TR}[a_2]) \vdash \text{modifies} fpt_s(a_1). \)

The above lemma handles locations on the heap, but the frame rule also concerns variables, which are the subject of the following two lemmas.

The following lemma states that free variables are preserved by the encoding. It can be proved by induction on the structure of SSL assertions.

**Lemma 15.** Let \( a \) be an assertion in SSL. Then \( \text{FV}(a) = \text{FV}(\text{TR}[a]) \).

The following lemma shows that the set of variables in a framed assertion (\( c \) in the frame rule of SSL) are such that \( \text{readVar}(\text{efs}(\text{TR}[c])) \) is a subset of \( \text{FV}(\text{TR}[c]) \). The lemma is proved by induction on the structure of SSL assertions.

**Lemma 16.** Let \( c \) be an assertion in SSL, then \( \text{readVar}(\text{efs}(\text{TR}[c])) \subseteq \text{FV}(\text{TR}[c]) \).

### 6.2.5 Translating SSL Proofs into UFRL

The following theorem shows that SSL Hoare formulas of the form \( \{a\} S\{a'\} \) can be translated into UFRL, by using read effect \( fpt_s(a) \), write effect \( fpt_s(a), \text{MV}(S) \) and fresh effect \( fpt_s(a') - r \), where \( r \) snapshots the set of locations of \( fpt_s(a) \) in the pre-state, and that the translation preserves validity. As can be seen in the lemma, a kind of converse holds, as some forms of UFRL Hoare formula translate back into SSL. The proof is found in Appendix G.
Theorem 8. Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement, and let $a$ and $a'$ be assertions in SSL, such that $\Gamma \vdash^s a \{a\} S \{a'\}$. Let $r$ be a region variable. Let $(\sigma, H)$ be $\Gamma$-state. If $\sigma, H \vdash^\Gamma \text{TR}[a] \Rightarrow r = \text{fpt}_s(a)$ and $r \notin \text{MV}(S)$, then

$$
\sigma, H \vdash^\Gamma a \{a\} S \{a'\} \iff \\
\sigma, H \vdash^\Gamma \text{reads}(\text{fpt}_s(a)) \\
\sigma, H \vdash^\Gamma \{\text{TR}[a]\} S \{\text{TR}[a']\} \\
[\text{modifies}(\text{fpt}_s(a), \text{MV}(S)), \text{fresh}(\text{fpt}_s(a') - r)]
$$

Def. 18 shows a syntactic mapping from the axioms and rules of SSL to those of UFRL. This mapping translates SSL axioms and rules into those of UFRL, however, the encoded ALLOC rule is an exception. UFRL has a special variable, alloc, that keeps track of the set of allocated locations globally; i.e. alloc is the domain of the heap. It is updated when executing the new statement. However, SSL does not have such a variable. Thus, the write effect of the encoded ALLOC adds "modifies alloc" to the frame condition.

Definition 18 (Syntactic Mapping from SSL to UFRL). Let $\Gamma$ be a well-formed type environment. Let $a$ and $a'$ be assertions in SSL. A syntactic mapping, $\text{TR}_s[\text{-}]$, from SSL axioms and rules to those of UFRL is defined below:

$$
\text{TR}_s[\vdash^\Gamma_a \{a\}] x := \text{new } T; \{a * \text{new}_s(T, x)\} = \\
[\text{reads}(\text{fpt}_s(a))] \\
\vdash^\Gamma_a \{\text{TR}[a]\} x := \text{new } T; \{\text{TR}[a * \text{new}_s(T, x)]\} \\
[\text{modifies } x, \text{modifies alloc, fresh}(\text{fpt}_s(\text{new}_s(T, x)))]
$$

$$
\text{TR}_s[\vdash^\Gamma_a \{a\} S \{a'\}] = \\
\vdash^\Gamma_a [\text{reads}(\text{fpt}_s(a))\{\text{TR}[a]\}] S \{\text{TR}[a']\} [\text{modifies}(\text{fpt}_s(a), \text{MV}(S)), \text{fresh}(\text{fpt}_s(a') - r)],
$$

where $r$ is a region variable such that $r \notin \text{MV}(S)$, $\text{TR}[a] \Rightarrow r = \text{fpt}_s(a)$ and

$S \neq x := \text{new } T;$.  

77
For the SSL rules, let $h_1, \ldots, h_n$ be hypotheses and $c$ be conclusion; then the syntactic mapping from a SSL rule to a UFRL rule is defined as below:

\[
\frac{\vdash^s \Gamma h_1, \ldots, \vdash^s h_n}{\vdash^s c} = \frac{\vdash^s \Gamma h_1, \ldots, \vdash^s h_n}{\vdash^s c}
\]

\[
\text{Theorem 9. Each translated SSL axiom is derivable, and each translated rule is admissible in the UFRL proof system.}
\]

The proof is by induction on the derivation and by cases in the last rule used, and can be found in Appendix H. The sequential case is not intuitive. An example is to show that how to use $SEQI_u$ to prove that the encoded sequence rule is admissible in UFRL. Particularly, the proof strategy of proving the side conditions on immunity is explained. Consider the example $x := y; x.f := 5; x.f := 6$. Assume $y.f \rightarrow 3$ before executing the first statement. In the proof, the following derivation can be achieved in SSL.

\[
\vdash^s \{y.f \rightarrow 3\} x := y; x.f := 5; \{x = y \& x.f \rightarrow 5\}
\]

\[
\vdash^s \{x = y \& x.f \rightarrow 5\} x.f := 6; \{x = y \& x.f \rightarrow 6\}
\]

\[
\vdash^s \{y.f \rightarrow 3\} x := y; x.f := 5; x.f := 6; \{x = y \& x.f \rightarrow 6\}
\] (SEQ$_s$)

By Def. 18, the two premises are encoded to

\[
\text{[reads region}\{y.f\}\text{]}
\]

\[
\vdash^u \{y.f = 3\} x := y; x.f := 5; \{x = y \& x.f = 5\}
\]

\[
\text{[modifies x, modifies region}\{y.f\}\text{]}
\]

\[
\text{reads region}\{x.f\}\text{]}
\]

\[
\vdash^u \{x = y \& x.f = 5\} x.f := 6; \{x = y \& x.f = 6\}
\]

\[
\text{[modifies region}\{x.f\}\text{]}
\]

(6.4)

(6.5)
And the proof obligation is to show that from Eq. (6.4) and Eq. (6.5), the translated conclusion below can be derived.

\[
\Gamma \uparrow \text{reads region}\{y.f\}
\]

\[
\Gamma_u \{y.f = 3\} x := y; \ x.f := 5; x.f := 6; \ \{x = y \&\& x.f = 6\}
\]

(6.6)

\[
\Gamma \uparrow \text{modifies region}\{y.f\}
\]

The immune side conditions are not satisfied. However, according to the postcondition of Eq. (6.4), it is known that \(y = x\) and \(y\) is not modified by the statement in Eq. (6.4). Hence, the variable \(y\) is substituted for \(x\) in the effects of Eq. (6.5), using the consequence rule, and get:

\[
\Gamma \uparrow \text{reads region}\{y.f\}
\]

\[
\Gamma_u \{x = y \&\& x.f = 5\} \ x.f := 6; \ \{x = y \&\& x.f = 6\}
\]

(6.7)

\[
\Gamma \uparrow \text{modifies region}\{y.f\}
\]

Now the side conditions about immunity are true. Eq. (6.6) is derived by using the rule \(SEQI_u\). The proof strategy generalizes the approach used in the example. Let \(S_1, S_2\) be a sequential statement. The effects of \(S_2\) is re-written by replacing all the variables in \(MV(S_1)\), i.e., \(\pi\), with the variables \(z\), such that \(a' \Rightarrow z = \pi\) and \(z \cap MV(S_1) = \emptyset\), where \(a'\) is the postcondition for \(S_1\). The detailed proof is shown in Appendix H.

**Corollary 6.** *The meaning of a SSL judgment is preserved by the syntactic mapping.*

### 6.3 Extending the UFRL (FRL) Proof System with Separating Conjunction

This section presents another way to allow SL and UFRL (or FRL) to interoperate. SL style assertions can appear in UFRL (or FRL itself), without using the somewhat verbose encoding of separating conjunction discussed previously. Thus, this section adds separating conjunction to the syntax of the assertions. The semantics of separating conjunction in UFRL (or FRL) is defined, and is proved equivalent to the one in SSL. Then the read effects of separating conjunction is defined. And the framing judgment is proved sound.
6.3.1 Extending the Syntax and the Semantics

To have the ability to write SL style specifications in UFRL (or FRL), there is no need to add the points-to assertion to the syntax, because the points-to assertion, \( x.f \rightarrow e \), has the same semantics as the equality assertion \( x.f = e \). Thus, the syntax of assertions shown in Fig. 3.2 is extended as follows:

\[
P ::= \ldots \mid P_1 \ast P_2
\]

In the syntax, dots “…” denotes the material defined previously. Given a SSL assertion \( a, TR_{rr}a \), which replaces each occurrence of \( \rightarrow \) in \( a \) with \( = \), is a valid assertion in the extended UFRL (or FRL). To ease the notational burden, sometimes, the points-to assertion and the equality assertion are used interchangeably in examples when the context is clear.

The separating conjunction is a supported UFRL (or FRL) assertion in the following sense.

**Definition 19** (Supported UFRL (or FRL) Assertions). Let \( P \) be an assertion in UFRL (or FRL). \( P \) is supported if there exists an SSL assertion, \( a \), such that \( P = TR[a] \).

Defining the semantics of separating conjunction, \( P_1 \ast P_2 \), in UFRL (or FRL) requires a definition of its footprint, \( fpt(P_1 \ast P_2) \). The semantics of \( fpt \) is defined by using the inverse of the translation function, \( TR^{-1} \). This inverse exists because, by definition, \( TR \) is injective, as can be shown by induction on the structure of SSL assertions (see Def. 16). So, for each supported UFRL (or FRL) assertion, \( P \), by definition there is some SSL assertion \( a \) such that \( TR(a) = P \); thus for each supported UFRL (or FRL) assertion \( P \), \( TR^{-1}[P] \) is defined to be the SSL assertion \( a \) such that \( TR(a) = P \).

Using \( TR^{-1} \), here defines the semantic footprint function for supported UFRL (or FRL) assertions, \( P \), by \( fpt(P) = fpt_s(TR^{-1}[P]) \).

Finally, the semantics of separating conjunction is defined as follows for supported UFRL (or FRL)
assertions $P_1$ and $P_2$:

$$\sigma, H \models^\Gamma P_1 \ast P_2 \iff \sigma, H \models^\Gamma P_1 \text{ and } \sigma, H \models^\Gamma P_2 \text{ and } \sigma, H \models^\Gamma \text{fpt}(P_1) \ast \text{fpt}(P_2) \quad (6.8)$$

The following lemma shows that Eq. (6.8) is a correct semantics.

**Lemma 17.** Let $\Gamma$ be a well-formed type environment. Let $(\sigma, H)$ be a $\Gamma$-state. Let $P_1$ and $P_2$ be supported assertions in extended UFRL (or FRL). Then

$$\sigma, H \models^\Gamma P_1 \ast P_2 \iff \sigma, H \models^\Gamma \text{fpt}^{-1}(P_1) \ast \text{fpt}^{-1}(P_2).$$

**Proof.** We assume $\sigma, H \models^\Gamma P_1 \ast P_2$ and calculate it as follows.

$$\sigma, H \models^\Gamma P_1 \ast P_2 \iff \langle \text{by Eq. (6.8)} \rangle$$

$$\sigma, H \models^\Gamma P_1 \text{ and } \sigma, H \models^\Gamma P_2 \text{ and } \sigma, H \models^\Gamma \text{fpt}(P_1) \ast \text{fpt}(P_2) \iff \langle \text{by definition fpt}(P) = \text{fpt}_s(\text{fpt}^{-1}(P)) \rangle$$

$$\sigma, H \models^\Gamma P_1 \text{ and } \sigma, H \models^\Gamma P_2 \text{ and } \sigma, H \models^\Gamma \text{fpt}_s(\text{fpt}^{-1}(P_1)) \ast \text{fpt}_s(\text{fpt}^{-1}(P_2)) \iff \langle \text{by semantics of assertions in Fig. 3.2} \rangle$$

$$\sigma, H \models^\Gamma P_1 \& P_2 \& \text{fpt}_s(\text{fpt}^{-1}(P_1)) \ast \text{fpt}_s(\text{fpt}^{-1}(P_2)) \iff \langle \text{by definition of the syntactical mapping from SSL to UFRL (Def. 16), as } P_1 \text{ and } P_2 \text{ are supported} \rangle$$

$$\sigma, H \models^\Gamma \text{fpt}^{-1}(P_1) \ast \text{fpt}^{-1}(P_2) \iff \langle \text{by Theorem 7} \rangle$$

$$\sigma, H \models^\Gamma \text{fpt}^{-1}(P_1) \ast \text{fpt}^{-1}(P_2)$$

Effects, Framing and Separator for SSL Formulas Recall that UFRL supports local reasoning by proving that the write effects of a statement are disjoint with the read effects of the predicates that describe the property of the program state. We define the read effects for $P_1 \ast P_2$ as follows:

$$\text{efs}(P_1 \ast P_2) = \text{efs}(P_1), \text{efs}(P_2) \quad (6.9)$$
Lemma 18 shows that the soundness of the frame validity (Def. 5), i.e., \( \text{true} \vdash^\Gamma \text{efs}(P_1 \ast P_2) \) is valid. The proof is by induction on the structure of assertions.

**Lemma 18 (Frame Soundness of Extended Assertions).** Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) and \( (\sigma', h') \) be two \( \Gamma \)-states. Let \( P \) be a supported assertion in extended UFRL. If \( (\sigma, h) \overset{\text{efs}(P)}{\equiv} (\sigma', h') \), then \( \mathcal{E}[\Gamma \vdash \text{fpt}(P) : \text{region}](\sigma) = \mathcal{E}[\Gamma \vdash \text{fpt}(P) : \text{region}](\sigma') \), and \( \sigma, h \vdash^\Gamma P \text{ iff } \sigma', h' \vdash^\Gamma P \).

The separating conjunction proves some properties about the separator (defined in Fig. 3.7).

**Lemma 19.** Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) be a \( \Gamma \)-state. Let \( P_1 \) and \( P_2 \) be supported assertions in extended UFRL. Then

\[
\sigma, h \vdash^\Gamma P_1 \ast P_2 \text{ implies } \sigma, h \vdash^\Gamma \text{efs}(P_2) // \text{modifies fpt}(P_1)
\]

Note that it is not valid that \( \sigma, h \vdash^\Gamma P_1 \ast P_2 \Rightarrow \sigma, h \vdash^\Gamma \text{efs}(P_2) // \text{modifies readVar(efs}(P_1)), \text{readR}(efs(P_1)) \). For example, let \( P_1 \) be \( x.f_1 = 4 \) and \( P_2 \) be \( x.f_2 = 5 \), and \( P_1 \ast P_2 \) is valid. Because \( \text{efs}(P_2) = \text{reads } x, \text{region}\{x.f_2\}, \text{and modifies readVar(efs}(P_1)), \text{readR}(efs(P_1)) = \text{modifies } x, \text{region}\{x.f_1\}, \) they are not disjoint sets.

### 6.3.2 Proof Rules

In the following , assume that all UFRL assertions involved in separating conjunctions are supported. This section discusses the introduction rule for separating conjunction, which is as follows:

\[
(I_{sc}) \quad \begin{array}{c}
\vdash^\Gamma_{u} [\delta]\{P\} \ S \ \{Q\}[\varepsilon] \\
\vdash^\Gamma_{u} [\delta]\{P \ast R\} \ S \ \{Q \ast R\}[\varepsilon]
\end{array} \quad \begin{array}{c}
P \& \& R \Rightarrow \text{efs}(R) // \text{modifies fpt}(P), \\
P \& \& R \Rightarrow \text{efs}(R) // \varepsilon \text{ and }
\end{array}
\]

where \( P \& \& R \Rightarrow \text{efs}(R) // \varepsilon \) and

\[
Q \& \& R \Rightarrow \text{efs}(R) // \text{modifies fpt}(Q)
\]

The two extra side conditions are used to conclude \( P \ast R \) and \( Q \ast R \), which is justified by the following lemma:
Lemma 20 (Soundness). $I_{sc}$ is admissible in the extended UFRL proof system.

Proof. $I_{sc}$ can be derived as follows:

\[
\frac{\Gamma \vdash \delta}{ \Gamma \vdash \delta \{P\} S \{Q\}\{\varepsilon\} \quad \text{where } P & & R \Rightarrow \text{efs}(R)/\varepsilon}
\]

\[
\frac{\Gamma \vdash \delta \{P & & R\} S \{Q & & R\}\{\varepsilon\}}{\Gamma \vdash \delta \{P * \} S \{Q * \}\{\varepsilon\} \quad \text{where } (*)}
\]

(*) is $P & & R \Rightarrow \text{efs}(R)/\text{modifies fpt}(P)$ and $Q & & R \Rightarrow \text{efs}(R)/\text{modifies fpt}_s(Q)$

\[\square\]

Lemma 21. Let $\Gamma$ be a well-formed type environment. Let $P_1$ and $P_2$ be supported assertions in extended UFRL proof system, and $(\sigma, h)$ be a $\Gamma$-state. If $\sigma, h \models^\Gamma P_1$ and $\sigma, h \models^\Gamma P_2$ and $\text{efs}(P_2)/\text{modifies fpt}(P_1)$, then $\sigma, h \models^\Gamma P_1 * P_2$.

Consider the example in Fig. 6.3. The example specifies a linked-list using the separating conjunction while the method append is specified in the style of UFRL, where its read effects are omitted, and are considered as reads alloc↓. Adopting the convention of VeriFast [41], ?v and ?vllst declare (universally-quantified) variables v and vllst respectively that scope over the entire specification of append. In the body of append, right after the loop, the following must be true:

\[
(\text{lst (this, vllst)} \& & (\text{lstseg (this, curr)} * (\text{lst (curr, ?cvllst)} \& &
\text{curr.next = null})) \& & \text{lst (n, [v]), (6.10})
\]

which (by the definition of the predicate lst) implies:

\[
(\text{lst (this, vllst)} \& & (\text{lstseg (this, curr)} * \text{curr.val } \rightarrow ?e * \text{curr.next } \rightarrow \text{null}) \& & \text{lst (n, [v]), (6.11})
\]

83
**predicate** lst(n : Node<T>, se : seq<T>)
reads fpt(lst(n, se));
decreases |se|;
{
  (n = null ⇒ se = []) &&
  (n ≠ null ⇒ n.val → se[0] * lst(n.next, se[1..]))
}

**predicate** lstseg(s: Node<T>, e : Node<T>, se : seq<T>)
reads fpt(lstseg(s, e, se));
decreases |se|;
{
  (s = e && se = []) ||
  (s ≠ e && (s.val → se[0] * lstseg(s.next, e, se[1..])))
}

class Node<T> {
  var val: T;  var next: Node<T>;

  **method** append(n: Node<T>)
  requires lst(n, [?v]) * lst(this, ?vlst);
  modifies region{last().next};
  ensures this.valst = old(this.valst) + [n.val];
  ensures this.repr = old(this.repr) + n.repr;
  ensures this.valid();
  {
    var curr: Node<T>; curr := this;;
    while (curr.next ≠ null)
      invariant lstseg(this, curr) * lst(curr, ?cvlst);
      invariant fpt(lstseg(this, curr)) + fpt(lst(curr, cvlst)) =
        fpt(lst(this, vlst));
      { curr := curr.next; }
    curr.next := n;
  }

  **function** last() : Node<T>
  { /* ... */ }

  /* ... other methods omitted */
}

Figure 6.3: A linked-list example written in UFRL with separating conjunction
which implies the precondition of the rule $UPD_u$. Using the rules $UPD_u$ and $SubEff_u$, the following is derived:

\[
\begin{align*}
& [\text{reads } curr, \text{ alloc}], \\
& \Gamma \vdash u \{curr \neq \text{null}\} curr\text{:next} := n; \{curr\text{:next} = n\} \\
& [\text{modifies region}\{curr\text{:next}\}]
\end{align*}
\]

By $CONSEQ_u$ and $I_{sc}$, the following is derived:

\[
\begin{align*}
& [\text{reads } curr, \text{ alloc}], \\
& \{curr \neq \text{null} \land \text{lstseg}(\text{this}, curr) \land curr\text{:val} \rightarrow ?cv \land \text{lst}(n, [v])\} \\
& \vdash u \{curr\text{:next} := n; \{curr\text{:next} = n\} \\
& \{curr\text{:next} = n \land \text{lstseg}(\text{this}, curr) \land curr\text{:val} \rightarrow ?cv \land \text{lst}(n, [v])\} \\
& [\text{modifies region}\{curr\text{:next}\}]
\end{align*}
\]

and the postcondition of Eq. (6.13) implies, by the definition of $\text{lst}$,

\[
lst(curr,[curr\text{:val}]+[v]) \land \text{lstseg}(this,n) \quad (6.14)
\]

To prove the postcondition, consider the second loop invariant in Fig. 6.3:

\[
fpt(\text{lstseg}(\text{this}, curr)) + fpt(\text{lst}(curr,[curr\text{:val}])) = fpt(\text{lst}(\text{this}, \text{vlst})), \quad (6.15)
\]

Together with Eq. (6.14), at the end of the method body, it must be true that $fpt(\text{lstseg}(this,curr)) + fpt(\text{lst}(curr,[curr\text{:val}]+[v])) = fpt(\text{lst}(this, \text{vlst} + [v]))$, which implies the postcondition of the procedure.
6.3.3 Encoding SSL specifications:

Using separating conjunctions in extended UFRL, the SSL Hoare-formulas are encoded by substituting $\Rightarrow$ for $\Rightarrow$ as follows:

$$\Gamma_s \vdash \{a\} \times := \text{new } T; \{a * \text{new}_s(T, x) \text{ iff } \text{reads } \text{fpt}_s(a)\}$$

$$\Gamma_u \vdash \{a[=\rightarrow]\} \times := \text{new } T; \{(a * \text{new}_s(T, x))[=\rightarrow]\}$$

$$\Gamma_u \vdash \{a[=\rightarrow]\} \times := \text{new } T; \{(a * \text{new}_s(T, x))[=\rightarrow]\}$$

where $\text{MV}(S)$ is the set of variables that $S$ may modify, and $r$ snapshots the set of locations of $\text{fpt}_s(a)$ in the pre-state. The encoded $\text{ALLOC}$ rule has the similar exception to Section 6.2.5.

To avoid complicated formulas due to the translation, proofs of later examples use the rule $I_{sc}$ if frames are constructed by separating conjunctions, otherwise, the rule $FRM_u$ is used in the examples. The places where the rule $I_{sc}$ is used can be considered as using the rule $FRM_u$ as well due to our results.

6.3.4 Summary

We have introduced two approaches to supporting separating conjunctions: (1) encoding them into assertions in UFRL; (2) adding them to the syntax and extending the UFRL proof system. The second approach takes advantage of the first one’s results, and makes the UFRL assertions more concise.
CHAPTER 7: RECURSIVE PREDICATES

This chapter presents the treatment for inductive predicates. Many examples in SL feature inductive predicates, as do some examples in this dissertation. Thus, the connection between SL and UFRL (or FRL) needs to treat such inductively-defined predicates. As part of this treatment, the syntax of assertions is extended with a limited form of recursive predicates. And it is shown how to translate abstract function definitions and calls in SL to recursive predicate definitions and calls in UFRL (or FRL).

7.1 Recursive Predicates in UFRL (FRL)

The following grammar shows the extension of the assertions defined in Fig. 3.1. In the syntax, dots “...” denotes the material defined previously. The extension allows predicate declarations and calls to predicates in assertions ($P$).

$$
\begin{align*}
\text{Predicate} & ::= \text{predicate } p \left( x : T \right) \text{ reads } \delta ; \ [\text{decreases } G ; ] \{ \ P \ \} \\
\ P & ::= \ldots | \ p(G) | x.p(G)
\end{align*}
$$

where $p$ is the predicate name and $G$ is either an expression or a region expression (as in Fig. 2.1). Assume that predicate names are unique in each program. A restricted form of recursive definition is allowed; mutual recursion is not allowed. The decreases clause is used to prescribe an argument that becomes strictly smaller each time a recursive predicate is called. This treatment is similar to Dafny [55, 32]. The body of a predicate is just an assertion. To make sure the predicate is monotonic, recursive calls of predicates can only appear in positive positions (e.g., not on the left side of an implication). And the recursive calls to predicates are not allowed inside unbounded universal quantifiers [32].

\footnote{The content in this chapter is submitted to Formal Aspects of Computing.}
To keep the spirit of a two-valued logic, a recursive predicate is allowed to be used only if it is provably terminating. To prove it terminates, a well-founded relation on the domain of a recursive predicate is enforced, e.g., a subregion relation \((\preceq)\) is defined on the type `region`. One of the proof obligations of its body is to show that the argument, which the decreases clause specifies, to each recursive predicate call goes down in this ordering [32].

The semantic function `body(T, p)` maps a pair of a type `T` and a predicate name `p` to its definition, where `T` is a reference type. Global predicates are considered to be wrapped in a distinguished class `Object`. The semantic function `formals(T, p)` maps a pair of a type `T` and a predicate name `p` to its declared formal parameters, where `T` is a reference type. The semantic function `rd` maps a predicate name to its read effect. The notation \(x \mapsto y\) means pointwise mapping. A semantic function for assertion is defined below:

\[
\mathcal{E}_p : \text{Assertion Typing Judgment} \rightarrow \text{Store} \times \text{Heap} \rightarrow \{\text{true}, \text{false}\}
\]

The satisfaction relation defined in Fig. 3.2 is defined by

\[
\mathcal{E}_p[\Gamma \vdash P : \text{bool}](\varphi)\sigma, H \iff \sigma, H \models P
\]

The semantics of a predicate call is defined as follows.

\[
\begin{align*}
\mathcal{E}_p[\Gamma \vdash p(G) : \text{bool}](\sigma, H) &= (\text{fix} \varphi(\sigma', H') \cdot \mathcal{E}_p[\Gamma \vdash \text{body(Object, p)} : \text{bool}](\sigma', H')(\sigma(\text{formals(Object, p)} \mapsto \overline{\sigma})), H) \\
&\quad \text{where } \overline{\sigma} = \overline{\mathcal{E}[\Gamma \vdash G : T]}(\sigma)
\end{align*}
\]

(7.1)

\[
\begin{align*}
\mathcal{E}_p[\Gamma \vdash x.p(G) : \text{bool}](\sigma, H) &= \sigma(x) = o \text{ and } o \neq \text{null} \text{ and} \\
&\quad (\text{fix} \varphi(\sigma', H') \cdot \mathcal{E}_p[\Gamma \vdash \text{body(T, p)} : \text{bool}](\sigma', H')(\sigma(\text{this, formals(T, p)} \mapsto (o, \overline{\sigma}))), H) \\
&\quad \text{where } T = \text{type(o)} \text{ and } \overline{\sigma} = \overline{\mathcal{E}[\Gamma \vdash G : T]}(\sigma)
\end{align*}
\]

(7.2)
The read effects of predicate calls are defined as follows:

\[ efs(p(F)) = efs(F), \delta[F/\bar{z}] \quad \text{where} \quad \delta = \text{rd}(p) \quad \text{and} \quad \bar{z} = \text{formals(Object, } p) \]

\[ efs(x.p(F)) = \text{reads } x, efs(F), \delta[(x,F)/(\text{this}, \bar{z})] \]

\[ \quad \text{where} \quad \delta = \text{rd}(p) \quad \text{and} \quad \bar{z} = \text{formals(type}(\sigma(x)), p) \]

The rules \( LIntro_1_u \) and \( LIntro_2_u \) introduce the form of a predicate call to left-hand side of the judgment. The rules \( RIntro_1_u \) and \( RIntro_2_u \) introduce the form of a predicate call to the right-hand side of the judgment. The type environment \( \Gamma(x) \) is omitted in the judgment.

\[
\begin{align*}
(\text{LIntro}_1_u) & \quad P' \vdash_\Gamma P \\
& \quad \quad \frac{}{p(F) \vdash_\Gamma P} \quad \text{where} \quad P' = \text{body}(\text{Object}, p)[F/\text{formals(Object, } p)]
\end{align*}
\]

\[
\begin{align*}
(\text{LIntro}_2_u) & \quad x \neq \text{null} \&\& P' \vdash_\Gamma P \\
& \quad \quad \frac{}{x.p(F) \vdash_\Gamma P} \quad \text{where} \quad P' = \text{body}(\Gamma(x), p)[F/\text{formals}(\Gamma(x), p)]
\end{align*}
\]

\[
\begin{align*}
(\text{RIntro}_1_u) & \quad P \vdash_\Gamma P' \\
& \quad \quad \frac{}{P \vdash_\Gamma p(F)} \quad \text{where} \quad P' = \text{body}(\text{Object}, p)[F/\text{formals(Object, } p)]
\end{align*}
\]

\[
\begin{align*}
(\text{RIntro}_2_u) & \quad P \vdash_\Gamma x \neq \text{null} \&\& P' \\
& \quad \quad \frac{}{P \vdash_\Gamma x.p(F)} \quad \text{where} \quad P' = \text{body}(\Gamma(x), p)[F/\text{formals}(\Gamma(x), p)]
\end{align*}
\]

Lemma 22. The rules \( LIntro_1_u, LIntro_2_u, RIntro_1_u \) and \( RIntro_2_u \) are sound.

Proof. As the meaning of a predicate is defined by its body, i.e., the predicate is true if and only if its body is true, the four proof rules are sound. \qed

7.2 Inductive Definition in SSL

The following grammar shows the extension of the SSL syntax given in Def. 13. It allows predicate calls in assertions.
\[ a ::= \ldots | p_s(\overline{e}) \]

where \( p_s \) is the predicate name and \( \overline{e} \) are arguments. Apply the definition of “inductive definition set” from Brotherston’s work [21] to SSL as follows:

**Definition 20 (Inductive Definition).** Let an inductive predicate \( p_s(z : T) \) in SSL. Then \( p_s \) is a set of conjunction of inductive cases. Each inductive case is in the form \( b \Rightarrow a \).

The following shows a valid inductive definition in SSL, which has two inductive cases, where Def. 20 is instantiated with \( b_1 := (n = \text{null}), a_1 := (se = []) \), \( b_2 := (n \neq \text{null}) \) and \( a_2 := (\exists m. \ n.val \mapsto se[0] \ast n.next \mapsto m \ast \text{list}(m, se[1..])) \), where \( se \) is a sequence.

\[
\text{list}(n, se) \overset{\text{def}}{=} (n = \text{null} \Rightarrow se = []) \land \\
(n \neq \text{null} \Rightarrow (\exists m. n.val \mapsto se[0] \ast n.next \mapsto m \ast \text{list}(m, se[1..])))
\]

(7.3)

The semantic function \( \text{idf} \) maps a predicate name to its induction definition, which is the conjunction of inductive cases. The semantic function \( \text{formals}_s \) maps a predicate name to its formal parameters.

A semantic function for assertion is defined below:

\[
\mathcal{E}_a : a \rightarrow \text{Store} \times \text{Heap} \rightarrow \{\text{true}, \text{false}\}
\]

The satisfaction relation defined by Def. 11 is defined by

\[
\mathcal{E}_a[\Gamma \vdash a](\sigma, h) \iff \sigma, h \vdash_{\overline{a}} a
\]

The semantics of inductive predicate \( p_s(\overline{e}) \) is defined as follows:

\[
\mathcal{E}_a[\Gamma \vdash p_s(\overline{e})](\sigma, h) = (\text{fix}(\lambda (\sigma', h') : \mathcal{E}_a[\Gamma \vdash \text{idf}(p_s)](\sigma', h'))(\sigma(\text{formals}_s(p_s) \mapsto \overline{e}, h))
\]

where \( \overline{e} = \mathcal{E}_s[\Gamma \vdash e : T](\sigma) \)

(7.4)

Let \( b \Rightarrow a \) be one of the inductive cases of predicate \( p_s(\overline{e}) \), then the rules \( L\text{Intro}_s \) and \( R\text{Intro}_s \) introduce the form of a predicate call to the left-hand side and the right-hand side of the judgment.
respectively.

\[
\frac{a \vdash_{\Gamma} a'}{p_s(\overline{e}) \vdash_{\Gamma} a'} \quad \text{where } a = (b \Rightarrow a)[\overline{e}/\text{formals}_s(p_s)]
\]

\[
\frac{a' \vdash_{\Gamma} a}{\text{where } a = (b \Rightarrow a)[\overline{e}/\text{formals}_s(p_s)]}
\]

7.3 Encoding

The translation of recursive predicate call is defined as follows:

\[
\text{TR}[[p_s(\overline{e})]] = p_s(\text{TR}[\overline{e}]).
\] (7.5)

Assume an inductive predicate \( p_s \) has \( n \) inductive cases. Fig. 7.1 on the following page shows the encoding of \( p_s \)'s inductive definition to a recursive predicate declaration in UFRL. The body of the generating recursive predicate is a conjunction of each encoded inductive case. The notation \( RE_1 ! ! \ldots ! ! RE_n \) means pairwise region disjointness. For each inductive predicate \( p_s : z : T \rightarrow \text{bool} \), there is a region function with the signature \( \text{region}_{p_s} : z : T \rightarrow \text{region} \) that computes the semantic footprint of the predicate \( p_s \)'s definition. The function's body is the semantic footprint of \( p_s \)'s definition. The region function is also used in the \text{decreases} clause. Fig. 7.2 on the next page shows the encoding of the inductive predicate in Eq. (7.3). Note that the invalid syntax can be solved by program instruments.

By the definition of \( \text{region}_{p_s} \) and the results in Section 6.2.3, it is known that \( \text{TR}[[p_s]] \vdash_{\Gamma} \text{region}_{p_s} \text{frm} \text{TR}[[p_s]]. \)

\textbf{Lemma 23.} Let \( \Gamma \) be a well-formed type environment. Let \( (\sigma, h) \) be a \( \Gamma \)-state, and \( p_s \) be an inductive predicate in SSL. Then

\[
\mathcal{E}_a[[\Gamma \vdash p_s(\overline{e}) : \text{bool}](\sigma, h)] = \mathcal{E}_p[[\Gamma \vdash \text{TR}[[p_s(\overline{e})]] : \text{bool}](\sigma, h)].
\]
\begin{align*}
\TR[p_s] &= \text{function } \text{region}_p_s(z:T) : \text{region} \\
&\quad \text{reads } \text{region}_p_s(z) ; \\
&\quad \text{decreases } \text{region}_p_s(z) ; \\
&\quad \{ \text{&& } \text{\textcircled{1} TR} \text{rr} \} \} \\
\end{align*}

Figure 7.1: Translation of inductive definition in SSL to recursive predicates in UFRL

\begin{align*}
\text{predicate } \text{list}(n : \text{Node}<T>, \text{se}: \text{sequence}<T>) \\
&\quad \text{reads } \text{region}_\text{list}(n, \text{se}) ; \\
&\quad \text{decreases } \text{region}_\text{list}(n, \text{se}) ; \\
&\quad \{ \\
&\quad \quad (n = \text{null } \Rightarrow \text{se} = []) \text{ && } \\
&\quad \quad (n \neq \text{null } \Rightarrow (\exists m. n.\text{val} = \text{se}[0] \text{ && } n.\text{next} = m \text{ && } \text{list}(m, \text{se}[1..]) \\
&\quad \quad \text{&& region}{n.\text{val}} !! \text{ region}{n.\text{next}} !! \text{ region}_\text{list}(m, \text{se}[1..]))) \\
&\quad \} \\
\text{function } \text{region}_\text{list}(n : \text{Node}<T>, \text{se}: \text{sequence}<T>) \\
&\quad \text{reads } \text{region}_\text{list}(n, \text{se}) ; \\
&\quad \text{decreases } \text{region}_\text{list}(n, \text{se}) ; \\
&\quad \{ \\
&\quad \quad \text{ret } := \\
&\quad \quad \text{if } (n = \text{null}) \text{ then region}{} + \\
&\quad \quad \text{if } (n \neq \text{null}) \text{ then} \\
&\quad \quad \text{region}{n.\text{val}} + \text{region}{n.\text{next}} + \text{region}_\text{list}(n.\text{next}, \text{se}[1..]); \\
&\} \\
\end{align*}

Figure 7.2: The encoding of the predicate Eq. (7.3)

The proof is found in Appendix I. By the syntactic mapping from SSL to UFRL proofs Def. 18,
the induction rule in SSL ($L_{Intro_s}$ and $R_{Intro_s}$) is encoded to the followings:

\[
\begin{align*}
(L_{Intro_s}) & \quad \frac{\text{TR}[a] \vdash^u \text{TR}[a']}{\text{TR}[p_s(c)] \vdash^u \text{TR}[a']} \\
\text{where } a &= (b \Rightarrow a)([\overline{c}/formals_s(p_s)]) \\
(R_{Intro_s}) & \quad \frac{\text{TR}[a'] \vdash^u \text{TR}[a]}{\text{TR}[a'] \vdash^u \text{TR}[p_s(\overline{c})]} \\
\text{where } a &= (b \Rightarrow a)([\overline{c}/formals_s(p_s)])
\end{align*}
\]

The encoded rules are admissible in the UFRL proof system by Theorem 7 and Lemma 23.
CHAPTER 8: REASONING ABOUT SUBTYPING

This chapter extends the programming language defined in Chapter 2 and Chapter 3 with inheritance, in a way that is similar to Java. To handle dynamic types, the FRL logic framework is extended by adding axioms and rules for dynamic and static method calls. A semantic model for behavior subtyping with explicit frame conditions is defined and proved sound.

8.1 Programming Language Extended with Inheritance

Fig. 8.1 shows the extensions of the program syntax from Fig. 2.1 and Fig. 3.1. In the syntax, dots “...” denotes the material defined previously. It contains typical object-oriented features, such as interface declarations, inheritance and method calls in statements. For simplicity, exception handling and overloading (e.g., field overloading and method overloading) are not provided. Recursive predicates presented in the previous chapter are not included for simplicity as well.

There is a designated variable super that is used to access members of the superclass of the current derived class. And there is a distinguished class named Object, which is the default superclass of a class that does not declare a superclass explicitly with the extends clause. Class
Table 8.1: Auxiliary functions used in the semantics

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>implements(T)</td>
<td>The interface names that are directly and transitively implemented by T, where T is a class name or an interface name, where T \not\in implements(T)</td>
</tr>
<tr>
<td>super(C)</td>
<td>The class name that C directly extends (if any)</td>
</tr>
<tr>
<td>supers(C)</td>
<td>The class names and interface names that are directly and transitively extended and implemented by class C, where C \not\in supers(C)</td>
</tr>
<tr>
<td>formals(T,m)</td>
<td>The method T.m’s formal parameters, where T is a class name or an interface name</td>
</tr>
<tr>
<td>body(C,m)</td>
<td>The method C.m’s body, where C is a class name</td>
</tr>
</tbody>
</table>

names and interface names are unique in each program. Interfaces may contain fields that are used in specifications.

For simplicity, all fields are protected and all methods are public.\(^{1}\) All classes (except \texttt{Object}) inherit from exactly one class and may implement multiple interfaces. Nested class declarations are not allowed for simplicity. All methods defined in a superclass are inherited by its subclasses. A method defined in a subclass with the same signature in the superclass overrides the superclass’s declaration.

Recall that each well-formed type environment, written \(\Gamma\), maps from identifiers to types:

\[
\Gamma \in \text{VarTypeEnv} = \text{Ids fin} \rightarrow T
\]

The collection of types \(T\) contains primitive types and reference types that are class names and interface names. The type of a method is of the form \(\overline{T} \rightarrow T’\). To streamline the presentation, the auxiliary functions defined in Table. 8.1 are used.

The subtype relation, \(\leq\), is a partial, reflexive and transitive relation on types. \texttt{Object} is the top of the subtype relation. The subtyping for a function type is contravariant for parameters and

\(^{1}\)This treatment simplifies the formalization.
covariant for the result. This is formalized as follows:

\[
\begin{align*}
C & \leq \textbf{Object} & C & \leq C & T \in \text{supers}(C) & I' \in \text{implements}(I) \\
& & & & \frac{C \leq T}{I \leq I'}
\end{align*}
\]

\[
\begin{align*}
T_1 & \leq T_2 & T_2 & \leq T_3 & \frac{T_2 \leq T_1}{T_1 \leq T_3} & \frac{T_2 \leq T_1'}{T_1 \leq T_1'} & \frac{T_2 \rightarrow T_2'}{T_1 \rightarrow T_1'}
\end{align*}
\]

The functions \(dfields(C)\) and \(dmeths(C)\) define the type of fields and methods that the class \(C\) defines respectively. The functions \(dfields(I)\) and \(dmeths(I)\) define the type of fields and methods that the interface \(I\) defines, respectively. The functions \(ifields(T)\) and \(imeths(T)\) define the type of fields and methods that the type \(T\) inherits, respectively. The function \(fields(T)\) is re-defined as the type of fields that the type \(T\) defines or inherits. This definition is compatible with the definition in Chapter 2, as the language defined there does not have inheritance. The function \(meths(T)\) defines the type of methods that the type \(T\) defines or inherits.

\[
dfields(T) = \{(f : T') \mid (\text{var } f : T') \text{ is declared in the reference type } T\}
\]

\[
ifields(T) = \begin{cases} 
\bigcup_{C \in \text{supers}(T)} dfields(C') \cup \bigcup_{I \in \text{implements}(T)} dfields(I) & \text{if } T \text{ is a class name} \\
\bigcup_{C \in \text{supers}(T)} dfields(\textbf{Object}) \cup \bigcup_{I \in \text{implements}(T)} dfields(I) & \text{if } T \text{ is an interface name}
\end{cases}
\]

\[
fields(T) = dfields(T) \cup ifields(T)
\]

\[
dmeths(T) = \{(T, m) \mapsto (T' \rightarrow T') \mid m(T) : T' \text{ is declared in the reference type } T\}
\]

\[
imeths(T) = \begin{cases} 
\bigcup_{C \in \text{supers}(T)} dmeths(C) \cup \bigcup_{I \in \text{implements}(T)} dmeths(I) & \text{if } T \text{ is a class name} \\
\bigcup_{C \in \text{supers}(T)} dmeths(\textbf{Object}) \cup \bigcup_{I \in \text{implements}(T)} dmeths(I) & \text{if } T \text{ is an interface name}
\end{cases}
\]

\[
meths(T) = dmeths(T) \cup imeths(T)
\]

The typing rule for the type test expression is shown as follows:

\[
\Gamma \vdash x : T' \quad \Gamma \vdash x \textbf{ is } T : \textbf{bool} \quad \text{where } \text{isRef}(T') \text{ and } T \leq : T'
\]

96
Typing rules for other expressions and region expressions are unchanged; see Appendix A. The following shows the typing rules for statements. They are adjusted from Fig. A.2. The typing rules for statements that are not shown here are unchanged.

\[
\begin{align*}
\Gamma \vdash x : T' & \quad \Gamma \vdash G : T \\
\Gamma \vdash x := G ; : ok(\Gamma) & \quad \Gamma \vdash x := \text{new} T ; : ok(\Gamma) \\
\text{where } x \neq \text{this} \text{ and } T \leq: T' & \quad \text{where } x \neq \text{this} \text{ and } T \leq: T'
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : T' & \\
\Gamma \vdash x := y.f ; : ok(\Gamma) & \quad \Gamma \vdash y : T \\
\text{where } x \neq \text{this}, (f : T) \in \text{fields}(\Gamma(y)) \text{ and } T \leq: T' & \quad \text{where } x \neq \text{this}, (f : T') \in \text{fields}(\Gamma(x)) \text{ and } T \leq: T'
\end{align*}
\]

The typing rules for the type cast and method call statements are shown as follows. Note that the form \texttt{super.m(E)} can only be called from code in a subclass, i.e., \texttt{this} \in \text{dom}(\Gamma).

\[
\begin{align*}
\Gamma \vdash x : T' & \quad \Gamma \vdash \text{super}.m(E) : T_1 \\
\Gamma \vdash x := (T)y ; : ok(\Gamma) & \quad \Gamma \vdash x := y.m(E) : ok(\Gamma) \\
\text{where } x \neq \text{this}, \text{isRef}(T') \text{ and } T \leq: T' & \quad \text{where } ((\Gamma(y), m) \mapsto (T_1 \rightarrow T')) \in \text{meths}(\Gamma(y)), T' \leq: T'' \text{ and } T_1 \leq: T
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash E : T_1 & \quad \Gamma \vdash x : T'' \\
\Gamma \vdash x := \text{super}.m(E) ; : ok(\Gamma) & \\
\text{where } \text{this} \in \text{dom}(\Gamma), C = \text{super}(\Gamma(\text{this})), (C, m) \mapsto (T \rightarrow T') \in \text{meths}(C), T' \leq: T'' \text{ and } T_1 \leq: T
\end{align*}
\]
The typing rules for assertions are adjusted as follows:

\[
\begin{align*}
\Gamma \vdash E_1 : T_1 & \quad \Gamma \vdash E_2 : T_2 \\
\Gamma \vdash E_1 = E_2 : \text{bool} \quad & \quad \Gamma \vdash E_1 \neq E_2 : \text{bool} \\
\text{where } T_1 \leq T_2 & \quad \text{or } T_2 \leq T_1 \\
\Gamma \vdash x : T' & \quad \Gamma \vdash E : T_2 \\
\Gamma \vdash x.f = E : \text{bool} \quad & \quad \text{where } \text{isRef}(T'), (f : T_1) \in \text{fields}(T') \text{ and } (T_1 \leq T_2 \text{ or } T_2 \leq T_1)
\end{align*}
\]

The typing rule for a dynamic type test assertion is shown as follows:

\[
\begin{align*}
\Gamma \vdash x : T' & \\
\Gamma \vdash x : T : \text{ bool} \quad \text{where } \text{isRef}(T') \text{ and } T \leq T'
\end{align*}
\]

### 8.2 Semantics

The function, \( \text{RefCtx} \), is used to denote sets of reference contexts that map references, \( \text{Ref} \), to class names:

\[
\rho \in \text{RefCtx} = \text{Ref} \xrightarrow{\text{fin}} \text{Name}
\]

A program \( \Gamma \)-state is extended to either be a triple of a store, a reference context, and a heap or an error:

\[
s \in \text{State} = \text{Store} \times \text{RefCtx} \times \text{Heap} \cup \{\text{err}\}
\]

A \( \Gamma \)-state \( (\rho, \sigma, H) \) is a state such that \( \text{dom}(\Gamma) = \text{dom}(\sigma) \).

There is a method environment, \( \theta \), which is a table of meanings of methods in all classes indexed by a pair of a class name and its method name, such that

\[
\theta = \text{fix}(\lambda g . \lambda (C, m) . \lambda s . \text{MS}[\Gamma \vdash \text{body}(C, m) : \text{ok}(\Gamma')](g)(s))
\]
The underlined lambda ($\lambda$) denotes a strict function that cannot recover from a nonterminating computation [81]. The semantic functions are re-defined as follows.

$E : \text{Expression Typing Judgment} \rightarrow \text{Store} \times \text{RefCtx} \rightarrow \text{Value}$

$MS : \text{Statement Typing Judgment} \rightarrow S \rightarrow \text{MethEnv} \rightarrow \text{State} \rightarrow \text{State}_\perp$

The semantics of a type test expression is defined as follows:

$E[\Gamma \vdash x \text{ is } T : \text{bool}](\sigma,\rho) = \sigma(x) \neq \text{null and } \rho(\sigma(x)) \ll T$

The semantics of other expressions is re-defined by using $\rho$ in place of the function $\text{type}$. By the definition of $\rho$ and $\text{type}$, this chapter’s $E[\Gamma \vdash G : T](\sigma,\rho)$ is equivalent to the previous $E[\Gamma \vdash G : T](\sigma)$ in Fig. 2.3. The semantics of statements is similar to Fig. 2.4, except the one for allocation:

$MS[\Gamma \vdash x := \text{new } T; : \text{ok}](\theta)(\sigma,\rho,h) = \text{let } (l,h') = \text{allocate}(T,h) \text{ in}
\text{let } (f_1 : T_1, \ldots, f_n : T_n) = \text{fields}(T) \text{ in}
\text{let } \sigma' = \sigma[x \mapsto l] \text{ in}
\text{let } \rho' = \text{Extend}(\rho,l,T) \text{ in}
(\sigma',\rho',h'[\sigma'(x),f_1] \mapsto \text{default}(T_1),\ldots,\sigma'(x),f_n) \mapsto \text{default}(T_n))$

The semantics for a type cast statement is defined below:

$MS[\Gamma \vdash x := (T)y; : \text{ok}](\theta)(\sigma,\rho,h) =
\text{if } \sigma(y) = \text{null or } \rho(\sigma(y)) \ll T \text{ then } (\sigma[x \mapsto \sigma(y)],\rho,h) \text{ else } \text{err}$
The semantics for a dynamic method call is defined below:

\[ MS[\Gamma \vdash x := y.m(\overline{G}); : ok(\Gamma)](\theta)(\sigma, \rho, h) = \]

\[
\begin{align*}
\text{if } \sigma(y) \neq \text{null} & \text{ then} \\
\phantom{=} & \text{let } \overline{z} = \text{formals}(\rho(\sigma(y)), m) \text{ in} \\
\phantom{=} & \text{let } \sigma'' = \text{Extend}(\sigma, (\text{this}, \overline{z}), (\sigma(y), \overline{E}[\Gamma \vdash G : T](\sigma, \rho))) \text{ in} \\
\phantom{=} & \text{if } (\rho(\sigma(y)), m) \in \theta \text{ then} \\
\phantom{=} & \text{let } v = (\theta(\rho(\sigma(y), m))(\sigma'', \rho, h)) \text{ in} \\
\phantom{=} & \text{if } v = (\sigma', \rho', h') \text{ then } (\sigma[x \mapsto \sigma'(\text{ret})], \rho', h') \\
\phantom{=} & \text{else if } v = \text{err} \text{ then } \text{err} \text{ else } \bot \\
\phantom{=} & \text{else err}
\end{align*}\]

else err

The semantics for a static method call does not need to pass the this parameter, as this points to the subclass that invokes super.m(\overline{G}).

\[ MS[\Gamma \vdash x := \text{super} . m(\overline{G}); : ok(\Gamma)](\theta)(\sigma, \rho, h) = \]

\[
\begin{align*}
\text{let } C = \text{super}(\Gamma(\text{this})) \text{ in} \\
\phantom{=} & \text{let } \overline{z} = \text{formals}(C, m) \text{ in} \\
\phantom{=} & \text{let } \sigma'' = \text{Extend}(\sigma, \overline{z}, \overline{E}[\Gamma \vdash G : T](\sigma, \rho)) \text{ in} \\
\phantom{=} & \text{if } (C, m) \in \theta \text{ then} \\
\phantom{=} & \text{let } v = (\theta(C, m))(\sigma'', \rho, h) \text{ in} \\
\phantom{=} & \text{if } v = (\sigma', \rho', h') \text{ then } (\sigma[x \mapsto \sigma'(\text{ret})], \rho', h') \\
\phantom{=} & \text{else if } v = \text{err} \text{ then } \text{err} \text{ else } \bot \\
\phantom{=} & \text{else err}
\end{align*}\]

Lemma 1 is adjusted with the new definition of the semantic state as follows:

**Lemma 1A.** Let \( \Gamma \) and \( \Gamma' \) be two well-formed type environments. Let \( S \) be a statement, such that \( \Gamma \vdash S : ok(\Gamma') \). Let \( \Gamma'' \) be a well-formed type environment, such that \( \text{dom}(\Gamma') \cap \text{dom}(\Gamma'') = \emptyset \) and \( \text{dom}(\Gamma') \cap \text{dom}(\Gamma'') = \emptyset \). Then
1. if $\mathcal{M}_S[\Gamma \vdash S : ok(\Gamma')]((\sigma, \rho, h)) \neq \text{err}$, then $\mathcal{M}_S[\Gamma, \Gamma'' \vdash S : ok(\Gamma', \Gamma'')]((\sigma, \rho, h)) \neq \text{err}$.

2. if $\mathcal{M}_S[\Gamma \vdash S : ok(\Gamma')]((\sigma, \rho, h)) = (\sigma', \rho', h')$, then $\mathcal{M}_S[\Gamma, \Gamma'' \vdash S : ok(\Gamma', \Gamma'')]((\sigma, \rho, h)) = (\sigma', \rho', h')$.

The semantics of assertions is re-defined by using $\rho$ in place of the function $\text{type}$. By the definition of $\rho$ and $\text{type}$, this chapter’s $\sigma, \rho, h \models^\Gamma P$ is equivalent to the previous $\sigma, h \models^\Gamma P$ in Fig. 3.2. The semantics of dynamic type test assertion is defined below:

$$\sigma, \rho, h \models^\Gamma x : T \text{ iff } \sigma(x) \neq \text{null and } \rho(\sigma(x)) = T$$

Also Lemma 2 is adjusted with the new definition of the semantic state as follows:

**Lemma 2A.** Let $\Gamma$ and $\Gamma'$ be two well-formed type environments such that $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$. Let $(\sigma, \rho, h)$ be a $\Gamma$-state. Then $\sigma, \rho, h \models^\Gamma P$ implies $(\sigma, \rho, h) \models^{\Gamma, \Gamma'} P$.

### 8.3 Effects

The definition of agreement needs to consider the reference context as well, i.e., if two states agree on a read effect, $\delta$, then the two states must agree on its dynamic types of objects as well.

**Definition 4A (Agreement on Read Effects).** Let $\Gamma$ be a well-formed type environment. Let $\delta$ be an effect that is well-typed in $\Gamma$. Let $\Gamma' \supseteq \Gamma$ and $\Gamma'' \supseteq \Gamma$. Let $(\sigma', \rho', h')$ and $(\rho'', \sigma'', h'')$ be a $\Gamma'$-state and a $\Gamma''$-state respectively. Then $(\sigma', \rho', h')$ and $(\rho'', \sigma'', h'')$ agree on $\delta$, written $(\sigma', \rho', h') \delta (\rho'', \sigma'', h'')$, if and only if:

1. for all $(\text{reads } x) \in \delta :: \sigma'(x) = \sigma''(x)$, and if $\sigma'(x) \in \text{dom}(\rho')$, then $\rho'(\sigma'(x)) = \rho''(\sigma''(x))$.

2. for all $(\text{reads region } \{x.f\}) \in \delta$ and for all $o$ such that $o = \sigma'(x)$ and $o \neq \text{null and } (f : T') \in \text{fields}(\rho'(o))$, $h'[o, f] = h''[o, f]$ and if $\rho'(h'[o, f]) \in \text{dom}(\rho')$, then $\rho''(h''[o, f]) \in \text{dom}(\rho'')$ and $\rho'(h'[o, f]) = \rho''(h''[o, f])$. 


The read effects of the type test expression is defined as \( efs(x \textbf{is } T) = \text{reads } x \). The read effects of other expressions and atomic assertions are unchanged.

### 8.3.1 The read effect of a class

In program verification, it is common practice to use an object invariant expressed by logical formulas and “abstract” data [37] to describe all possible states of the object. An invariant is either encoded through methods’ pre- and postconditions, or is explicitly specified (e.g., by the keyword \texttt{invariant}). To frame an invariant, ghost fields are commonly used in the dynamic frames based approaches, e.g., RL, FRL and Dafny. Ghost fields are specification-only fields that can be manipulated by specifications as a program runs, but cannot change a program’s (non ghost) data or execution path. Consider the example in Fig. 8.2. The predicate \texttt{valid} is an invariant of the class \texttt{Node<T>}. The field \texttt{fpt} is a ghost field that computes the regions that frame the predicate \texttt{valid}. The method \texttt{add} prepends the node \texttt{n} to the linked-list. At the last line of its implementation, the ghost field \texttt{fpt} of the object \texttt{ret} is updated to frame its predicate \texttt{valid}. It is said that \texttt{reads fpt} is the read effects of the class \texttt{Node<T>}. For ease of the discussion, it is assumed that there is a function \texttt{bnd} that maps a class name to its read effects.

### 8.4 Supertype Abstraction and Local Reasoning

\textit{Supertype abstraction} [48] allows one to use a supertype’s method specification to reason about calls to a subtype’s method. Leavens and Naumann [47] have shown that \textit{behavioral subtyping} is necessary and sufficient for the validity of supertype abstraction. They define behavioral subtyping in terms of specification refinement, and define specification refinement in terms of preconditions and postconditions, but give no explicit treatment of frame conditions. To apply their result to the framework of local reasoning, the constraints for frame conditions are needed. This section defines the problem in the framework of FRL and achieves behavioral subtyping by using the techniques
class Node<T> {
    var val : T;  var next : Node<T>;
    var fpt: region;

predicate valid()
    reads this, this.fpt;
    {
        region{this.*} <= this.fpt &&
        (this.next = null => region{this.*} = this.fpt) &&
        (this.next != null => region{this.next.*} <= this.fpt &&
        next.fpt < this.fpt && region{this.*} != next.fpt &&
        this.fpt = region{this.*} + next.fpt && next.valid())
    }

    method add(n: Node<T>) : Node<T>
        requires n != null && n.next = null;
        requires n.valid() && this.valid();
        requires n.fpt != this.fpt;
        modifies region{n.next}, region{n.fpt};
        ensures ret = n && ret.valid();
        ensures ret.fpt = region{ret.*} + this.fpt;
        {
            ret := n;
            ret.next := this;
            ret.fpt := region{ret.*} + this.fpt;
        }
    }

Figure 8.2: An example of framing invariant

of encapsulation and specification inheritance [28].

8.4.1 Problem

Suppose \( m \) is an instance of \( T \)’s method with specification \( \{P_T\} T.m(\overline{x}:T^\tau) \{Q_T\}[\varepsilon_T] \) in FRL. Let \( R \) be a predicate whose read effects are separate from the write effects of \( \varepsilon_T \). In the spirit of
supertype abstraction, it would be ideal if the following were valid:

\[
\text{for all } o : T :: \{ o \neq \text{null} \& \& P_T \& \& R \} \ o.m(\overline{G}) \ \{ Q_T \& \& R \}[\varepsilon_T].
\] (8.1)

Eq. (8.1) means that the method \( m \)'s implementations in \( T \)'s subtypes have to comply with \( T.m \)'s specification. Let \( S \) be a subtype of \( T \) and \( \{ P_S \} m(\overline{x} : T') \{ Q_S \}[\varepsilon_S] \) be \( m \)'s specification in \( S \). To make Eq. (8.1) valid, a behavioral subtyping constraint is enforced on the specification of \( S.m \) [47], i.e., \( \{ P_S \} m(\overline{x} : T') \{ Q_S \}[\varepsilon_S] \ refines \ \{ P_T \} \ T.m(\overline{x} : T') \ \{ Q_T \}[\varepsilon_T] \). According to result in the work of Leavens and Naumann [47], such constrains are \( P_T \Rightarrow P_S \) and \( \text{old}(P_T) \land Q_S \Rightarrow Q_T \).

Because their work ignores frame conditions with the assumption that they could be encoded to postconditions, the relation between \( \varepsilon_T \) and \( \varepsilon_S \) is not clear. Therefore, this dissertation focus on the effects of overridden methods and formalizes this framing problem in the FRL proof system.

Let \( \delta \) be the read effect of \( R \) in a state where \( P_T \) holds, i.e., such that \( P_T \models \Gamma \ \delta \ \text{frm} \ \ R \). As \( R \) is preserved during the execution of the method \( T.m \), then it must be true that \( P_T \ \& \& \ R \Rightarrow \ \delta / \varepsilon_T \).

As \( P_T \Rightarrow P_S \), using the rule \( \text{FrmProjCtx} \) in Fig. 3.6, it must be that \( P_S \models \Gamma \ \delta \ \text{frm} \ \ R \).

Write effects only make sense when the precondition is true. Furthermore, for supertype abstraction, one can assume that the supertype's precondition is true; so suppose that \( P_T \) is true, in which case \( P_T \ \& \& \ R \ \& \& \ \delta / \varepsilon_S \) is true. Consider different cases of the relation between \( \varepsilon_S \) and \( \varepsilon_T \).

1. Suppose \( \varepsilon_S = \varepsilon_T \). This is the case where \( P_T \ \& \& \ R \ \& \& \ \delta / \varepsilon_S \). As \( P_T \Rightarrow P_S \), for validity it must be that \( P_S \ \& \& \ R \ \& \& \ \delta / \varepsilon_S \). Thus, \( S.m \) automatically preserves \( R \).

2. Suppose \( \varepsilon_S < \varepsilon_T \). As \( P_T \ \& \& \ R \ \& \& \ \delta / \varepsilon_T \), it must be that \( P_T \ \& \& \ R \ \& \& \ \delta / \varepsilon_S \). And because \( P_T \Rightarrow P_S \), validity requires that \( P_S \ \& \& \ R \ \& \& \ \delta / \varepsilon_S \), which also preserves \( R \).

3. Suppose \( \varepsilon_S \leq \varepsilon_T \). In this case, \( \varepsilon_S \) may contain additional fields that are introduced by the type \( S \). This is the so called “the extended state problem” [50, 64]. A solution to the extended state problem is to divide the effect \( \varepsilon_S \) into two parts: \( \varepsilon_{S_1} \) and \( \varepsilon_{S_2} \), where \( \varepsilon_{S_1} \leq \varepsilon_T \) and \( \varepsilon_{S_2} \cap \varepsilon_T = \emptyset \). Following the previous two cases, validity requires that \( P_S \ \& \& \ R \ \& \& \ \delta / \varepsilon_{S_1} \).
Since reasoning at the level of the supertype knows nothing about $\varepsilon S_2$, the regions in $\varepsilon S_2$ must be such that $R$ cannot possibly depend on them. Following standard software engineering practice, it is called lack of dependency on $\varepsilon S_2$ “encapsulation”.

8.4.2 Encapsulation

In object-oriented programming, an object contains field names and methods that manipulate these fields. The values of these fields represent the state of the object. If fields can only be accessed through their class, in the sense that the only way to read or write these fields is by calling a method defined in (or inherited by) the class, then those fields are encapsulated by the class. Fig. 8.3 shows three examples of encapsulation. The examples of Cell and ReCell are adapted from the work of Parkinson and Bierman [76]. The class Cell is the base class, and the classes ReCell and FCell are its derived classes. The class ReCell declares an additional field $bak$ with type int. The class FCell declares an additional field $fcc$ with type Cell. Fig. 8.4 illustrates three objects of these three classes residing in disjoint parts of the heap; these three objects are $\sigma(c), \sigma(rc)$, and $\sigma(fc)$. In the store, $(c : \sigma(c))$ means that the value of the variable $c$ has value $\sigma(c)$. The left side of the store shows the types of variables, e.g., the variable $c$ has type Cell, the variable $rc$ has type ReCell, and the variable $fc$ has type FCell. In the heap, $((\sigma(c), val) : 0)$ means that the location $(\sigma(c), val)$ stores the value 0. The fields of the object $c$ and $rc$ are integers, thus are encapsulated as they are protected. The field $fcc$ of the object $fc$ has reference type, and the object $fc.fcc$ is created by the object $fc$. Thus, the data of $fc$ is encapsulated as well.

In the figure, dashed boxes indicate regions where an object’s data is stored. Each dashed box is said to frame the object. For example, region $R_c$ frames the object $c$, $R_{rc}$ frames the object $rc$, and $R_{fc}$ frames the object $fc$. The relation between these regions can be expressed as: $R_c !! R_{rc} !! R_{fc}$.

\footnote{The formula $R_1 !! R_2 !! R_3$ means that the three regions are pointwise disjoint.}
i.e., there is no sharing among those objects. Since these fields are all protected, this relation also implies that the objects are encapsulated.

An object is always encapsulated if for all states, either its frame is a subregion of other objects’ frames, or its frame is disjoint with them. If all the objects of type \( C \) are encapsulated, then the type \( C \) is said to be encapsulated. This idea is formalized in the following two definitions.

**Definition 21 (Class \( C \) encapsulates \( RE \)).** Let \( \Gamma \) be a well-formed type environment, and \( C \) be a class. Let \( RE \) be a subregion of \( \text{bnd}(C) \), i.e., \( RE < \text{bnd}(C) \). Then the class \( C \) encapsulates \( RE \) only if for all \( \Gamma \)-states \((\sigma, \rho, h)\) and for all \( x : C, x' : C', x \neq x' (\sigma, \rho, h) \triangleright_\Gamma RE[x/\text{this}] ! ! \text{bnd}(C')[x'/\text{this}].\)

**Definition 22 (Encapsulation).** Let \( \Gamma \) be a well-formed type environment, and \( C \) be a class. Then
the class $C$ is encapsulated only if for all $x : C, x' : C'$, $\Gamma$-states $(\sigma, \rho, h)$, such that $x \neq x'$, either $(\sigma, \rho, h) \models^\Gamma bnd(C)[x/this] < bnd(C')[x'/this]$ or for all subregions of $bnd(C)$, $RE$, $C$ encapsulates $RE$.

In Def. 22, the case where $bnd(C)[x/this] < bnd(C')[x'/this]$ is that one object is a substructure of the other, i.e., in Fig. 8.4, the class $Cell$ is a substructure of the class $FCell$.

The following lemma shows that if $RE$ is encapsulated by a class, then the regions in the read effect of $RE$ are encapsulated as well. This property is used in the definition of specification refinement.

**Lemma 24.** Let $x$ and $x'$ be two variables whose types are some class $C$ and $C'$ respectively, where $x \neq x'$. Let $RE$ be a region expression such that $C$ encapsulates $RE$. Then $RE[x/this]$ is $P/\text{modifies} bnd(C')[x'/this]$-immune.

**Proof.** Let $\text{region}\{x.f_1, \ldots, f_n\}$ in $RE$ be arbitrary. By the definition of read effect in Fig. 3.5, $\text{efs}(\text{region}\{x.f_1, \ldots, f_n\}) = \text{reads} x, \text{region}\{x.f_1\} \ldots \text{region}\{x.f_1, \ldots, f_{n-1}\}$. Let $R$ be the union of all regions in $\text{efs}(\text{region}\{x.f_1, \ldots, f_n\})$. Let $r' = \text{region}\{x'.g_1, \ldots, g_m\}$ in
Let $\text{bnd}(C')$ be arbitrary. By the definition of immune (Def. 6), the proof obligation is to show that regions in $R$ are all disjoint from $r'$. The rest of the proof proceeds by contradiction. Let $\text{region}\{x.f_1,\ldots,f_i\} \in R$, where $1 < i \leq n - 1$, such that $x.f_1,\ldots,f_i = x'.g_1,\ldots,g_m$. Then, there are two cases: (1) when $i = n - 1$, then it must be that $x'.g_1,\ldots,g_m,\ldots,f_n = x.f_1,\ldots,f_n$; (2) $i < n - 1$, then it must be that $x'.g_1,\ldots,g_m.f_{i+1},\ldots,f_n = x.f_1,\ldots,f_n$. The two cases both contradict the definition that $RE$ is encapsulated by the class $C$, i.e., $RE ! ! RE'$.

However, aliasing may break representation encapsulation by argument exposure and representation exposure [58, 65, 70]. Argument exposure happens when a type $T$’s representation is aliased by the reference to a client, through arguments of $T$’s methods. Consider the example in Fig. 8.5. The class $ECell$ inherits the class $Cell$, and declares an additional field $ecc$ with type $Cell$. Its constructor is an example of argument exposure as the field $ecc$ is aliased with the object $c$ passed to it.

Consider the following client code.

```java
var c : Cell; c := new Cell; var e : ECell; e := new ECell(c);
```

In the client code, the object $c$ is used to construct the object $e$, which leads to the aliasing between the object $e$ and the object $c$. Thus, changing the states of the object $c$ may change the states of the object $e$. At the end of the client code, the read effect of the object $c$ is $\text{bnd}(Cell)[c/this] = \text{reads region}\{c.val\}$, and the read effect of the object $e$ is $\text{bnd}(ECell)[e/this] = (\text{reads region}\{e.val\} + \text{region}\{e.ecc\} + \text{region}\{e.ecc.val\})$. Let $P$ be $c \neq \text{null} \&\& e \neq \text{null} \&\& e.ecc \neq \text{null}$. At the end of the client code, where $c = e.ecc$, shown in Fig. 8.6, the frame of the object $c$ is a subregion of the frame of the object $e$. Thus, the object $c$ is encapsulated. But the object $e$ is not encapsulated, as its frame is either not a subregion or not disjoint from the frame of $c$.

Representation exposure happens when a type $T$’s representation object is aliased by its clients through $T$’s methods’ return values. An example is the following method declaration:
```java
class ECell extends Cell {
    var ecc : Cell;

    ECell(c : Cell)
    requires c≠null;
    modifies region(this.*);
    ensures this.get() = 0 && this.ecc = c;
    { 
        this.val := 0; this.ecc := c;
    }

    function wf_set() : region {
        ret := super.wf_set() + region(ecc.val);
    }

    method set(v : int)
    requires this.ecc≠null;
    modifies wf_set();
    ensures this.get() = v && this.ecc.get() = v;
    { super.set(v); ecc.set(v); } }  
```

Figure 8.5: The specification of the class ECell

```java
class C{ var f : T; method m () : T { ret := this.f; } }
```

where f has reference type. In this case, a subregion of the the class C’s frame may be shared with other objects. Thus, the class C is not encapsulated. Section 8.6 introduces the methodology of capturing exposed regions in the case of argument exposure and representation exposure.

8.5 The Proof System

Recall that previously the effect, $\varepsilon_S$ in a subtype $S$ is divided into two parts, $\varepsilon_{S_1}$ and $\varepsilon_{S_2}$, where $\varepsilon_{S_1} \triangleq \varepsilon_{S_2}$, $\varepsilon_{S_2} \cap \varepsilon_T = \emptyset$ and writeR($\varepsilon_{S_2}$), are encapsulated by $S$. As those locations may frame parts of the extended state, thus, it must be true that for each method of $S$, writeR($\varepsilon_{S_2}$) $\triangleq$
The previous analysis leads to the following definition of specification refinement.

**Definition 23** (Specification Refinement). Let $S$ and $T$ be two types, such that $S \subseteq T$. Let $\{P_T\} T.m(x:T) \{Q_T\}[\varepsilon_T]$ and $\{P_S\} S.m(x:T') \{Q_S\}[\varepsilon_S, \text{modifies RE}]$ be specifications of a method $m$ in $T$ and in $S$ respectively, where $x:T'$ are its formal parameters. Then the specification of $S.m$ refines the specification of $T.m$ if $P_T \Rightarrow P_S, \text{old}(P_T) \land Q_S \Rightarrow Q_T, \varepsilon_S \subseteq \varepsilon_T, \text{modifies RE} \notin \varepsilon_T$ and RE is $P_T/\varepsilon_T$-immune.

To make sure $S.m$’s specification is refined by $T.m$’s specification, this dissertation adopts specification inheritance [28] as in JML [45]. The formula $\{P_T\}\{Q_T\}[\varepsilon_T]$ and $\text{also}\{P_S\}\{Q_S\}[\varepsilon_S, \text{RE}]$ is defined as follows, where $\varepsilon_S \subseteq \varepsilon_T, \text{modifies RE} \notin \varepsilon_T$ and RE is $P_T/\varepsilon_T$-immune:

$$\{P_T || P_S\}\{(\text{old}(P_T) \Rightarrow Q_T) \& \&(\text{old}(P_S) \Rightarrow Q_S)\}[\varepsilon_T, \varepsilon_S, \text{RE}], \quad (8.2)$$

The following lemma justifies the definition of the semantics of \text{also}.

**Lemma 25.** Let $T$ and $S$ be two types, such that $S \subseteq T$ and $S \neq T$. Let $T$’s method $m$ be specified as: $\{P_T\} T.m(x:T')\{Q_T\}[\varepsilon_T]$. Let $S.m$ be specified by $\{P_T\} S.m(x:T')\{Q_T\}[\varepsilon_T] \text{also}$. Then...

---

*(Figure 8.6: Argument exposure example)*
\{P\}_S.m(x:T^r)\{Q_s\}[\varepsilon_s,RE], \text{where } \varepsilon_s \leq \varepsilon_T, \text{modifies } RE \notin \varepsilon_T \text{ and } RE \text{ is } P_T/\varepsilon_T\text{-immune.}

Then the specification of \(T.m\) is refined by the specification of \(S.m\).

**Proof.** Let \(\Gamma\) be a well-formed type environment. Define \(R\) as \(\forall \text{ region}\{x_i, y_i\} \in RE :: x_i.f_i = z_i\), where \(z_i\) is fresh; this \(R\) is implicitly always true by the program semantics. The proof is shown in the following derivation.

\[
\vdash^{\Gamma}_{\tau} \{P_T \mid P_S\} \cdot (\text{old}(P_T) \Rightarrow Q_T) \&\& (\text{old}(P_S) \Rightarrow Q_S)[\varepsilon_T, \varepsilon_S, RE] \]

where \(P_T \Rightarrow P_T \mid P_S\) and \(P_T \&\& (\text{old}(P_T) \Rightarrow Q_T) \Rightarrow Q_T\)

\[
\vdash^{\Gamma}_{\tau} \{P_T\} \cdot (Q_T \&\& (\text{old}(P_S) \Rightarrow Q_S))[\varepsilon_T, \varepsilon_S, RE] \]

where \(Q_T \&\& (\text{old}(P_S) \Rightarrow Q_S) \Rightarrow Q_T\)

\[
\vdash^{\Gamma}_{\tau} \{P_T\} \cdot (Q_T)[\varepsilon_T, \varepsilon_S, RE] \]

where \(\varepsilon_s \leq \varepsilon_T\)

\[
\vdash^{\Gamma}_{\tau} \{P_T \&\& R\} \cdot (Q_T)[\varepsilon_T, \text{modifies } RE] \]

where modifies \(RE \notin \varepsilon_T\) and \(RE\) is \(P_T/\varepsilon_T\)-immune

\[
\vdash^{\Gamma}_{\tau} \{P_T \&\& R\} \cdot (Q_T)[\varepsilon_T] \]

where \(P_T \Leftrightarrow (P_T \&\& R)\)

\[
\vdash^{\Gamma}_{\tau} \{P_T\} \cdot (Q_T)[\varepsilon_T] \]

\[
\vdash^{\Gamma}_{\tau} \{P\} \cdot S\{Q\}[\varepsilon] \]

8.5.1 Correctness Judgment

A judgment with hypothesis is written as follows:

\[
\Delta \vdash^{\Gamma}_{\tau} \{P\} \cdot S\{Q\}[\varepsilon] \]

where \(\Delta\) is a specification context that maps pairs of class and method names to the corresponding method’s specification. Each method specification is written in the form \(\{P\}_T.m(x:T^r)\{Q\}[\varepsilon].\)
The specifications are obtained from declared specifications and combinations of specification from supertypes, e.g., a specification inheritance in JML [45]. A specification context \( \Delta \) is \( \Gamma \)-valid if all the specifications in \( \Delta \) are well-typed under \( \Gamma \). A \( \Delta \)-method environment \( \theta \) means \( \text{dom}(\Delta) = \text{dom}(\theta) \).

**Definition 24** (Behaviorally-Subtyped Specification Context). Let \( \Gamma \) be a well-formed type environment. A specification context \( \Delta \) is \( \Gamma \)-valid behaviorally-subtyped if for all specifications in \( \text{dom}(\Delta) \) are well-typed under \( \Gamma \), and for each \( (T, m) \in \text{dom}(\Delta) \), if \( S \preceq T \), then \( (S, m) \in \text{dom}(\Delta) \) and \( \Delta(S.m) \) refines \( \Delta(T.m) \).

Def. 25 defines the meaning of a specification context.

**Definition 25** (Specification Context Interpretation). Let \( \Gamma \) be a well-formed type environment, and \( \Delta \) be a \( \Gamma \)-valid behaviorally-subtyped specification context. Then, \( \theta \) is a \( \Delta \)-interpretation if for each \( \Delta(T, m) = \{P\} \; T.m(x:T') \{Q\}[\varepsilon] \), for all \( \Gamma \)-states, \( (\sigma, \rho, h) \):

1. \( (\theta(T, m))((\sigma, \rho, h) \neq \text{err} \iff (\sigma, \rho, h) \models \Gamma P \)

2. If \( (\sigma, \rho, h) \models \Gamma P \) and \( (\sigma', \rho', h') = (\theta(T, m))(\sigma, \rho, h) \), then \( (\sigma', \rho', h') \models \Gamma' Q \) and

   (a) for all \( x \in \text{dom}(\sigma) \), if \( \sigma(x) \neq \sigma'(x) \), then \( \text{modifies} \; x \in \varepsilon \)

   (b) for all \( (o, f) \in \text{dom}(h) \), if \( h'(o, f) \neq h(o, f) \), then \( (o, f) \in \mathcal{E}[\Gamma \vdash \text{writeR}(\varepsilon) : \text{region}](\sigma, \rho) \)

   (c) for all \( (o, f) \in \mathcal{E}[\Gamma' \vdash \text{freshR}(\varepsilon) : \text{region}](\sigma', \rho'), (o, f) \in (\text{dom}(h') - \text{dom}(h)) \)

In a verification logic, a method call is interpreted by checking its precondition, havocing its frame condition, and assuming its postcondition. If its precondition is not true, the verification fails. Following the Banerjee and Naumann’s work [2], this case is called a \( p \)-fault. To capture \( p \)-fault in the semantics, the semantic function for statements is re=defined as follows.

\( \mathcal{M}S : \text{Typing Judgment} \to S \to \text{MethEnv} \to \text{Specification Context} \to \text{State} \to \text{State}_\bot + \{p \text{-fault}\} \)
This chapter’s $\mathcal{MS}[[\Gamma \vdash S : ok(\Gamma')]](\theta)(\Delta)(\sigma, \rho, h)$ is equivalent to the previous $\mathcal{MS}[[\Gamma \vdash S : ok(\Gamma')]](\theta)(\sigma, \rho, h)$ except for the semantics of a dynamical method call:

$$\mathcal{MS}[[\Gamma \vdash x := y.m(\overline{G}); : ok(\Gamma')]](\theta)(\sigma, \rho, h) =$$

$$\text{if } \sigma(y) \neq \text{null} \text{ then}$$

$$\text{let } \overline{z} = \text{formals}(\rho(\sigma(y)), m) \text{ in}$$

$$\text{let } \overline{v} = \overline{\mathcal{E}[[\Gamma \vdash G : T]](\sigma, \rho)} \text{ in}$$

$$\text{let } \sigma'' = \text{Extend}(\sigma, (\text{this}, \overline{z}), (\sigma(y), \overline{v})) \text{ in}$$

$$\text{let } \{P\} \cdot \{Q\}[\overline{e}] = \Delta(\rho(\sigma(y)), m) \text{ in}$$

$$\text{if } \sigma, \rho, h \vdash_{\Gamma} P[\overline{v}/\overline{z}] \text{ then}$$

$$\text{if } (\rho(\sigma(y)), m) \in \theta \text{ then}$$

$$\text{let } v = (\theta(\rho(\sigma(y), m)))(\sigma'', \rho, h) \text{ in}$$

$$\text{if } v = (\sigma', \rho', h') \text{ then } (\sigma[x \mapsto \sigma'(\text{set}]), \rho', h')$$

$$\text{else if } v = \text{err then } err \text{ else } \bot$$

$$\text{else } err$$

$$\text{else } p\text{-fault}$$

$$\text{else } err$$
and a static method call:

\[ \mathcal{M}S[\Gamma \vdash x := \texttt{super.m}(G); \texttt{ok}((\theta))(\sigma, \rho, h)] = \]

let \( C = \texttt{super}(\Gamma(\texttt{this})) \) in

let \( z = \texttt{formals}(C, m) \) in

let \( v = \mathcal{E}[\Gamma \vdash G : T](\sigma, \rho) \) in

let \( \sigma'' = \text{Extend}(\sigma, z, v) \) in

let \( \{P\} S \{Q\}[\varepsilon] = \Delta(C, m) \) in

if \( (\sigma, \rho, h) \models \Gamma P[v/z] \) then

if \( (C, m) \in \theta \) then

let \( v = (\theta(C, m))(\sigma'', \rho, h) \) in

if \( v = (\sigma', \rho', h') \) then \( (\sigma[x \mapsto \texttt{ret}], \rho', h') \)

else if \( v = \texttt{err} \) then \( \texttt{err} \) else \( \bot \)

else \( \texttt{err} \)

else \( \texttt{p-fault} \)

else \( \texttt{err} \)

A valid FRL Hoare-Formula is defined as follows.

**Definition 26** (Valid FRL Hoare-Formula with Hypothesis). Let \( \Gamma \) be a well-formed type environment, and \( \Delta \) be a \( \Gamma \)-valid behaviorally-subtyped specification context, and \( \theta \) be a \( \Delta \)-interpretation. Let \( S \) be a statement. Let \( P \) and \( Q \) be assertions, \( \varepsilon \) be effects, and \( (\sigma, \rho, h) \) be a \( \Gamma \)-state. Then \( \{P\} S \{Q\}[\varepsilon] \) is valid in \( (\sigma, \rho, h) \) under \( \Gamma \) and \( \theta \) written \( (\sigma, \rho, h); \theta \models \Gamma \{P\} S \{Q\}[\varepsilon] \), if and only if whenever \( (\sigma, \rho, h) \models \Gamma P \), then

1. \( \mathcal{M}S[\Gamma \vdash S : \texttt{ok}(\Gamma')][(\theta))(\sigma, \rho, h)] \neq \texttt{err} \),

2. \( \mathcal{M}S[\Gamma \vdash S : \texttt{ok}(\Gamma')][(\theta))(\sigma, \rho, h)] \neq \texttt{p-fault} \),

3. if \( (\sigma', \rho', h') = \mathcal{M}S[\Gamma \vdash S : \texttt{ok}(\Gamma')][(\theta))(\sigma, \rho, h)] \), then \( (\sigma', \rho', h') \models \Gamma'' Q \) and
(a) for all \( x \in \text{dom}(\sigma) \): \( \sigma'(x) \neq \sigma(x) \) : modifies \( x \in \varepsilon \)

(b) for all \((o, f) \in \text{dom}(h), \) if \( h'[o, f] \neq h[o, f] \), then \((o, f) \in \mathcal{E} \triangleright \text{write}(\varepsilon) : \text{region} \triangleright (\sigma, \rho) \)

(c) for all \((o, f) \in \mathcal{E} \triangleright \text{fresh}(\varepsilon) : \text{region} \triangleright (\sigma', \rho') :: (o, f) \in (\text{dom}(h') - \text{dom}(h)) \).

A FRL judgment \( \Delta \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \) is valid if and only if for all \( \Delta \)-interpretation \( \theta \), and all \( \Gamma \)-states \( s :: s; \theta \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \).

The axioms and inference rules defined in Fig. 4.1 are unchanged. Structural rules defined in Fig. 4.2 and Fig. 4.3 are adjusted by adding hypothesis. The axiom for the type cast statement is shown below:

\[
\text{(TypeCast} \triangleright \text{r)} \quad \Gamma \vdash \{ y = \text{null} \mid y : T' \} \quad x := (T)y; \quad \{ x = y \} [\text{modifies} \ x] \quad \text{where} \quad T' \leq : T
\]

The axiom for dynamic method calls that is adapted from Banerjee and Naumann’s work [2] as follows:

\[
\text{(Dcall} \triangleright \text{r)} \quad \Delta, \{ P \} T.m(z : T \mid Q)[\varepsilon] \vdash \Gamma \{ x \neq \text{null} \& \& P[\overline{G/\varepsilon}] \} x.m(\overline{G}) \{ Q[\overline{G/\varepsilon}] \}[\varepsilon[\overline{G/\varepsilon}]]
\]

where \( \Gamma(x) = T \)

where \( P[\overline{G/\varepsilon}] \) simultaneously substitutes \( \overline{G} \) for \( \varepsilon \) in \( P \). The axiom for static method calls is:

\[
\text{(SCall} \triangleright \text{r)} \quad \Delta, \{ P \} C.m(z : T \mid Q)[\varepsilon] \vdash \Gamma \{ P[\overline{G/\varepsilon}] \} C.m(\overline{G}) \{ Q[\overline{G/\varepsilon}] \}[\varepsilon[\overline{G/\varepsilon}]]
\]

The following two structural rules show that additional type declarations and method declarations do not invalidate proved statements.

\[
\begin{align*}
\text{(TypeExt} \triangleright \text{r)} & \quad \Delta \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \quad \Delta \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \\
\text{(MethExt} \triangleright \text{r)} & \quad \Delta \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon] \quad \Delta, \Delta' \vdash \Gamma \{ P \} S \{ Q \}[\varepsilon]
\end{align*}
\]

**Lemma 26.** The rules TypeCast\(_{r}\), DCall\(_{r}\), SCall\(_{r}\), TypeExt\(_{r}\), and MethExt\(_{r}\), are sound.

**Proof.** Each rule is proved in turn. Let \((\sigma, \rho, h) \) and \((\sigma', \rho', h') \) be pre- and post-state respectively.

**TypeCast\(_{r}\):** In this case, the precondition, \( y = \text{null} \mid y : T' \), is assumed. There are two cases:
• $y = \text{null}$: By its semantics, $\sigma' = \sigma[x \mapsto \sigma(y)], \rho' = \rho$ and $h' = h$. It must be true that $(\sigma', \rho', h') \models^\Gamma x = y$. Moreover, the write effect modifies $x$ is correct as $x$ is the only variable that is modified.

• $y : T'$: In this case, $\rho(\sigma(y)) = T'$. By the side condition, $T' \leq T$, it is true that $\rho(\sigma(y)) \leq T$. By its semantics, $\sigma' = \sigma[x \mapsto \sigma(y)], \rho' = \rho$ and $h' = h$. It must be true that $(\sigma', \rho', h') \models^\Gamma x = y$. Moreover, the write effect modifies $x$ is correct as $x$ is the only variable that is modified.

$DCall_r$: In this case, the precondition, $x \neq \text{null} \& \& P[\theta / z]$, is assumed. By the side condition, the static type of $x$ is $T$. Because $\Delta$ is behaviorally-subtyped, it is sound to use the specification of $T.m$ as hypothesis. Let $\theta$ be its interpretation, such that $(T, m) \in \text{dom}(\theta)$. By the definition of specification context interpretation (Def. 25) and the assumption that the precondition is true, it is true $\theta(T, m)(\sigma, \rho, h) \neq \text{err}$, $(\sigma', \rho', h') = (\theta(T, m))(\sigma, \rho, h)$ and $(\sigma', \rho', h') \models^\Gamma Q[\theta / z]$. And the effect, $\varepsilon[\theta / z]$, is correct.

$SCall_r$: As the statement explicitly indicate the method $C.m$ is called, it is sound to use the specification of $C.m$ as hypothesis. The proof is similar to the proof of the rule $DCall_r$, thus, is omitted.

$TypeExt_r$: By definition of a well-formedness type environment, $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$. By Lemma 2, it must be true that $\sigma, \rho, h \models^\Gamma P$ implies $\sigma, \rho, h \models^\Gamma, \Gamma' P$. By Lemma 1, $\mathcal{MS}[\Gamma, \Gamma' \vdash S : \text{ok}(\Gamma'')] (\sigma, \rho, h) = (\sigma', \rho', h')$. By Lemma 2, the postcondition follows as well. As the original specification is well-typed, $\text{dom}(\Gamma') \cap \mathcal{FV}(S) = \emptyset$. Thus, the effect is just $\varepsilon$ as well.

$MethExt_r$ Let $\Gamma' = \{ P_{T'} \} T'.m(x : \overline{T})\{Q_{T'}\}[\varepsilon_{T'}]$. There are two cases.

1. the method $T'.m$ inherits from $T.m$, where $S \leq T$. By the definition of behaviorally-subtyped specification context, the specification of $T'.m$ refines the specification of $T.m$. The result follows supertype abstraction.
2. the method $T'.m$ is new and does not inherit from any supertype’s methods. Thus, $T'.m$ cannot be invoked by $S$.

Definition 27 (Modular Soundness). Let $\Gamma$ be a type environment. Let $\Delta$ be a $\Gamma$-valid behaviorally-subtyped specification context. Let $P$ and $Q$ be assertions, $S$ be a statement, and $\varepsilon$ be effects. The judgment, $\Delta \vdash$$\Gamma\ {\{P\} \ S \ {Q}\}[\varepsilon]$, is modularly sound if and only if for all $\Gamma' \supseteq \Gamma$, for all $\Gamma'$-valid behaviorally-subtyped specification contexts, $\Delta'$, such that $\Delta' \supseteq \Delta$, and for all $\Delta'$-interpretations, $\theta'$,

$$\Delta \vdash$$\Gamma\ {\{P\} \ S \ {Q}\}[\varepsilon] \Rightarrow \theta' \vdash$$\Gamma'\ {\{P\} \ S \ {Q}\}[\varepsilon].$$

Theorem 10. Let $\Gamma$ be a well-formed type environment. Let $\Delta$ be a $\Gamma$-valid behaviorally-subtyped specification context. Let $P$ and $Q$ be assertions, $S$ be a statement, and $\varepsilon$ be effects. The judgment, $\Delta \vdash$$\Gamma\ {\{P\} \ S \ {Q}\}[\varepsilon]$, is modularly sound.

Proof. Assume $\Delta \vdash$$\Gamma\ {\{P\} \ S \ {Q}\}[\varepsilon]$. By Lemma 26, it must be true that $\Delta, \Delta' \vdash$$\Gamma'\ {\{P\} \ S \ {Q}\}[\varepsilon]$.

8.6 Examples

This section explains how encapsulation is specified and proved by examples. To capture exposed regions, each reference type is equipped with a field, \texttt{df}, which stores an object’s frame, e.g., \texttt{df} stores the dynamic frames of a linked-list. Another field \texttt{exposed} is used to store the locations that may be shared with other objects. It has default value \texttt{region[]} as fields are protected in this dissertation. It is updated in the methods where arguments exposures and representation exposure may happen. Since all the objects have these two fields, this dissertation assumes that they are declared in the class \texttt{Object} such that:
As all the classes inherit from the class `Object`, the two fields are inherited in all types. A type’s invariant, which is the condition that an object has to hold for all states, is specified by explicit pre- and postconditions of all the methods of the type.

In addition, to make sure $\varepsilon_S \leq \varepsilon_T$, incrementally defined region functions are used to simulate data groups [50]. Incremental definition can be enforced by syntactically checking, i.e., the function declared in the superclass has to be invoked in its subclasses.

Protected field names cannot be used in the specification of public methods, as they are invisible and meaningless to non-privileged clients. Therefore, the keyword `spec.public` is used in the field declaration. The declaration

```plaintext
var spec_public val : int;
```

is a shorthand for the declaration

```plaintext
var val : int;
public var abstract _val : int;
rep _val <-> val;
```

The `spec.public` modifier is adopted from JML [22]. The `abstract` modifier defines fields that are only used for specifications. The meaning of an abstract field is defined by an abstraction function whose body is declared by the `rep` clause. As abstraction is not what this dissertation focuses on, it is not formalized. Functions are just methods without side effects. For simplicity, formalizing functions and pure methods are not provided. The theoretical foundation for these can be found in the work of Banerjee et al. [3].

Fig. 8.7 shows the specifications of the classes `Cell`. The field `df` in the class `Cell` stores the regions that frame its data representation, which is the region `region{this.val}`. As its type is not a reference type, the field `exposed` is always `region{}`. This condition is defined in the body of the predicate `inv`. 

```plaintext
class Object { var df : region; var exposed : region; }  
```
class Cell{
  var spec_public val : int;

  function wf_set() : region
    reads wf_set();
  { ret := region{this.val}; }

  predicate inv()
    reads this.df;
  { df = region{this.val} && this.exposed = region{} } }

method Cell()
  modifies this.df;
  ensures this.val = 0 && inv();
  { this.val := 0;
    this.df := region{this.val};
    this.exposed := region{}; }

function get() : int
  requires inv();
  reads wf_set();
  ensures ret = this.val;
  { ret := val; }

method set(v : int)
  requires inv();
  modifies wf_set();
  ensures this.val = v && inv();
  { val := v; }
}

Figure 8.7: The specification of class Cell

Fig. 8.8 shows the revised specification of the class ECell. The field df in the class ECell stores the regions that frame its data representation, which are the regions region{this.val} that inherits from its supertype, region{this.ecc} that is introduced by the class ECell, and
also the frame of the object \texttt{this.ecc}, when it is not null. This is specified by the first assertion in the body of the predicate \texttt{inv}. The assertion \texttt{this.ecc \neq null ? ecc.df : region{}},
is a conditional region expression, which abbreviates the assertion \texttt{this.ecc = t \&\& t \neq null ? ecc.df : region{}}, where \texttt{t} is fresh.

The method \texttt{ECell.set} updates both \texttt{val} and \texttt{ecc.val} with the new value. Its write effects is specified by the function \texttt{wf.set}. To make sure an incremental definition of the function, \texttt{super.wf.set} is forced to be called in its body. This is enforced by a syntactic check. Its body adds the region \texttt{region{ecc.val}} as well, which belongs to the extended state of the class \texttt{ECell}. By the definition of specification refinement (Def. 23), the proof obligation is to show that the region \texttt{region{ecc.val}} is encapsulated, i.e., it is disjoint from what may be exposed, i.e., \texttt{this.exposed}. This disjointness condition has to be true for all the states of the class \texttt{ECell}’s objects. Therefore, this condition is defined in the predicate \texttt{inv}, which is enforced to be true before and after each method after the object is constructed. However, the postcondition of the constructor implies that \texttt{this.exposed = ecc.df}, and by the specification of the class \texttt{Cell}, \texttt{ecc.val = ecc.df}. Therefore, the disjointness condition is violated and the error is captured.
class ECell extends Cell {
    var spec_public ecc : Cell;

    predicate inv() {
        this.df = region{this.val} + region{this.ecc} +
        (this.ecc ≠ null ? ecc.df : region{}} &&
        this.exposed != region{ecc.val}
    }

    function wf_set() : region
    reads wf_set();
    { ret := super.wf_set() + region{ecc.val}; }

    method ECell(c : Cell)
    requires c ≠ null;
    modifies region{this.*};
    ensures this.val = 0 && this.ecc = c;
    ensures this.exposed = ecc.df;
    ensures inv();
    {
        this.val := 0;
        this.ecc := c;
        this.df := region{this.val} + region{this.ecc} + ecc.df;
        this.exposed = ecc.df;
    }

    method set(v : int)
    also
    requires inv() \DRAND this.ecc ≠ null;
    modifies wf_set();
    ensures this.get() = v && ecc.get() = v;
    ensures inv();
    {
        super.set(v);
        ecc.set(v);
    }
}

Figure 8.8: The revised specification of class ECell
CHAPTER 9: APPLICATIONS

This chapter shows several potential applications of the results in this dissertation.

9.1 A Footprint Function

The specification language can be further extended with a footprint function, say \( fpt \), for supported assertions. However, such a footprint function would not be well-defined for arbitrary assertions, since not all are supported, and thus not all footprints would be semantic footprints. Note that, by construction, an SSL assertion \( a \) and its translation \( TR[\llbracket a \rrbracket] \) have the same semantic footprints, i.e., \( fpt_s(a) = fpt(TR[\llbracket a \rrbracket]) \), where \( fpt \) is the semantic footprint function for the UFRL (or FRL) assertions. The specification of the method \texttt{mark} (Fig. 9.4) is one example of using the \( fpt \) function. In this case, \( fpt(dag(d)) \) returns the set of locations of the DAG \( d \) that satisfy the predicate \( dag \).

9.2 Intraoperation of FRL and SSL

The results in this dissertation allow specifications written in the style of either UFRL, FRL or SSL to be understood in one UFRL proof system. A program verifier built in UFRL should identify specifications written in FRL and SSL, and encode them to UFRL by using the corresponding translating rules automatically. This section introduces a scheme that interprets these styles of specifications. Given a method specification, there are two cases.

1. If both the read effect the and write effect (including the fresh effect) of the method are specified, then verifies check the implementation of the method by using the axioms and proof rules in UFRL. There is no translation involved.

2. If the read effect of the method is not specified, then there are two cases.

\[\text{\footnotesize Part of the content in this chapter is submitted to Formal Aspect of Computing.}\]

122
(a) If the write effect of the method is specified, or if the method is decorated by the keyword \textit{pure} (that is a shorthand for a frame of \texttt{modifies }\varnothing\texttt{)\texttt{, then verifiers consider that the specification is written in the style of FRL, set the read effects to the default value, reads }\texttt{alloc},\texttt{ and verify the implementation of the method by using the axioms and proof rules in UFRL.}

(b) Otherwise, the specifications are considered as written in the style of SSL. After a syntactical check on the assertions appear in the specification, i.e., they should be all supported, verifiers translate the specification to UFRL and verify the implementation by using the axioms and proof rules in UFRL.

9.3 Hypothetical Reasoning and Interoperation between Modules

A program may conceptually consist of distinct modules or components, each of which manipulates a separate internal resources, e.g., part of the heap. Different modules’ specifications may be specified in the style of either SSL or FRL. This section shows how these different styles of specifications interoperate with each other.

Assume the form of program correctness judgment with hypothesis in UFRL is $\Delta_u \vdash^\Gamma_u [\delta] \{P_1\} S \{P_2\} [\varepsilon]$, which states that $S$ satisfies its Hoare-formula under certain hypotheses, $\Delta_u$, which map pairs of a class and a method name to the corresponding method’s specification. Hypotheses are given by the grammar:

\[
\Delta_u ::= \emptyset \mid \Delta_{u_1}, \Delta_{u_2} \mid [\delta] \{P\} T.m(x:T) \{Q\} [\varepsilon]
\]

Recall the hypotheses defined in Section 8.5 as follows:

\[
\Delta_r ::= \emptyset \mid \Delta_{r_1}, \Delta_{r_2} \mid \{P\} T.m(x:T) \{Q\} [\varepsilon]
\]

Hypotheses specified by SSL are given by the following grammar [73]:

\[
\Delta_s ::= \emptyset \mid \Delta_{s_1}, \Delta_{s_2} \mid ssl\{a\} T.m(x:T) \{a'\} [X]
\]

123
where \( X = \text{MV}(S) \) and \( S = \text{body}(C,m) \).

There is a syntactic translation function, \( \text{TR}_\Delta \), that maps the hypotheses in FRL and SSL to those in UFRL as follows:

\[
\begin{align*}
\text{TR}_\Delta[\emptyset] &= \emptyset \\
\text{TR}_\Delta[\Delta_{r_1}, \Delta_{r_2}] &= \text{TR}_\Delta[\Delta_{r_1}], \text{TR}_\Delta[\Delta_{r_2}] \\
\text{TR}_\Delta[\{P\}T.m(\overline{x:T})\{Q\}[\varepsilon]] &= [\text{reads alloc}]\{P\}T.m(\overline{x:T})\{Q\}[\varepsilon] \\
\text{TR}_\Delta[\Delta_{s_1}, \Delta_{s_2}] &= \text{TR}_\Delta[\Delta_{s_1}], \text{TR}_\Delta[\Delta_{s_2}] \\
\text{TR}_\Delta[\text{ssl}\{a\}T.m(\overline{x:T})\{a'\}[X]] &= [\text{reads } r\downarrow]T.\text{TR}[\{a\}]T.m(\overline{x:T})\{\text{TR}[a']\} \\
&\quad [\text{modifies } (r\downarrow, X), \text{fresh}(\text{fpt}_s(a') - r)]
\end{align*}
\]

where \( \text{TR}[a] \Rightarrow r = \text{fpt}_s(a) \) and \( r \notin X \)

Another way to encode SSL hypotheses would be

\[
\begin{align*}
\text{TR}_\Delta[\text{ssl}\{a\}T.m(\overline{x:T})\{a'\}[X]] &= \\
&[\text{reads } r\downarrow]T.\text{TR}[\{a\}] \&\& r = \text{fpt}_s(a)T.m(\overline{x:T})\{\text{TR}[a']\} \\
&[\text{modifies } (r\downarrow, X), \text{fresh}(\text{fpt}_s(a') - r)]
\end{align*}
\]

where \( r \notin X \)

Consider the example shown in Fig. 9.1 and Fig. 9.2 that define two classes, NumberS and NumberR. They are specified in the style of SSL and FRL respectively. The method setX declared in the class NumberR does not need precondition, as \( \text{this} \neq \text{null} \) is implicit by the program’s semantics. However, the precondition of the method setX declared in the class NumberS is needed, because the location region{this.x} has to be requested there, otherwise, it is not a valid SSL Hoare-formula. The method addOne declared in the class NumberR adds one to the value stored in the region region{n.x} and assigns to \( \text{this}.x \). Thus, its write effect is just region{this.x}. Consider the following client code.
class NumberS{
    var x : int;

    method NumberS()
        ensures this.x \geq 0;
        { this.x := 0; }

    method setX(v: int)
        requires \exists y. this.x \mapsto y;
        ensures this.x \mapsto v;
        { this.x := v; }

    method getX() : int
        requires \exists v. this.x \mapsto v;
        ensures this.x \mapsto v * ret = v;
        { ret := this.x; }
}

Figure 9.1: The class NumberS specified in the style of SSL

class NumberR{
    var x : int;

    method NumberR()
        ensures this.x = 0;
        { this.x := 0; }

    method setX(v: int)
        ensures this.x = v;
        { this.x := v; }

    method addOne(n : NumberS)
        requires n \neq null;
        modifies region(this.x);
        ensures this.x = n.x + 1;
        { this.x := n.getX() + 1; }

    method getX() : int
        modifies \emptyset;
        ensures ret = this.x;
        { ret := this.x; }
}

Figure 9.2: The class NumberR specified in the style of FRL

var sNumber; sNumber := new NumberS();

var rNumber; rNumber := new NumberR();

sNumber.setX(5); rNumber.addOne(sNumber);

assert sNumber.getX() = 5; assert rNumber.getX() = 6;

To prove the assertion is true, the two styles of specifications are translated to the specifications in UFRL shown in Fig. 9.3.

The read effects of $\Delta u_1$ and $\Delta u_2$ can be extended to alloc $\downarrow$ by using the rule SubEff$_u$.  

125
After the declaration and initialization (var sNumber; sNumber := new NumberS; var rNumber; rNumber := new NumberR;), it must be true that sNumber.x = 0 & rNumber.x = 0, which implies the precondition of sNumber.setSX(5). Thus, its postcondition is assumed right after it. As the read effects of rNumber.x = 0 is separate from the method’s write effects, using the rule FRM_u, it must be true that sNumber.x = 5 & rNumber.x = 0, which implies the precondition of rNumber.addOne(sNumber). Thus its postcondition is assumed right after it. As the read effects of sNumber.x = 5 is separate from the method’s write effects, using the rule FRM_u, it must be true that sNumber.x = 5 & rNumber.x = 6. In order to use the rule SEQ1_u, the following side condition has to be true:

region{rNumber.x} is sNumber.x ⊠0/modifies region{sNumber.x}-immune.  

(9.1)
By the definition of immune (Def. 6), the proof obligation is to show:

\[ \text{efs}(\text{region}\{rNumber.x\}) / \text{modifies region}\{sNumber.x\}, \quad (9.2) \]

which is true. Then the two statements’ write effects are accumulated by using \( CONSEQ_u \) and \( SEQ1_u \):

\[
[\text{alloc}] \\
\{sNumber.x = 0\} \\
\Delta \vdash^\Gamma_u sNumber.setX(5); \ rNumber.addOne(sNumber); \\
\{sNumber.x = 5 \&\& \ rNumber.x = sNumber.x + 1\} \\
[\text{modifies region}\{sNumber.x\}, \text{region}\{rNumber.x\}] \\
\]

where \( \Delta = \Delta_{u1}, \Delta_{u2}, \Delta_{u3}, \Delta_{u4}, \Delta_{u5} \). Thus, it can be proved that \( sNumber.getX() = 5 \) and \( rNumber.getX() = 6 \).

### 9.4 The DAG Example

This section specifies and verifies marking a directed acyclic graph. Fig. 9.4 on the following page specifies directed acyclic graphs (DAGs), where sharing is permitted between sub-DAGs, but cycles are not permitted. A predicate \( \text{dag} \) describes its structure written in the style of SL. The use of the conjunction (instead of separating conjunction) indicates that sub-DAGs may share some locations.

It is proved that the body of the method \( \text{mark} \) satisfies its specification under the hypothesis that recursive calls satisfy the specification being proved. Another method hypothesis is the specification of \( \text{unmarked} \). When reasoning about \( \text{mark} \), the specification of the method \( \text{unmarked} \) is used, instead of its body, i.e., if \( \text{unmarked} \)'s precondition is satisfied, its postcondition is assumed after calling it. Because \( \text{mark} \)'s precondition implies the one of \( \text{unmarked} \), the function \( \text{unmarked} \) can be used to specify the write effect of the method \( \text{mark} \). Its read effect is not specified, thus
class Dag { var mark : int; var l : Dag; var r : Dag; }

predicate dag(d:Dag)
  reads fpt(dag(d));
  decreases fpt(dag(d));
  { d ≠ null ⇒ ∃ i, j, k.(d.mark → i * d.l→j * d.r→k *
    (dag(j) ∧ dag(k)))}

function unmarked(d: Dag) : region
  requires dag(d);
  reads fpt(dag(d));
  ensures ∀ n : Dag.(region{n.mark} ⊆ fpt(dag(d)) & n.mark → 0 ⇐⇒
    region{n.mark} ⊆ ret)
  decreases fpt(dag(d));
  {
    if (d == null) ret := region();
    else{
      ret := region();
      if (d.mark = 0) {
        ret := ret + region{d.mark};
      }
      ret := ret + unmarked(d.l);
      ret := ret + unmarked(d.r);
    }
  }

method mark (d: Dag)
  requires dag(d);
  requires d ≠ null ∧ d.mark → 1 ⇒
    ∀ n:Dag.(region{n.mark} ⊆ fpt(dag(d)) ⇒ n.mark → 1);
  modifies unmarked(d);
  ensures d ≠ null ⇒
    ∀ n:Dag.(region{n.mark} ⊆ fpt(dag(d)) ⇒ n.mark → 1);
  decreases fpt(dag(d));
  {
    if (d ≠ null & d.mark = 0) {
      d.mark := 1; mark(d.l); mark(d.r);
    }
  }

Figure 9.4: The specification of marking a DAG
is alloc↓ by default. Assume all the nodes in a DAG are not marked before mark is invoked. The algorithm marks its left sub-DAGs first. Suppose the node \( n \) is shared by the left and right sub-DAGs. \( n \) and \( n \)'s sub-DAGS are marked when marking the left sub-DAGs. Therefore, the second precondition is true. Proving the body of the method mark considers the following three cases.

1. \( d = \text{null} \): The second precondition is vacuously true; the write effect is an empty set. The call does not do anything, which is consistent with its write effects. The postcondition is vacuously true.

2. \( \text{dag}(d) \land d \neq \text{null} \land d.\text{mark} \leftarrow 1 \): According to the precondition, the DAG \( d \) is all marked, which is also what the postcondition describes. For the write effects, also under this assumption that the DAG is marked, the set of locations that satisfies the postcondition of unmarked is an empty set. The call does not do anything, which is consistent with its write effects. Similar to the previous case, the precondition implies the postcondition.

3. \( \text{dag}(d) \land d \neq \text{null} \land d.\text{mark} \leftarrow 0 \): This case means that the current node is not marked and its sub-DAGs may not be marked. Assume \( i \) and \( j \) are the witnesses of the existential variables in the predicate \( \text{dag} \). The following must be true:

\[
d.\text{mark} \leftarrow 0 \land d.l \leftarrow i \land d.r \leftarrow j \land (\text{dag}(d.l) \land \text{dag}(d.r)),
\]

which implies the precondition of the rule \( \text{UPD}_u \), the following is derived:

\[
\begin{align*}
\text{reads } d \\
\Gamma \vdash u \{d \neq \text{null}\} d.\text{mark} := 1; \{d.\text{mark} \leftarrow 1\}
\end{align*}
\]

One can translate Eq. (9.5) into a formula in UFRL by Def. 16, or use the result in Section 6.3.
without translation. To avoid big formulas, the second approach is explored:

\[ d \neq \text{null} \vdash \Gamma_u \begin{cases} \text{reads } d, \text{region}\{d.l\}, \text{region}\{d.r\}, \text{fpt}(\text{dag}(d.l) \land \text{dag}(d.r)) \\ \text{frm}(d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \end{cases} \]  

(9.6)

Thus, the read effects are separate from the write effects, \text{region}\{d.mark\}. Using the rule \( I_{sc} \), the following is derived:

\[ \begin{align*} &\text{reads } d \\ &\{d \neq \text{null} \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))\} \\ &\vdash \Gamma_u \text{d.mark} := 1; \\ &\{\text{d.mark} \mapsto 1 \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))\} \\ &\text{modifies region}\{\text{d.mark}\} \end{align*} \]  

(9.7)

which implies the precondition of \text{mark}(d.l). Thus the following must be true (noting that preconditions, or postconditions, written on different lines of a method specification are conjoined):

\[ \begin{align*} &\text{reads alloc} \\ &\begin{cases} \text{dag}(d.l) \land (d \neq \text{null} \land \text{d.mark} \mapsto 1 \Rightarrow \\ \forall n : \text{Node. (region}\{n.mark\} \leq \text{fpt}(\text{dag}(d)) \Rightarrow n.mark \mapsto 1)) \end{cases} \\ &\vdash \Gamma_u \text{mark}(d.l); \\ &\{d.l \neq \text{null} \Rightarrow \forall n : \text{Dag. (region}\{n.mark\} \leq \text{fpt}(\text{dag}(d.l)) \Rightarrow n.mark \mapsto 1\} \\ &\text{modifies unmarked}(d.l) \end{align*} \]  

(9.8)

The rule \text{SubEff}_u is used on Eq. (9.7) and Eq. (9.8) to match up the effects for the rule \text{SEQ1}_u. Then the rule \text{CONSEQ}_u is used on Eq. (9.8) to match up the postcondition of \text{d.mark} := 1 and the precondition of \text{mark}(d.l) and to get rid of the implication in the precondition.
Thus, the following is derived:

\[
\text{reads alloc}, d
\]

\[
\{d \neq \text{null} \land d.l \rightarrow i \land d.r \rightarrow j \land (\text{dag}(d.l) \land \text{dag}(d.r))\}
\]

\[\Gamma \vdash_u \text{d.mark} := 1;\]  \hspace{1cm} (9.9)

\[
\{d.mark \rightarrow 1 \land d.l \rightarrow i \land d.r \rightarrow j \land (\text{dag}(d.l) \land \text{dag}(d.r))\}
\]

[modifies region\{d.mark\}]

and

\[
\text{reads alloc}, d
\]

\[
\{\text{dag}(d.l)\}
\]

\[\Gamma \vdash_u \text{mark}(d.l);\]

\[
\{d.l \neq \text{null} \Rightarrow \forall n : \text{Dag. region\{n.mark\} \leq fpt(\text{dag}(d.l)) \Rightarrow n.mark \rightarrow 1}\}
\]

[modifies unmarked(d.l)]  \hspace{1cm} (9.10)

By using the rule \(I_{sc}, FRM_u\) and \(CONSEQ_u\) for Eq. (9.10), the following is derived:

\[
\text{reads alloc}, d
\]

\[
\{\text{dag}(d.l)\}
\]

\[\Gamma \vdash_u \text{mark}(d.l);\]

\[
\begin{align*}
(d.l \neq \text{null} \Rightarrow \forall n : & \text{Dag. region\{n.mark\} \leq fpt(\text{dag}(d.l)) \Rightarrow} \\
& (n.mark \rightarrow 1) \land (d.mark \rightarrow 1 \land d.l \rightarrow i \land d.r \rightarrow j \land (\text{dag}(d.l) \land \text{dag}(d.r)))
\end{align*}
\]

[modifies unmarked(d.l)]  \hspace{1cm} (9.11)

In order to use the rule \(SEQ1_u\), the following side condition has to be true:

unmarked(d.l) is \((d \neq \text{null} \land d.l \rightarrow i \land d.r \rightarrow j \land (\text{dag}(d.l) \land \text{dag}(d.r))/ \text{region\{d.mark\}}-\text{immune} \hspace{1cm} (9.12)\)
By the definition of immune (Def. 6), the proof obligation is to show that for all modifies \( RE \) in \( \text{unmarked}(d.l) \):

\[
(d \neq \text{null} \land d.l \rightarrow i \ast d.r \rightarrow j \ast (\text{dag}(d.l) \land \text{dag}(d.r)) \text{ implies } \text{efs}(RE)/region\{d.mark\}.
\]

Here shows the above is true by contradiction. Suppose that there is some \( RE \), such that \( \text{efs}(RE) \) contains the location \( \text{region}\{d.mark\} \). Then \( RE \) must have the form \( d.mark.f \), for some field name \( f \), by definition of effects (Fig. 3.5). Because the type of \( mark \) is \text{int}, not a reference, this is impossible.

Now the two statements’ write effects can be accumulated. And the following is derived:

\[
\begin{align*}
&\left[ \text{reads alloc}\downarrow, d \right] \\
&\{d \neq \text{null} \land d.l \rightarrow i \ast d.r \rightarrow j \ast (\text{dag}(d.l) \land \text{dag}(d.r))\} \\
&\Gamma_u \vdash \text{d.mark} := 1; \text{mark}(d.l); \\
&\left\{ (d.l \neq \text{null} \Rightarrow \forall n: \text{Dag.(region}\{n.mark\} \leq \text{fpt}(\text{dag}(d.l)) \Rightarrow \\
&\quad (n.mark \rightarrow 1)) \land (d.mark \rightarrow 1 \land d.l \rightarrow i \ast d.r \rightarrow j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \right\} \\
&\left[ \text{modifies region}\{d.mark\}, \text{unmarked}(d.l) \right]
\end{align*}
\]

The postcondition of the above implies the precondition of the method \( \text{mark}(d.r) \). Using the rule \( CONSEQ_u \) (getting rid of the implication in the precondition), the following must be true:

\[
\begin{align*}
&\left[ \text{reads alloc}\downarrow \right] \\
&\{\text{dag}(d.r)\} \\
&\Gamma_u \vdash \text{mark}(d.r); \\
&\{d.r \neq \text{null} \Rightarrow \forall n: \text{Dag.(region}\{n.mark\} \leq \text{fpt}(\text{dag}(d.r)) \Rightarrow n.mark \rightarrow 1\} \\
&\left[ \text{modifies unmarked}(d.r) \right]
\end{align*}
\]

(9.14)
As the function `unmarked` only collects unmarked locations, the following must be true:

\[(d.l \not= \text{null} \Rightarrow (\forall n : \text{Dag}(.\text{region}\{n.\text{mark}\} \leq fpt(\text{dag}(d.l)) \Rightarrow n.\text{mark} \mapsto 1)))\]

\[\Rightarrow \text{reads} fpt(\text{dag}(d.l)) :/ \text{modifies} \text{unmarked}(d.r)\]  
(9.15)

By using the rules \(I_{sc}, FRM_u, CONSEQ_u\) and \(SubEff_u\), the following is derived.

\[
\begin{align*}
\langle \text{reads alloc}\rangle, d \\
\{ (d.l \not= \text{null} \Rightarrow \forall n : \text{Dag}(.\text{region}\{n.\text{mark}\} \leq fpt(\text{dag}(d.l)) \Rightarrow \\
\text{n.mark} \mapsto 1)) \land (d.\text{mark} \mapsto 1 \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \}
\end{align*}
\]

\[\text{mark}(d.r);\]

\[\Gamma_u \vdash (d.r \not= \text{null} \Rightarrow \forall n : \text{Dag}(.\text{region}\{n.\text{mark}\} \leq fpt(\text{dag}(d.r)) \Rightarrow \\
\text{n.mark} \mapsto 1)) \land \\
(d.l \not= \text{null} \Rightarrow \forall n : \text{Dag}(.\text{region}\{n.\text{mark}\} \leq fpt(\text{dag}(d.l)) \Rightarrow \\
\text{n.mark} \mapsto 1)) \land (d.\text{mark} \mapsto 1 \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \}
\]

\[\{ \text{modifies unmarked}(d.r) \}\]  
(9.16)

Again, the side condition has to be true in order to use the rule \(SEQI_u\), i.e., \(\text{unmarked}(d.r)\) is

\[(d \not= \text{null} \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r)))/
\]

\[(\text{region}\{d.\text{mark}\}, \text{unmarked}(d.l))\text{-immune} \quad (9.17)\]

By the definition of immune (Def. 6), the proof obligation is to show:

\[\text{• for all modifies } RE \in \text{region}\{d.\text{mark}\}:\]

\[(d \not= \text{null} \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \text{ implies } efs(RE) :/ \text{unmarked}(d.r).\]

In this case, \(RE\) is just \(\text{region}\{d.\text{mark}\}\), by the assumption, which is disjoint with \(fpt(\text{dag}(d.l) \land \text{dag}(d.r))\) that is \(\text{unmarked}(d.r)\)’s superset.
• for all modifies $RE \in \text{unmarked}(d.l)$:

$$(d \neq \text{null} \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))) \implies \text{efs}(RE) / \text{unmarked}(d.r).$$

It is proved by contradiction. Suppose under the assumption, there is some $RE$ that has the form $\text{region}\{d.f_1, \ldots, f_n, \text{mark}\}$, then $\text{efs}(RE)$ is $\text{region}\{d.f_1, \ldots, f_n\}$, where $f_i \in \{l, r\}$, and $1 \leq i \leq n$. Moreover $d.f_1, \ldots, f_n$ has the type $\text{Dag}$. However, all the regions in $\text{unmarked}(d.r)$ has the form $\text{region}\{d.f_1, \ldots, f_m, \text{mark}\}$, and $d.f_1, \ldots, f_m, \text{mark}$ has the type $\text{int}$, where $f_j \in \{l, r\}$, and $1 \leq j \leq m$. Thus, there is no overlapping between the two sets of locations.

Now by using the rule $\text{SEQ1}_u$, the following is derived:

$$\begin{align*}
\begin{array}{l}
\{d \neq \text{null} \ast d.l \mapsto i \ast d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))\} \\
\text{d.mark} := 1; \text{mark}(d.l); \text{mark}(d.r);
\end{array}
\end{align*}
$$

$$\begin{array}{l}
\Gamma_u \frac{\begin{array}{l}
(d.r \neq \text{null} \Rightarrow \forall n: \text{Dag}(\text{region}\{n.mark\} \leq \text{fpt}(\text{dag}(d.r)) \Rightarrow \\
n.mark \mapsto 1)) \land \\
(d.l \neq \text{null} \Rightarrow \forall n: \text{Dag}(\text{region}\{n.mark\} \leq \text{fpt}(\text{dag}(d.l)) \Rightarrow \\
n.mark \mapsto 1)) \land (\text{d.mark} \mapsto 1 * d.l \mapsto i * d.r \mapsto j \ast (\text{dag}(d.l) \land \text{dag}(d.r))))
\end{array}}{[\text{modifies region}\{d.mark\}, \text{unmarked}(d.l), \text{unmarked}(d.r)]}
\end{array}
$$

(9.18)

The postcondition above can imply the one for $\text{mark}$, thus the program is verified.

Remark: in this example, the write effects of $\text{mark}$ are not necessarily precise. Let $\varepsilon_l$ and $\varepsilon_r$ be the write effects of $\text{mark}(d.l)$ and $\text{mark}(d.r)$ respectively. Suppose the location $\text{region}\{x.mark\}$ is contained in both write effects. To use the sequence rule, the proof obligation is to show that $\varepsilon_r$ is $\text{dag}(d) / \varepsilon_l$-immune. By the definition of immune (Def. 6), the proof obligation is to show that for all modifies $RE \in \varepsilon_r :: RE$ is $\text{dag}(d) / \varepsilon_l$-immune. In this case, the proof obligation is to show that $\text{dag}(d)$ implies $\text{efs}(\text{region}\{x.mark\}) / \varepsilon_1$, by Def. 6. By the definition of read effects, $\text{efs}(\text{region}\{x.mark\}) = \text{reads } x$. There are two cases.
1. $x = d$. In this case, the proof obligation is to show that $\text{dag}(d)$ implies $\text{reads } d \neq \text{region}\{d.\text{mark}\}$, which is true.

2. $x = d.f_1, \ldots, f_n$, where $f_1, \ldots, f_n$ are either the field name $l$ or the field name $r$. In this case, the proof obligation is to show that $\text{dag}(d)$ implies $(\text{reads region}\{d.f_1, \ldots, f_n\}, \text{efs}(d.f_1, \ldots, f_{n-1}) \mathbin{\cup} \text{region}\{d.f_1, \ldots, f_n.\text{mark}\})$, which is true because, the field $f_n$ has type $\text{Dag}$, the field $\text{mark}$ has the type $\text{bool}$, thus $\text{region}\{d.f_1, \ldots, f_n\} \neq \text{region}\{d.f_1, \ldots, f_n.\text{mark}\}$. Similarly, the regions contained in the read effect $\text{efs}(d.f_1, \ldots, f_{n-1})$ are all disjoint with the region $\text{region}\{d.f_1, \ldots, f_n.\text{mark}\}$.

9.5 An Integrated Example

This subsection demonstrates mixed specification and verification in FRL and SSL, using an order program for a coffee shop as an example. Parts of this program are specified in the style of FRL, parts in SSL, and parts in a mixed style. Consider a client code shown in Fig. 9.5.

```plaintext
var menu : Menu; menu := new Menu;
var shop : CoffeeShop; shop := new CoffeeShop(menu);
var shop1 : CoffeeShop; shop1 := new CoffeeShop(menu);
shop.takeOrder(1,1); shop.takeOrder(1,3); shop.takeOrder(2,3);
shop.takeOrder(4,5); shop1.takeOrder(3,3); shop1.takeOrder(1,1);
shop1.takeOrder(2,4); shop1.takeOrder(4,6);
shop.service();
```

Figure 9.5: A client code of a coffee shop

Two shop objects share one menu object. Taking orders and performing services only read the menu. Thus, it can be proved that executing shop1’s method service preserves shop2’s property, as the write effects of shop1.service in Fig. 9.11 do not overlap the read effects.
of shop2’s predicate. In particular, the read effects of menu’s predicate are separate from the write effects of shop1.service. This is credited to FRL’s flexibility of specifying write effects. Another example that showcases such a benefit is the specification of iterator written in FRL in Fig. 9.7. The keyword pure is another way to specify that hasNext does not have write effects. If the iterator methods hasNext and next were specified in SSL, then their frames would contain the footprints of their preconditions, so the underlying data structure would be modifiable. These larger write effects would also propagate to service, since that method needs to call the iterator methods, so its write effects of service would have to contain the footprint of the iterator methods. These larger write effects could cause trouble in some cases.

In addition, the SSL style of specifications has been used in the example as well, i.e., the specification of add in Fig. 9.6. Moreover, the use of separating conjunction makes the specifications concise.

Here explains the example in detail. The program is deployed to a digital device on each table. Customers or waiters order coffee by choosing item numbers from the menu. For each item on the menu, the system will look for its identifier (which is used in some other internal systems). For simplicity, assume that each order only contains one item. Each table may have multiple orders.

The coffee shop maintains a list of orders and the menu; each order stores a table number, the menu item number, and whether it has already been served. The list of orders is implemented by a generic linked-list List<T> in Fig. 9.6. The class List<T> is implemented by a list of Node<T> that may be invisible to clients. For the convenience, the specifications of the class Node<T> that are used to verify the implementation of the class List<T> are summarized in Table. 9.1. The specifications and implementations of the class List<T> are shown in Fig. 9.6 on the following page. One can add a node to the list by invoking the method add, test whether a list is empty or not by invoking the pure method isEmpty, and obtain its iterator by invoking the pure method iterator. Fig. 9.7 shows an implementation of List<T>’s iterator. The field curr denotes the cursor position.
class List<T>{
    var h : Node<T>;

    predicate vList(se: seq<T>)
        reads fpt(vList(se));
    { lst(h, se)  }

    method List<T>()
        requires true;
        modifies region{this.*};
        ensures vList([]);
    { h := null; }

    method add(t : T)
        requires vList(?vlst);
        ensures vList(vlst + [t]);
    {
        var n: Node<T>;
        n := new Node<T>(t);
        if h = null then h := n;
        else h.append(n);
        /* calls append method of node h */
    }

    method isEmpty() : int
        requires vList(?vlst);
        reads region{this.*};
        ensures (h = null ⇒ ret = 1) &&
            (h ≠ null ⇒ ret = 0)
    { if(h = null) ret := 1; else ret := 0; }

    method iterator() : ListIterator<T>
        requires vList(?vlst);
        fresh region{ret.*};
        ensures ret ≠ null && ret.list = this
            && ret.curr = this.h && ret.vLIter(vlst);
    { ret := new ListIterator<T>(this); }

    /* ... other methods omitted */
}

Figure 9.6: The specification of a generic linked-list written in a mixed style
class ListIterator<T>{
    var list : List<T>;   var curr : Node<T>;

    method ListIterator(l : List<T>)
        requires l != null && l.vList(?vlst);
        modifies region{this.*};
        ensures list = l && curr = l.h
                       && vLIter(vlst);
        { list := l; curr := l.h; }

    method hasNext() : int
        requires vLIter();
        ensures (curr != null => ret = 1)
                    && (curr = null => ret = 0);
        { 
            if (curr != null) then ret := 1;
            else ret := 0;
        }

    method next() : T
        requires vLIter() && hasNext();
        modifies region{this.curr};
        ensures (curr = old(curr.next)) &&
                    ret = old(curr.get());
        { ret := curr.get(); curr := curr.next; }

    predicate vLIter()
        reads fpt(vLIter());
        { list != null && list.vList(?vlst)
                      && vLIter(vlst) }

    predicate vLIter(vlst: seq<T>)
        reads fpt(vLIter(vlst));
        { list.vList(vlst) &&
                       region{curr.*} <= fpt(list.vList(vlst))
        }

    /= ... other methods omitted */
}

Figure 9.7: The class ListIterator specified in the style of FRL
Table 9.1: Selected specifications for the class Node<T>

<table>
<thead>
<tr>
<th>Method</th>
<th>Precondition</th>
<th>Postcondition</th>
<th>Write effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node&lt;T&gt;(v)</td>
<td>true</td>
<td>lst(this, [v])</td>
<td>region{this.*}</td>
</tr>
<tr>
<td>get()</td>
<td>true</td>
<td>ret = this.val</td>
<td>Ø</td>
</tr>
<tr>
<td>append(n)</td>
<td>lst(n, [?v]) + lst(this, ?vlst)</td>
<td>lst(this, vlst + [v])</td>
<td>region{last().next}</td>
</tr>
</tbody>
</table>

Fig. 9.8 specifies a generic dictionary as a mapping. The generic Dictionary<Key,Value> is implemented by an acyclic list of Pair<Key,Value> that may be invisible to the clients. A generic mathematical sequence map<Key,Value> is used as an abstract model of the values stored in Dictionary<Key,Value>. Operations and formulas for a map are defined in Table. 2.1. The pure method lookup returns a value for a given key. Its precondition makes sure that the key is in the domain of the dictionary.

The class order contains table, itemId and served. The field table records the number of the table in an order. The field itemId stores a coffee’s identifier. The field served tracks whether the order is served. The class CoffeeShop maintains a List of Order and a menu that is initialized by the parameter of the constructor of CoffeeShop, and stores the mapping between Coffee’s numbers and identifiers. For simplicity, the details of Menu is omitted. The method takeOrder looks up the coffee’s identifier in the menu, generates a new order and adds it to the order list. The method service sets the orders to be served. The predicate cshop specifies the structure of a CoffeeShop. The formal parameter lseq specifies the sequence of Order. The formal parameter oseq specifies the contents of orders in the list. The following formula specifies that the sequence of Order contains the expected contents:

$$\forall i. 0 \leq i < |lseq| \Rightarrow lseq[i].vOrder(oseq[3 \ast i..3 \ast i + 2]),$$

where oseq[i..j] generates a new sequence that starts from the element oseq[i] and end with the element oseq[j]. It is well-formed if 0 \leq i \leq j \leq |oseq|. The sequence oseq is the
**predicate** \(\text{dic}(p : \text{Pair<Key, Value>}, m : \text{map<Key, Value>})\)
- **reads** \(\text{fpt}(\text{dic}(p, m))\);
- **decreases** \(|m|\);

{ 
\(p = \text{null} \Rightarrow |m| = 0\) \land \
p \neq \text{null} \Rightarrow p.\text{key} \in m \land p.\text{val} \mapsto m[p.\text{key}] * \)
\(\text{dic}(p.\text{next}, (\text{map } i \mid i \in m \land i \neq p.\text{key} :: m[i]))\)
}

**class** Pair<Key, Value>{
- **var** key : Key;
- **var** val : Value;
- **var** next : Pair<Key, Value>;
}

**class** Dictionary<Key, Value>{
- **var** head : Pair<Key, Value>;

**predicate** \(\text{vDic}(m: \text{map<Key, Value>})\)
- **reads** \(\text{vDic}(m)\);

{ 
\(\text{dic}(\text{head}, m)\)
}

**method** lookup(k: Key) : Value
- **requires** \(\text{vDic}(?m) \land k \in m;\)
- **ensures** \(\text{vDic}(m) \land \text{ret} = m[k];\)

{ /* ... omitted */ }

/* ... other methods omitted */
}

**Figure 9.8:** The specification of a generic dictionary written in the style of SSL

flattened sequence of 3-element array. Each array corresponds the three fields of an order. The formal parameter \(m\) specifies the menu. In the dynamic frames approach [43, 44], this can be specified by declaring these three parameters as ghost fields and updating them when it is needed.

**Abstraction:** Although information hiding and abstraction are not a focus of this dissertation, they
class Order{
    var table int;
    var itemId : int;
    var served : int;

    predicate vOrder(se: seq<int>)
        reads region{this.*};
    {
        |se| = 3 &&
        this.table ↦ se[0] *
        this.itemId ↦ se[1] *
        this.served ↦ se[2]
    }

    method Order(t: int, item: int)
        modifies region{this.*};
        ensures vOrder([t, item, 0]);
    {
        this.table := t; this.itemId := item;
        this.served := 0;
    }

    method served()
        requires this.served ↦ _;
        ensures this.served ↦ 1;
    {
        this.served := 1;
    }

    method isServed() : int
        reads region{this.served};
        ensures ret = served;
    {
        if(served = 1) then ret := 1; else ret := 0;
    }

    /* ... other methods omitted */
}

Figure 9.9: The class Order
class CoffeeShop{
    var orders : List<Order>;    var menu : Dictionary<int, int>;

    predicate cshop(lseq: seq<Order>, oseq: seq<int>, m: map<int, int>)
        reads fpt (cshop(lseq, oseq, m));
    { orders ≠ null * menu ≠ null * orders.vList(lseq) * menu.vDis(m) *
        ∀ i. 0 <= i && i <|lseq| ⇒ lseq[i].vOrder(oseq[3*i..3*i+2])
    }

    function severd_seq(se: seq<T> ) : seq<T>
        requires ∃ i. (i >= 0 ⇒ |seq| = 3 * i);
        reads Ø;
        decreases |se|;
    { if se = [] then ret := [];
        else ret := se[2 := 1] + severd_seq(se[3..])
    }

    method CoffeeShop(menu : Menu)
        requires menu ≠ null && menu.vDic(?m);
        modifies region{this.*};
        ensures cshop([], [], m);
        /* ... omit the postcondition about menu */
    { orders = new List<Order>(); /* ... omitted */ }

    method takeOrder(item: int, table: int) : Order
        requires cshop(?lseq, ?oseq, ?m) && item ∈ m;
        ensures cshop(lseq + [ret], oseq + [table, item, m[item]], m)
    { var itemId = menu.lookup(item);
        ret := new Order(table, itemId);
        orders.add(ret);
    }

Figure 9.10: The specification of an application program written in a mixed style (part 1)
method service()
  requires cshop(?lseq, ?oseq, ?m);
  modifies filter(fpt(cshop(lseq, oseq, m)), Order, served);
  ensures cshop(lseq, severd_seq(oseq),m);
{
  var iter := orders.iterator();
  while (iter.hasnext()){
    var o = iter.next();
    if (o.isServed ≠ 1)
      o.served();
  }
}

/* ... other methods omitted */

Figure 9.11: The specification of an application program written in a mixed style (part 2)

figure prominently in other works on SL [75, 76]. This technique can also be handled. In the
example, assume that the classes List<T> and Dictionary<Key, Value> are libraries,
and are declared in a separate module from clients. Their implementations are hidden from its clients. Thus, their clients can only see their predicate names. Table. 9.2 summarizes the set of predicate names that are used to visible to clients.

Therefore, the class CoffeeShop uses the name of predicates vList and vDis to define its own predicate; the actual formulas that are defined by those predicated are abstracted away. Thus, CoffeeShop does not know the internal representation of List, thus is not influenced by the change of List's representation, i.e., replacing a linked list with an array.

However, some specifications use the hidden fields to describe observable behaviors of methods. For example, the write effects of the method next in Fig. 9.7 exposes the field curr that is supposed to be a private field. This can be solved by (at least) two established methodologies: data groups [50, 59] and model variables [23, 49]. Following JML [45], the second approach is
explored. Model variables are used to define abstract values. For example, the specifications of `ListIterator<T>` can be revised by declaring

```java
public model var _curr;
private represents _curr <- curr;
```

Here `_curr` is a model variable represented by the private field `curr`. The represents clause says that the value of `_curr` is the value of the field `curr`. That is, the value of `_curr` changes immediately when the value of `curr` changes. Moreover, the location `this._curr` is connected with the location `this.curr` implicitly. Thus the write effects of the method `next` can be rewritten as:

```java
modifies region(this._curr);
```

And also the specifications that use `this.curr` can be rewritten by substituting `this._curr` for it. For simplicity, in the remainder of this dissertation, program fields are used and are considered to be publicly accessible in specifications.

**Interoperation:** The specifications in this example are written in different styles, nevertheless, they can be combined and used in verification. The example is used to show how to verify that the implementation of the method `takeOrder` satisfies its specification.

A preliminary step in making the different styles interoperate with each other (following Section 9.2) is to translate specifications without explicit effects into UFRL, giving them explicit read and write effects. For the SL specifications, these effects are derived from the footprint of the SL
precondition. For example, the specification of the method `lookup` in Fig. 9.8 is encoded in UFRL as:

\[
\begin{align*}
\text{[reads fpt]} & (vDic(m) \land k \in m)] \\
\{vDic(?m) \land k \in m\} \triangledown := \text{lookup}(k:Key); \{vDic(m) \land v = m[k]\} \\
\text{[modifies fpt]} & (vDic(m) \land k \in m)]
\end{align*}
\]  

(9.19)

By using the rule `SubEff_u`, the following is derived:

\[
\begin{align*}
\text{[reads alloc]} \\
\{vDic(?m) \land k \in m\} \\
\vdash_u \text{lookup}(k:Key) \text{returns} \ (v:Value) \\
\{vDic(m) \land v = m[k]\} \\
\text{[modifies fpt]} & (vDic(m) \land k \in m)]
\end{align*}
\]  

(9.20)

Specifications with explicit write effects are encoded into those in UFRL with read effects that are `reads alloc]`. For example the specification of `takeOrder` is encoded in UFRL as:

\[
\begin{align*}
\text{[reads alloc]} \\
\{cshop(?lseq, ?oseq, ?m) \land item \in m\} \\
\vdash_u \text{takeOrder}(item : \text{int}, table : \text{int})\text{returns} (ret : \text{Order}) \\
\{cshop(lseq + \{ret\}, oseq + \{table, item, m.[item]\}, m)\} \\
\text{[modifies fpt]} & (cshop(lseq,oseq,m) \land item \in m)]
\end{align*}
\]  

(9.21)

Proceeding to the verification of the body of `takeOrder`, its precondition is assumed:

\[
cshop(lseq,oseq,m) \land item \in m,
\]  

(9.22)

which implies the precondition of `menu.lookup` by using the definition of the predicate `cshop` in Fig. 9.10. For the write effects, by the definition of the predicate `cshop` again, it must be true that `fpt(vDic(m)) \leq fpt(cshop(lseq,oseq,m) \land item \in m).`
Thus, the method call `menu.lookup` is allowed in the body of the method `takeOrder`. After finishing executing method `menu.lookup`, its postcondition gets assumed: `menu.vDis(m) \land itemId = m[item]`.

As the precondition of the constructor of `Order` is true, and it only changes the values in the newly allocated locations on the heap; it does not change existing locations. Thus, it is allowed in the body of the method `takeOrder`. After it finishes executing, and by using the rule $I_{sc}$, it must be true that $(menu.vDis(m) \land itemId = m[item]) \ast \text{ret.vOrder([table, itemId, 0])} \ast \text{orders.vList(lseq)}$, which implies the precondition of the method `orders.add(ret)`. For the write effects, according to Section 9.2, its specification is encoded as:

$$[\text{reads alloc}]\{vList(?vlst)\} \text{add(}t\text{); } \{vList(vlst + [t])\}\{\text{modifies fpt}(vList(vlst))\}. \tag{9.23}$$

Together with the definition of the predicate $c\text{shop}$ in Fig. 9.10, the following must be true:

$$\text{fpt(orders.vlst(lseq))} \leq \text{fpt(cshop(lseq, oseq, m) \&\& item } \in m\text{)}, \tag{9.24}$$

Thus, `orders.add` is allowed in the body of `takeOrder`. After finishing executing it, its postcondition gets assumed. And using the rule $I_{sc}$, it must be true that $(menu.vDis(m) \land itemId = m[item]) \ast \text{ret.vOrder([table, itemId, 0])} \ast \text{orders.vList(lseq + [ret])}$, which implies the postcondition of `takeOrder`. Thus, the implementation is verified.

**Verifying a Client of `CoffeeShop`:** For simplicity, assume the items that customers chose are all available, i.e., always exist in the internal system. Using the specification of `CoffeeShop`, consider the client code in Fig. 9.5. Although the two instances, `shop` and `shop1`, share `menu`, the write effects of `service` claim that only the fields `served` of the object `Order` may be modified. Thus, the following is true:

$$\text{reads fpt(shop1.cshop(?l, ?o, m))};$$

$$\text{modifies filter(fpt(shop.cshop(lseq, oseq, m)), Order, served)}. \tag{9.25}$$
class DCell extends Cell {
    method set(v : int)
    also
        requires inv();
        modifies wf_set();
        ensures this.val = 2*v;
        { super.set(2*v); }
}

Figure 9.12: The specification of the class DCell

Then using the rule $FRM_u$ and the rule $CONSEQ_u$, it can be proved that shop1 is not served. Note that in the body of service, an iterator is used. As it only reads the underlying data structure that is traversing, the iterator is specified in the style of FRL; the underlying data structure is specified to be untouched. That allows the write effects of service to be precise.

9.6 Examples on Behavioral Subtyping

This section presents examples of reasoning about inheritance. Fig. 9.12, Fig. 9.13 and Fig. 9.14 are examples adapted from Parkinson and Bierman’s work [76]. The class DCell reuses the field name val of its superclass and stores double the value which is passed to. There is no need to override the pure method wf_set and the predicate inv as there is no additional fields.

The combined specification of DCell.set is shown below:

\[
\{ \text{this.inv()}) \\
\text{DCell.set}(v : \text{int}) \\
\{ \text{this.val} = v \&\& \text{this.inv()} \&\& \text{this.val} = 2*v \} \\
[ \text{modifies this.wf_set()} ]
\]

The postcondition implies false. Thus, the body of DCell.set is not a correct implementation against its specification. If the implementation is while true { skip; }, then it would
satisfy the specification, as the definition of the correctness judgment assumes termination. This implementation does not terminate, so the specification is vacuously satisfied.

The class ReCell extends from class Cell in Fig. 8.7 by introducing the field bak. The method ReCell.set overrides the one declared in its superclass; it updates val with the new value x and stores its old value in the field bak. Its write effect is specified by the pure method $wf\_set()$ that overrides $Cell.wf\_set()$ as well, where $super.wf\_set$ is invoked in its body.

The specification of ReCell.set is shown below. Following the definition of specification inheritance (Eq. (8.2)), they are the combination of the one specified in the class ReCell and the one of Cell.set.

\[
\{\text{super.inv()} \implies (\exists d.\text{this.val} = d) \land \text{this.inv()})
\]

ReCell.set(v : int)

\[
\{\text{old(super.inv())} \implies \text{this.val} = v \land super.inv() \land
\]

old((\exists d.\text{this.val} = d) \land \text{this.inv()}) \implies \text{this.val} = v \land \text{this.bak} = d
\]

[modifies this.wf_set()]

By the program semantics, the assertion $\exists d.\text{this.val} = d$ is implicit. Thus, by the rule $CONSEQ_r$, the above specification is derived to

\[
\{\exists d.\text{this.val} = d \land \text{this.inv()})
\]

ReCell.set(v : int)

\[
\{\text{this.val} = v \land \text{this.bak} = d
\]

[modifies this.wf_set()]

Fig. 9.14 shows the specification of the class TCell. It declares an additional field val2. A type invariant, this.val = this.val2, restricts the behavior of its super class Cell [76]. The
combined specification of \texttt{TCell.set} is shown below:

\begin{verbatim}
\{ \texttt{super.inv()} \parallel \texttt{this.inv()} \}
\texttt{TCell.set}(v: \texttt{int})
\{ \{ \texttt{old(super.inv())} \Rightarrow \texttt{this.val} = v \&\& \texttt{super.inv}() \} \&\&
\{ \texttt{old(this.inv())} \Rightarrow \texttt{this.val} = v \&\& \texttt{this.inv}() \}
\}[	exttt{modifies this.wf_set()}]
\end{verbatim}

By the rule \textit{CONSEQ}, the above specification is derived to

\begin{verbatim}
\{ \texttt{this.inv()} \}
\texttt{TCell.set}(v: \texttt{int})
\{ \{ \texttt{old(this.inv())} \Rightarrow \texttt{this.val} = v \&\& \texttt{this.inv}() \}
\}[	exttt{modifies this.wf_set()}]
\end{verbatim}

It can be proved that the implementation of the method \texttt{TCell.set} satisfies the above specification.

Fig. 9.15 declares the class \texttt{OCell} that inherits from the class \texttt{Cell}, and declares two additional fields, \texttt{c} and \texttt{o}. Together with the inherited field \texttt{val}, they compose \texttt{OCell}'s data representation. The field \texttt{c} is created by \texttt{OCell}'s constructor and is encapsulated. But the field \texttt{o} is initialized by the constructor's parameter. That causes argument exposure. Therefore, the last line of the constructor updates the field exposed by \texttt{o.df}.

The method \texttt{OCell.set} updates both \texttt{this.val} and \texttt{this.c.val} with the new value. Its write effects is specified by the function \texttt{wf_set}. Its definition returns regions that come from two parts: one part is from its supertype, i.e., \texttt{super.wf_set()}; the other is from the extended state, i.e., \texttt{c.df}. Thus, the definition of the predicate \texttt{inv} describes the disjointness, i.e., \texttt{this.exposed!!c.df}, which means that \texttt{c.df} is encapsulated by \texttt{OCell}. Therefore, the specification of \texttt{OCell.set} refines the specification of \texttt{Cell.set}. 

149
class ReCell extends Cell
{
    var spec_public bak : int;

    predicate inv()
        reads this.df;
        {
            this.df = region{this.val} + region{this.bak} &&
            this.exposed = region{}
        }

    pure method wf_set() : region
        reads wf_set;
        {
            ret := super.wf_set() + region{this.bak};
        }

    method ReCell()
        modifies region{this.*};
        ensures this.val = 0 && this.bak = 0;
        ensures inv();
        {
            this.val := 0;
            this.bak := 0;
            this.df := region{this.val} + region{this.bak};
            this.exposed := region{};
        }

    method set(v : int)
        also
        requires (∃ d this.val = d) && inv();
        modifies wf_set();
        ensures val = v && bak = d;
        {
            bak := val; super.set(v);
        }
}

Figure 9.13: The specification of the class ReCell
class TCell extends Cell {
    var spec_public val2 : int;

    predicate inv()
        reads this.df;
    {
        this.df = region{this.val} + region{this.val2} &&
        this.exposed = region{} &&
        this.val = this.val2
    }

    pure method wf_set() : region
        reads wf_set;
    {
        ret := super.wf_set() + region{this.val2};
    }

    method TCell()
        modifies region{this.*};
    ensures this.val = 0 && this.val2 = 0;
    ensures inv();
    {
        this.val := 0;
        this.val2 := 0;
        this.df := region{this.val} + region{this.val2};
        this.exposed := region{};
    }

    method set(v : int)
    also
        requires inv();
    modifies wf_set();
    ensures this.val = v && inv();
    {
        super.set(v);
        this.val2 := v;
    }
}

Figure 9.14: The specification of the class TCell
class OCell extends Cell {
    var spec_public c : Cell;
    var spec_public o : Object;

    pure wf_set() : region
        reads wf_set();
    { ret := super.wf_set() + c.df }

    predicate inv()
        reads this.df;
    { (df = region(this.val) + region(this.c) + region(this.o) +
            (c != null ? c.df : region{ }) +
            o != null ? o.df : region{ }) &&
            (this.exposed != c.df) )
    }

    method OCell(o : Object)
        requires o != null;
        modifies region{ this.* };
        ensures this.o = o && inv();
    {
        this.o := o;
        this.c := new Cell();
        this.df := region{ this.* } + c.df + o.df;
        this.exposed := this.o.df + this.c.exposed;
    }

    method set(v : int)
        also
        requires c != null && inv();
        modifies wf_set();
        ensures v = this.val && v = c.get() && inv();
    { c.set(this.v); super.set(v); }
}

Figure 9.15: The specification of the class OCell
CHAPTER 10: CONCLUSION AND FUTURE WORK

This dissertation has presented the logic, UFRL, which is able to reason about object-based programs specified in the styles of FRL and SSL. This is accomplished by a translation from SSL to UFRL which preserves not only the meaning of assertions, but which can also translate proofs in SSL into UFRL proofs. Thus, UFRL provides a single mechanism that allows FRL and SSL to interoperate with each other, allowing designers flexibility in writing specifications in either style or in a mix of styles. Also, a frame condition for behavioral subtyping is defined and proved sound.

10.1 Future Work

10.1.1 Formalization

The programming language defined in this dissertation lacks many features, such as exceptions, access modes, module declarations, etc. The problems of data abstraction and information hiding are not addressed in this dissertation. Extending the language with these features would make the programs closer to the programs used in practice, such as Java programs. That would ease the transfer of the ideas from this dissertation on UFRL into JML [22].

This dissertation only formalized a restricted form of recursive predicates. Mutual recursive predicates, functions and pure methods [3] that are used in the examples should be added to the assertion language.

10.1.2 Encoding or incorporate other methodologies

Parkinson and Bierman [75, 76] develop abstract predicate families to reason about inheritance based on the separation logic that requires a second-order quantifier. UFRL could be extended to encode these abstract predicate families as well. Then, it would be possible to compare the results on reasoning about inheritance. One way of encoding would use model fields or pure methods.
The ownership model has been extensively studied in many works, such as [8, 10, 63, 66, 70]. It is valuable to connect FRL or UFRL with one of these ownership models. In particular, FRL or UFRL specifications could be simplified if they could generate equivalent conditions for some properties and proof obligations that are generated by the Universe Type System [63]. Another methodology that FRL or UFRL could incorporate is the work on typestate [17, 18, 27, 85]. Typestates provide a way to more abstractly write specifications. A typestate transition graph can be generated, which give designers and programmers intuitions about the object. FRL may use ghost variables to express typestates.

10.1.3 Implementation

Both FRL and UFRL can be encoded into first-order logic with modular verification. Firstly, the quantifiers in FRL (UFRL) are first-order. This allows implementations to use a theorem prover like an SMT solver, such as Z3 [26] or CVC4 [12]. Secondly, the type region in FRL (UFRL) is a set of locations. Region operators, i.e., union, difference and intersection, are translated into corresponding set operations, which are first-order operations. Thirdly, automated verification tools prove programs in a method-modular way. When verifying a method body, its precondition is assumed. Automated verification tools check whether the locations that are intended to update by a method body are a subset of the method’s frame conditions. Instead of directly accumulating effects and composing each proof rule similarly to the approach used in the dissertation, verification tools for FRL (UFRL) can be implemented by computing weakest preconditions or by symbolic execution.

The intermediate verification language Boogie [8, 52] can be used to generate verification conditions. The \( fpt_s \) function would be encoded to an uninterpreted function and axioms in the generated Boogie program [35].
APPENDIX A: TYPING RULES
This section shows the typing rules. The predicate isRef(T) returns true just when T is a reference type in the program. Typing rules for expressions and region expressions are shown Fig. A.1. Typing rules for statements are shown in Fig. A.2. The typing rules for assertions are defined in Fig. A.3.

\[
\begin{align*}
\Gamma \vdash n : \text{int} & \quad \Gamma \vdash x : T \\
\text{where } \Gamma(x) = T & \quad \Gamma \vdash \text{null} : T \\
\text{where isRef}(T) & \\
\Gamma \vdash E_1 : T_1 & \quad \Gamma \vdash E_2 : T_2 & \quad \Gamma \vdash \oplus : T_1 \rightarrow T_2 \rightarrow T \\
\Gamma \vdash E_1 \oplus E_2 : T & & \Gamma \vdash \text{region\{} : \text{region} \\
\Gamma \vdash x : T & \\
\Gamma \vdash \text{region\{}x.f\} : \text{region} & \quad \Gamma \vdash \text{region\{}x.*\} : \text{region} \\
\text{where } (f : T') \in \text{fields}(T) \text{ and isRef}(T) & \quad \text{where isRef}(T) \\
\Gamma \vdash E : \text{bool} & \quad \Gamma \vdash RE_1 : \text{region} & \quad \Gamma \vdash RE_2 : \text{region} \\
\Gamma \vdash E \? RE_1 : RE_2 : \text{region} & \\
\Gamma \vdash \text{filter\{}RE, T, f\} : \text{region} & \quad \Gamma \vdash \text{filter\{}RE, T\} : \text{region} \\
\text{where isRef}(T) & \quad \text{where isRef}(T) \\
\Gamma \vdash RE_1 : \text{region} & \quad \Gamma \vdash RE_2 : \text{region} \\
\Gamma \vdash RE_1 \otimes RE_2 : \text{region} & \\
\end{align*}
\]

Figure A.1: The typing rules for pure expressions and region expressions
\[
\begin{align*}
\Gamma \vdash \text{skip} : \text{ok}(\Gamma) & \quad \Gamma \vdash \text{var} \; x : T ; : \text{ok}(\Gamma, x : T) \\
& \quad \text{where } x \notin \text{dom}(\Gamma) & \quad \Gamma \vdash x : T \quad \Gamma \vdash G : T \\
& \quad \frac{\Gamma \vdash x := G ; : \text{ok}(\Gamma)}{\Gamma \vdash \text{where } x \neq \textbf{this}}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : T & \quad \Gamma \vdash y : T' \\
& \quad \frac{\Gamma \vdash x := y.f ; : \text{ok}(\Gamma)}{\Gamma \vdash \text{where } x \neq \textbf{this}, \text{isRef}(T') \text{ and } (f : T) \in \text{fields}(T')} \\
\Gamma \vdash x : T' & \quad \Gamma \vdash G : T \\
& \quad \frac{\Gamma \vdash x.f := G ; : \text{ok}(\Gamma)}{\Gamma \vdash \text{where isRef}(T') \text{ and } (f : T) \in \text{fields}(T')} \\
\Gamma \vdash E : \text{bool} & \quad \Gamma \vdash S_1 : \text{ok}(\Gamma_1) \quad \Gamma \vdash S_2 : \text{ok}(\Gamma_2) \\
& \quad \frac{\Gamma \vdash \text{if } E \text{ then } \{S_1\} \text{ else } \{S_2\} : \text{ok}(\Gamma)}{\Gamma \vdash E : \text{bool} \quad \Gamma \vdash S : \text{ok}(\Gamma')} \\
\Gamma \vdash \text{while } E \{S\} : \text{ok}(\Gamma) & \quad \Gamma \vdash E : \text{bool} \quad \Gamma \vdash S : \text{ok}(\Gamma') \\
\Gamma \vdash S_1 : \text{ok}(\Gamma') & \quad \Gamma'' \vdash S_2 : \text{ok}(\Gamma') \\
& \quad \frac{\Gamma \vdash S_1S_2 : \text{ok}(\Gamma')}{\Gamma \vdash \text{if } E \text{ then } \{S_1\} \text{ else } \{S_2\} : \text{ok}(\Gamma)}
\end{align*}
\]

Figure A.2: The typing rules for statements
Figure A.3: The typing rules for assertions
APPENDIX B: PROOF OF THEOREM 1
Theorem 1: The judgment \( \vdash_{r}^{\Gamma} \{ P \} S \{ Q \}[\varepsilon] \) that is derivable by the axioms and inference rules in Fig. 4.1, and the structural rules in Fig. 4.2 and Fig. 4.3 are valid.

Proof. The proof is done by induction on the derivation and by cases on the last rule used. In each axiom, it is shown that the judgment is valid according to the statement’s semantics. In each inference rule, it is shown that the proof rule derives valid conclusions from valid premises when its side conditions is satisfied. Let \( S \) be a statement and \((\sigma, h)\) be an arbitrary state, and without loss of generality, let \((\sigma', h') = MS[\vdash S : ok(\Gamma')][(\sigma, h)]\). Assume \( \vdash_{r}^{\Gamma} \{ P \} S \{ Q \}[\varepsilon] \), and \( \sigma, h \vdash_{r}^{\Gamma} P \). Then the proof obligation is to prove \( \sigma', h' \vdash_{r}^{\Gamma'} Q \), and that all the changed locations are in \( \varepsilon \). There are 6 base cases.

1. (\text{SKIP}_r) In this case, \( S \) is \text{skip}; \( P \) is \text{true}, \( Q \) is \text{true}, \( \varepsilon \) is \( \emptyset \). By the program semantics Fig. 4.1, \( \sigma' = \sigma, h' = h \) and \( \Gamma' = \Gamma \). Thus, \( \sigma', h' \vdash_{r}^{\Gamma'} \text{true} \). For the frame condition, \( S \) does not change anything, thus, it is \( \emptyset \).

2. (\text{VAR}_r) In this case, \( S \) is \text{var} \( x : T \); \( P \) is \text{true}, \( Q \) is \( x = \text{default}(T) \) and \( \varepsilon \) is \( \emptyset \). By the program semantics Fig. 4.1, \( \Gamma' = \Gamma, (x : T), \sigma' = \text{Extend}(\sigma, x, \text{default}(T)) \) and \( h' = h \). Thus \( (\sigma', h') \) entails \( Q \). For the frame condition, as the statement does not change anything existing in the prestate, thus, it is \( \emptyset \).

3. (\text{ALLOC}_r) In this case, \( S \) is \( x := \text{new} T \); \( P \) is \text{true}, \( Q \) is \( x = \text{default}(T) \) and \( \varepsilon = \text{modifies} x, \text{alloc}, \text{fresh(region}\{x.\star\}) \). By the program semantics Fig. 4.1, \( \Gamma' = \Gamma, \sigma' = \sigma(x \mapsto l) \) and \( h' = h''[\{l, f \mapsto \text{default}(T)\}] \), where \( (l, h'') = \text{allocate}(T, h) \). Thus, \( (\sigma', h') \) entails \( Q \).

For the frame condition, \( S \) only updates the variable \( x \) and \text{alloc}. By the semantics, the function \text{allocate} returns a new heap. So \text{fresh(region}\{x.\star\}) is the fresh effect.

4. (\text{ASSGN}_r) In this case, \( S \) is \( x := G \); \( P \) is \( x = x' \), \( Q \) is \( \{ x = G/(x \mapsto x') \} \) and \( \varepsilon = \text{modifies} x \), where \( x \notin \text{FV}(G) \). By the program semantics Fig. 4.1, \( \Gamma' = \Gamma, (\sigma', h') = \)
\[ (\sigma[x \mapsto E[\Gamma \vdash G : T]](\sigma)), h), \] which entails \( Q \).

For the frame condition, this statement only updates variable \( x \). Therefore, \( \varepsilon \) is modifies \( x \) is correct.

5. (\textit{UPD}_r) In this case, \( S \) is \( x.f := G \); \( P \) is \( x \neq \text{null} \), \( Q \) is \( x.f = G \) and \( \varepsilon \) is modifies region\( \{x.f\} \). By the program semantics Fig. 4.1, \( \Gamma' = \Gamma \), \( (\sigma', h') = (\sigma, h([E[\Gamma \vdash x : T](\sigma), f]) \mapsto E[\Gamma \vdash G : T'](\sigma))]}, which entails \( Q \).

For the frame condition, this statement changes the singleton heap location \( (\sigma(x), f) \). Therefore, \( \varepsilon \) is modifies region\( \{x.f\} \) is correct.

6. (\textit{ACC}_r) In this case, \( S \) is \( x := x'.f \); \( P \) is \( x' \neq \text{null} \), \( Q \) is \( x = x_1.f \), and \( \varepsilon \) is modifies \( x \), where \( x \neq x' \). By the program semantics Fig. 4.1, \( \Gamma' = \Gamma \), \( (\sigma', h') = (\sigma[x \mapsto h([E[\Gamma \vdash x' : T](\sigma), f])], h) \), which entails \( Q \).

For the frame condition, this statement only updates variable \( x \). Therefore, \( \varepsilon = \text{modifies} \) \( x \) is correct.

The inductive hypothesis is that for all substatements \( S_i \), if \( \vdash^{\Gamma_i} \{P_i\} S_i \{Q_i\}[\varepsilon_i] \), and \( \sigma_i, h_i \vdash^{\Gamma_i} P_i \), then \( \sigma'_i, h'_i \vdash^{\Gamma'_i} Q_i \).

1. (\textit{IF}_r) In this case, \( S \) is \textbf{if} \( E \) \textbf{then} \( \{S_1\} \) \textbf{else} \( \{S_2\} \). There are two cases:

   • \( E \). By the inductive hypothesis, it must be true that \( \sigma, h \vdash^{\Gamma} P \&\& E \), (\( \sigma', h' \) = \( \mathcal{M}_S[\Gamma \vdash S_1 : \text{ok}(\Gamma_1)](\sigma, h) \)), which entails \( Q \). And the frame condition is correct.

   • \( \neg E \). By the inductive hypothesis, it must be true that \( \sigma, h \vdash^{\Gamma} P \&\& \neg E \), (\( \sigma'', h'' \) = \( \mathcal{M}_S[\Gamma \vdash S_2 : \text{ok}(\Gamma_2)](\sigma, h) \)), which entails \( Q \). And the frame condition is correct.

By the program semantics Fig. 4.1, if \( P \) holds in the prestate, no matter which path the program takes, if the program terminates, \( Q \) holds.
2. (WHILE.) In this case, $S$ is while $E$ do $\{S\}$. $P = I$, $Q = I \&\& E$ and the frame conditions is $\varepsilon$. The premise is $\vdash^Γ r \{I \&\& E\} \{S\} \{I\}[ε].$

By the program semantics Fig. 4.1, let $g$ be a recursive point function, such that

$$g = \lambda s.\text{if } E \Gamma \neq \text{bool}(\sigma) \neq 0 \text{then let } s' = MS[Γ \vdash S : ok(Γ')](σ, H)\text{ing } s' \text{else } s$$

By definition, $\text{fix}$ is a fixed point function, so $\text{fix}(g) = g$. Then the following proves $\text{fix}(g)(σ, h) \vdash^{Γ'} I$ by fixed-point induction.

Base Case: $\bot \vdash^{Γ'} I$ holds vacuously. It requires to prove all members in $\bot$ implies $I$, but there is nothing in $\bot$. Hence it is vacuously true.

Inductive Case: Let $(σ'', h'') \vdash^{Γ'} I$ hold for an arbitrary iteration of $g$, and $ε$ is the frame condition. Then the proof obligation is to show that $\text{fix}(g)(σ'', h'') \vdash^{Γ'} I$ holds, and the changed locations on the heap is $ε$.

There are two cases:

- $E$. By the semantics, $\text{fix}(g)(σ'', h'') = g(MS[Γ \vdash S : ok(Γ')](σ'', h'')).$ By the inductive hypothesis, $g(MS[Γ \vdash S : ok(Γ')](σ'', h'')) \vdash^{Γ'} I$ holds. Hence $\text{fix}(g)(σ'', h'') \vdash^{Γ'} I$ holds. For the frame condition, since the fixed point function always returns the same function $g$, which is framed by $ε$ by the induction hypothesis, therefore $ε$ is the frame condition for an arbitrary iteration.

- $¬E$. By the semantics, $\text{fix}(g)(σ'', h'') = (σ'', h'')$. Therefore, by the inductive hypothesis, $\text{fix}(g)(σ'', h'') \vdash^{Γ'} I$ holds. For the frame condition, since the state does not change, the frame is $\text{region}\{\}$, which is the subset of $ε$.

Now it has been shown that if the loop exits, which means that $¬E$ holds, the loop invariant $I$ holds. Therefore, $Q$ holds and $ε$ is its frame condition.
3. \((SEQ1_r)\) In this case, \(S\) is \(S_1S_2\), where \(S_1 \neq \text{var } x : T_i\). Let \((\sigma, h)\) be a state, such that \((\sigma, h) \vdash^\Gamma P\). By the inductive hypothesis for \(S_1\) and \(S_2\), \((\sigma'', h'') \vdash^\Gamma P_1\). By the second premise and the semantics, \((\sigma', h') = MS[\Gamma \vdash S_1 : ok(\Gamma')]\(\sigma, h\), \((\sigma'', h'') \vdash^\Gamma P_1\). Hence \((\sigma', h') \vdash^r P'\).

For the frame condition, the proof obligation is to show \((\sigma, h) \rightarrow (\sigma', h') \vdash^\Gamma (\varepsilon_1, \varepsilon_2, \text{fresh}(RE))\), which is proved by Lemma 7. It is instantiated with \(G_0 := \Gamma, G_1 := \Gamma''\), \(G_2 := \Gamma', RE_1 := RE_2, (\sigma_0, h_0) := (\sigma, h), (\sigma_1, h_1) := (\sigma'', h''), (\sigma_2, h_2) := (\sigma', h')\) and \(\varepsilon_1 := (\varepsilon_1, \text{fresh}(RE))\). The following conditions, which are required by the Lemma, are satisfied:

- \((\sigma, h) \vdash^\Gamma P\) and \((\sigma'', h'') \vdash^{\Gamma''} P_1\) from the above;
- \((\sigma, h) \rightarrow (\sigma'', h'') \vdash^\Gamma (\varepsilon_1, \text{fresh}(RE))\) by the inductive hypothesis;
- \((\sigma'', h'') \rightarrow (\sigma', h') \vdash^{r''} (\varepsilon_2, \text{modifies } RE_1)\) by the inductive hypothesis.
- \(\varepsilon_2\) is \(P/\varepsilon_1\)-immune by the given side condition;
- for all \(\text{fresh}(RE) \in \varepsilon_1 :: RE\) is \(P/(\varepsilon_2, \text{modifies } RE_1)\)-immune by the given side condition.
- \(\varepsilon[\Gamma'' \vdash RE_1 : \text{region}]\(\sigma'\) \cap \sigma(\text{alloc}) = \emptyset\), \(RE\) are freshly allocated regions by \(S_1\), i.e., \(\varepsilon[\Gamma'' \vdash RE_1 : \text{region}]\(\sigma'\) \subseteq (\sigma''(\text{alloc}) - \sigma(\text{alloc}))\).

4. \((SEQ2_r)\) In this case, \(S\) is \(\text{var } x : T; S_2\). This case follows the inductive hypothesis and the program semantics.

5. \((SUBEFF_r)\) By the inductive hypothesis, \(\vdash^r \{P\}S\{Q\}[\varepsilon]\). Hence when applying the frame condition \(\varepsilon' \supseteq \varepsilon\), the locations that may be changed are also contained in \(\varepsilon'\). Therefore \(\varepsilon'\) is a correct frame.

6. \((FRM_r)\) In this case, by the inductive hypothesis, it must be true that \(\vdash^r \{P\}S\{Q\}[\varepsilon]\). And
by the assumption, it must be true that \( P \vdash^\Gamma \delta \) from \( Q \) and \( P \&\& R \Rightarrow \delta / \varepsilon \). The proof obligation is to show \( \vdash^\Gamma \{ P \&\& R \} S\{ Q \&\& R \}[\varepsilon] \). Because \( P \&\& R \) implies \( P \), Thus, it must be true that \( \vdash^\Gamma \{ P \&\& R \} S\{ Q \&\& R \}[\varepsilon] \). Let \(( \sigma', H') = MS[\Gamma \vdash S: ok(\Gamma')]\( (\sigma, H) \). The proof obligation is to show that \( (\sigma', H') \vdash^{\Gamma'} R \). By \( (\sigma, H) \vdash^{\Gamma} P \&\& R \) and the side condition \( P \&\& R \Rightarrow \delta / \varepsilon \), it must be true that \( (\sigma, H) \vdash^{\Gamma} \delta / \varepsilon \). As the write effect is \( (\sigma, H) \rightarrow (\sigma', H') \vdash^{\Gamma} \varepsilon \), it must be true that \( (\sigma, H) \equiv^{\delta} (\sigma', H') \). By the definition of framing (Def. 5) and \( (\sigma, H) \vdash^{\Gamma} P \&\& R \), it must be true that \( (\sigma', H') \vdash^{\Gamma'} R \).

7. \((\text{CONSEQ}_r)\) In this case, by the inductive hypothesis, it must be true that \( \vdash^\Gamma \{ P' \} S\{ Q' \}[\varepsilon] \). By the premise, \( P \Rightarrow P' \) and \( Q' \Rightarrow Q \). Hence \( \vdash^\Gamma \{ P \} S\{ Q \}[\varepsilon] \) is valid.

\[\square\]
APPENDIX C: PROOF OF THEOREM 3
Theorem 3: Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement. Let $P_1$ and $P_2$ be assertions. Let $\varepsilon$ be effects. Then

$$\vdash^\Gamma_r \{P_1\} S \{P_2\} \{\varepsilon\} \iff \vdash^\Gamma_u [\text{reads } r] \{P_1\} S \{P_2\} \{\varepsilon\}$$

where $P_1 \Rightarrow r = \text{alloc}$ and modifies $r \notin \varepsilon$

Proof. The left hand side implies the right hand side is firstly proved; i.e., that if there is a proof in FRL, then the encoded proof is in UFRL. The proof is done by the induction on FRL derivation and by cases on the last rule used. There are 6 base cases.

1. SKIP: In this case, suppose that the FRL proof consists of the axiom $\text{SKIP}_r$, which is $\vdash^\Gamma_r \{\text{true}\} \text{skip}; \{\text{true}\}[\emptyset]$. Then, the proof obligation is to show that the judgment $\vdash^\Gamma_u [\text{reads } r] \{\text{true}\} \text{skip}; \{\text{true}\}[\emptyset]$, where $\text{true} \Rightarrow r = \text{alloc}$, is derivable in UFRL.

   It can be done by using the axiom $\text{SKIP}_u$ and the structural rule $\text{SubEff}_u$.

2. VAR: In this case, suppose that the FRL proof consists of the axiom $\text{VAR}_r$, which is $\vdash^\Gamma_r \{\text{true}\} \text{var } x : T; \{x = \text{default}(T)\}[\emptyset]$. Then, the proof obligation is to show that the judgment $\vdash^\Gamma_u [\text{reads } r] \{\text{true}\} \text{var } x : T; \{x = \text{default}(T)\}[\emptyset]$, where $\text{true} \Rightarrow r = \text{alloc}$, is derived in UFRL. It can be done by using the axiom $\text{VAR}_u$ and the structural rule $\text{SubEff}_u$.

3. ALLOC: In this case, suppose that the FRL proof consists of the axiom $\text{ALLOC}_r$, which is $\vdash^\Gamma_r \{\text{true}\} x := \text{new } T; \{\text{new}(T, x)\} \{\text{modifies } x, \text{alloc}, \text{fresh}(\text{region}\{x.\})\}$. Then, the proof obligation is to show the following judgment is derivable in UFRL.

   $$\vdash^\Gamma_u [\text{reads } r] \{\text{true}\} x := \text{new } T; \{\text{new}(T, x) \{\text{modifies } x, \text{alloc}, \text{fresh}(\text{region}\{x.\})\}\}$$

   where $\text{true} \Rightarrow r = \text{alloc}$

   It can be done by using the axiom $\text{ALLOC}_u$ and the structural rule $\text{SubEff}_u$.

4. UPD: Suppose that the FRL proof consists of the axiom $\text{UPD}_r$, which is $\vdash^\Gamma_r \{x \neq \text{null}\} x.f := G; \{x.f = G\} \{\text{modifies } \text{region}\{x.f\}\}$, where $x \notin \text{FV}(G)$. Then, the
proof obligation is to show that the following judgment is derivable.

\[ \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \]

where \( x \notin \text{FV}(G) \) and \( x \neq null \Rightarrow r = \text{alloc} \)

It can be done by using the axiom \text{UPD}_u \text{ and the structural rule SubEff}_u.  

5. ASGN: Suppose that the FRL proof consists of the axiom ASGN, which is

\[ \vdash \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \]

where \( x \notin \text{FV}(G) \) and \( true \Rightarrow r = \text{alloc} \)

It can be done by using the axiom ASGN and the structural rule SubEff.  

6. ACC: Suppose that the FRL proof consists of the axiom ACC, which is

\[ \vdash \Gamma \vdash u_r \text{reads } \Gamma \vdash u_r \text{reads } \]

where \( x \neq x' \) and \( x' \neq null \Rightarrow r = \text{alloc} \)

It can be done by using the axiom ACC and the structural rule SubEff.  

The inductive hypothesis is that for all substatements \( S_i \), it is true that

\[ \vdash \Gamma \vdash P \vdash S_i \{Q_i \} [\varepsilon_i] \text{ iff } \vdash \Gamma \vdash \text{reads } \Gamma \vdash S_i \{Q_i \} [\varepsilon_i], \text{ where } P_i \Rightarrow r = \text{alloc} \text{ and } \text{modifies } r \notin \varepsilon. \]

1. \text{IF:} In this case, suppose that the FRL proof consists of the rule \text{IF}, which is

\[ \vdash \Gamma \vdash \{P \& \; E\} \{P'\} [\varepsilon] \quad \vdash \Gamma \vdash \{P \& \; \neg E\} \{P'\} [\varepsilon] \]

\[ \vdash \Gamma \vdash \{P \text{if } E \text{ then } S_1 \} \{P'\} [\varepsilon] \]

167
Then the proof obligation is to show that the following is derivable.

\[
\Gamma \vdash u \text{reads } r \downarrow (P \land E) S_1 \{P^r\}[\varepsilon]\quad \Gamma \vdash u \text{reads } r \downarrow (P \land \neg E) S_2 \{P^r\}[\varepsilon]
\]

\[
\Gamma \vdash u \text{reads } r \downarrow \{P\} \text{if } E \text{ then } \{S_1\} \text{ else } \{S_2\}\{P^r\}[\varepsilon]
\]

where \( P \Rightarrow r = \text{alloc} \)

By the inductive hypothesis, the two premises are assumed. Using the rule \( IF_u \), the following is derived

\[
\Gamma \vdash u \text{reads } r \downarrow (P \land E) S_1\{Q\}[\varepsilon]\quad \Gamma \vdash u \text{reads } r \downarrow (P \land \neg E) S_2\{Q\}[\varepsilon]
\]

\[
\Gamma \vdash u \text{reads } r \downarrow, \delta_E \downarrow \{P\} \text{if } E \{S_1\} \text{ else } \{S_2\}\{Q\}[\varepsilon]
\]

where \( \delta_E = \text{efs}(E), P \Rightarrow r = \text{alloc} \) and \( \text{modifies } r \notin \varepsilon \)

Then, using the rule \( \text{SubEff}_u \), the conclusion of is derived.

2. \( \text{WHILE} \): Suppose that the FRL proof consists of the rule \( \text{WHILE}_r \), which is

\[
\Gamma \vdash \{P \land E\} S\{P\}[\varepsilon, \text{fresh(RE)}]
\]

\[
\Gamma \vdash \{P \land r = \text{alloc}\} \text{while } E \{S\}\{P \land \neg E\}[\varepsilon]
\]

where \( P \Rightarrow \text{RE} ! ! r, \varepsilon \) is fresh-free, \( \varepsilon \) is \( P/\varepsilon \)-immune and \( \text{modifies } r \notin \varepsilon \)

where \( r \) snapshots the domain of the heap in the pre-state. Then the proof obligation is to show the following is derivable.

\[
\Gamma \vdash u \text{reads } r \downarrow \{P \land E\} S\{P\}[\varepsilon, \text{fresh(RE)}]
\]

\[
\Gamma \vdash u \text{reads } r \downarrow \{P \land r = \text{alloc}\} \text{while } E \{S\}\{P \land \neg E\}[\varepsilon]
\]

where \( P \Rightarrow r = \text{alloc} \) and \( \text{modifies } r \notin \varepsilon \)

By the inductive hypothesis, the premises is assumed. To use the rule \( \text{WHILE}_u \), its side conditions have to be true. In addition to the side condition that is given by the assumption, it needs to prove that \( \text{reads } r \) is \( P/\varepsilon \)-immune, which is true because \( \text{modifies } r \notin \varepsilon \).
Then, using the rule $WHILE_u$, the following is derived

$$\Gamma \vdash_{u} \text{reads} \ r \{P \ & \ E\} \ S\{P\} \ [\varepsilon, \ fresh(RE)]$$

$$\vdash_{u} \text{reads} \ r\downarrow, \delta_E\{P \ & \ & r = alloc\} \ while \ E\{S\} \ {P \ & \ & -E} \ [\varepsilon]$$

where $P \Rightarrow r = alloc, P \Rightarrow RE \ ! \ ! r, \varepsilon$ is fresh-free, $\varepsilon$ is $P/\varepsilon$-immune

and modifies $r \notin \varepsilon$,

where $\delta_E = efs(E)$. Then, the conclusion is derived by using the structural rule $SubEff_u$.

3. $SEQ1$: Suppose that the FRL proof consists of the rule $SEQ1_r$, which is

$$\vdash_{r} \{P\} \ S_1\{P_1\}[\varepsilon_1, fresh(RE)] \quad \vdash_{r} \{P_1\} \ S_2\{P'\}[\varepsilon_2, RE]$$

$$\vdash_{r} \{P\} \ S_1S_2\{P'\}[\varepsilon_1, \varepsilon_2, fresh(RE)]$$

where $S_1 \neq \text{var} \ x : T;\varepsilon_1$ is fresh-free, $\varepsilon_2$ is $P/\varepsilon_1$-immune, and

$RE$ is $P_1/(\text{modifies} \ RE, \varepsilon_2)$-immune

Then the proof obligation is to show that the following is derivable.

$$\vdash_{u} \text{reads} \ r\downarrow\{P\} \ S_1\{P_1\}[\varepsilon_1, fresh(RE)] \quad \vdash_{u} \text{reads} \ r\downarrow\{P_1\} \ S_2\{P'\}[\varepsilon_2, RE]$$

$$\vdash_{u} \text{reads} \ r\downarrow\{P\} \ S_1S_2\{P'\}[\varepsilon_1, \varepsilon_2, fresh(RE)]$$

where $P \Rightarrow r = alloc$ and modifies $r \notin (\varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the two premises are assumed. To use the rule $SEQ1_u$, check the side condition reads $r$ is $P/\varepsilon_1$-immune, which is true because modifies $r \notin (\varepsilon_1, \varepsilon_2)$ by the inductive hypothesis. Then, the conclusion is derived by using the rule $SEQ1_u$.

4. $SEQ2$: Suppose that the FRL proof consists of the rule $SEQ1_r$:

$$\vdash_{r} \{P \ & \ & x = default(T)\} : S\{Q\}[modifies \ x\varepsilon]$$

$$\vdash_{r} \{P\} \ var \ x : T; S\{P'\}[\varepsilon]$$
Then, the proof obligation is to show that the following is derivable.

\[\Gamma \text{reads } r \downarrow \{P \&\& x = \text{default}(T)\} S \{Q\}[\text{modifies } x, \varepsilon] \]

\[\Gamma \text{reads } r \downarrow \{P\} \text{var } x : T; S \{P'\}[\varepsilon] \]

where \( P \Rightarrow r = \text{alloc} \) and \( r \neq x; \)

By the inductive hypothesis, the assumption is assumed. Then the conclusion is derived by using the rule \( SEQ2_u. \)

5. \( FRM: \) Suppose that the FRL proof consists of the rule \( FRM_r: \)

\[\Gamma \text{reads } r \downarrow \{P\} S \{P'\}[\varepsilon] \quad P \vdash \Gamma \delta \text{frm } Q \]

\[\vdash \Gamma \{P \&\& Q\} S \{P' \&\& Q\}[\varepsilon] \quad \text{where } P \&\& Q \Rightarrow \delta/\varepsilon \]

Then the proof obligation is to show that the following is derivable in UFRL.

\[\Gamma \text{reads } r \downarrow \{P\} S \{P'\}[\varepsilon] \quad P \vdash \Gamma \delta \text{frm } Q \]

\[\Gamma \text{reads } r \downarrow \{P \&\& Q\} S \{P' \&\& Q\}[\varepsilon] \quad \text{where } P \&\& Q \Rightarrow \delta/\varepsilon, P \Rightarrow r = \text{alloc} \text{ and } \text{reads } r \notin \varepsilon \]

By inductive hypothesis, the two premises of the above equations are assumed. Then, the conclusion can be derived by using the rule \( FRM_u. \)

6. \( SubEff: \) Suppose that the FRL proof consists of the rule \( SubEff_r: \)

\[\Gamma \text{reads } r \downarrow \{P\} S \{P'\}[\varepsilon] \quad P \vdash \Gamma \varepsilon \leq \varepsilon' \]

\[\vdash \Gamma \{P\} S \{P'\}[\varepsilon'] \]

Then the proof obligation is to show that the following is derivable in UFRL.

\[\Gamma \text{reads } r \downarrow \{P\} S \{P'\}[\varepsilon] \quad P \vdash \Gamma \varepsilon \leq \varepsilon' \]

\[\vdash \Gamma \text{reads } r \downarrow \{P\} S \{P'\}[\varepsilon'] \quad \text{where } P \Rightarrow r = \text{alloc} \text{ and } \text{reads } r \notin \varepsilon \]
By the inductive hypothesis, the two premises are assumed. Then, the conclusion can be derived by using the rule \( \text{SubEff}_u \), because the side condition \( \text{regRW}(\varepsilon, \text{reads } r) \leq \text{regRW}(\varepsilon', \text{reads } r) \) is true.

7. **CONSEQ**: suppose that the FRL proof consists of the rule **CONSEQ**: $\vdash r \{P_1\} S \{P'_1\}[\varepsilon] \quad P'_1 \Rightarrow P_2$

Then, the proof obligation is to show that the following is derivable.

$\vdash u \{P_2\} S\{P'_2\}[\varepsilon]$

where $P \Rightarrow r = \text{alloc}$ and \( \text{reads } r \notin \varepsilon \)

By the inductive hypothesis, the two premises are assumed. Then, its conclusion can be derived by using the rule **CONSEQ\(_u\)**.

8. **ConEff**: suppose that the FRL proof consists of the rule **ConEff**: $\vdash r \{P \& \& E\} S \{P'\}[\varepsilon_1] \quad \vdash \{P \& \& \neg E\} S \{P'\}[\varepsilon_2]$

Then, the proof obligation is to show that the following is derivable in UFRL.

$\vdash u \{\text{reads } r\downarrow\} \{P \& \& E\} S \{P'\}[\varepsilon_1] \quad \vdash \{\text{reads } r\downarrow\} \{P \& \& \neg E\} S \{P'\}[\varepsilon_2]$

where $P \Rightarrow r = \text{alloc}$ and \( \text{reads } r \notin \varepsilon \)

By the inductive hypothesis, the two premises are assumed. Then, its conclusion can be derived by using the rule **ConEff\(_u\)**.

9. **ConMask1**: Suppose that the FRL proof consists of the rule **ConMask1**: $\vdash r \{P, E \? \varepsilon_1 : \varepsilon_2\} S \{P'\}[\varepsilon, \varepsilon_1]$

where $P \Rightarrow E$
Then, the proof obligation is to show that the following is derivable in UFRL.

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]
\]

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon, \varepsilon_1]
\]

where \( P \Rightarrow E, P \Rightarrow r = \text{alloc} \) and \( \text{reads } r \notin \varepsilon \)

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule \( \text{ConMask1}_u \)

10. \( \text{ConMask2} \): suppose that the FRL proof consists of the rule \( \text{ConMask2}_r \):

\[
\Gamma \vdash \{P\} \mathcal{S} \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]
\]

\[
\Gamma \vdash \{P\} \mathcal{S} \{P'\}[\varepsilon, \varepsilon_2]
\]

where \( P \Rightarrow \neg E \)

Then, the proof obligation is to show that the following is derivable in UFRL.

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]
\]

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon, \varepsilon_2]
\]

where \( P \Rightarrow \neg E, P \Rightarrow r = \text{alloc} \) and \( \text{reads } r \notin \varepsilon \)

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule \( \text{ConMask2}_u \).

11. \( \text{PostToFr} \): Suppose that the FRL proof consists of the rule \( \text{PostToFr}_r \):

\[
\Gamma \vdash \{P\} \mathcal{S} \{P'\}[\varepsilon]
\]

\[
\Gamma \vdash \{P\} \mathcal{S} \{P'\}[\varepsilon, E ? \text{fresh}(RE_1) : \text{fresh}(RE_2)]
\]

where \( P \Rightarrow (E \&\& RE_1 \text{ !! alloc}) \) and \( P \Rightarrow (\neg E \&\& RE_2 \text{ !! alloc}) \)

Then, the proof obligation is to show the following is derivable in UFRL.

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon]
\]

\[
\Gamma \vdash \text{reads } r \downarrow \{P\} \mathcal{S} \{P'\}[\varepsilon, E ? \text{fresh}(RE_1) : \text{fresh}(RE_2)]
\]

where \( P \Rightarrow r = \text{alloc}, \text{reads } r \notin \varepsilon, P \Rightarrow (E \&\& RE_1 \text{ !! alloc}) \)

and \( P \Rightarrow (\neg E \&\& RE_2 \text{ !! alloc}) \)
By the inductive hypothesis, the premise is assumed. Then, the conclusion is derived by using the rule \textit{PostToFr\textsubscript{u}}.

12. \textit{FrToPost}: Suppose that the FRL proof consists of the rule Fr\textsubscript{ToPost}:

\[
\Gamma \vdash \{P\} S \{P'\} [\varepsilon, E : \text{fresh}(RE_1) : \text{fresh}(RE_2)] \\
\Gamma \vdash \{P\} S \{P' \& \& (b \Rightarrow RE_1 \& \& r) \& \& (\neg b \Rightarrow RE_2 \& \& r)\} \\
[\varepsilon, E : \text{fresh}(RE_1) : \text{fresh}(RE_2)]
\]

where \( P \Rightarrow r = \text{alloc}, P \Rightarrow b = E, \text{modifies}\ b \notin \varepsilon\) and \text{modifies}\ r \notin \varepsilon

Then, the proof obligation is to show that the following is derivable

\[
\Gamma \vdash \{\text{reads} \ r\downarrow\} \{P\} S \{P'\} [\varepsilon, E : \text{fresh}(RE_1) : \text{fresh}(RE_2)] \\
[\text{reads} \ r\downarrow] \\
\Gamma \vdash \{P\} S \{P' \& \& (b \Rightarrow RE_1 \& \& r) \& \& (\neg b \Rightarrow RE_2 \& \& r)\} \\
[\varepsilon, E : \text{fresh}(RE_1) : \text{fresh}(RE_2)]
\]

where \( P \Rightarrow r = \text{alloc}, P \Rightarrow b = E, \text{modifies}\ b \notin \varepsilon\) and \text{modifies}\ r \notin \varepsilon

By the inductive hypothesis, the premise is assumed. Then, the conclusion can be derived by using the rule Fr\textsubscript{ToPost\textsubscript{u}}.

13. \textit{VarMask1}: Suppose that the FRL proof consists of the rule Var\textsubscript{Mask1}:

\[
\Gamma \vdash \{P\} S \{P'\} [E : \text{modifies} x, \varepsilon_1] : \varepsilon_2 \\
\Gamma \vdash \{P\} S \{P'\} [E : \varepsilon_1] : \varepsilon_2
\]

where \( P \Rightarrow b = E, P \Rightarrow b, P \parallel P' \Rightarrow x = y, P \& \& b \Rightarrow \text{reads} y/(x, \varepsilon)\)

and \text{modifies}\ b \notin (\varepsilon_1, \varepsilon_2)

173
Then, the proof obligation is to show that the following is derivable in UFRL.

\[
\Gamma \vdash \{ \text{reads } r \} \{ P \} S \{ P' \} \{ E ? (\text{modifies } x, \varepsilon_1) : \varepsilon_2 \}
\]

where \( P \Rightarrow r = \text{alloc}, \ P \Rightarrow b = E, \ P \Rightarrow b, \ P' \Rightarrow x = y, \)

\( P \&\& b \Rightarrow \text{reads } y/(x, \varepsilon) \), modifies \( r \notin (\varepsilon_1, \varepsilon_2) \) and modifies \( b \notin (\varepsilon_1, \varepsilon_2) \)

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule \( \text{VarMask1}_u \).

14. \( \text{VarMask2} \): Suppose that the FRL proof consists of the rule \( \text{VarMask2}_r \):

\[
\Gamma \vdash \{ P \} S \{ P' \} \{ E ? \varepsilon_1 : (\text{modifies } x, \varepsilon_2) \}
\]

where \( P \Rightarrow b = E, P \Rightarrow \neg b, P' \Rightarrow x = y, P \&\& \neg b \Rightarrow \text{reads } y/(x, \varepsilon) \)

and modifies \( b \notin (\varepsilon_1, \varepsilon_2) \)

Then, the proof obligation is to show that the following is derivable in UFRL.

\[
\Gamma \vdash \{ \text{reads } r \} \{ P \} S \{ P' \} \{ E ? \varepsilon_1 : (\text{modifies } x, \varepsilon_2) \}
\]

where \( P \Rightarrow r = \text{alloc}, \ P \Rightarrow b = E, \ P \Rightarrow \neg b, \ P' \Rightarrow x = y, \)

\( P \&\& \neg b \Rightarrow \text{reads } y/(x, \varepsilon), \text{reads } r \notin (\varepsilon_1, \varepsilon_2) \) and modifies \( b \notin (\varepsilon_1, \varepsilon_2) \)

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule \( \text{VarMask2}_u \).
15. **FieldMask1**: Suppose that the FRL proof consists of the rule $\text{FieldMask1}_r$:

\[
\vdash^\Gamma_r \{P\} S\{P'\}[\varepsilon, E \ ? \ (\text{modifies region}\{x.f\}, \varepsilon_1) : \varepsilon_2] \\
\vdash^\Gamma_r \{P\} S\{P'\}[\varepsilon, E \ ? \ \varepsilon_1 : \varepsilon_2]
\]

where $P \Rightarrow b = E, P \Rightarrow b, P \parallel P' \Rightarrow x.f = y,$

$P' \&\& b \Rightarrow \text{reads } x/\text{modifies } \varepsilon,$

$P' \&\& b \Rightarrow \text{reads } y/\text{modifies } \varepsilon$ and $\text{modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)

Then, the proof obligation is to show that the following is derivable in UFRL.

\[
\vdash^\Gamma_u [\text{reads } r\!]\{P\} S\{P'\}[\varepsilon, E \ ? \ (\text{modifies region}\{x.f\}, \varepsilon_1) : \varepsilon_2] \\
\vdash^\Gamma_u [\text{reads } r\!]\{P\} S\{P'\}[\varepsilon, E \ ? \ \varepsilon_1 : \varepsilon_2]
\]

where $P \Rightarrow r = \text{alloc}, P \Rightarrow b = E, P \Rightarrow b, P \parallel P' \Rightarrow x.f = y,$

$P' \&\& b \Rightarrow \text{reads } x/\text{modifies } \varepsilon, P' \&\& b \Rightarrow \text{reads } y/\text{modifies } \varepsilon$

\text{modifies } r \notin (\varepsilon, \varepsilon_1, \varepsilon_2) \text{ and } \text{modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)

(C.1)

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule $\text{FieldMask1}_u$.

16. **FieldMask2**: Suppose that the FRL proof consists of the rule $\text{FieldMask2}_r$:

\[
\vdash^\Gamma_r \{P\} S\{P'\}[\varepsilon, E \ ? \ \varepsilon_1 : (\text{modifies region}\{x.f\}, \varepsilon_2)] \\
\vdash^\Gamma_r \{P\} S\{P'\}[\varepsilon, E \ ? \ \varepsilon_1 : \varepsilon_2]
\]

where $P \Rightarrow b = E, P \Rightarrow \neg b, P \parallel P' \Rightarrow x.f = y,$

$P' \&\& \neg b \Rightarrow \text{reads } x/\text{modifies } \varepsilon,$

$P' \&\& \neg b \Rightarrow \text{reads } y/\text{modifies } \varepsilon$ and $\text{modifies } b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$
Then the proof obligation is to show that the following is derivable in UFRL.

$$\vdash^\Gamma \mathit{reads} \{P\} S \{P'\}[\varepsilon, E ? \varepsilon_1 : \mathit{modifies} \mathit{region}\{x.f\}, \varepsilon_2]$$

$$\vdash^\Gamma \mathit{reads} \{P\} S \{P'\}[\varepsilon, E ? \varepsilon_1 : \varepsilon_2]$$

where $P \Rightarrow r = \mathit{alloc}$, $P \Rightarrow b = E$, $P \Rightarrow \neg b$, $P \parallel P' \Rightarrow x.f = y,$

$P' \& \& \neg b \Rightarrow \mathit{reads} x/\mathit{modifies} \varepsilon, P' \& \& \neg b \Rightarrow \mathit{reads} y/\mathit{modifies} \varepsilon$

modifies $r \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$ and modifies $b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule $\mathit{FieldMask2}_u$.

Next, it is proved from the right side of the left side. It means that if there is a proof $\vdash^\Gamma_u \mathit{reads} \{P_1\} S \{P_2\}[\varepsilon]$ in UFRL, where $P_1 \Rightarrow r = \mathit{alloc}$ and modifies $r \notin \varepsilon$, then there is a proof $\vdash^\Gamma_r \{P_1\} S \{P_2\}[\varepsilon]$ in FRL. The proof is done by the induction on the UFRL derivation and by cases on the last rule used, and is omitted. 

$\square$
APPENDIX D: PROOF OF THEOREM 5
Theorem 5: An assertion in SL is supported if and only if it has semantic footprint.

Proof. Let $\Gamma$ be a well-formed type environment. Let $(\sigma, h)$ be a state and $a$ be an assertion in SL, such that $\sigma, h \vdash^\Gamma_a$. Let $\mathcal{H} \overset{\text{def}}{=} \{h'|h' \subseteq h \land \sigma, h' \vdash^\Gamma_a\}$. Any subset of $\mathcal{H}$ defines a partial order, i.e., $\mathcal{H}_1 \subseteq \mathcal{H}_2$ iff $\mathcal{H}_1, \mathcal{H}_2 \in \mathcal{P}(\mathcal{H})$ and $\mathcal{H}_1 \subseteq \mathcal{H}_2$. Define $(\downarrow \mathcal{H}_i) \overset{\text{def}}{=} \{\mathcal{H}' | \mathcal{H}' \subseteq \mathcal{H}_i \land \mathcal{H}' \in \mathcal{P}(\mathcal{H})\}$, where $\mathcal{H}_i \in \mathcal{P}(\mathcal{H})$. For any pair of $\mathcal{H}_1$ and $\mathcal{H}_2$, $(\downarrow \mathcal{H}_1) \cap (\downarrow \mathcal{H}_2)$ is a partial order. Let $\bigcap_{\mathcal{H}}$ define the greatest lower bound of any subset of the intersection. Let $\bigcap_{\mathcal{H}}$ define the greatest lower bound of any subset of the intersection. If it has a greatest lower bound of $\mathcal{H}_1$ and $\mathcal{H}_2$, then

$$\mathcal{H}_a \leq (\mathcal{H}_1 \bigcap_{\mathcal{H}} \mathcal{H}_2) \text{ iff } (\mathcal{H}_a \leq \mathcal{H}_1 \text{ and } \mathcal{H}_a \leq \mathcal{H}_2).$$

Thus, $\mathcal{H}_a$ is the least subheap for an assertion $a$ in Definition 14. Next, it is shown that $\text{dom}(\mathcal{H}_a)$ is $a$’s semantic footprint. Let $\mathcal{R} \overset{\text{def}}{=} \{r | \sigma, r \vdash^\Gamma_a\}$. Any subset of $\mathcal{R}$ defines a partial order in a way similar to $\mathcal{H}$. Let $\bigcap_{\mathcal{R}}$ define the greatest lower bound of any subset of $\mathcal{R}$. Let $\text{DOM}$ be a functor from $\mathcal{P}(\mathcal{H})$ to $\mathcal{P}(\mathcal{R})$, such that $\text{DOM}(|h_1, h_2, \ldots, h_n|) = \{\text{dom}(h_1), \text{dom}(h_2), \ldots, \text{dom}(h_n)\}$. If $(\mathcal{H}_1) \leq (\mathcal{H}_2)$, then $\text{DOM}(\mathcal{H}_1) \leq \text{DOM}(\mathcal{H}_2)$. Thus $\mathcal{H}_p \leq (\mathcal{H}_1 \bigcap_{\mathcal{H}} \mathcal{H}_2) \text{ iff } \text{DOM}(\mathcal{H}_p) \leq \text{DOM}(\mathcal{H}_1) \bigcap_{\mathcal{R}} \text{DOM}(\mathcal{H}_2).$ □
**Theorem 7:** Let $\Gamma$ be a well-formed type environment. Let $a$ be an assertion in SSL. Then $\sigma, h \models^{\Gamma}_s a$ iff $\sigma, h \models^{\Gamma}_u \text{TR}[[a]]$.

**Proof.** The proof is by the induction on the assertion’s structure. Here it is only shown the most interesting case that encodes the separating conjunction. Other proofs are found in the KIV project [6]. It is an inductive case when $a$ is of the form $a_1 \ast a_2$. The inductive hypothesis is that for all subassertions $a_i$, $\sigma, h \models^{\Gamma}_s a_i$ iff $\sigma, h \models^{\Gamma}_u \text{TR}[[a_i]]$.

From the left side to the right side is firstly proved. Assume $\sigma, h \models^{\Gamma}_s a_1 \ast a_2$. The proof obligation is to show that $\sigma, h \models^{\Gamma}_u \text{TR}[[a_1]] \land \text{TR}[[a_2]] \land \land (fpt_s(a_1) \cdot ! fpt_s(a_2))$.

\[
\sigma, h \models^{\Gamma}_s a_1 \ast a_2
\]

iff \(\langle\text{by the semantics of separation logic (Def. 11)}\rangle\)

exists $h_1, h_2 :: (h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2)$

iff \(\langle\text{by let fresh variables, } h_1 \text{ and } h_2, \text{ be the witnesses of the existential variables.}\rangle\)

$h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2$

implies \(\langle\text{by truth of assertions is preserved under heap extension ( Lemma 9)}\rangle\)

$h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2 \land \sigma, h \models^{\Gamma}_s a_1 \land \sigma, h \models^{\Gamma}_s a_2$

implies \(\langle\text{by let } r_1 \text{ and } r_2 \text{ be fresh, and by Theorem 6}\rangle\)

$h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2 \land \sigma, h \models^{\Gamma}_s a_1 \land \sigma, h \models^{\Gamma}_s a_2 \land E[\Gamma \vdash fpt_s(a_1) : \text{region}] (\sigma) = r_1 \land E[\Gamma \vdash fpt_s(a_2) : \text{region}] (\sigma) = r_2$

implies \(\langle\text{by Corollary 4 and } h_1 \perp h_2\rangle\)

$h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2 \land \sigma, h \models^{\Gamma}_s a_1 \land \sigma, h \models^{\Gamma}_s a_2 \land E[\Gamma \vdash fpt_s(a_1) : \text{region}] (\sigma) = r_1 \land E[\Gamma \vdash fpt_s(a_2) : \text{region}] (\sigma) = r_2 \land r_1 \cdot ! r_2$

iff \(\langle\text{by inductive hypothesis}\rangle\)

$h_1 \perp h_2 \land h = h_1 \cdot h_2 \land \sigma, h_1 \models^{\Gamma}_s a_1 \land \sigma, h_2 \models^{\Gamma}_s a_2 \land \sigma, h \models^{\Gamma}_s a_1 \land \sigma, h \models^{\Gamma}_s a_2 \land E[\Gamma \vdash fpt_s(a_1) : \text{region}] (\sigma) = r_1 \land E[\Gamma \vdash fpt_s(a_2) : \text{region}] (\sigma) = r_2 \land r_1 \cdot ! r_2$

and $\sigma, h \models^{\Gamma}_u \text{TR}[[a_1]]$ and $\sigma, h \models^{\Gamma}_u \text{TR}[[a_2]]$
iff 〈by the semantics of assertions (Fig. 3.2)〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \text{TR}[[a_2]] \land (\text{fpt}_s(a_1) \land \text{fpt}_s(a_2)) \]

iff 〈by Mapping from SSL to UFRL (Def. 16)〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1 * a_2]] \]

Next, it is proved it from the right side to the left side. Assume
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \text{TR}[[a_2]] \land (\text{fpt}_s(a_1) \land \text{fpt}_s(a_2)) \]. The proof obligation is to show that
\[ \sigma, h \models^\Gamma a_1 * a_2. \]

iff 〈by the semantics of assertions (Fig. 3.2)〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \sigma, h \models^\Gamma \text{TR}[[a_2]] \land \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_1) : \text{region}]](\sigma) = r_1 \text{ and } \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_2) : \text{region}]](\sigma) = r_2 \text{ and } r_1 \land r_2 \]

iff 〈by inductive hypothesis〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \sigma, h \models^\Gamma \text{TR}[[a_2]] \land \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_1) : \text{region}]](\sigma) = r_1 \text{ and } \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_2) : \text{region}]](\sigma) = r_2 \text{ and } r_1 \land r_2 \text{ and } \sigma, h \models^\Gamma a_1 \land \sigma, h \models^\Gamma a_2 \]

iff 〈by Corollary 5〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \sigma, h \models^\Gamma \text{TR}[[a_2]] \land \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_1) : \text{region}]](\sigma) = r_1 \text{ and } \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_2) : \text{region}]](\sigma) = r_2 \text{ and } r_1 \land r_2 \]

implies 〈Lemma 9〉
\[ \sigma, h \models^\Gamma \text{TR}[[a_1]] \land \sigma, h \models^\Gamma \text{TR}[[a_2]] \land \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_1) : \text{region}]](\sigma) = r_1 \text{ and } \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a_2) : \text{region}]](\sigma) = r_2 \text{ and } r_1 \land r_2 \]

and \( \sigma, h \models^\Gamma a_1 \land \sigma, h \models^\Gamma a_2 \land \sigma, h \models^\Gamma (\text{dom}(h) - r_1) \models^\Gamma a_2 \)

implies 〈by Corollary 4, \( r_1 \cup (\text{dom}(h) - r_1) = \text{dom}(h), \text{ and } h_1 = h \upharpoonright r_1 \text{ and } h_2 = h \upharpoonright (\text{dom}(h) - r_1) \)〉
\[ \text{exists } h_1, h_2 :: (h_1 \perp h_2 \text{ and } h = h_1 \cdot h_2 \text{ and } \sigma, h_1 \models^\Gamma a_1 \text{ and } \sigma, h_2 \models^\Gamma a_2) \]

iff 〈by the semantics of separation logic (Def. 11)〉
\[ \sigma, h \models^r_s a_1 \ast a_2 \]
APPENDIX F: PROOF OF LEMMA 13
Lemma 13: Let $\Gamma$ be a well-formed type environment. Let $a$ and $a'$ be assertions and $S$ be a statement, such that $\models^s_s \{a\}S\{a'\}$. Let $(\sigma, H)$ be an arbitrary state. If $\sigma, H \models^s_s \{a\}$ and $\mathcal{M}\mathcal{S}\in\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H) = (\sigma', H')$, then:

1. for all $x \in \text{dom}(\sigma) :: \sigma'(x) \neq \sigma(x)$ implies $x \in \mathcal{M}\mathcal{V}(S)$.

2. for all $(o, f) \in \text{dom}(H) :: H'[o, f] \neq H[o, f]$ implies $(o, f) \in \mathcal{E}\in\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)$.

3. for all $(o, f) \in (\mathcal{E}\in\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma') - \mathcal{E}\in\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) :: (o, f) \in (\text{dom}(H') - \text{dom}(H))$.

Proof. Let $a, a', S, (\sigma, H)$ be given, such that $\models^s_s \{a\}S\{a'\}$. Let $(\sigma', H')$ be such that $\sigma', H') = \mathcal{M}\mathcal{S}\in\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H)$.

For property 1, it must be shown that for all $x \in \text{dom}(\sigma) :: \sigma'(x) \neq \sigma(x)$ implies $x \in \mathcal{M}\mathcal{V}(S)$. The proof is by induction on the structure of the statement $S$ and the definition of $\mathcal{M}\mathcal{V}(S)$. There are 6 base cases.

1. (SKIP) In this case, $S$ has the form skip$. According to its semantics Fig. 2.4, $\sigma = \sigma'$. Thus, it is vacuously true.

2. (VAR) In this case, $S$ has the form var $x : T$; According to its semantics Fig. 2.4, $\sigma' = \text{Extend}(\sigma, x, \text{default}(T))$. Thus, it is vacuously true, as $\text{Extend}$ only extends $\sigma$ by definition.

3. (ALLOC) In this case, $S$ has the form $y := \text{new} T$; for some variable $y$. According to the semantics Fig. 2.4, $\sigma' = \sigma[y \mapsto l]$, where $l$ is some new location. Thus, no other variables are mapped to different values by $\sigma'$. For $y$, it must be true that $\sigma'(y) \neq \sigma(y)$, and $y \in \mathcal{M}\mathcal{V}(y := \text{new} T;) = \{y\}$, according to Fig. 6.2.

4. (ASGN) In this case, $S$ has the form $y := e$; for some variable $y$. According to its semantics Fig. 2.4, $\sigma' = \sigma[y \mapsto v]$, where $v$ is the value of $e$. For $y$, $\sigma'(y) \neq \sigma(y)$, and $y \in \mathcal{M}\mathcal{V}(y := e) = \{y\}$, according to Fig. 6.2.
5. \textit{(UPD)} In this case, \( S \) has the form \( y.f := e; \). According to its semantics Fig. 2.4, \( \sigma' = \sigma \). Thus, it is vacuously true.

6. \textit{(ACC)} In this case, \( S \) has the form \( y := x'.f; \). According to its semantics Fig. 2.4, \( \sigma' = \sigma[y \mapsto v] \), where \( v \) is the value of \( x'.f \). Thus, \( \sigma'(y) \neq \sigma(y) \), and \( y \in \mathcal{M}V(y := x'.f) = \{ y \} \), according to Fig. 6.2.

The inductive hypothesis is that for all substatements \( S_i, (\sigma_i, H_i) \), and \( (\sigma'_i, H'_i) \), \textit{for all} \( x \in \text{dom}(\sigma_i) :: \sigma'_i(x) \neq \sigma_i(x) \) implies \( x \in \mathcal{M}V(S_i) \). There are 3 inductive cases.

1. \textit{(IF)} In this case, \( S \) has the form \textbf{if} \( e \) \{ \( S_1 \) \} \textbf{else} \{ \( S_2 \) \}. According to its semantics Fig. 2.4, if \( \mathcal{E}[[\Gamma \vdash e : \text{bool}]](\sigma) \neq 0 \), then the result follows from the inductive hypothesis applied to \( S_1 \). Similarly if \( \neg \mathcal{E}[[\Gamma \vdash e : \text{bool}]](\sigma) \), the result also follows similarly.

2. \textit{(WHILE)} In this case, \( S \) has the form \textbf{while} \( e \) \{ \( S \) \}. According to its semantics Fig. 2.4 on page 23, there exists \( n \geq 0 \), such that \( \sigma' = \sigma_n \) and \( \neg \mathcal{E}[[\Gamma \vdash e : \text{bool}]](\sigma_n) \). The proof is done by induction on \( n \). The base case is \( n = 0 \). According to the semantics Fig. 2.4, \( \sigma' = \sigma \).

Thus, it is vacuously true. For the inductive case, assume \textit{for all} \( x \in \text{dom}(\sigma) :: \sigma_{n-1}(x) \neq \sigma(x) \) implies \( x \in \mathcal{M}V(S) \). And by the inductive hypothesis, \textit{for all} \( x \in \text{dom}(\sigma_{n-1}) :: \sigma_n(x) \neq \sigma_{n-1}(x) \) implies \( x \in \mathcal{M}V(S) \). Thus, \textit{for all} \( x \in \text{dom}(\sigma) :: \sigma(x) \neq \sigma'(x) \) implies \( x \in \mathcal{M}V(S) \).

3. \textit{(SEQ)} In this case, \( S \) has the form \( S_1S_2 \). By definition, \( \mathcal{M}V(S_1S_2) = \mathcal{M}V(S_1) \cup \mathcal{M}V(S_2) \).

According to the statement’s semantics Fig. 2.4, assume \( \sigma_1 \) is the post-states of \( S_1 \). By the inductive hypothesis, \textit{for all} \( x \in \text{dom}(\sigma) :: \sigma_1(x) \neq \sigma(x) \) implies \( x \in \mathcal{M}V(S_1) \), and \textit{for all} \( x \in \text{dom}(\sigma_1) :: \sigma'(x) \neq \sigma_1(x) \) implies \( x \in \mathcal{M}V(S_2) \). Thus, \textit{for all} \( x \in \text{dom}(\sigma) :: \sigma'(x) \neq \sigma(x) \) implies \( x \in \mathcal{M}V(S_1S_2) \).
For property 2, it must be shown that for all \((o,f) \in \text{dom}(H)\): \(H'[o,f] \neq H[o,f]\) implies \((o,f) \in \mathcal{E}[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)\). Assume that \(\sigma, H \models^\Gamma_s \{a\} S \{a'\}\), \(\sigma, H \models^\Gamma_s a\) and \(\mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H')\). The proof is done in calculational style, starting from the assumptions.

\[
\begin{align*}
\sigma, H \models^\Gamma_s \{a\} S \{a'\} \text{ and } & \sigma, H \models^\Gamma_s a \text{ and } \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H') \\
\text{iff} & \langle \text{by assumption } \models^\Gamma_s \{a\} S\{a'\} \text{,} \\
& \langle \text{thus } (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s \{a\} S \{a'\} \text{,} \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H') \text{ and } (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{,} \\
\text{iff} & \langle \text{by Corollary 5: } \sigma, H \models^\Gamma_s a \text{ and } \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H') \text{ and } (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{,} \\
\text{iff} & \langle \text{by the definition of SSL valid Hoare-formula (Def. 17)} \rangle \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H') \text{ and } (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \models^\Gamma_s a \text{ and } \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \models^\Gamma_s a \text{ and } \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{,} \\
\text{iff} & \langle \text{by frame property of SL} \rangle \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \rangle = (\sigma', H') \text{ and } (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \models^\Gamma_s a \text{ and } \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{ and } \\
& \mathcal{M}S[\Gamma \vdash S : \text{ok}(\Gamma')] \langle \sigma, H \models^\Gamma_s a \text{ and } \\
& (\sigma, H \models^\Gamma_s \{a\} S \{a'\}) \iff \sigma, H \models^\Gamma_s a \text{,} \\
\text{and } H'' \vdash \text{dom}(H) - \mathcal{E}[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \text{ and } \\
H' = H'' \cdot H \vdash \text{dom}(H) - \mathcal{E}[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \text{ implies } (A \text{ and } B \implies B)
\[ H'' \downarrow H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
\[ H' = H'' \cdot H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \]
iff \[
\langle \text{by} \quad \text{for all } (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \implies \ldots \text{ implies } \rangle 
\]
\[
(o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \text{ is a tautology}
\]
\[ H'' \downarrow H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
\[ H' = H'' \cdot H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
for all \((o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \implies H''[o, f] \neq \]
\[ H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)[o, f] \text{ implies } (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \]
implies \[
\langle \text{by} \quad H' = H'' \cdot H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \cap E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) = \emptyset \rangle 
\]
\[ H'' \downarrow H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
\[ H' = H'' \cdot H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
and for all \((o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \implies H'[o, f] \neq \]
\[ H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)[o, f] \text{ implies } (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \]
implies \[
\langle \text{by Corollary 4, } E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \subseteq \text{dom}(H), \text{ twice} \rangle 
\]
\[ H'' \downarrow H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
\[ H' = H'' \cdot H \downarrow (\text{dom}(H) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \text{ and } \]
for all \((o, f) \in \text{dom}(H) : H'[o, f] \neq H[o, f] \text{ implies } (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \]
implies \[
\langle \text{by } A \text{ and } B \text{ implies } B \rangle 
\]
for all \((o, f) \in \text{dom}(H) : H'[o, f] \neq H[o, f] \text{ implies } (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \]
For property 3, it must be shown that \(\forall (o, f) \in (E[\Gamma' \vdash \text{fpt}_s(a') : \text{region}](\sigma) - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \implies (o, f) \in (\text{dom}(H') - \text{dom}(H)).\) Assume that \(\sigma, H \models_s \{a\} S \{a'\}, \sigma, H \models_s a \text{ and } MS[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H) = (\sigma', H').\) The proof is done in calculational style, starting from the assumptions.

\(\sigma, H \models_s \{a\} S \{a'\} \text{ and } \sigma, H \models_s a \text{ and } MS[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H) = (\sigma', H')\)
iff \[
\left( \begin{array}{l}
\text{by assumption } \models_s^\Gamma \{a\} S\{a'\}, \\
\text{thus } (\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\end{array} \right)
\]

iff \[
(\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H')
\]

iff \[
(\text{by Corollary 5: } \sigma, H \models_s^\Gamma a \iff \sigma, H \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H') \text{ and } \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\]

iff \[
(\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H') \text{ and } \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\]

iff \[
(\text{by the definition of SSL valid Hoare-formula (Def. 17)})
\]

\[
(\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H') \text{ and } \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\]

iff \[
(\text{by frame property of SL})
\]

\[
(\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H') \text{ and } \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\]

iff \[
(\text{by definition of SSL valid Hoare-formula (Def. 17)})
\]

\[
(\sigma, H) | E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma) \models_s^\Gamma \{a\} S\{a'\} \iff \sigma, H \models_s^\Gamma a \text{ and}
\]

\[
\mathcal{MS}[\Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) = (\sigma', H') \text{ and } \sigma, H \models_s^\Gamma \{a\} S\{a'\}
\]

implies \[
(\sigma', H'') \models_s^{\Gamma'} a' \text{ and } H'' \models_H (\text{dom}(H) - E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma)) \text{ and}
\]

\[
H' = H'' \cdot H | (\text{dom}(H) - E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma))
\]

iff \[
(\text{by } (\text{for all } (o, f) \in (r' - r) : (o, f) \in (r' - r)) \text{ is a tautology})
\]

\[
\sigma', H'' \models_s^{\Gamma'} a' \text{ and } H'' \models_H (\text{dom}(H) - E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma)) \text{ and}
\]

\[
H' = H'' \cdot H | (\text{dom}(H) - E[\Gamma \vdash fpt_s(a) : \text{region}] (\sigma))
\]

iff \[
(\text{by } (\text{for all } (o, f) \in (r' - r) : (o, f) \in (r' - r)) \text{ is a tautology})
\]
\( \sigma', H'' \models_{\pi} a' \) and \( H'' \uparrow H \vdash (\text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \) and

\[ H' = H'' \cdot H \downarrow \text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma) \] and

for all \( (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \):

\( (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \)

implies \( \langle \text{by Corollary 4, } \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma) \subseteq \text{dom}(H) \rangle \)

\( \sigma', H'' \models_{\pi} a' \) and \( H'' \uparrow H \vdash (\text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \) and

\[ H' = H'' \cdot H \downarrow \text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma) \] and

for all \( (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \) :

\( (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \)

implies \( \langle \text{by } H' = H'' \cdot H \downarrow \text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma) \rangle \) and Corollary 4

\( \sigma', H'' \models_{\pi} a' \) and \( H'' \uparrow H \vdash (\text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \) and

\[ H' = H'' \cdot H \downarrow \text{dom}(H) - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma) \] and

for all \( (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \) :

\( (o, f) \in (\text{dom}(H') - \text{dom}(H)) \)

implies \( \langle \text{by } A \text{ and } B \text{ implies } B \rangle \)

\( \langle \text{for all } (o, f) \in (\mathcal{E}[[\Gamma' \vdash \text{fpt}_s(a') : \text{region}]](\sigma') - \mathcal{E}[[\Gamma \vdash \text{fpt}_s(a) : \text{region}]](\sigma)) \rangle \)

\( \langle \text{for all } (o, f) \in (\text{dom}(H') - \text{dom}(H)) \rangle \)
APPENDIX G: PROOF OF THEOREM 8
**Theorem 8:** Let $\Gamma$ be a well-formed type environment. Let $S$ be a statement, and let $a$ and $a'$ be assertions in SSL, such that $\vdash^S \{a\} S \{a'\}$. Let $r$ be a region variable. Let $(\sigma, H)$ be a $\Gamma$-state. If $\sigma, H \vdash \Gamma [\{a\} \Rightarrow r = \text{fpt}_s(a) \land r \notin \text{MV}(S)]$, then

$$\sigma, H \vdash^S \{a\} S \{a'\} \iff \sigma, H \vdash^\Gamma [\{\text{fpt}_s(a)\} \{\text{TR}[a]\} S \{\text{TR}[a']\}][\text{modifies} \text{fpt}_s(a), \text{MV}(S), \text{fresh} (\text{fpt}_s(a') - r)]$$

**Proof.** Assume that $\sigma, H \vdash^S \{a\} S \{a'\}$. The lemma is proved by mutual implication. First it is proved that the left side implies the right side.

$$\sigma, H \vdash^S \{a\} S \{a'\} \quad \text{by assumption} \quad \vdash^S \{a\} S \{a'\}.$$  

$$\text{iff} \quad \left( \text{thus } (\sigma, H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \vdash^S \{a\} S \{a'\} \iff \sigma, H \vdash^S \{a\} S \{a'\} \right) \quad (\sigma, H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \vdash^S \{a\} S \{a'\}$$

**iff**  

(by the definition of SSL valid Hoare-formula (Def. 17))

$$(\sigma, H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \vdash^S \{a\} S \{a'\}$$

implies (by Lemma 13)

$$(\sigma, H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \vdash^S \{a\} S \{a'\}$$

$$(\sigma', H') = \text{MS}[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma), H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \neq \text{err} \land$$

if $(\sigma', H') = \text{MS}[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma), H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)$, then $\sigma', H' \vdash^S a'$

and for all $x \in \text{dom}(\sigma)$: $\sigma'(x) \neq \sigma(x) : x \in \text{MV}(S)$ and

for all $(o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)$:

$H'[o, f] \neq H) E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)[o, f] : (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)$ and

for all $(o, f) \in (E[\Gamma' \vdash \text{fpt}_s(a') : \text{region}](\sigma') \setminus E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma))$:

$(o, f) \in (\text{dom}(H') \setminus E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma))$
implies \( (H - H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \perp \)

\[
H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \quad \text{and} \quad H = (H - H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \cdot H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)
\]

\( (\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \models^* a \) implies

\[\mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \neq \text{err} \text{ and}
\]

if \( (\sigma', H') = \mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \), then

\( \sigma', H' \models^* a' \) and \( (\sigma', H'') = \mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H) \) and

for all \( x \in \text{dom}((\sigma)) : \sigma'(x) \neq \sigma(x) : x \in \text{MV}(S) \) and

for all \( (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) : H'[o, f] \neq \)

\( H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)[o, f] : (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \) and

for all \( (o, f) \in (E[\Gamma' \vdash \text{fpt}_s(a') : \text{region}](\sigma') - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) ::
\]

\( (o, f) \in (\text{dom}(H') - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \)

implies (by the frame property of SL)

\( (\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \models^* a \) implies

\[\mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \neq \text{err} \text{ and}
\]

if \( (\sigma', H') = \mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \), then

\( \sigma', H' \models^* a' \) and \( (\sigma', H'') = \mathcal{MS}[\Gamma \vdash \text{S} : \text{ok}(\Gamma')]((\sigma, H) \) and

\( H'' = H' \cdot (H - H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \) and

\( H' \perp (H - H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \) and

for all \( x \in \text{dom}((\sigma)) : \sigma'(x) \neq \sigma(x) : x \in \text{MV}(S) \) and

for all \( (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) : H'[o, f] \neq \)

\( H \downarrow E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)[o, f] : (o, f) \in E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \) and

for all \( (o, f) \in (E[\Gamma' \vdash \text{fpt}_s(a') : \text{region}](\sigma') - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) ::
\]

\( (o, f) \in (\text{dom}(H') - E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma)) \)

implies (by Corollary 5)
\[ \sigma, H \vDash_{\mathcal{S}}^{\ast} a \text{ implies } \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, \mathcal{H} E [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \neq \text{err} \text{ and} \]

if \((\sigma', H') = \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, \mathcal{H} E [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma))\), then \(\sigma', H' \vDash_{\mathcal{S}}^{\ast} a'\)

and \((\sigma', H'') = \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H)\) and

\[ H'' = H' \cdot (H - H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \text{ and} \]

\[ H' \uplus (H - H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \text{ and} \]

for all \(x \in \text{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathcal{M} V(S)\) and

for all \((o, f) \in \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) : H'[o, f] \neq\]

\[ H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)[o, f] : (o, f) \in \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) \text{ and} \]

for all \((o, f) \in (\mathcal{E} [\Gamma \vdash fpt_{s} (a') : \text{region}] (\sigma') - \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) : (o, f) \in (\text{dom}(H') - \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma))\)

iff \((\text{by Theorem 7, twice})\)

\[ \sigma, H \vDash_{\mathcal{S}}^{\ast} \mathcal{R} \Sigma [ a ] \text{ implies } \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, \mathcal{H} E [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \neq \text{err} \text{ and} \]

if \((\sigma', H') = \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, \mathcal{H} E [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma))\), then

\[ \sigma', H' \vDash_{\mathcal{S}}^{\ast} \mathcal{R} \Sigma [ a' ] \text{ and } (\sigma', H'') = \mathcal{M} S \Sigma [ \Gamma \vdash S : \text{ok}(\Gamma')] (\sigma, H) \text{ and} \]

\[ H'' = H' \cdot (H - H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \text{ and} \]

\[ H' \uplus (H - H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \text{ and} \]

for all \(x \in \text{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathcal{M} V(S)\) and

for all \((o, f) \in \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) : H'[o, f] \neq\]

\[ H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)[o, f] : (o, f) \in \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) \text{ and} \]

for all \((o, f) \in (\mathcal{E} [\Gamma \vdash fpt_{s} (a') : \text{region}] (\sigma') - \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) : (o, f) \in (\text{dom}(H') - \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma))\)

implies \(\begin{cases} \text{by } \text{dom}(H') - \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) = \text{dom}(H') - \text{dom}(H), \text{ because} \\ \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma) \subseteq \text{dom}(H) \text{ by Corollary 4, and} \end{cases} \)

\[ H' \uplus (H - H \cdot \mathcal{E} [\Gamma \vdash fpt_{s} (a) : \text{region}] (\sigma)) \]
\[\sigma, H \vdash \Gamma \] TR[a] \ implies \ MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma) \neq err\] and if \((\sigma', H') = MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\), then
\[\sigma', H' \vdash \Gamma' \] TR[a'] and \((\sigma', H'') = MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H)\) and \(H'' = H' \cdot (H - H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\) and \(H' \vdash (H - H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\)
and for all \(x \in dom(\sigma) : \sigma'(x) \neq \sigma(x) : x \in MV(S)\) and
for all \((o, f) \in E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma) \mid H'[o, f] \neq H'|E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma, o, f) \in E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\) and for all \((o, f) \in (E[\Gamma' \vdash fpt_\sigma(a') : \text{region}](\sigma') - E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)) :: (o, f) \in (dom(H') - dom(H))\)

\[
\text{by Corollary 4, and } H' \vdash (H - H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma) \]
\[
\sigma, H \vdash \Gamma \] TR[a] \ implies \ MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma) \neq err\] and if \((\sigma', H') = MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H) \mid E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\), then
\[\sigma', H' \vdash \Gamma' \] TR[a'] and \((\sigma', H'') = MS[\Gamma \vdash S : ok(\Gamma')](\sigma, H)\) and
for all \(x \in dom(\sigma) : \sigma'(x) \neq \sigma(x) : x \in MV(S)\) and
for all \((o, f) \in dom(H) : H'[o, f] \neq H[o, f] : (o, f) \in E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)\) and for all \((o, f) \in (E[\Gamma' \vdash fpt_\sigma(a') : \text{region}](\sigma') - E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma)) :: (o, f) \in (dom(H') - dom(H))\)

iff \((\text{by assumption } E[\Gamma \vdash fpt_\sigma(a) : \text{region}](\sigma) = E[\Gamma \vdash r : \text{region}](\sigma))\)

194
Next, let \( r = \text{fpt}_s(a) \). The proof goes from the right side to the left side.

\[
\sigma, H \models \Gamma \implies M_S[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H) \implies E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \neq \text{err}
\]

iff \( \langle \text{by } E[\Gamma \vdash r : \text{region}](\sigma) = E[\Gamma' \vdash r : \text{region}](\sigma') \text{ because } r \notin \text{MV}(S) \rangle \)

\[
\sigma, H \models \Gamma \implies M_S[\Gamma \vdash S : \text{ok}(\Gamma')](\sigma, H) \implies E[\Gamma \vdash \text{fpt}_s(a) : \text{region}](\sigma) \neq \text{err}
\]

where \( \text{fpt}_s(a) \) modifies \( r = \text{fpt}_s(a) \)
\[
\sigma, H \models \Gamma \vdash \text{region}(\sigma) \iff \text{iff}
\]

\[
\begin{align*}
&\sigma, H \models \Gamma \vdash \text{region}(\sigma) \iff \text{iff} \\
&\text{iff (by Theorem 7, twice)} \\
&\sigma, H \models \Gamma \vdash \text{region}(\sigma) \iff \text{iff}
\end{align*}
\]
APPENDIX H: PROOF OF THEOREM 9
Theorem 9: Each translated SSL axiom is derivable, and each translated rule is derivable in the UFRL proof system.

Proof. The proof is by the induction on the derivation and by cases in the last rule used. In each case, it is shown that the translated proof axioms and rules are derivable in UFRL.

1. SKIP: by Def. 18, the encoded axiom is the axiom $\text{SKIP}_u$.

2. VAR: by Def. 18, the encoded axiom is the axiom $\text{VAR}_u$.

3. ALLOC: by the rule $\text{ALLOC}_s$ and Def. 18, the translated rule is is shown below:

$$
\begin{align*}
\text{reads } fpt_s(a) \\
\Gamma \vdash \{ \text{TR}[a]\} x := \text{new } T; \{ \text{TR}[a \ast new_s(T, x)]\} \\
\text{modifies } fpt_s(a, x, \text{alloc}, \text{fresh}(fpt_s(new_s(T, x))))
\end{align*}
$$

where $x \notin FV(a)$

By definition of the predicate $new_s(T, x)$, it is known that $fpt_s(new_s(T, x)) = \text{region}\{x.\}$. Given the axiom $\text{ALLOC}_u$, the translated rule is derived by using the rules $\text{FRM}_u$ and $\text{SubEff}_u$.

The derivation is shown in Fig. H.1. The program semantics assumes that the location for each field in a class is disjoint with each other, thus $new(T, x) \text{ iff } new_s(T, x)$.

4. ACC: by the rule $\text{ACC}_s$ and Def. 18, the translated rule is shown below:

$$
\begin{align*}
\text{region}\{x'.f\} \\
\Gamma \vdash \{ x'.f = z \} x := x' f; \{ x'.f = z \text{ \&\& } x = z \} \\
\text{modifies region}\{x'.f\}, x
\end{align*}
$$

where $x \neq x', x' \neq z \text{ and } x \neq z$

where the fresh effect is empty, thus, it is omitted; it is true that $fpt_s(x = z) \text{ iff } fpt_s(x'.f \mapsto z)$, thus, it is omitted. Given the axiom $\text{ACC}_u$, by definition of read effects for assertions in Fig. 3.5, it must be true that $(x'.f = z) \vdash (\text{reads } x', \text{region}\{x'.f\}, z) \text{ frm } x'.f = z$. By the side
conditions and the definition of separator, it must be true that \( \text{reads}(x', \text{region}(x'.f), z) \vdash \text{modifies } x \). Hence \( x'.f = z \) is the frame. Using the rules \( FRM_u \) and \( SubEff_u \), the translated rule is derived in Fig. H.2.

5. **UPD**: by the rule \( UPD_s \) and Def. 18, the translated rule is derived as follows:

\[
\Gamma \vdash [\text{reads region}\{x.f\}]
\{
\exists z.x.f = z \}x.f := E\{x.f = E\}[\text{modifies region}\{x.f\}]
\text{(H.3)}
\]

where \( x \notin \text{FV}(E) \)

where the fresh effect is empty, thus, it is omitted. Note that \( x.f \mapsto _{\_} \) is an abbreviation for \( \exists z.x.f \mapsto _{\_} \) Thus \( x.f \mapsto _{\_} \) is translated to \( \exists z.x.f = z \). The translated rule is derived by using the rules \( SubEff_u \) and \( CONSEQ_u \). The derivation is shown in Fig. H.3.

6. **SEQ**: by the rule \( SEQ_s \) and Def. 18, the translated rule is derived as follows:

\[
\Gamma \vdash [\text{reads } r_1]\{\text{TR}[a]\} S_1 \{\text{TR}[b]\}[\text{modifies } r_1, \text{MV}(S_1), \text{fresh } (r_2 - r_1)]
\]

\[
\Gamma \vdash [\text{reads } r_2]\{\text{TR}[b]\} S_2 \{\text{TR}[a']\}[\text{modifies } r_2, \text{MV}(S_2), \text{fresh } (fpt_s(a') - r_2)]
\]

\[
\Gamma \vdash [\text{reads } r_1]\{\text{TR}[a]\} S_1 S_2 \{\text{TR}[a']\}[\text{modifies } r_1, \text{MV}(S_1 S_2), \text{fresh } (fpt_s(a') - r_1)]
\]

where \( \text{TR}[a] \Rightarrow r_1 = fpt_s(a), r_1 \notin \text{MV}(S_1), \text{TR}[b] \Rightarrow r_2 = fpt_s(b) \text{ and } r_2 \notin \text{MV}(S_2) \)

\( \text{(H.4)} \)

There are two cases:

(a) \( S_1 = \text{var } x T; \): In this case, by the rule \( VAR_s \), it must be true that \( b = a \ast \text{default}(T) \).
\( \text{MV}(\text{var } x : T) = \emptyset \) and \( r_1 = r_2 = \text{fpt}_s(a) \). Then the proof obligation is to show

\[ \vdash_{u} \text{reads } r_1 \downarrow \{ \text{TR}[a] \} \text{var } x : T ; \{ \text{TR}[a * \text{default}(T)] \} \{ \text{modifies } r_1 \downarrow \} \]

\[ \vdash_{u} \{ \text{TR}[a * \text{default}(T)] \} S_2 \{ \text{TR}[a'] \} \]

\[ \{ \text{modifies } r_1 \downarrow, \text{MV}(S_2), \text{fresh } (\text{fpt}_s(a') - r_1) \} \]

\[ \vdash \text{reads } r_1 \downarrow \{ \text{TR}[a] \} \text{var } x : T ; S_2 \{ \text{TR}[a'] \} \]

\[ \{ \text{modifies } r_1 \downarrow, \text{MV}(S_2), \text{fresh } (\text{fpt}_s(a') - r_1) \} \]

where \( \text{TR}[a * \text{default}(T)] \Rightarrow r_2 = \text{fpt}_s(a) \) and \( r_2 \notin \text{MV}(S_2) \).

Using the rule \( \text{SubEff}_u \) on the second premise, the following is derived

\[ \vdash \text{reads } r_1 \downarrow \]

\[ \vdash_u \{ \text{TR}[a * \text{default}(T)] \} S_2 \{ \text{TR}[a'] \} \]

\[ \{ \text{modifies } r_1 \downarrow, x, \text{MV}(S_2), \text{fresh } (\text{fpt}_s(a') - r_1) \} \]

Using the rule \( \text{SEQ2}_u \), the conclusion of Eq. (H.5) is derived.

(b) \( S_1 \neq \text{var } x : T ; \) consider the following cases:

\[ \text{• } S_1 \text{ does not allocate new locations, i.e., } r_1 = r_2. \text{ The rule } \text{SEQ1}_u \text{ is instantiated with } RE := \text{region}\}, \ \text{RE}_1 := \text{region}\} \ \text{and } \text{RE}_2 := \text{region}\}. \text{ If the immunity side conditions are satisfied, then the conclusion of (H.4) is derived by using the rule } \text{SEQ1}_u. \text{ Otherwise, for all } x \in \text{MV}(S_1) \text{ and } x \in \text{FV}(b), \text{ there exists } z, \text{ such that } b \text{ implies } x = z \text{ and } z \notin \text{MV}(S_1). \text{ The variable } z \text{ is substituted for } x \text{ in } \text{fpt}_s(b). \text{ Then the second premise of Eq. (H.4) is re-written as:} \]

\[ \vdash \{ \text{reads } r_1 \downarrow [\{ z / \text{MV}(S_1) \}] \} \]

\[ \vdash u \{ \text{TR}[b] \} S_2 \{ \text{TR}[a'] \} \]

\[ \{ \text{modifies } r_1 \downarrow [\{ z / \text{MV}(S_1) \}], \text{MV}(S_2), \text{fresh } (\text{fpt}_s(a') - r_1) \} \]

where \( r_1 \downarrow [\{ z / \text{MV}(S_1) \}] \) means that \( \text{for all } RE \in r_1 \downarrow :: \text{RE}[\text{MV}(S_1) / z] \). From the first premise of Eq. (H.4) and Eq. (H.7), the immunity side conditions are satisfied. After using the rule \( \text{SEQ1}_u \).
the following is derived.

$\Gamma \vdash_u \{ \text{reads } r_1 \downarrow, r_1 \downarrow [\bar{z}/\text{MV}(S_1)] \}

\begin{align*}
\vdash_u & \{ \text{TR}[[a]] \} S_1 S_2 \{ \text{TR}[[a']] \} \\
\text{modifies } & r_1 \downarrow, r_2 \downarrow [\text{MV}(S_1)/\bar{z}], \text{MV}(S_1 S_2), \text{fresh}(fpt_s(a') - r_1 \downarrow [\text{MV}(S_1)/\bar{z}]) \\
\end{align*}

(H.8)

Because for all $RE \in r_1 :: RE$ in $r_2 \downarrow [\overline{y/x}]$, Eq. (H.8) can be simplified to

$\begin{align*}
\vdash_u & \{ \text{reads } r_1 \downarrow \} \\
\text{modifies } & r_1 \downarrow, \text{MV}(S_1 S_2), \text{fresh}(fpt_s(a') - r_1) \\
\end{align*}

- $S_1$ allocates some new locations. Then the second premise of Eq. (H.4) can be re-written as:

$\begin{align*}
\vdash_u & \{ \text{reads } r_1 \downarrow, (r_2 - r_1) \} \\
\text{modifies } & r_1 \downarrow, (r_2 - r_1), \text{MV}(S_2), \text{fresh}(fpt_s(a') - r_2) \\
\end{align*}

The rule $SEQ1_u$ is instantiated with $RE := r_2 - r_1$, $RE_1 := r_2 - r_1$ and $RE_2 := r_2 - r_1$. If the immunity side conditions are satisfied, then union the fresh effects of the two statements and get $fpt_s(a') - r_1$. Hence, the conclusion of Eq. (H.4) is derived by using the rule $SEQ1_u$. Otherwise, the treatment is similar to the previous case.
7. **IF**: by the rule $IF_s$ and Def. 18, the translated rule is shown as follows:

$$
\begin{align*}
\Gamma & \text{reads } fpt_s(a) \\
\Gamma & \{ \text{TR}[a] \land E \} \text{ } S_1 \{ \text{TR}[a'] \} \\
\text{modifies } fpt_s(a), MV(S_1), \text{fresh}(fpt_s(a') - r) \\
\Gamma & \{ \text{TR}[a] \land \neg E \} \text{ } S_2 \{ \text{TR}[a'] \} \\
\text{modifies } fpt_s(a), MV(S_2), \text{fresh}(fpt_s(a') - r) \\
\end{align*}
$$

Note that $fpt_s(E)$ and $fpt_s(\neg E)$ are both region{}, thus are omitted. By the inductive hypothesis, the premise of Eq. (H.9) is assumed. Then, using the rule $IF_u$, the following is derived.

$$
\begin{align*}
\Gamma & \text{reads } fpt_s(a), \text{reads } efs(E) \\
\Gamma & \{ \text{TR}[a] \} \text{ } \text{if } E \text{then} S_1 \text{else} S_2 \{ \text{TR}[a'] \} \\
\text{modifies } fpt_s(a), MV(S_1), MV(S_2), \text{fresh}(fpt_s(a') - r) \\
\end{align*}
$$

Now, consider to use the rule $SubEff_u$. Because $regRW(\text{reads } fpt_s(a), \text{reads } efs(E), \text{modifies } fpt_s(a), MV(S_1), MV(S_2), \text{fresh}(fpt_s(a') - r)) = fpt_s(a)$, the following side condition is true:

$$
fpt_s(a) \leq regRW(\text{reads } fpt_s(a), \text{modifies } fpt_s(a), MV(S_1), MV(S_2), \text{fresh}(fpt_s(a') - r))
$$

Therefore, after using the rule $SubEff_u$, the conclusion of Eq. (H.9) is derived.
8. **WHILE** by the rule \( \text{WHILE}_s \) and Def. 18, the translated rule is shown below:

\[
\begin{align*}
\Gamma_u & \quad \text{reads} \ fpt_s(I) \{ \TR[II] \& \& S\{ \TR[II] \} \} \{ \text{modifies} \ fpt_s(I), MV(S) \} \\
\Gamma_u & \quad \{ \TR[II] \} \text{while } E \{ S \} \{ \TR[II] \& \& \neg E \} \\
& \quad \{ \text{modifies} \ fpt_s(I), MV(S) \} \\
\end{align*}
\]

(H.11)

The rule \( \text{WHILE}_u \) is instantiated with \( RE := \text{region}\{\} \). The treatment about the immunity side condition is similar to that of the sequence rule. If it is satisfied, then the rule \( \text{WHILE}_u \) is used and the following is derived.

\[
\begin{align*}
\Gamma_u & \quad \{ \TR[II] \} \text{while } E \{ S \} \{ \TR[II] \& \& \neg E \} \\
& \quad \{ \text{modifies} \ fpt_s(I), MV(S) \} \\
\end{align*}
\]

(H.12)

Similarly to the case of the rule \( IF_u \), using the rule \( \text{SubEff}_u \), the conclusion of Eq. (H.11) is derived.

If the immunity side condition is not satisfied, for all \( x \in MV(S) \) and \( x \in FV(I) \), there exists \( z \) such that \( I \) implies \( x = z \) and \( z \notin MV(S) \). The variable \( z \) is substituted for \( x \) in \( fpt_s(I) \). Then the immunity side condition is satisfied. the rules \( \text{WHILE}_u \) and \( \text{SubEff}_u \) are used and the conclusion is derived.
9. *FRM*: by the rule $FRM_s$ and Def. 18, the translated rule is shown as follows:

$$[\text{reads } r_1 \downarrow]$$

$$\vdash \Gamma \{ \text{TR}[a] \} S \{ \text{TR}[a'] \}$$

$$[\text{modifies } r_1 \downarrow, MV(S), \text{fresh } (fpt_s(a') - r_1)]$$

$$[\text{reads } r_1 + r_2]$$

$$\vdash \Gamma \{ \text{TR}[a] \land \text{TR}[c] \land (r_1 \neq r_2) \} S \{ \text{TR}[a'] \land \text{TR}[c] \land (fpt_s(a') \neq r_2) \}$$

$$[\text{modifies } fpt_s(a) + r, MV(S), \text{fresh } (fpt_s(a') + r_2 - r')]$$

where $\text{TR}[a] \Rightarrow r_1 = fpt_s(a), \text{TR}[c] \Rightarrow r_2 = fpt_s(c), r_1 \notin MV(S), r_2 \notin MV(S)$,

$$\text{TR}[a] \land \text{TR}[c] \land (fpt_s(a') \neq r_2) \Rightarrow r' = r_1 + r_2, \text{and } MV(S) \cap FV(c) = \emptyset$$

(H.13)

By the inductive hypothesis, the premise of Eq. (H.13) is assumed. The rule $FRM_u$ is instantiated with $Q := \text{TR}[c]$ and $\eta := efs(\text{TR}[c])$. The proof obligation is to show the side condition, which is:

$$\text{TR}[a] \land \text{TR}[c] \land (r_1 \neq r_2) \text{ implies } efs(\text{TR}[c]) / (\text{modifies } MV(S), fpt_s(a))$$

(H.14)

By Lemma 14 and by the definition of separator (Fig. 3.7), Eq. (H.14) is true. After using the rule $FRM_u$, the following is derived.

$$[\text{reads } r_1 \downarrow]$$

$$\vdash \Gamma \{ \text{TR}[a] \land \text{TR}[c] \land (r_1 \neq r_2) \} S \{ \text{TR}[a'] \land \text{TR}[c] \}$$

$$[\text{modifies } r_1 \downarrow, MV(S), \text{fresh } (fpt_s(a') - r_1)]$$

(H.15)

Now, consider to use the rule $FRM_u$ again. It is instantiated with $Q := r_1 + r_2$ and $\eta := \text{reads } r_1, \text{reads } r_2$. The proof obligation is to show the side condition is true, which is:

$$(\text{TR}[a] \land \text{TR}[c] \land (r_1 \neq r_2)) \text{ implies } (\text{reads } r_1, \text{reads } r_2) / (\text{modifies } r_1 \downarrow, MV(S))$$

(H.16)

By $r_1 \notin MV(S)$ and $r_2 \notin MV(S)$, Eq. (H.16) is true. Note that $\text{modifies } r_1 \downarrow$ means that values in the locations contained in $r_1$ may be modified. The variable $r_1$ is not changed. After using
the rule $FRM_u$, the following is derived.

$$\Gamma \uparrow_{\text{reads } r_1 \downarrow} \vdash \{\text{reads } r_1 \downarrow\} \{\text{modifies } r_1 \downarrow, \text{fresh } (fpt_s(a') - r_1)\}$$

(H.17)

Because $\text{TR}[c]$ is preserved by $S$, $r_2 = fpt_s(c)$ in the post-state. Thus, after using the rule $CONSEQ_u$, the following is derived.

$$\Gamma \uparrow_{\text{reads } r_1 \downarrow} \vdash \{\text{reads } r_1 \downarrow\} \{\text{modifies } r_1 \downarrow, \text{MV}(S), \text{fresh } (fpt_s(a') - r_1)\}$$

(H.18)

Now the proof obligation is to show that $fpt_s(a') \in fpt_s(c)$ in the poststate. By the definition of SSL Hoare-formula, it is known that $fpt_s(a') = r_1 + RE$, where $RE$ are possibly empty regions that do not exist in the pre-state, hence $RE \in fpt_s(c)$. Hence, $fpt_s(a') \in fpt_s(c)$ is true. Then, after using the rule $CONSEQ_u$, the following is derived.

$$\Gamma \uparrow_{\text{reads } r_1 \downarrow} \vdash \{\text{reads } r_1 \downarrow\} \{\text{modifies } r_1 \downarrow, \text{MV}(S), \text{fresh } (fpt_s(a') - r_1)\}$$

(H.19)

Now, consider the fresh effects. By the side condition $r' = r_1 + r_2$, it must be true that

$$fpt_s(a') - r_1 = fpt_s(a') + r_2 - r_1 - r_2 = fpt_s(a') + r_2 - r'.$$

Finally, using the rule $SubEff_u$ to loosen the read effects, the conclusion of Eq. (H.13) is derived.
\[
\text{FRM}_u \quad \Gamma \vdash \emptyset \quad \{ \text{true} \} \times := \text{new } T; \{ \text{new}(T,x) \}
\]
\[
\quad \text{modifies } x, \text{alloc, fresh region}\{x.\}
\]
\[
\quad \text{true } \vdash \emptyset \quad \text{efs}(\text{TR}[\alpha]) \text{ frm TR}[\alpha]
\]
\[
\quad \text{where } \text{TR}[\alpha] \Rightarrow \text{efs}(\text{TR}[\alpha])/(x, \text{alloc}) \text{ and }
\]
\[
\quad \text{FV}(\alpha) \cap \{x\} = \emptyset \Rightarrow \text{FV}(\text{TR}[\alpha]) \cap \{x\} = \emptyset (\text{ Lemma 15})
\]
\[
\text{SubEff}_u \quad \vdash \emptyset \quad \{ \text{TR}[\alpha] \} \times := \text{new } T; \{ \text{TR}[\alpha] \text{ & new}(T,x) \}
\]
\[
\quad \text{modifies } x, \text{alloc, fresh region}\{x.\}
\]
\[
\quad \text{(Subeffect) } \vdash \emptyset \preceq \text{reads fpt}_s(a)
\]
\[
\text{SubEffect}_u \quad \vdash \emptyset \quad \{ \text{TR}[\alpha] \} \times := \text{new } T; \{ \text{TR}[\alpha] \text{ & new}(T,x) \}
\]
\[
\quad \text{modifies } x, \text{alloc, fresh region}\{x.\}
\]
\[
\quad \text{(Subeffect) } \vdash \emptyset \preceq \text{reads fpt}_s(a)
\]
\[
\text{SubEffect}_u \quad \vdash \emptyset \quad \{ \text{TR}[\alpha] \} \times := \text{new } T; \{ \text{TR}[\alpha] \text{ & new}(T,x) \}
\]
\[
\quad \text{modifies } x, \text{alloc, fpt}_s(a), \text{fresh region}\{x.\}
\]
\[
\quad \text{where } \text{TR}[\alpha] \Rightarrow \text{alloc and fpt}_s(a) \preceq r
\]
\[
\quad \text{(FrToPost}_u \quad \vdash \emptyset \quad \{ \text{TR}[\alpha] \}
\]
\[
\quad \{ \text{TR}[\alpha] \text{ & new}(T,x) \text{ & (fpt}_s(a) \text{ ! region}\{x.\})\}
\]
\[
\quad \text{modifies } x, \text{alloc, fpt}_s(a), \text{fresh(region}\{x.\}])
\]

Figure H.1: The derivation of rule TR\(_R[ALLOC_s]\)
$\Gamma$ denotes the set of modules that are read or modified.

(ACC) $\vdash_{u} [\text{efs}(x.f)] \{x' \neq \text{null}\} x := x' . f ; \{x = x'.f\}$ modifies $x$

(FRM) $\vdash_{u} [\text{reg}(x.f), x, z] \text{frm}(x.f = z)$

(SubEffect) $\vdash_{u} \text{modifies } x \leq \text{modifies } x, \text{reg}(x.f)$

(1) $x'.f = z \& \& x \neq y \& \& x' \neq y \Rightarrow ((\text{reg}(x.f), x', y)/x)$ and $x' \neq \text{null} \Rightarrow \exists z.(x'.f = z)$

(2) $x' \neq \text{null} \Rightarrow \text{regR}(\text{efs}(x.f), x, \text{reg}(x.f)) \leq \text{regR}(\text{reg}(x.f), x)$

Figure H.2: The derivation of rule $\text{TR}_R[[\text{ACC}_s]]$

(UPD) $\vdash_{u} [\text{reads } x, \text{efs}(E)] \{x \neq \text{null}\} x . f := E; \{x . f = E\}$

(CONSEQ) $\vdash_{u} [\text{reads } x, \text{efs}(E)] \{x \neq \text{null}\} x . f := E; \{x . f = E\}$ modifies $x.f$

(SubEffect) $\vdash_{u} \text{reads readR}(\text{reads } x, \text{efs}(E)) \leq \text{reads region}(x.f)$

(1) $x' \neq \text{null} \Rightarrow \exists z.x.f = z$

Figure H.3: The derivation of rule $\text{TR}_R[[\text{UPD}_s]]$
APPENDIX I: PROOF OF LEMMA 23
Lemma 23: Let $(\sigma, h)$ be a state, and $p_s$ be an inductive predicate in SSL. Then

$$E_a[\Gamma \vdash p_s(\bar{v}) : \texttt{bool}] (\sigma, h) = E_p[\Gamma \vdash TR[\Gamma p_s(\bar{v})] : \texttt{bool}] (\sigma, h).$$

Proof. The proof is an inductive case of the proof of Theorem 7. The inductive hypothesis is that

for all subassertions $a_i$, $E_a[\Gamma \vdash a_i : \texttt{bool}] (\sigma, h) = E_p[\Gamma \vdash TR[a_i] : \texttt{bool}] (\sigma, h)$. Let $b_1 \Rightarrow a_1 \cdots b_n \Rightarrow a_n$ be inductive cases for $p_s$. We prove it as follows.

$$E_a[\Gamma \vdash p_s(\bar{v}) : \texttt{bool}] (\sigma, h)$$

iff \hspace{1em} \langle\text{by semantics of inductive predicates Eq. (7.4)}\rangle

$$(\text{fix}_A(\sigma', h'). \ E_a[\Gamma \vdash (b_1 \Rightarrow a_1) \land \ldots \land (b_n \Rightarrow a_n) : \texttt{bool}] (\sigma', h'))$$

$$(\sigma(formals_s(p_s) \rightarrow E_a[\Gamma \vdash e : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by semantics of assertions Def. 11}\rangle

$$(\text{fix}_A(\sigma', h'). \ E_a[\Gamma \vdash b_1 \Rightarrow a_1 : \texttt{bool}] (\sigma', h') \text{ and } \ldots \text{ and}$$

$$E_a[\Gamma \vdash b_n \Rightarrow a_n : \texttt{bool}] (\sigma', h'))(\sigma(formals_s(p_s) \rightarrow E_a[\Gamma \vdash e : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by inductive hypothesis}\rangle

$$(\text{fix}_A(\sigma', h'). \ E_p[\Gamma \vdash TR[b_1 \Rightarrow a_1] : \texttt{bool}] (\sigma', h') \text{ and } \ldots \text{ and}$$

$$E_p[\Gamma \vdash TR[b_n \Rightarrow a_n] : \texttt{bool}] (\sigma', h'))(\sigma(formals_s(p_s) \rightarrow E_a[\Gamma \vdash e : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by Lemma 11: } E_a[\Gamma \vdash e : T]](\sigma) = E[\Gamma \vdash TR[e] : T]](\sigma)\rangle

$$(\text{fix}_A(\sigma', h'). \ E_p[\Gamma \vdash TR[b_1 \Rightarrow a_1] : \texttt{bool}] (\sigma', h') \text{ and } \ldots \text{ and}$$

$$E_p[\Gamma \vdash TR[b_n \Rightarrow a_n] : \texttt{bool}] (\sigma', h'))(\sigma(formals_s(p_s) \rightarrow E_a[\Gamma \vdash e : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by the semantics of assertions (Fig. 3.2)}\rangle

$$(\text{fix}_A(\sigma', h'). \ E_p[\Gamma \vdash TR[b_1 \Rightarrow a_1] \& \ldots \& TR[b_n \Rightarrow a_n] : \texttt{bool}] (\sigma', h'))$$

$$(\sigma(formals_s(p_s) \rightarrow E[\Gamma \vdash TR[e] : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by the definition of encoding inductive predicates in Fig. 7.1}\rangle

$$(\text{fix}_A(\sigma', h'). \ E_p[\Gamma \vdash idf(p_s)] : \texttt{bool}] (\sigma', h'))$$

$$(\sigma(formals_s(p_s) \rightarrow E[\Gamma \vdash TR[e] : T]](\sigma)), h)$$

iff \hspace{1em} \langle\text{by semantics of recursive predicate}\rangle
\[ \mathcal{E}_p[\Gamma \vdash TR[\bar{p}(\bar{v})] : \text{bool}](\sigma, h) \]
LIST OF REFERENCES


