Reasoning About Frame Properties in Object-oriented Programs

2017

Yuyan Bao
University of Central Florida

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 Engieering & 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

LIST OF TABLES

CHAPTER 1: INTRODUCTION[1]

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:

$$(FRM_s)\ \frac{\vdash_s^\Gamma \{a\}\ S\ \{a'\}}{\vdash_s^\Gamma \{a * c\}\ S\ \{a' * c\}} \quad \textbf{where } \mathtt{MV}(S) \cap \mathtt{FV}(c) = \varnothing$$

where $\mathtt{FV}(c)$ returns the set of free variables in c and $\mathtt{MV}(S)$ returns the set of variables that may be modified by *S*.is 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 dag:

$$dag(d) \stackrel{\text{def}}{=} d \neq null \Rightarrow \exists\, i, j, k.(d.mark \mapsto i * d.l \mapsto j * d.r \mapsto k * (dag(j) \wedge dag(k)))$$

where the assertions of the form $x.f \mapsto e$ 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:

$$(FRM_{rl})\ \frac{\vdash^{\Gamma}_{rl} \{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^{\Gamma}_{rl} \delta\ frm\ Q}{\vdash^{\Gamma}_{rl} \{P\ \&\&\ Q\}\ S\ \{P'\ \&\&\ Q\}[\varepsilon]}\ \textbf{\textit{where}}\ P\ \&\&\ Q \Rightarrow \delta \cdot\!/\!\cdot \varepsilon$$

The formula ε is a *write effect* that denotes the set of variables and locations that may be modified by *S*. The formula δ 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 dynamicallly 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.val := e;`) is just $n \neq null$ in region logic, or $\exists\ x.n.val \mapsto x$ in separation logic, no matter whether n is a node in a list or not.

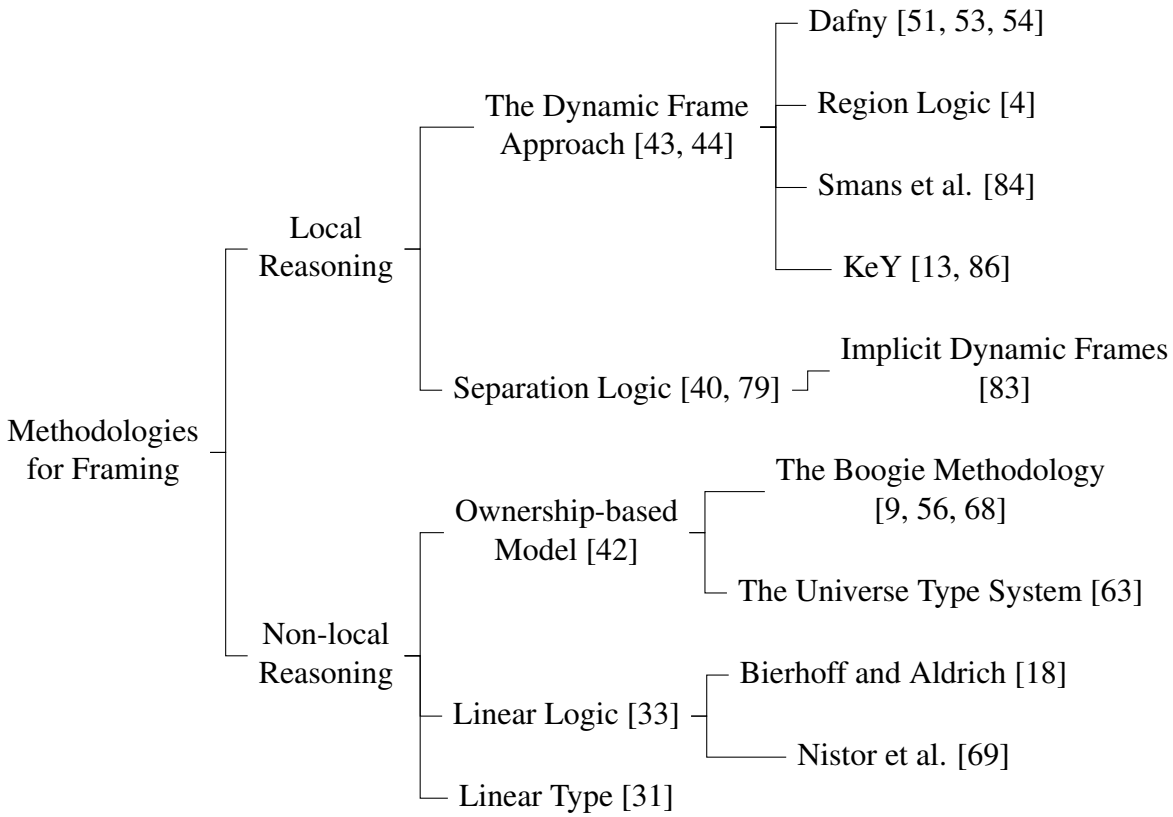

Figure 1.1: Taxonomy of methodologies for framing

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 **upapck**, 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: $P \multimap Q$, 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 **`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, **`adopt`** e_1 **`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.

Region Logic:

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 **`rgn`** or 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 **`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, **`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 $\mathbf{rd} G\text{`}f$ allows one to read the field of objects in $G$ [4, p.22]; e.g., the RL read effect $\mathbf{rd}\ x\text{`}f$ is equivalent to the FRL read effect $\mathbf{reads}\ \mathbf{region}\{x.f\}$.

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 $\mathtt{fptp}(x.f = E) = \mathbf{rd}\ x, x.f, \mathtt{ftpt}(E)$, where **rd** is the keyword for read effects (this work uses **reads** instead). The form $\mathbf{rd}\ x.f$ abbreviates the form $\mathbf{rd}\ \{x\}\text{`}f$ [4, p.23]. Although $x\text{`}f$ may not be the same as the region expression $\mathbf{region}\{x.f\}$ as explained previously, $\mathbf{rd}\ x\text{`}f$ is semantically equivalent to as $\mathbf{reads}\ \mathbf{region}\{x.f\}$ 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 **old** expressions to specify which fields of

threatened objects must not change. Consider the following example adapted from Leino's work [54].

```
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 `\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 **`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 Spec$^\sharp$ specification language [10] encodes an ownership-based model with ghost variables. In Spec$^\sharp$, 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 α separate-conjuncts from the assertion c. However, the work in this dissertation needs methodology to express encapsulation.

CHAPTER 2: PROGRAMMING LANGUAGE[1]

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 **`this`** is the receiver object; the variable **`ret`** stores the return value of a method if the method has one; the variable **`alloc`** stores the domain of the heap. In the syntax, the notation n is a numeral, x is a variable name (or a pseudo-variable, such as **`alloc`**), and f is a field name.

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \overline{\mathit{Class}}\ \ S \\
\mathit{Class} & ::= & \texttt{class}\ \mathsf{C}\ \{\ \overline{\mathit{Member}}\ \} \\
\mathit{Member} & ::= & \mathit{Field} \mid \mathit{Method} \\
\mathit{Field} & ::= & \texttt{var}\ f\texttt{:}T\texttt{;} \\
\mathit{Method} & ::= & \texttt{method}\ m\,(\overline{x : T})\ [\texttt{:}T']\{\ \overline{S}\ \} \\
T & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{region} \mid C \mid C\texttt{<}\overline{T}\texttt{>} \\
E & ::= & n \mid x \mid \texttt{null} \mid E_1 \oplus E_2 \\
RE & ::= & x \mid \texttt{region}\{\} \mid \texttt{region}\{x.f\} \mid \texttt{region}\{x.*\} \mid E\ \texttt{?}\ RE_1\ \texttt{:}\ RE_2 \\
 & \mid & \texttt{filter}\{RE,T,f\} \mid \texttt{filter}\{RE,T\} \mid RE_1 \otimes RE_2 \\
G & ::= & E \mid RE \\
S & ::= & \texttt{skip;} \mid \texttt{var}\ x\texttt{:}T\texttt{;} \mid x\texttt{:=}G\texttt{;} \mid x_1\texttt{:=}x_2.f\texttt{;} \mid x.f\texttt{:=}G\texttt{;} \\
 & \mid & x\texttt{:=new}\ T\texttt{;} \mid \texttt{if}\ E\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\} \mid \texttt{while}\ E\ \{S\} \mid S_1S_2 \\
\oplus & ::= & = \mid + \mid - \mid * \mid \leqslant \ldots \\
\otimes & ::= & + \mid - \mid *
\end{array}
$$

Figure 2.1: The syntax of the programming language

[1]Subsets of the presented language in this chapter was presented at *FTfJP* '15 [7] and is submitted to *Formal Aspects of Computing*.

$\texttt{MV}(\textbf{skip;}) = \varnothing$ $\quad \texttt{MV}(\textbf{var}\ x : T;) = \varnothing$ $\quad \texttt{MV}(x := \textbf{new}\ T;) = \{x\}$
$\texttt{MV}(x := G;) = \{x\}$ $\quad \texttt{MV}(x.f := G;) = \varnothing$ $\quad \texttt{MV}(x := x'.f;) = \{x\}$
$\texttt{MV}(\textbf{if}\ E\ \textbf{then}\ \{S_1\}\textbf{else}\ \{S_2\}) = \texttt{MV}(S_1) \cup \texttt{MV}(S_2)$
$\texttt{MV}(\textbf{while}\ E\ \{S\}) = \texttt{MV}(S)$
$\texttt{MV}(S_1 S_2) = \texttt{MV}(S_1) \cup \texttt{MV}(S_2)$

Figure 2.2: The definition of the function MV.

A class consists of fields and methods. A field is declared with type integer, a user-defined datatype, or **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 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 **region**, which is a set of locations. The region expression **region**$\{\}$ denotes the empty region. The region expression of the form **region**$\{x.f\}$ 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 **region**$\{x.*\}$ denotes a set that contains the abstract locations represented by the reference x and all its fields. [2] 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 **filter**$\{RE, T, f\}$ 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 **filter**$\{RE, T\}$ 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 **filter**$\{RE, T\} =$

[2] Since FRL does not have subclassing or subtyping, the fields in **region**$\{x.*\}$ are based on the static type of the reference denoted by x, which will also be its dynamic type.

Table 2.1: Features of the programming language that are not formalized in this dissertation.

seq<T>	seqnence type
$\|s\|$	the length of the sequence s
$s[i]$	the element at index i of the sequence s if $0 \leqslant i$ and $i < \|s\|$
$s[i..]$	generate a new sequence that has the same elements in the same order as s, but the first one, if $0 \leqslant i$ and $i < \|s\|$
$s[i..j]$	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 $0 \leqslant i$ and $i < \|s\|$
$s_1 + s_2$	sequence concatenation
map<K, V>	map type
$m[k]$	the value of a given key k in a map m, if k is in the domain of m
k **in** m	test whether the key k is in the domian of the map m
k !**in** m	test whether the key k is not in the domian of the map m
$m[k := v]$	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
map $i \mid i$ **in** m && $i \neq k :: m[i]$	generate a new map that is the same as the map m excluding the key k.

$\{o_1.f_1, o_1.f_2\}$. 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

$$\textbf{predicate}\ \texttt{p}(\overline{z:T})\ \textbf{reads}\ \texttt{r}\ \{\ \texttt{P}\ \}$$

is syntax sugar for the declaration

$$\textbf{method}\ \texttt{p}(\overline{z:T})\ \textbf{reads}\ \texttt{r}\ \{\ \textbf{ret}\ :=\ \texttt{P};\ \}$$

Predicates are used for the purpose of specification, and cannot be invoked by programs.

There is a type environment, Γ, which maps variables to types:

$$\Gamma \in \mathit{TypeEnv} = \mathit{var} \rightarrow \mathit{T}.$$

A type environment, Γ, is *well-formed* if it is a partial function, i.e., for all $x \in \mathit{dom}(\Gamma)$, $\Gamma(x)$ 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, σ, is a partial function that maps a variable to its value. A heap, h or H, is a finite partial map from Loc to values. The set *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* $(o, f) \in \mathit{dom}(H)$*.* $H[o, f]$ *is the value that* H *associates to* (o, f)*.*

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

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

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:* $\mathit{dom}(H_1 \cdot H_2) = \mathit{dom}(H_1) \cup \mathit{dom}(H_2)$*, and for all* $(o, f) \in \mathit{dom}(H_1 \cdot H_2)$:

$$(H_1 \cdot H_2)[o, f] = \begin{cases} H_1[o, f], & \text{if } (o, f) \in \mathit{dom}(H_1), \\ H_2[o, f], & \text{if } (o, f) \in \mathit{dom}(H_2). \end{cases}$$

Let H be a heap and R be a region. The restriction of H to R, *written $H \restriction R$ is defined by: $dom(H \restriction R) = dom(H) \cap R$ and $\forall (o, f) \in (H \restriction R) :: (H \restriction R)[o, f] = H[o, f]$. We use err to denote an error region or an error heap; the restriction of a heap H to an error region is defined by: $H \restriction err = err$.* ∎

Notations: Let f and g be two partial functions. Then $f \preccurlyeq g$ means that $dom(f) \preccurlyeq dom(g)$ and $\forall\, x \in dom(f) :: f(x) = g(x)$. And $f * g$ means that $dom(f) \cap dom(g) = \varnothing$. 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 MO gives the semantics of operators. A *Value* is either a Boolean, an object reference (which may be $null$), an integer or a set of locations: $Value = Boolean + Object + Int + PowerSet(Loc)$. The auxiliary function *fields*(T) takes a reference type T and returns a list of its declared field names and their types. The function $type(o)$ takes a reference o and returns its type. Pure expressions evaluate to *Value*s; 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 **`region`**$\{x.f\}$ is evaluated to an empty set when $x = null$. The pair (**`null`**, f) is not allowed in the regions of our language's semantics.

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

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

$\mathcal{E}$: *Expression Typing Judgment* → *Store* → *Value*

$\mathcal{E}[\![\Gamma \vdash x : T]\!](\sigma) = \sigma(x) \qquad \mathcal{E}[\![\Gamma \vdash \texttt{null} : T]\!](\sigma) = null \qquad \mathcal{E}[\![\Gamma \vdash n : \texttt{int}]\!](\sigma) = \mathcal{N}[\![n]\!]$

$\mathcal{E}[\![\Gamma \vdash E_1 \oplus E_2 : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash E_1 : T_1]\!](\sigma)\ \mathcal{MO}[\![\oplus]\!]\ \mathcal{E}[\![\Gamma \vdash E_2 : T_2]\!](\sigma)$

$\mathcal{E}[\![\Gamma \vdash \texttt{region}\{\} : \texttt{region}]\!](\sigma) = \varnothing$

$\mathcal{E}[\![\Gamma \vdash \texttt{region}\{x.f\} : \texttt{region}]\!](\sigma) = \textbf{if } \sigma(x) = null \textbf{ then } \varnothing \textbf{ else } \{(\sigma(x), f)\}$

$\mathcal{E}[\![\Gamma \vdash \texttt{region}\{x.*\} : \texttt{region}]\!](\sigma) =$
$\quad \textbf{if } \sigma(x) = null \textbf{ then } \varnothing \textbf{ else } \{(o, f) \mid o = \sigma(x) \text{ and } T = \mathit{type}(o) \text{ and } (f : T') \in \mathit{fields}(T)\}$

$\mathcal{E}[\![\Gamma \vdash E \ ?\ RE_1 \ ?\ RE_2 : \texttt{region}]\!](\sigma) =$
$\quad \textbf{if } \mathcal{E}[\![\Gamma \vdash E : \texttt{bool}]\!](\sigma) \textbf{ then } \mathcal{E}[\![\Gamma \vdash RE_1 : \texttt{region}]\!](\sigma) \textbf{ else } \mathcal{E}[\![\Gamma \vdash RE_2 : \texttt{region}]\!](\sigma)$

$\mathcal{E}[\![\Gamma \vdash \texttt{filter}\{RE, T\} : \texttt{region}]\!](\sigma) =$
$\quad \{(o, f) | (o, f) \in \mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma) \wedge \mathit{type}(o) = T\}$

$\mathcal{E}[\![\Gamma \vdash \texttt{filter}\{RE, T, f\} : \texttt{region}]\!](\sigma) =$
$\quad \{(o, f') | (o, f') \in \mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma) \wedge f' = f \wedge \mathit{type}(o) = T\}$

$\mathcal{E}[\![\Gamma \vdash RE_1 \otimes RE_2 : \texttt{region}]\!](\sigma) =$
$\quad \mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma)\ \mathcal{MO}[\![\otimes]\!]\ \mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma)$

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 ($\underline{\lambda}$) 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* $\Gamma \vdash S : ok(\Gamma')$. *Let* Γ'' *be a well-formed type environment, such that* $\mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma'') = \varnothing$ *and*

$$
\begin{array}{l}
\mathcal{MS} : \textit{Statement Typing Judgment} \to \textit{State} \to \textit{State}_\perp \\
\mathcal{MS}[\![\Gamma \vdash \texttt{skip};: ok(\Gamma)]\!](\sigma, H) = (\sigma, H) \\
\mathcal{MS}[\![\Gamma \vdash \texttt{var}\ x : T;: ok(\Gamma, x : T)]\!](\sigma, H) = (\sigma[x \mapsto \textit{default}(T)], H) \\
\mathcal{MS}[\![\Gamma \vdash x := G;: ok(\Gamma)]\!](\sigma, H) = \ (\sigma[x \mapsto \mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma)], H) \\
\mathcal{MS}[\![\Gamma \vdash x.f := G;: ok(\Gamma)]\!](\sigma, H) = \\
\quad \textbf{if}\ \sigma(x) \neq null\ \textbf{then}\ \ (\sigma, H[(\sigma(x), f) \mapsto \mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma)])\ \textbf{else}\ \ err \\
\mathcal{MS}[\![\Gamma \vdash x_1 := x_2.f;: ok(\Gamma)]\!](\sigma, H) = \\
\quad \textbf{if}\ \sigma(x_2) \neq null\ \textbf{then}\ \ (\sigma[x_1 \mapsto H[(\sigma(x_2), f)]], H)\ \textbf{else}\ \ err \\
\mathcal{MS}[\![\Gamma \vdash x := \texttt{new}\ T;: ok(\Gamma)]\!](\sigma, H) = \textbf{let}\ (l, H') \ = \ allocate(T, H)\ \textbf{in} \\
\quad \textbf{let}\ (f_1, \ldots, f_n) \ = \ \textit{fields}(T)\ \textbf{in} \\
\quad\quad \textbf{let}\ \sigma' \ = \ \sigma[x \mapsto l]\ \textbf{in} \\
\quad\quad\quad (\sigma', H'[(\sigma'(x), f_1) \mapsto \textit{default}(T_1), \ldots, (\sigma'(x), f_n) \mapsto \textit{default}(T_n)]) \\
\mathcal{MS}[\![\Gamma \vdash \texttt{if}\ E\ \texttt{then}\{S_1\}\texttt{else}\{S_2\} : ok(\Gamma)]\!](\sigma, H) = \\
\quad \textbf{if}\ \mathcal{E}[\![\Gamma \vdash E : \texttt{bool}]\!](\sigma)\ \textbf{then}\ \ \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma_1)]\!](\sigma, H) \\
\quad \textbf{else}\ \ \mathcal{MS}[\![\Gamma \vdash S_2 : ok(\Gamma_2)]\!](\sigma, H) \\
\mathcal{MS}[\![\Gamma \vdash \texttt{while}\ E\ \{S\};: ok(\Gamma)]\!](\sigma, H) = \\
\quad \textit{fix}(\lambda g\ .\ \underline{\lambda} s\ .\ \ \textbf{let}\ v \ = \ \mathcal{E}[\![\Gamma \vdash E : \texttt{bool}]\!](\sigma)\ \textbf{in} \\
\quad\quad\quad \textbf{if}\ v \neq 0\ \textbf{then}\ \ \textbf{let}\ s' \ = \ \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](s)\ \textbf{in}\ g \circ s' \\
\quad\quad\quad \textbf{else}\ \ \textbf{if}\ v = 0\ \textbf{then}\ \ s\ \textbf{else}\ err)(\sigma, H) \\
\mathcal{MS}[\![\Gamma \vdash S_1 S_2 : ok(\Gamma')]\!](\sigma, H) = \textbf{let}\ s' \ = \ \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma'')]\!](\sigma, H)\ \textbf{in} \\
\quad \textbf{if}\ s' \neq err\ \textbf{then}\ \ \mathcal{MS}[\![\Gamma'' \vdash S_2 : ok(\Gamma')]\!](s')\ \textbf{else}\ \ err
\end{array}
$$

Figure 2.4: The semantics of statements

dom$(\Gamma') \cap$ *dom*$(\Gamma'') = \varnothing$. *Then*

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

2. *if* $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, h) = (\sigma', h')$, *then* $\mathcal{MS}[\![\Gamma, \Gamma'' \vdash S : ok(\Gamma', \Gamma'')]\!](\sigma, h) = (\sigma', h')$.

CHAPTER 3: ASSERTION LANGUAGES AND FRAMING[1]

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.

$$
\begin{array}{rcl}
B & ::= & E_1 = E_2 \mid E_1 \neq E_2 \\
P & ::= & B \mid x.f = E \mid RE_1 \leqslant RE_2 \mid P_1 \;\&\&\; P_2 \mid P_1 \;||\; P_2 \mid \neg P \\
 & \mid & \forall\ x\texttt{:int::}P \mid \forall\ x\texttt{:}T\texttt{:region}\{x.f\}\leqslant RE\texttt{:}P \mid \exists\ x\texttt{:int::}P \\
 & \mid & \exists\ x\texttt{:}T\texttt{:region}\{x.f\}\leqslant RE\texttt{:}P
\end{array}
$$

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 $RE_1 \leqslant RE_2$ checks that RE_1 is a subregion of RE_2. The assertion RE_1 `!!` 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

[1]The content in this chapter was partially presented at *FTfJP* '15 [7]. And part of the content is submitted to *Formal Aspects of Computing*.

$\sigma, H \models^{\Gamma} E_1 = E_2$ *iff* $\mathcal{E}[\![\Gamma \vdash E_1 : T_1]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash E_2 : T_2]\!](\sigma)$
$\sigma, H \models^{\Gamma} E_1 \neq E_2$ *iff* $\mathcal{E}[\![\Gamma \vdash E_1 : T_1]\!](\sigma) \neq \mathcal{E}[\![\Gamma \vdash E_2 : T_2]\!](\sigma)$
$\sigma, H \models^{\Gamma} x.f = E$ *iff* $(\sigma(x), f) \in dom(H)$ *and* $H[\sigma(x), f] = \mathcal{E}[\![\Gamma \vdash E : T]\!](\sigma)$
$\sigma, H \models^{\Gamma} RE_1 \leqslant RE_2$ *iff* $\mathcal{E}[\![\Gamma \vdash RE_1 : \mathbf{region}]\!](\sigma) \subseteq \mathcal{E}[\![\Gamma \vdash RE_2 : \mathbf{region}]\!](\sigma)$
$\sigma, H \models^{\Gamma} RE_1 \ !\ !\ RE_2$ *iff* $\mathcal{E}[\![\Gamma \vdash RE_1 : \mathbf{region}]\!](\sigma) \cap \mathcal{E}[\![\Gamma \vdash RE_2 : \mathbf{region}]\!](\sigma) = \varnothing$
$\sigma, H \models^{\Gamma} P_1 \ \&\&\ P_2$ *iff* $\sigma, H \models^{\Gamma} P_1$ *and* $\sigma, H \models^{\Gamma} P_2$
$\sigma, H \models^{\Gamma} P_1 \ ||\ P_2$ *iff* $\sigma, H \models^{\Gamma} P_1$ *or* $\sigma, H \models^{\Gamma} P_2$
$\sigma, H \models^{\Gamma} \neg P$ *iff* $\sigma, H \not\models P$
$\sigma, H \models^{\Gamma} \forall\, x : \mathbf{int} :: P$ *iff for all* $v :: \sigma[x \mapsto v], H \models^{\Gamma,\, x:\mathbf{int}} P$
$\sigma, H \models^{\Gamma} \forall\, x : T : \mathbf{region}\{x.f\} \leqslant RE : P$ *iff for all* $o : (o, f) \in \mathcal{E}[\![\Gamma \vdash RE : \mathbf{region}]\!](\sigma)$ *and* $type(o) = T : \sigma[x \mapsto o], H \models^{\Gamma,\, x:T} P$
$\sigma, H \models^{\Gamma} \exists\, x : \mathbf{int} :: P$ *iff exists* $v :: \sigma[x \mapsto v], H \models^{\Gamma,\, x:\mathbf{int}} P$
$\sigma, H \models^{\Gamma} \exists\, x : T : \mathbf{region}\{x.f\} \leqslant RE : P$ *iff exists* $o : (o, f) \in \mathcal{E}[\![\Gamma \vdash RE : \mathbf{region}]\!](\sigma)$ *and* $type(o) = T : (\sigma[x \mapsto o], H) \models^{\Gamma,\, x:T} P$

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* Γ *and* Γ' *be two well-formed type environments. Let* (σ, H) *be a* Γ*-state. Then* $\sigma, H \models^{\Gamma} P$ *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., (**modifies** x, **region**$\{y.f\}$) is short for (**modifies** x, **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

ε, δ ::= $\varnothing$ | $\varepsilon_1, \varepsilon_2$ | E ? $\varepsilon_1 : \varepsilon_2$ | **reads** RE | **reads** x | **modifies** RE
| **modifies** x | **fresh**(RE)

Figure 3.3: The grammar of effects

avoid ambiguity, the notation **reads** $x\downarrow$ means that reading the locations that are in x, where x has type **region**; the notation **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 Γ; for example, **reads** x is meaningless if x in not in the domain of Γ.

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

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

where M *is either* **reads**, **modifies**, *or* **fresh**. ∎

A correct method must have an actual write effect that is a sub-effect of its specified effect.[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.

[2]The sub-effect rules are also applicable for read effects.

$$\vdash^{\Gamma} \varepsilon \leqslant \varepsilon \qquad \vdash^{\Gamma} \varepsilon, \varepsilon' \leqslant \varepsilon', \varepsilon \qquad \frac{\varepsilon' \text{ is a write or read effect}}{\vdash^{\Gamma} \varepsilon \leqslant \varepsilon, \varepsilon'} \qquad \vdash^{\Gamma} \textbf{fresh}\, RE, \varepsilon \leqslant \varepsilon$$

$$\mathit{false} \vdash^{\Gamma} \varepsilon \leqslant \varepsilon' \qquad \frac{P \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_2 \qquad P \vdash^{\Gamma} \varepsilon_2 \leqslant \varepsilon_3}{P \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_3} \qquad \frac{P' \Rightarrow P \qquad P \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_2}{P' \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_2}$$

$$\frac{P \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_2}{P \vdash^{\Gamma} \varepsilon_1, \varepsilon \leqslant \varepsilon_2, \varepsilon} \qquad \vdash^{\Gamma} \textbf{modifies}\, RE_1, RE_2 \quad \lesseqgtr \textbf{modifies}\, RE_1 + RE_2$$

$$\vdash^{\Gamma} \textbf{reads}\, RE_1, RE_2 \quad \lesseqgtr \textbf{reads}\, RE_1 + RE_2$$

$$\vdash^{\Gamma} \textbf{modifies filter}\{RE, T, f\} \leqslant \textbf{modifies}\, RE$$

$$\vdash^{\Gamma} \textbf{modifies filter}\{RE, T\} \leqslant \textbf{modifies}\, RE$$

$$RE_1 \leqslant RE_2 \vdash^{\Gamma} \textbf{modifies}\, RE_1 \leqslant \textbf{modifies}\, RE_2$$

$$RE_1 \leqslant RE_2 \vdash^{\Gamma} \textbf{reads}\, RE_1 \leqslant \textbf{reads}\, RE_2 \qquad \vdash^{\Gamma} E?\varepsilon_1 : \varepsilon_2 \leqslant \varepsilon_1, \varepsilon_2$$

$$\frac{\begin{array}{cc} P \mathbin{\&\&} E_1 \mathbin{\&\&} E_2 \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_3 & P \mathbin{\&\&} \neg E_1 \mathbin{\&\&} E_2 \vdash^{\Gamma} \varepsilon_2 \leqslant \varepsilon_3 \\ P \mathbin{\&\&} E_1 \mathbin{\&\&} \neg E_2 \vdash^{\Gamma} \varepsilon_1 \leqslant \varepsilon_4 & P \mathbin{\&\&} \neg E_1 \mathbin{\&\&} \neg E_2 \vdash^{\Gamma} \varepsilon_2 \leqslant \varepsilon_4 \end{array}}{P \vdash^{\Gamma} E_1?\varepsilon_1 : \varepsilon_2 \leqslant E_2?\varepsilon_3 : \varepsilon_4}$$

Figure 3.4: The sub-effect rules

- *writeR* discards all but region expressions in write effects; for example, *writeR*(**reads** x, **modifies** y, **modifies region**$\{x.f\}$) is equal to **region**$\{x.f\}$.
- *readR* discards all but region expressions in read effects; e.g., *readR*(**reads** x, **reads region**$\{x.f\}$) = **region**$\{x.f\}$.
- *freshR* discards all but region expressions in fresh effects; e.g., *freshR*(**reads** x, **modifies region**$\{x.f\}$, **fresh region**$\{y.*\}$) = **region**$\{y.*\}$.
- *readVar* discards all but variables in read effects; for example, *readVar*(**reads** x, **reads**

region$\{x.f\}) = x$.

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

Write effects and fresh effects make sense for two consecutive states, written $(\sigma, h) \hookrightarrow (\sigma', h')$. 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.

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

1. *for all* $x \in dom(\Gamma)$*, either* $\sigma(x) = \sigma'(x)$ *or* **modifies** x *is in* ε*;*
2. *for all* $(o, f) \in \sigma($**alloc**$)$*, either* $h[o, f] = h'[o, f]$ *or there is RE such that* **modifies** *RE is in* ε*, such that* $(o, f) \in \mathcal{E}[\![\Gamma \vdash RE :$ **region**$]\!](\sigma)$*;*
3. *for all RE such that* **fresh**(RE) *is in* ε*, it must be true that* $\mathcal{E}[\![\Gamma \vdash RE :$ **region**$]\!](\sigma') \subseteq ($ $\sigma'($**alloc**$) - \sigma($**alloc**$)$ $)$. ∎

Def. 4 says that if two states agree on a read effect, δ, then the values that depend on δ 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* Γ *be a well-formed type environment. Let* δ *be a well-formed effect in* Γ*. Let* $\Gamma' \geqslant \Gamma$ *and* $\Gamma'' \geqslant \Gamma$*. Let* (σ', h') *and* (σ'', h'') *be a* Γ'*-state and a* Γ''*-state respectively. Then it is said that* (σ', h') *and* (σ'', h'') agree *on* δ*, written* $(\sigma', h') \overset{\delta}{\equiv} (\sigma'', h'')$*, when the following holds:*

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

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 **reads** R' *frm* P. If $R' \leqslant \bar{R}$, i.e., R' `!!` 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*$(x.f = y) =$ **reads** x, **region**$\{x.f\}$, y.

$$
\begin{array}{ll}
\mathit{efs}(x) = \textbf{reads}\ x & \mathit{efs}(n) = \varnothing \qquad \mathit{efs}(null) = \varnothing \\
\mathit{efs}(E_1 \oplus E_2) = \mathit{efs}(E_1), \mathit{efs}(E_2) & \mathit{efs}(\textbf{region}\{\}) = \varnothing \\
\mathit{efs}(\textbf{region}\{x.f\}) = \textbf{reads}\ x & \mathit{efs}(\textbf{region}\{x.*\}) = \textbf{reads}\ x \\
\multicolumn{2}{l}{\mathit{efs}(E\ ?\ RE_1 : RE_2) = \mathit{efs}(E), E\ ?\ \mathit{efs}(RE_1) : \mathit{efs}(RE_2)} \\
\mathit{efs}(\textbf{filter}\{RE, f\}) = \mathit{efs}(RE) & \mathit{efs}(\textbf{filter}\{RE, T, f\}) = \mathit{efs}(RE) \\
\mathit{efs}(RE_1 \otimes RE_2) = \mathit{efs}(RE_1), \mathit{efs}(RE_2) & \mathit{efs}(E_1 = E_2) = \mathit{efs}(E_1), \mathit{efs}(E_2) \\
\mathit{efs}(E_1 \neq E_2) = \mathit{efs}(E_1), \mathit{efs}(E_2) & \mathit{efs}(x.f = E) = \textbf{reads}\ x, \textbf{region}\{x.f\}, \mathit{efs}(E) \\
\mathit{efs}(RE_1 \leqslant RE_2) = \mathit{efs}(RE_1), \mathit{efs}(RE_2) & \mathit{efs}(RE_1\ !!\ RE_2) = \mathit{efs}(RE_1), \mathit{efs}(RE_2)
\end{array}
$$

Figure 3.5: The read effects of expressions, region expressions and atomic assertions

The framing judgment, $P \vdash^{\Gamma} \delta \mathit{frm}\ Q$, means that read effects, δ, contains the variables and locations that are needed to evaluate Q in a Γ-state that satisfies P. Fig. 3.6 shows the judgment for assertions.

(*FrmFtpt*)
$\mathit{true} \vdash^{\Gamma} \mathit{efs}(P)\ \mathit{frm}\ P$
where P is atomic

(*FrmNeg*)
$\mathit{true} \vdash^{\Gamma} \mathit{efs}(P)\ \mathit{frm}\ \neg P$
where P is atomic

(*FrmSub*)
$$\frac{R \vdash^{\Gamma} \delta_1\ \mathit{frm}\ Q \qquad Q \vdash^{\Gamma} \delta_1 \leqslant \delta_2}{P \vdash^{\Gamma} \delta_2\ \mathit{frm}\ Q}$$
where $P \Rightarrow R$

(*FrmConj*)
$$\frac{P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_1 \qquad P\ \&\&\ Q_1 \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_2}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ (Q_1\ \&\&\ Q_2)}$$

(*FrmDisj*)
$$\frac{P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_1 \qquad P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_2}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ (Q_1\ ||\ Q_2)}$$

(*FrmEq*)
$$\frac{P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_1}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q_2}$$
where $Q_1 \iff Q_2$

(*FrmProjCtx*)
$$\frac{P\ \&\&\ Q \vdash^{\Gamma} \delta\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ Q}$$

(*Frm*$\forall_1$)
$$\frac{P \vdash^{\Gamma, x:\mathtt{int}} (\delta, \mathbf{reads}\ x)\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm} \forall\ x : \mathbf{int} :: Q}$$

(*Frm*$\forall_2$)
$$\frac{P \vdash^{\Gamma} \mathbf{reads}\ \mathit{efs}(RE) \leqslant \delta \qquad P \wedge \mathbf{region}\{x.f\} \leqslant RE \vdash^{\Gamma, x:T} (\varepsilon,\ x)\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm} \forall\ x : T : \mathbf{region}\{x.f\} \leqslant RE : Q}$$

(*Frm*$\exists_1$)
$$\frac{P \vdash^{\Gamma, x:\mathtt{int}} (\delta, \mathbf{reads}\ x)\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm} \exists\ x : \mathbf{int} :: Q}$$

(*Frm*$\exists_2$)
$$\frac{P \vdash^{\Gamma} \mathbf{reads}\ \mathit{efs}(RE) \leqslant \delta \qquad P\ \&\&\ \mathbf{region}\{x.f\} \leqslant RE \vdash^{\Gamma, x:T} (\delta, x)\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm} \exists\ x : T : \mathbf{region}\{x.f\} \leqslant RE : Q}$$

Figure 3.6: The inference rules for the framing judgment

The following defines the meaning of a framing judgment.

Definition 5 (Frame Validity)**.** *Let* Γ *be a well-formed type environment. Let* P *and* Q *be asser-*

tions, and δ be a read effect. The framing judgment $P \vdash^{\Gamma} \delta$ frm Q is valid, written $P \models^{\Gamma} \delta$ frm Q, if and only if for all Γ-states (σ, h) and (σ', h'), if $(\sigma, h) \overset{\delta}{\equiv} (\sigma', h')$ and $\sigma, h \models^{\Gamma} P \,\&\&\, Q$, then $\sigma', h' \models^{\Gamma} Q$. ∎

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

Lemma 3 (Framing Soundness for Expressions)**.** *Let Γ be a well-formed type environment. Let (σ, h) and (σ', h') be two Γ-states. Let G be an expression. If $(\sigma, h) \overset{efs(G)}{\equiv} (\sigma', h')$, then it must be true that $\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma')$.*

Proof. The proof is straightforward by structural induction on expressions, thus is omitted. □

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

Proof. By induction on a derivation of a framing judgment $P \vdash^{\Gamma} \delta$ *frm* Q. □

3.4 Separator and Immune

The notation, $\not\cdot$, is used to define the disjointness on effects in Fig. 3.7 on the following page, where δ is a read effect and ε is a write effect. $\delta \not\cdot \varepsilon$ means that the read effects in δ are disjoint with the write effects in ε. The effect, **reads** δ, where δ is not a conditional effect, is treated as **reads if** $true$ **then** δ **else** $\varnothing$. For example, let *RE* be **if** $x.f{=}0$ **then region**$\{y.f\}$ **else region**$\{\}$. Suppose $x \neq y$ and $x.f \neq 0$. The separation of **reads region**$\{y.f\}$ and **modifies** *RE* can be derived to **reads region**$\{y.f\} \not\cdot$ **modifies region**$\{\}$ by the rule ConMask introduced in the next section.

$$
\begin{array}{rcl}
\delta \not/ \varnothing & = & \mathit{true} \\
\varnothing \not/ \varepsilon & = & \mathit{true} \\
\mathbf{reads}\ y \not/ \mathbf{modifies}\ x & = & y \not\equiv x \\
\mathbf{reads}\ y \not/ \mathbf{modifies}\ \mathit{RE} & = & \mathit{true} \\
\mathbf{reads}\ \mathit{RE}_1 \not/ \mathbf{modifies}\ x & = & \mathit{true} \\
\mathbf{reads}\ \mathit{RE}_1 \not/ \mathbf{modifies}\ \mathit{RE}_2 & = & \mathit{RE}_1\ !!\ \mathit{RE}_2 \\
\delta \not/ (\varepsilon, \varepsilon') & = & (\delta \not/ \varepsilon) \wedge (\delta \not/ \varepsilon') \\
(\delta, \delta') \not/ \varepsilon & = & (\delta \not/ \varepsilon) \wedge (\delta' \not/ \varepsilon) \\
\delta \not/ (E\ ?\ \varepsilon_1\ :\ \varepsilon_2) & = & \begin{cases} \delta \not/ \varepsilon_1 & \text{if} \quad E \\ \delta \not/ \varepsilon_2 & \text{if} \quad \neg E \end{cases} \\
(E\ ?\ \delta_1\ :\ \delta_2) \not/ \varepsilon & = & \begin{cases} \delta_1 \not/ \varepsilon & \text{if} \quad E \\ \delta_2 \not/ \varepsilon & \text{if} \quad \neg E \end{cases}
\end{array}
$$

Figure 3.7: The definition of separator

The following lemma says if read effects, δ, and write effects, ε are separate, then the values on δ are preserved.

Lemma 5 (Separator Agreement)**.** *Let Γ be a well-formed type environment. Let ε and δ be effects in Γ. Let (σ, h) and (σ', h') be two Γ-states. If $(\sigma, h) \rightarrow (\sigma', h') \models^{\Gamma} \varepsilon$ and $(\sigma, h) \models^{\Gamma} \delta \not/ \varepsilon$, then $(\sigma, h) \overset{\delta}{\equiv} (\sigma', h')$.*

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

1. Let **reads** x in δ be arbitrary. Since $(\sigma, H) \models^{\Gamma} \delta \not/ \varepsilon$, **modifies** $x \notin \varepsilon$. 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 **reads** *RE* in δ be arbitrary. It is to show that for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{RE} : \mathbf{region}]\!](\sigma)$, $h[o, f] = h'[o, f]$. By the definition of separator (Fig. 3.7), for any **modifies** *RE*′ in ε, it must be true that $\sigma, h \models^{\Gamma} \mathit{RE}\ !!\ \mathit{RE}'$. By the assumption that $\sigma, h \models^{\Gamma} \delta \not/ \varepsilon$, it must be

true that $(o, f) \notin \mathcal{E}[\![\Gamma \vdash RE' : \mathbf{region}]\!](\sigma, h)$. 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 **modifies** x, and that of the second statement is **region**$\{x, f\}$. The effect of their composition is not necessarily **modifies** (x, **region**$\{x, f\}$), as **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 **modifies region**$\{y.f\}$ which is sound owing to the postcondition of x := y, which is x = y. The effect **modifies region**$\{y.f\}$ is immune from updating x. Immunity is used in the proof of Theorem 1 on page 49 .

Definition 6 (Immune)**.** *Let RE be a region expression, P be an assertion, and ε and δ be two effects. Then RE is* immune *from ε under P, written RE is P/ε-immune, if and only if P implies efs(RE)$\cdot/\cdot\varepsilon$.*

Effect δ is immune from ε under P, if and only if for all **modifies** *RE in δ :: RE is P/ε-immune.*

∎

This notion is used to prevent naive accumulation of write effects. To explain this, let ε_1 and ε_2 be the two write effects of two sequential statements. Intuitively, if the variables and regions that ε_1 contains overlap with the variables and regions that ε_2 depends on, then ε_2 is not ε_1-immune.

Consider the example *x.f := x; x.f := x;*. Assume the precondition of the first update statement is $x \neq null$. The write effects of both update statements, ε_1 and ε_2, are **modifies region**$\{x.f\}$.

The proof obligation is to show that ε_2 is $x \neq null/\varepsilon_1$-immune. Informally, the write effect ε_2 relies on the variable x. But, the write effect ε_1 does not contain **modifies** x. Therefore, **modifies region**$\{x.f\}$ is $x \neq null/$**modifies region**$\{x.f\}$-immune. A proof of this is calculated as follows.

modifies region$\{x.f\}$ is $x \neq null/$**modifies region**$\{x.f\}$-immune

iff $\langle$by the definition of Immune (Def. 6)$\rangle$

for all **modifies** *RE* in **modifies region**$\{x.f\}$ :: *RE* is

$x \neq null/$**modifies region**$\{x.f\}$-immune

iff $\langle$by *RE* is **region**$\{$x.f$\}\rangle$

region$\{x.f\}$ is $x \neq null/$**modifies region**$\{x.f\}$-immune

iff $\langle$by the definition of Immune (Def. 6)$\rangle$

$x \neq null$ *implies efs*(**region**$\{x.f\}$)$\not/$**modifies region**$\{x.f\}$

iff $\langle$by the definition of read effects (Fig. 3.5)$\rangle$

$x \neq null$ *implies* **reads** $x\not/$**modifies region**$\{x.f\}$

iff $\langle$by the definition of separator (Fig. 3.7)$\rangle$

true

However, note that if the first statement were $x := y$, then the effect **modifies region**$\{x.f\}$ would not be $x \neq null/$**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 $x \neq null \;\&\&\; x.f \neq null$. In this case, ε_1 is **modifies region**$\{x.f.g\}$, and ε_2 is **modifies region**$\{x.f\}$. The following shows that **modifies region**$\{x.f.g\}$ is $x \neq null$/**modifies region**$\{x.f\}$-immune is false.

modifies region$\{x.f.g\}$ is $x \neq null$/**modifies region**$\{x.f\}$-immune

iff $\langle$by the definition of Immune (Def. 6)$\rangle$

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

$x \neq null$/**modifies region**$\{x.f\}$-immune

iff $\langle$by *RE* is **region**$\{$x.f.g$\}\rangle$

region$\{x.f.g\}$ is $x \neq null$/**modifies region**$\{x.f\}$-immune

iff $\langle$by the definition of Immune (Def. 6)$\rangle$

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

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

$x \neq null$ *implies* (**reads** x, **region**$\{x.f\}$)$\cdot/$**modifies region**$\{x.f\}$

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

false

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

Proof. By the assumption that *RE* is P/ε-immune, and by the definition of immunity (Def. 6), it must be true that $\sigma, h \models^{\Gamma} P \Rightarrow efs(RE)\cdot/\varepsilon$. By the assumption that $(\sigma, h) \rightarrow (\sigma', h') \models^{\Gamma} \varepsilon$ and by the separator agreement (Lemma 5), it must be true that $(\sigma, h) \overset{efs(RE)}{\equiv} (\sigma', h')$. Then by the

soundness of framing for expressions (Lemma 3), it must be true that $\mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash RE : \texttt{region}]\!](\sigma')$. □

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 Γ_0, Γ_1 and Γ_2 be well-formed type environments. Let (σ_0, h_0), (σ_1, h_1), (σ_2, h_2) be Γ_0, Γ_1 and Γ_2-states respectively. Let ε_1 and ε_2 be two effects, and P and P' be two assertions. If the following hold:*

1. $\sigma_0, h_0 \models^{\Gamma_0} P$ *and* $\sigma_1, h_1 \models^{\Gamma_1} P'$*;*
2. $(\sigma_0, h_0) \rightarrow (\sigma_1, h_1) \models^{\Gamma_0} \varepsilon_1$*;*
3. $(\sigma_1, h_1) \rightarrow (\sigma_2, h_2) \models^{\Gamma_1} \varepsilon_2, \textbf{\texttt{modifies}}\ RE$*;*
4. ε_2 *is* P/ε_1*-immune;*
5. *for all* $\textbf{\texttt{fresh}}(RE') \in \varepsilon_1$ *::* RE' *is* $P/(\varepsilon_2, \textbf{\texttt{modifies}}\ RE)$*-immune;*
6. $\mathcal{E}[\![\Gamma_1 \vdash RE : \textbf{\texttt{region}}]\!](\sigma_1, h_1) \cap \sigma_0(\textbf{\texttt{alloc}}) = \varnothing$.

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

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, $\textbf{\texttt{modifies}}\ x$ is either in ε_1 or in ε_2 or both.

For condition (2) in Def. 3, let $(o, f) \in \sigma_0(\mathbf{alloc})$, 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 **modifies** RE_0 in ε_1 and $(o, f) \in \mathcal{E}[\![\Gamma_0 \vdash RE_0 : \mathbf{region}]\!](\sigma_0, h_0)$. Thus, **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 ε_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, **modifies** RE' in $(\varepsilon_1, \varepsilon_2)$.
   - $(o, f) \in \mathcal{E}[\![\Gamma_1 \vdash RE]\!](\sigma_1, h_1)$. By assumption (6), $\mathcal{E}[\![\Gamma_1 \vdash RE]\!](\sigma_1, h_1) \cap \sigma_0(\mathbf{alloc}) = \varnothing$. That implies $(o, f) \notin \sigma_0(\mathbf{alloc})$, which contradicts the assumption that $(o, f) \in \sigma_0(\mathbf{alloc})$.

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

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

□

CHAPTER 4: FINE-GRAINED REGION LOGIC[1]

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 ε, and the locations specified to be fresh in ε 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* Γ *be a well-formed type environment. Let S be a statement, let* P_1 *and* P_2 *be assertions, let* ε *be effects, and let* (σ, H) *be a* Γ*-state. Then* $\{P_1\}\, S\, \{P_2\}[\varepsilon]$ is valid in (σ, H)*, written* $\sigma, H \models^{\Gamma}_{r} \{P_1\}\, S\, \{P_2\}[\varepsilon]$*, if and only if whenever* $\sigma, H \models^{\Gamma} P_1$*, then*

1. $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) \neq$ *err;*
2. *if* $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$*, then*
   - $\sigma', H' \models^{\Gamma'} P_2$
   - *for all* $x \in dom(\sigma) : \sigma'(x) \neq \sigma(x) :$ **`modifies`** $x \in \varepsilon$
   - *for all* $(o, f) \in dom(H) : H'[o, f] \neq H[o, f] :$ $(o, f) \in \mathcal{E}[\![\Gamma \vdash writeR(\varepsilon) :$ **`region`**$]\!](\sigma)$*, and*
   - *for all* $(o, f) \in \mathcal{E}[\![\Gamma' \vdash freshR(\varepsilon) :$ **`region`**$]\!](\sigma')::\ (o, f) \in (dom(H') - dom(H))$.

[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* $\models_r^\Gamma\ \{P_1\}\ S\ \{P_2\}[\varepsilon]$*, if and only if for all* Γ*-states* $(\sigma, H) :: \sigma, H \models_r^\Gamma\ \{P_1\}\ S\ \{P_2\}[\varepsilon]$. ∎

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 **`alloc`**.

4.1 Axioms and Inference Rules

Fig. 4.1 on the following page shows the axioms and inference rules for statements. The predicate $true$ 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, $new(T, x)$, means that $x.f_1 = \mathit{default}(T_1)\ \&\&\ \ldots\ \&\&\ x.f_n = \mathit{default}(T_n)$, where the $f_i : T_i$ are defined by $(f_1 : T_1, \ldots, f_n : T_n) = \mathit{fields}(T)$.

($SKIP_r$)
$\vdash_r^\Gamma \{true\}\mathbf{skip};\{true\}[\varnothing]$

(VAR_r)
$\vdash_r^\Gamma \{true\}\mathbf{var}\ x:T;\{x = default(T)\}[\varnothing]$

($ALLOC_r$)
$\vdash_r^\Gamma \begin{array}{l}\{true\}\ x := \mathbf{new}\ T;\ \{new(T,x)\} \\ [\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})]\end{array}$

($ASGN_r$)
$\vdash_r^\Gamma \{true\}\ x := G;\ \{x = G\}\ [\mathbf{modifies}\ x]$ **where** $x \notin \mathrm{FV}(G)$

(ACC_r)
$\vdash_r^\Gamma \{x' \neq null\}\ x := x'.f;\ \{x = x'.f\}\ [\mathbf{modifies}\ x]$ **where** $x \neq x'$

(UPD_r)
$\vdash_r^\Gamma \{x \neq null\}\ x.f := G;\ \{x.f = G\}[\mathbf{modifies}\ \mathbf{region}\{x.f\}]$

($SEQ1_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S_1\ \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \qquad \vdash_r^{\Gamma'} \{P_1\}\ S_2\ \{P'\}[\varepsilon_2, \mathbf{modifies}\ RE_1]}{\vdash_r^\Gamma \{P\}\ S_1S_2\ \{P'\}[\ \varepsilon_1, \varepsilon_2, \mathbf{fresh}(RE)]}$$
where $S_1 \neq \mathbf{var}\ x:T;$, ε_1 is fresh-free, ε_2 is P/ε_1-immune,
RE is $P_1/(\varepsilon_2, \mathbf{modifies}\ \ RE_1)$-immune and $P_1 \Rightarrow RE_1 \leqslant RE$

($SEQ2_r$)
$$\frac{\vdash_r^{\Gamma,\ x:T} \{P\ \&\&\ x = default(T)\}\ S\ \{Q\}[\mathbf{modifies}\ x, \varepsilon]}{\vdash_r^\Gamma \{P\}\ \mathbf{var}\ x:T;S\ \{Q\}[\varepsilon]}$$

(IF_r)
$$\frac{\vdash_r^\Gamma \{P\ \&\&\ E\}\ S_1\ \{P'\}[\varepsilon] \qquad \vdash_r^\Gamma \{P\ \&\&\ \neg E\}\ S_2\ \{P'\}[\varepsilon]}{\vdash_r^\Gamma \{P\}\mathbf{if}\ E\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}\{P'\}[\varepsilon]}$$

($WHILE_r$)
$$\frac{\vdash_r^\Gamma \{P\ \ \&\&\ \ E\}\ S\ \{P\}\ [\varepsilon, \mathbf{modifies}\ RE]}{\vdash_r^\Gamma \{P\ \&\&\ r = \mathbf{alloc}\}\ \mathbf{while}\ E\ \{S\}\ \{P\ \ \&\&\ \neg E\}\ [\varepsilon]},$$
where $P \Rightarrow RE\ !!\ r, \varepsilon$ is fresh-free, ε is P/ε-immune, and $\mathbf{modifies}\ r \notin \varepsilon$

Figure 4.1: The correctness axioms and proof rules for statements in FRL

(FRM_r)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \delta\ \mathit{frm}\ Q}{\vdash_r^\Gamma \{P \mathbin{\&\&} Q\}\ S\ \{P' \mathbin{\&\&} Q\}[\varepsilon]}$$
where $P \mathbin{\&\&} Q \Rightarrow \delta \diagup \varepsilon$

($SUBEFF_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \varepsilon \leqslant \varepsilon'}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon']}$$

($CONSEQ_r$)
$$\frac{\vdash_r^\Gamma \{P_1\}\ S\ \{P_1'\}[\varepsilon]}{\vdash_r^\Gamma \{P_2\}\ S\{P_2'\}[\varepsilon]}$$
where $P_2 \Rightarrow P_1$ and $P_1' \Rightarrow P_2'$

($ConEff_r$)
$$\frac{\vdash_r^\Gamma \{P \mathbin{\&\&} E\}\ S\ \{P'\}[\varepsilon_1] \qquad \vdash_r^\Gamma \{P \mathbin{\&\&} \neg E\}\ S\ \{P'\}[\varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

($ConMask1_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \mathbf{modifies}\ (E\ ?\ \varepsilon_1\ :\ \varepsilon_2)]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_1]}$$
where $P \Rightarrow E$ and $\mathbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

($ConMask2_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \mathbf{modifies}\ (E\ ?\ \varepsilon_1\ :\ \varepsilon_2)]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_2]}$$
where $P \Rightarrow \neg E$ and $\mathbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

($PostToFr_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}$$
where $P \Rightarrow r = \mathbf{alloc}, P \Rightarrow (E \mathbin{\&\&} RE_1\ !!\ r), P \Rightarrow (\neg E \mathbin{\&\&} RE_2\ !!\ r)$ and
$\mathbf{modifies}\ r \notin \varepsilon$

Figure 4.2: The structural rules in FRL (1)

($FrToPost_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}{\vdash_r^\Gamma \begin{array}{l}\{P\}\ S\ \{P'\ \&\&\ (b \Rightarrow RE_1\ !!\ r)\ \&\&\ (\neg b \Rightarrow RE_2\ !!\ r)\} \\ [\varepsilon, E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]\end{array}}$$
where $P \Rightarrow b = E, P \Rightarrow b, P \Rightarrow r = \mathbf{alloc}, \mathbf{modifies}\ b \notin \varepsilon$ and $\mathbf{modifies}\ r \notin \varepsilon$

($VarMask1_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[E\ ?\ \mathbf{modifies}\ x, \varepsilon_1 : \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow E, P \mid\mid P' \Rightarrow x = y$ and $P\ \&\&\ E \Rightarrow \mathbf{reads}\ y / (x, \varepsilon)$

($VarMask2_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ (\mathbf{modifies}\ x, \varepsilon_2)]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\mathbf{if}\ E\ \mathbf{then}\ \varepsilon_1\ \mathbf{else}\ \varepsilon_2]}$$
where $P \Rightarrow \neg E, P \mid\mid P' \Rightarrow x = y$ and $P\ \&\&\ \neg E \Rightarrow \mathbf{reads}\ y / (x, \varepsilon)$

($FieldMask1_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ (\mathbf{modifies}\ \mathbf{region}\{x.f\}, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow E, P \mid\mid P' \Rightarrow x.f = y, P'\ \&\&\ E \Rightarrow \mathbf{reads}\ x / \mathbf{modifies}\ \varepsilon,$
$P'\ \&\&\ E \Rightarrow \mathbf{reads}\ y / \mathbf{modifies}\ \varepsilon$

($FieldMask2_r$)
$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \varepsilon_1\ :\ \mathbf{modifies}\ \mathbf{region}\{x.f\}, \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow \neg E, P \mid\mid P' \Rightarrow x.f = y, P'\ \&\&\ \neg E \Rightarrow \mathbf{reads}\ x / \mathbf{modifies}\ \varepsilon$
and $P'\ \&\&\ \neg E \Rightarrow \mathbf{reads}\ y / \mathbf{modifies}\ \varepsilon$

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_1S_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: $x :=$ **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

$$\vdash_r^\Gamma \begin{array}{l} \{true\}\texttt{x := new T;}\{new(T,x)\} \\ [\,\textbf{modifies}\ x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \end{array} \tag{4.1}$$

$$\vdash_r^\Gamma \{x \neq null\}x.f := 5;\{x.f = 5\}[\,\textbf{modifies region}\{x.f\}] \tag{4.2}$$

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

$$\vdash_r^\Gamma \{x \neq null\}x.f := 5;\{x.f = 5\}[\,\textbf{modifies region}\{x.*\}] \tag{4.3}$$

Then, using the *CONSEQ*$_r$ rule on Eq. (4.1), the following is derived

$$\vdash_r^\Gamma \begin{array}{l} \{true\}\texttt{x := new T;}\{x \neq null\} \\ [\,\textbf{modifies}\ x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \end{array} \tag{4.4}$$

In order to use the rule *SEQ1*$_r$ on Eq. (4.4) and Eq. (4.3), the rule *SEQ1*$_r$ is instantiated with *RE* := **region**$\{x.*\}$, *RE*$_1$:= **region**$\{x.*\}$, ε_1 := **modifies** x,**modifies alloc** and $\varepsilon_2 := \varnothing$. Then, the immune side conditions has to be true, which are:

$$\textbf{modifies}\ x \text{ is } true/\textbf{modifies}\ x, \textbf{modifies alloc}\text{-immune} \tag{4.5}$$

and

$$\varnothing \text{ is } true/\textbf{modifies}\ x, \textbf{modifies alloc}\text{-immune} \tag{4.6}$$

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

$$\textit{for all}\ \textbf{modifies}\ RE \in (\textbf{modifies}\ x) :: RE \text{ is}$$
$$true/\textbf{modifies}\ x, \textbf{modifies}\ \textbf{alloc}\text{-immune} \quad (4.7)$$

Eq. (4.7) is vacuously true, since no region expression *RE* can be a variable x. Now, using the rule $SEQ1_r$, the following is derived

$$\vdash_r^\Gamma \begin{array}{l} \{true\}x := \textbf{new}\ T; x.f := 5; \{x.f = 5\} \\ [\,\textbf{modifies}\ x, \textbf{modifies}\ \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \end{array}$$

In this case, the write effect of the second statement, **modifies region**$\{x.*\}$, 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:

$$\frac{\vdash_r^\Gamma \{P\}\ S_1\ \{P_1\}[\varepsilon_1] \qquad \vdash_r^\Gamma \{P_1\}\ S_2\ \{P'\}[\varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S_1 S_2\ \{P'\}[\varepsilon_1, \varepsilon_2]}$$

where ε_1 is fresh-free and ε_2 is P/ε_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 ε_1 contains have to be disjoint with those that ε_2 depends on. Examples have been given in Section 3.4 in Chapter 3. Similarly, for the read effect, variables and regions that ε_1 contains have to be disjoint with those that δ_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 ε_2 is P/ε_1-immune, where ε_1 and ε_2 are both **modifies region**$\{x.f\}$, and P is $x \neq null$. Here shows that δ_2 is P/ε_1-immune as follows, where δ_2 is **reads** x.

reads x is $x \neq null$/**modifies region**$\{x.f\}$-immune

iff $\langle$by the definition of Immune (Def. 6)$\rangle$

for all **reads** *RE* $\in$ **reads** x :: *RE* is $x \neq null$/**modifies region**$\{x.f\}$-immune

iff $\langle$by there does not exist such *RE*$\rangle$

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

$$\vdash_r^{\Gamma} \{true\}\mathbf{var}\ y : \mathbf{int}; \{y = 0\}[\varnothing] \tag{4.8}$$

After using the axiom *ASGN*$_r$, the following is derived

$$\vdash_r^{\Gamma} \{true\}\ y := 5;\ \{y = 5\}\ [\mathbf{modifies}\ y] \tag{4.9}$$

By the rule *CONSEQ*$_r$ on Eq. (4.9), the following is derived

$$\vdash_r^{\Gamma} \{y = 0\}y := 5;\ \{y = 5\}\ [\mathbf{modifies}\ y] \tag{4.10}$$

Using the rule *SEQ2*$_r$ on Eq. (4.8) and Eq. (4.10), the following is derived $\vdash_r^{\Gamma} \{true\}\mathbf{var}\ y : \mathbf{int}; y := 5;\ \{y = 5\}\ [\varnothing]$

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 $\Gamma = \texttt{alloc} : \texttt{region}, f : \texttt{region}, y : \texttt{int}, x : C$:

$$B \stackrel{\text{def}}{=} x := \texttt{new}\, T;\; f := f + \texttt{region}\{x.*\};\; y := y - 1;$$
$$S \stackrel{\text{def}}{=} f := \texttt{region}\{\};\; y := 5;\; \texttt{while}\, y\, \{B\}$$

The proof obligation is to show that

$$\vdash_r^{\Gamma} \begin{array}{l} \{true\}\, S\, \{y = 0\} \\ [\texttt{modifies}\, f, \texttt{modifies alloc}, \texttt{modifies}\, x, \texttt{modifies}\, y, \texttt{fresh}(f)] \end{array} \tag{4.11}$$

After using the axiom *ASGN*$_r$, once for each of the following, the following is derived

$$\vdash_r^{\Gamma} \{true\}\, f := \texttt{region}\{\}; \{f = \texttt{region}\{\}\}[\texttt{modifies}\, f] \tag{4.12}$$

$$\vdash_r^{\Gamma} \{true\}\, y := 5; \{y = 5\}[\texttt{modifies}\, y] \tag{4.13}$$

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

$$\vdash_r^{\Gamma} \{f = \texttt{region}\{\}\}\, y := 5; \{f = \texttt{region}\{\}\, \&\&\, y = 5\}[\texttt{modifies}\, y] \tag{4.14}$$

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

$$\vdash_r^{\Gamma} \{true\}\, f := \texttt{region}\{\};\; y := 5; \{f = \texttt{region}\{\}\, \&\&\, y = 5\}[\texttt{modifies}\, f, \texttt{modifies}\, y] \tag{4.15}$$

Now, consider the loop. Let variable g be fresh; g is used to snapshot the initial value of **alloc**. For the loop body B, the proof obligation is to derive

$$\vdash_r^\Gamma \{g \,!!\, f\} B \{g \,!!\, f\} [\textbf{modifies} f, \textbf{modifies} x, \textbf{modifies} y, \textbf{modifiesalloc}] \quad (4.16)$$

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

$$\begin{array}{l} \vdash_r^\Gamma \quad \{g \,!!\, f \;\&\&\; g = \textbf{alloc}\}\, \textbf{while}\, y\, \{B\}\, \{g \,!!\, f \;\&\&\; y = 0\} \\ \qquad [\,\textbf{modifies}\, x, \textbf{modifies}\, y, \textbf{modifies}\, f, \textbf{modifies alloc}\,] \end{array} \quad (4.17)$$

The rule *PostToFr*$_r$ is instantiated with $r := g$ and *RE* $:= f$. The following is derived.

$$\begin{array}{l} \vdash_r^\Gamma \quad \{g \,!!\, f \;\&\&\; g = \textbf{alloc}\}\, \textbf{while}\, y\, \{B\}\, \{g \,!!\, f \;\&\&\; y = 0\} \\ \qquad [\,\textbf{modifies}\, x, \textbf{modifies}\, y, \textbf{modifies}\, f, \textbf{modifies alloc}, \textbf{fresh}(f)] \end{array} \quad (4.18)$$

After using the rule *CONSEQ*$_r$ from the above, the following is derived.

$$\begin{array}{l} \vdash_r^\Gamma \quad \{g \,!!\, f \;\&\&\; g = \textbf{alloc}\}\, \textbf{while}\, y\, \{B\}\, \{y = 0\} \\ \qquad [\,\textbf{modifies}\, x, \textbf{modifies}\, y, \textbf{modifies}\, f, \textbf{modifies alloc}, \textbf{fresh}(f)] \end{array} \quad (4.19)$$

The postcondition of Eq. (4.15) implies the precondition of Eq. (4.19). After using the rule *CONSEQ*$_r$ on Eq. (4.15), the following is derived.

$$\begin{array}{l} \vdash_r^\Gamma \quad \{true\}\, f := \textbf{region}\{\};\; y := 5;\; \{g \,!!\, f \;\&\&\; g = \textbf{alloc}\} \\ \qquad [\,\textbf{modifies}\, f, \textbf{modifies}\, y] \end{array} \quad (4.20)$$

From Eq. (4.20) and Eq. (4.19), the rule *SEQ1*$_r$ is instantiated with *RE* $=$ **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 *ALLOC*$_r$, the following is derived.

$$\begin{array}{l} \vdash_r^\Gamma \quad \{true\}\, x := \textbf{new}\, T;\; \{new(T, x)\} \\ \qquad [\,\textbf{modifies}\, x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \end{array} \quad (4.21)$$

Then by the rule *FRM*$_r$ from the above, the following is derived.

$$\vdash_r^\Gamma \begin{array}{l} \{g \;!!\; f\}\; x := \mathbf{new}\, T;\; \{new(T,x)\;\&\&\; g \;!!\; f\} \\ [\,\mathbf{modifies}\; x, \mathbf{modifies\ alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{4.22}$$

The rule *FrToPost*$_r$ is instantiated with $r := g$ and *RE* = **region**$\{x.*\}$. And **reads** $g \not\cdot$ (**modifies** x, **modifies alloc**) is true. After applying the rule, the following is derived.

$$\vdash_r^\Gamma \begin{array}{l} \{g \;!!\; f\}\; x := \mathbf{new}\, T;\; \{new(T,x)\;\&\&\; g \;!!\; f\;\&\&\; g \;!!\; \mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\; x, \mathbf{modifies\ alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{4.23}$$

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

$$\vdash_r^\Gamma \{true\}\; f := f' + \mathbf{region}\{x.*\};\; \{f = f' + \mathbf{region}\{x.*\}\}\; [\,\mathbf{modifies}\; f] \tag{4.24}$$

After using the rule *FRM*$_r$, the following is derived.

$$\vdash_r^\Gamma \begin{array}{l} \{new(T,x)\;\&\&\; g \;!!\; f\;\&\&\; g \;!!\; \mathbf{region}\{x.*\}\} \\ f := f' + \mathbf{region}\{x.*\}; \\ \{f = f' + \mathbf{region}\{x.*\}\;\&\&\; new(T,x)\;\&\&\; g \;!!\; f\;\&\&\; g \;!!\; \mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\; f] \end{array} \tag{4.25}$$

From Eq. (4.23) and Eq. (4.25), the rule *SEQ1*$_r$ is instantiated with *RE* = **region**$\{\}$. As the immunity conditions are vacuously true, the following is derived.

$$\vdash_r^\Gamma \begin{array}{l} \{g \;!!\; f\} \\ x := \mathbf{new}\, T;\; f := f' + \mathbf{region}\{x.*\}; \\ \{f = f' + \mathbf{region}\{x.*\}\;\&\&\; new(T,x)\;\&\&\; g \;!!\; f\;\&\&\; g \;!!\; \mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\; f, \mathbf{modifies}\; x, \mathbf{modifies\ alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{4.26}$$

Then by the rules *CONSEQ*$_r$ and *SubEff*$_r$, the following is derived.

$$\vdash_r^{\Gamma} \begin{array}{l} \{g \;!!\; f\}\, x := \mathbf{new}\, T;\; f := f' + \mathbf{region}\{x.*\}; \{g \;!!\; f\} \\ [\,\mathbf{modifies}\; f, \mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc}] \end{array} \tag{4.27}$$

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. $\vdash_r^{\Gamma} \{true\}\; y := y' - 1; \{y = y' - 1\}[\,\mathbf{modifies}\; f]$ Then, by the rules *FRM*$_r$, *SEQ1*$_r$ and *CONSEQ*$_r$, Eq. (4.16) is derived.

4.2 Soundness

Theorem 1. *The judgment* $\vdash_r^{\Gamma} \{P\}S\{Q\}[\varepsilon]$ *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, 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[1]

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 $[\delta]\{P_1\}S\{P_2\}[\varepsilon]$, where δ are read effects (on the heap) and ε are write effects; thus (ε, δ) contains all the heap locations that *S* may access. Note that δ and ε 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 Γ be a well-formed type environment. Let S be a statement. Let P_1 and P_2 be assertions. Let ε be effects and δ be read effects, let (σ, H) be a Γ-state. Then* $[\delta]\{P_1\}S\{P_2\}[\varepsilon]$ is valid *in (σ, H), written $\sigma, H \models_u^\Gamma [\delta]\{P_1\}S\{P_2\}[\varepsilon]$, if and only if whenever $\sigma, H \models^\Gamma P_1$, then:*

1. $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H \restriction \mathcal{E}[\![\Gamma \vdash regRW(\varepsilon, \delta) : \texttt{region}]\!](\sigma)) \neq err$, *and*
2. *if* $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H \restriction \mathcal{E}[\![\Gamma \vdash regRW(\varepsilon, \delta) : \texttt{region}]\!](\sigma))$, *then the following all hold:*
   - $\sigma', H' \models^{\Gamma'} P_2$,

[1]The content in this chapter is submitted to *Formal Aspects of Computing*.

- *for all* $x \in dom(\sigma) : \sigma'(x) \neq \sigma(x) :$ **modifies** $x \in \varepsilon$,
- *for all* $(o, f) \in dom(H) : H'[o, f] \neq H[o, f] :$
$$(o, f) \in \mathcal{E}[\![\Gamma \vdash writeR(\varepsilon) : \textbf{region}]\!](\sigma), \text{ and}$$
- *for all* $(o, f) \in \mathcal{E}[\![\Gamma' \vdash freshR(\varepsilon) : \textbf{region}]\!](\sigma') :: (o, f) \in (dom(H') - dom(H))$.

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

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

$$[\textbf{reads region}\{x.f\}]\{x \neq null\}y := x.f; \{y = x.f\}[\textbf{modifies } y]. \tag{5.1}$$

Eq. (5.1) is a valid UFRL Hoare-formula, because $regRW(\textbf{reads region}\{x.f\}, \textbf{modifies } y) = \textbf{region}\{x.f\}$. The region $\textbf{region}\{x.f\}$ 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 $[\varnothing]\{x \neq null\}y := x.f; \{y = x.f\}[\textbf{modifies } y]$ is not a valid UFRL Hoare-formula, as $regRW(\textbf{reads } \varnothing, \textbf{modifies } y) = \textbf{region}\{\}$. As another example, consider the following formula:

$$[\varnothing]\{x \neq null\}x.f := y; \{x.f = y\}[\textbf{modifies region}\{x.f\}]. \tag{5.2}$$

Eq. (5.2) is a valid UFRL Hoare-formula, because $regRW(\varnothing, \textbf{modifies region}\{x.f\}) = \textbf{region}\{x.f\}$.

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].

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 (δ and η) 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.

$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, \texttt{fresh}(RE)]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\ \&\&\ r\ !!\ RE\}[\varepsilon]}\ \textit{where}\ P \Rightarrow r = \texttt{alloc}$$

This uses the subeffect rule, because $regRW(\delta, \texttt{fresh}(RE), \varepsilon) \leqslant regRW(\delta, \varepsilon)$, and *regRW* ignores fresh effects.

($SKIP_u$)
$\vdash_u^\Gamma [\varnothing]\{true\}\mathbf{skip};\{true\}[\varnothing]$

(VAR_u)
$\vdash_u^\Gamma [\varnothing]\{true\}\mathbf{var}\ x:T;\{x = \mathit{default}(T)\}[\varnothing]$

($ALLOC_u$)
$$\vdash_u^\Gamma \begin{array}{l}[\varnothing]\\ \{true\}\ x := \mathbf{new}\ T;\ \{new(T,x)\}\\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})]\end{array}$$

($ASGN_u$)
$\vdash_u^\Gamma [\eta]\{true\}\ x := G;\ \{x = G\}\ [\,\mathbf{modifies}\ x]$ **where** $x \notin \mathrm{FV}(G)$ and $\eta = \mathit{efs}(G)$

(UPD_u)
$\vdash_u^\Gamma [x,\eta]\ \{x \neq null\}\ x.f := G;\ \{x.f = G\}[\,\mathbf{modifies}\ \mathbf{region}\{x.f\}]$**where** $\eta = \mathit{efs}(G)$

(ACC_u)
$\vdash_u^\Gamma [\eta]\{x' \neq null\}\ x := x'.f;\ \{x = x'.f\}\ [\,\mathbf{modifies}\ x]$, **where** $x \neq x'$ and $\eta = \mathit{efs}(x'.f)$

(IF_u)
$$\frac{\vdash_u^\Gamma [\delta]\{P\ \&\&\ E\}\ S_1\{Q\}\ [\varepsilon] \qquad \vdash_u^\Gamma [\delta]\{P\ \&\&\ \neg E\}\ S_2\{Q\}\ [\varepsilon]}{\vdash_u^\Gamma [\delta,\delta_E]\{P\}\ \mathbf{if}\ E\ \{S_1\}\mathbf{else}\{S_2\}\{Q\}\ [\varepsilon]} \quad \textbf{where } \delta_E = \mathit{efs}(E)$$

($SEQ1_u$)
$$\frac{\vdash_u^\Gamma [\delta_1]\{P\}\ S_1\ \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \qquad \vdash_u^{\Gamma'} [\delta_2, \mathbf{reads}\ RE_1]\{P_1\}\ S_2\ \{P'\}[\,\varepsilon_2, \mathbf{modifies}\ RE_2]}{\vdash_u^\Gamma [\delta_1,\delta_2]\{P\}\ S_1S_2\ \{P'\}[\varepsilon_1,\varepsilon_2,\mathbf{fresh}(RE)]}$$
where $S_1 \neq \mathbf{var}\ x:T;$, ε_1 is fresh-free, δ_2 is P/ε_1-immune, ε_2 is P/ε_1-immune, RE is $P_1/(\mathbf{modifies}\ \ RE_2,\varepsilon_2)$-immune, $RE_1 \leqslant RE$ *and* $RE_2 \leqslant RE$

($SEQ2_u$)
$$\frac{\vdash_u^{\Gamma,\ x:T} [\delta, \mathbf{reads}\ x]\{P\ \&\&\ x = \mathit{default}(T)\}\ S\ \{Q\}[\,\mathbf{modifies}\ x,\varepsilon]}{\vdash_u^\Gamma [\delta]\{P\}\ \mathbf{var}\ x:T;S\ \{Q\}[\varepsilon]}$$

($WHILE_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\ \&\&\ E\}\ S\{P\}\ [\varepsilon,\mathbf{modifies}\ RE]}{\vdash_u^\Gamma [\delta,\delta_E]\{P\ \&\&\ r = \mathbf{alloc}\}\ \mathbf{while}\ E\ \{S\}\ \{P\ \&\&\ \neg E\}\ [\varepsilon]}$$
where $\delta_E = \mathit{efs}(E), P \Rightarrow RE\ !!\ r, \varepsilon$ is fresh-free, $\mathbf{modifies}\ r \notin \varepsilon, \delta$ is P/ε-immune *and* ε is P/ε-immune

Figure 5.1: The correctness axioms and proof rules for statements in UFRL

(FRM_u)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \eta\, frm\, Q}{\vdash_u^\Gamma [\delta]\{P \,\&\&\, Q\}\ S\ \{P' \,\&\&\, Q\}[\varepsilon]} \quad \textbf{where } P \,\&\&\, Q \Rightarrow \eta / \varepsilon$$

($SubEff_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P_1\}S\{P_2\}[\varepsilon] \qquad \vdash_{P_1} \varepsilon \leqslant \varepsilon'}{\vdash_u^\Gamma [\delta']\{P_1\}S\{P_2\}[\varepsilon']} \quad \textbf{where } P_1 \Rightarrow \mathit{regRW}(\varepsilon, \delta) \leqslant \mathit{regRW}(\varepsilon', \delta')$$

($CONSEQ_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P_1\}\ S\ \{P_1'\}[\varepsilon]}{\vdash_u^\Gamma [\delta]\{P_2\}\ S\{P_2'\}[\varepsilon]} \quad \textbf{where } P_2 \Rightarrow P_1 \text{ and } P_1' \Rightarrow P_2'$$

($ConEff_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P \,\&\&\, E\}\ S\ \{P'\}[\varepsilon_1] \qquad \vdash_u^\Gamma [\delta]\{P \,\&\&\, \neg E\}\ S\ \{P'\}[\varepsilon_2]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

($ConMask1_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_1]} \quad \textbf{where } P \Rightarrow E$$

($ConMask2_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_2]} \quad \textbf{where } P \Rightarrow \neg E$$

($PostToFr_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \texttt{fresh}(RE_1)\ :\ \texttt{fresh}(RE_2)]}$$
where $P \Rightarrow (E \,\&\&\, RE_1\ !!\ \texttt{alloc})$ and $P \Rightarrow (\neg E \wedge RE_2\ !!\ \texttt{alloc})$

($FrToPost_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon, E\ ?\ \texttt{fresh}(RE_1)\ :\ \texttt{fresh}(RE_2)]}{\vdash_u^\Gamma \begin{array}{l}[\delta] \\ \{P\}\ S\ \{P' \,\&\&\, (b \Rightarrow RE_1\ !!\ r) \,\&\&\, (\neg b \Rightarrow RE_2\ !!\ r)\} \\ [\varepsilon, E\ ?\ \texttt{fresh}(RE_1)\ :\ \texttt{fresh}(RE_2)]\end{array}}$$
where $P \Rightarrow b = E, P \Rightarrow r = \texttt{alloc}, P \Rightarrow E, \texttt{modifies}\ b \notin \varepsilon$ and $\texttt{modifies}\ r \notin \varepsilon$

Figure 5.2: The structural rules in UFRL (1)

($VarMask1_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\ E\ ?\ (\textbf{modifies}\ x, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow E, P \mid\mid P' \Rightarrow x = y$ and $P\ \&\&\ E \Rightarrow \textbf{reads}\ y \not\cdot (x, \varepsilon)$,

($VarMask2_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ (\textbf{modifies}\ x, \varepsilon_2)]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow \neg E, P \mid\mid P' \Rightarrow x = y$ and $P\ \&\&\ \neg E \Rightarrow \textbf{reads}\ y \not\cdot (x, \varepsilon)$

($FieldMask1_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ (\textbf{modifies region}\{x.f\}, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow E, P \mid\mid P' \Rightarrow x.f = y, P'\ \&\&\ E \Rightarrow \textbf{reads}\ x \not\cdot \textbf{modifies}\ \varepsilon$
and $P'\ \&\&\ E \Rightarrow \textbf{reads}\ y \not\cdot \textbf{modifies}\ \varepsilon$

($FieldMask2_u$)
$$\frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ (\textbf{modifies region}\{x.f\}, \varepsilon_2)]}{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$
where $P \Rightarrow \neg E, P \mid\mid P' \Rightarrow x.f = y, P'\ \&\&\ \neg E \Rightarrow \textbf{reads}\ x \not\cdot \textbf{modifies}\ \varepsilon$
and $P'\ \&\&\ \neg E \Rightarrow \textbf{reads}\ y \not\cdot \textbf{modifies}\ \varepsilon$

Figure 5.3: The structural rules in UFRL (2)

5.1.1 The Sequence Rules

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: $x := \textbf{new}\ T; y := x.f$, 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:

$$\begin{array}{ll} & [\varnothing] \\ \vdash_u^\Gamma & \{true\}x := \mathbf{new}\, T; \{new(T, x)\} \\ & [\,\mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{5.3}$$

$$\vdash_u^\Gamma [\mathbf{reads}\; x, \mathbf{region}\{x.f\}]\{x \neq null\}y := x.f; \{y = x.f\}[\,\mathbf{modifies}\; y] \tag{5.4}$$

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

$$\vdash_u^\Gamma [\mathbf{reads}\; x, \mathbf{region}\{x.*\}]\{x \neq null\}y := x.f; \{y = x.f\}[\,\mathbf{modifies}\; y] \tag{5.5}$$

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

$$\begin{array}{ll} & [\varnothing] \\ \vdash_u^\Gamma & \{true\}x := \mathbf{new}\, T; \{x \neq null\} \\ & [\,\mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{5.6}$$

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

$$\mathbf{reads}\; x \text{ is } true/(\mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc})\text{-immune} \tag{5.7}$$

and

$$\mathbf{modifies}\; y \text{ is } true/(\mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc})\text{-immune} \tag{5.8}$$

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

$$\begin{array}{l} \textit{for all}\; \mathbf{reads}\; RE \in (\mathbf{reads}\; x) :: RE \text{ is} \\ \qquad true/(\mathbf{modifies}\; x, \mathbf{modifies}\; \mathbf{alloc})\text{-immune} \end{array} \tag{5.9}$$

and

for all **modifies** *RE* $\in$ (**modifies** y) :: *RE* is

$$true/(\textbf{modifies}\ x, \textbf{modifies}\ \textbf{alloc})\text{-immune} \quad (5.10)$$

Eq. (5.9) and Eq. (5.10) are vacuously true. Now, using the rule $SEQ1_u$, the following is derived

$$\begin{array}{ll} & [\textbf{reads}\ x] \\ \vdash_u^\Gamma & \{true\}\texttt{x := new T; y := x.f;}\{y = x.f\} \\ & [\textbf{modifies}\ x, \textbf{modifies}\ \textbf{alloc}, \textbf{modifies}\ y, \textbf{fresh}(\textbf{region}\{x.*\})] \end{array}$$

In this case, the **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* $\vdash_u^\Gamma [\delta]\{P\}S\{Q\}[\varepsilon]$ *that 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 (σ, h) be Γ-state. Assume $\vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$ and $\sigma, h \models^\Gamma P$. Then it needs to be true that $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, h\restriction\mathcal{E}[\![\Gamma \vdash regRW(\varepsilon, \delta) : \textbf{region}]\!](\sigma)) \neq err$.

1. ($SKIP_u$) In this case, S is **skip**;, P is $true$, and δ and ε are both $\varnothing$. As it is known that $h\restriction\mathcal{E}[\![\Gamma \vdash regRW(\varepsilon, \delta) : \textbf{region}]\!](\sigma) = \varnothing$, by the program semantics Fig. 2.4, it must be true that $\mathcal{MS}[\![\Gamma \vdash \textbf{skip}; : ok(\Gamma)]\!](\sigma, \varnothing) \neq err$.

2. (VAR_u) In this case, S is **var** $x : T;$, P is $true$, and δ and ε are both $\varnothing$. As it is known that $h\restriction\mathcal{E}[\![\Gamma \vdash \mathit{regRW}(\varepsilon, \delta) : \mathbf{region}]\!](\sigma) = \varnothing$, by the program semantics Fig. 2.4, it must be true that $\mathcal{MS}[\![\Gamma \vdash \mathbf{var}\ x : T; : ok(\Gamma, x : T)]\!](\sigma, \varnothing) \neq \mathit{err}$.

3. ($ALLOC_u$) In this case, S is $x := \mathbf{new}\, T;$, P is $true$ and ε and δ are both $\varnothing$. As it is known that $h\restriction\mathcal{E}[\![\Gamma \vdash \mathit{regRW}(\varepsilon, \delta) : \mathbf{region}]\!](\sigma) = \varnothing$, by the program semantics Fig. 2.4, it must be true that $\mathcal{MS}[\![\Gamma \vdash x := \mathbf{new}\, T; : ok(\Gamma)]\!](\sigma, \varnothing) \neq \mathit{err}$.

4. ($ASSGN_u$) In this case, S is $x := G;$, P is $x = x'$ and δ is $\mathit{efs}(G)$ and ε is **modifies** x, where $x \notin \mathtt{FV}(G)$. Since it is known that $h\restriction\mathcal{E}[\![\Gamma \vdash \mathit{regRW}(\varepsilon, \delta) : \mathbf{region}]\!](\sigma) = \varnothing$, by the program semantics Fig. 2.4 on page 23, it must be true that $\mathcal{MS}[\![\Gamma \vdash x := G; : ok(\Gamma)]\!](\sigma, \varnothing) \neq \mathit{err}$.

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

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

Other inductive cases follow inductive hypotheses.

□

CHAPTER 6: Interoperability[1]

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 **`reads alloc`**$\downarrow$.

Lemma 8. *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 let (σ, H) be a Γ-state. Then*

$$\sigma, H \models_r^\Gamma \{P_1\}S\{P_2\}[\varepsilon] \textit{ iff } \sigma, H \models_u^\Gamma [\textbf{\texttt{reads alloc}}\downarrow]\{P_1\}S\{P_2\}[\varepsilon].$$

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

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

iff $\langle$by the definition of FRL valid Hoare-formula (Def. 7).$\rangle$

[1]The content in this chapter is submitted to *Formal Aspect of Computing*.

$\sigma, H \models^{\Gamma} P_1$ *implies* $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) \neq \mathit{err}$ and

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$, then $\sigma', H' \models^{\Gamma'} P_2$ and

(*for all* $x \in \mathit{dom}(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* **modifies** $x \in \varepsilon$) and

(*for all* $(o, f) \in \mathit{dom}(H) :: (H'[o, f] \neq H[o, f]$ *implies*

$(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{writeR}(\varepsilon) : \mathbf{region}]\!](\sigma))$ and

(*for all* $(o, f) \in \mathcal{E}[\![\Gamma' \vdash \mathit{freshR}(\varepsilon) : \mathbf{region}]\!](\sigma') :: (o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))))$

iff $\langle$by $H = H{\restriction}\mathit{dom}(H), \mathit{dom}(H) = \mathcal{E}[\![\Gamma \vdash \mathit{regRW}(\varepsilon, \mathbf{alloc}{\downarrow}) : \mathbf{region}]\!](\sigma)\rangle$

$\sigma, H \models^{\Gamma} P_1$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) \neq \mathit{err}$ and if

$(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{regRW}(\varepsilon, \mathbf{alloc}{\downarrow}) : \mathbf{region}]\!](\sigma))$,

then $(\sigma', H' \models^{\Gamma'} P_2)$ and (*for all* $x \in \mathit{dom}(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* **modifies** $x \in \varepsilon$) and

(*for all* $(o, f) \in \mathit{dom}(H) :: (H'[o, f] \neq H[o, f]$ *implies*

$(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{writeR}(\varepsilon) : \mathbf{region}]\!](\sigma))$) and

(*for all* $(o, f) \in \mathcal{E}[\![\Gamma' \vdash \mathit{freshR}(\varepsilon) : \mathbf{region}]\!](\sigma') :: (o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))))$

iff $\langle$by the definition of UFRL valid Hoare-formula (Def. 8)$\rangle$

$\sigma, H \models_u^{\Gamma} \{P_1\}\, S\, \{P_2\}[\varepsilon][\mathbf{reads\ alloc}{\downarrow}]$

□

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*

$$\sigma, H \models_u^{\Gamma} [\eta]\{P_1\}S\{P_2\}[\varepsilon] \textit{ implies } \sigma, H \models_r^{\Gamma} \{P_1\}S\{P_2\}[\varepsilon].$$

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* $\mathtt{TR}_R[\![-]\!]$ *from FRL rules*

to those of UFRL is defined below:

For the FRL axioms: $TR_R[\![\vdash_r^\Gamma \{P_1\}\, S\, \{P_2\}[\varepsilon]]\!] \;=\; \vdash_u^\Gamma [\mathbf{reads\ alloc}\downarrow]\{P_1\}\, S\, \{P_2\}\, [\varepsilon]$.

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:*

$$TR_R[\![\frac{\vdash_r^\Gamma h_1 \ldots \vdash_r^\Gamma h_n}{\vdash_r^\Gamma c}]\!] \;=\; \frac{TR_R[\![\vdash_r^\Gamma h_1]\!] \ldots TR_R[\![\vdash_r^\Gamma h_n]\!]}{TR_R[\![\vdash_r^\Gamma c]\!]}$$

∎

Theorem 3. *Let* Γ *be a well-formed type environment. Let S be a statement. Let* P_1 *and* P_2 *be assertions. Let* ε *be effects. Then*

$$\vdash_r^\Gamma \{P_1\}S\{P_2\}[\varepsilon] \;\; \textit{iff} \;\; \vdash_u^\Gamma [\mathbf{reads}\, r\downarrow]\{P_1\}S\{P_2\}[\varepsilon]$$
$$\textbf{\textit{where}}\ P_1 \Rightarrow r = \mathbf{alloc}\ \textit{and}\ \mathbf{modifies}\ r \notin \varepsilon$$

The proof is found in Appendix C.

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

Corollary 3. *Let* Γ *be a well-formed type environment. Let S be a statement. Let* P_1 *and* P_2 *be assertions. Let* ε *be effects and* η *be read effects. Then*

$$\vdash_u^\Gamma [\eta]\{P_1\}S\{P_2\}[\varepsilon] \textit{ implies } \vdash_r^\Gamma \{P_1\}S\{P_2\}[\varepsilon].$$

The proof uses the subeffect rule to convert η into **reads** r, where $P_1 \Rightarrow r = \mathbf{alloc}$, 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, $a_1 * a_2$, denotes that assertions a_1 and a_2 hold in separate parts of the current heap. The separating implication, $a_1 -\!\!* a_2$, 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:*

$$e ::= x \mid \texttt{null} \mid n$$
$$a ::= e = e \mid x.f \mapsto e \mid a * a \mid a -\!\!* a \mid a \wedge a \mid a \vee a \mid a \Rightarrow a \mid \exists x.a$$ ∎

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 $\mathcal{N}$ is the standard meaning function*

for numeric literals and (σ, h) *is a state, then the semantics of expressions in separation logic is:*

$$\mathcal{E}_s : \textit{Typing Judgment} \rightarrow e \rightarrow \textit{State} \rightarrow \textit{Value}$$

$$\mathcal{E}_s[\![\Gamma \vdash x : T]\!](\sigma, h) = \sigma(x) \qquad \mathcal{E}_s[\![\Gamma \vdash n : \textbf{\texttt{int}}]\!](\sigma, h) = \mathcal{N}[\![n]\!]$$

$$\mathcal{E}_s[\![\Gamma \vdash \texttt{null} : T]\!](\sigma, h) = null$$

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

$$\begin{aligned}
\sigma, h \models_s^\Gamma e = e' &\iff \mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma) = \mathcal{E}_s[\![\Gamma \vdash e' : T]\!](\sigma) \\
\sigma, h \models_s^\Gamma x.f \mapsto e &\iff (\sigma(x), f) \in \textit{dom}(h) \textit{ and } h[(\sigma(x), f)] = \mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma) \\
\sigma, h \models_s^\Gamma a_1 * a_2 &\iff \textit{exists } h_1, h_2.(h_1 \perp h_2 \textit{ and } h = h_1 \cdot h_2 \textit{ and } \sigma, h_1 \models_s^\Gamma a_1 \textit{ and } \\
&\qquad \sigma, h_2 \models_s^\Gamma a_2) \\
\sigma, h \models_s^\Gamma a_1 -\!\!* a_2 &\iff \textit{for all } h'.(h' \perp h \textit{ and } \sigma, h' \models_s^\Gamma a_1 \textit{ implies } \sigma, h \cdot h' \models_s^\Gamma a_2) \\
\sigma, h \models_s^\Gamma a_1 \wedge a_2 &\iff \sigma, h \models_s^\Gamma a_1 \textit{ and } \sigma, h \models_s^\Gamma a_2 \\
\sigma, h \models_s^\Gamma a_1 \vee a_2 &\iff \sigma, h \models_s^\Gamma a_1 \textit{ or } \sigma, h \models_s^\Gamma a_2 \\
\sigma, h \models_s^\Gamma a_1 \Rightarrow a_2 &\iff \textit{for all } h'.(\sigma, h \cdot h' \models_s^\Gamma a_1 \textit{ implies } \sigma, h \cdot h' \models_s^\Gamma a_2) \\
\sigma, h \models_s^\Gamma \exists x.a &\iff \textit{exists } v.(\sigma[x \mapsto v], h \models_s^\Gamma a)
\end{aligned}$$

∎

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 ($\restriction$) 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 $r \subseteq \textit{dom}(H)$, define

$\sigma, H\restriction r \models^{\Gamma}_{sl} a$ if and only if $\sigma, (H\restriction r) \models^{\Gamma}_{s} a$, however, for clarity, the following definitions of validity for separating conjunction and implication are used.

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

$$\sigma, H\restriction r \models^{\Gamma}_{sl} a_1 * a_2 \textit{ iff } exists\ r_1, r_2 :: (r_1 \cap r_2 = \varnothing \ and\ r = r_1 \cup r_2 \ and\ \sigma, H\restriction r_1 \models^{\Gamma}_{sl} a_1 \ and \\ \sigma, H\restriction r_2 \models^{\Gamma}_{sl} a_2)$$

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

$$\begin{aligned}
\sigma, H\restriction r \models^{\Gamma}_{sl} a_1 -\!\!* a_2 \textit{ iff for all } h', r' :: \quad & (dom(h') \cap r = \varnothing \ and\ \sigma, h'\restriction r' \models^{\Gamma}_{sl} a_1 \textit{ implies} \\
& \sigma, (H \cup h')\restriction (r \cup r') \models^{\Gamma}_{sl} a_2) \\
\sigma, H\restriction r \models^{\Gamma}_{sl} a_1 \Rightarrow a_2 \textit{ iff for all } h', r' :: \quad & (dom(h') \cap r = \varnothing \ and\ \sigma, (H \cup h)\restriction (r \cup r') \models^{\Gamma}_{sl} a_1 \\
& \textit{implies } \sigma, (H \cup h')\restriction (r \cup r') \models^{\Gamma}_{sl} a_2)
\end{aligned}$$

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

Theorem 4. *Let Γ be a well-formed type environment. Let σ be a store, h and H be heaps, and r be a region, such that $r \subseteq dom(H)$ and $h = H\restriction r$, then $\sigma, h \models^{\Gamma}_{s} a$ iff $\sigma, H\restriction r \models^{\Gamma}_{sl} a$.*

The above theorem chooses $dom(h)$ 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.

6.2.2 *Supported Separation Logic*

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 (σ, 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 $\{(\sigma(x), f) \mapsto \mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma)\}$. 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:

$$\texttt{MinReg}(a, \sigma, h) = \bigcap\{r \mid r \subseteq \textit{dom}(h) \textit{ and } (\sigma, h \vDash_s^\Gamma a \textit{ implies } \sigma, (h\restriction r) \vDash_s^\Gamma a)\},$$

where (σ, h) is a state. However, $\sigma, (h\restriction\texttt{MinReg}(a, \sigma, h)) \vDash_s^\Gamma a$ is not always true. For example, consider $(x.f \mapsto 5) \vee (y.g \mapsto 6)$ 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 $\sigma, (h \restriction \varnothing) \vDash_s^\Gamma (x.f \mapsto 5) \vee (y.g \mapsto 6)$ is false. So, some assertions containing disjunction do not have a semantic footprint. Semantic footprints are defined as follows.

Definition 12 (Semantic Footprint)**.** *Let Γ be a well-formed type environment. Let a be an assertion*

in SL, and (σ, h) *be a* Γ*-state. Then* $\texttt{MinReg}(a, \sigma, h)$ *is the semantic footprint of* a *if and only if* $\sigma, (h\restriction \texttt{MinReg}(a, \sigma, h)) \models^{\Gamma}_{s} a$*. 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].[2] 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:*

$$e ::= x \mid \texttt{null} \mid n$$
$$b ::= e_1 = e_2 \mid e_1 \neq e_2$$
$$a ::= b \mid x.f \mapsto e \mid a_1 * a_2 \mid a_1 \wedge a_2 \mid b \Rightarrow a \mid \exists\, x.\, (y.f \mapsto \mathtt{x} * a)$$ ∎

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, $a_1 * a_2$, needs this property. Given the truth of $a_1 * a_2$ 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

[2] For the work with classical separation logic, the *emp* predicate is needed.

and r_2, must satisfy $r_1 \cup r_2 = \mathit{dom}(h)$, which is required by its semantics. Picking $h\restriction r_1$ as the witness for h_1 pushes the proof to take $h\restriction(\mathit{dom}(h) - r_1)$ as the witness for h_2. Lemma 9 below can be applied in this scenario as $h\restriction r_2 \subseteq h\restriction(\mathit{dom}(h) - r_1)$, as $r_1 \subseteq \mathit{dom}(h)$ and $r_2 \subseteq \mathit{dom}(h)$.

Lemma 9. *Let Γ be a well-formed type environment. Let a be an SSL assertion, and (σ, h) be a Γ-state. Let h' be a heap, such that $h \subseteq h'$. Then $\sigma, h \models^{\Gamma}_{s} a \Rightarrow \sigma, h' \models^{\Gamma}_{s} a$.*

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 Γ be a well-formed type environment. Let (σ, h) be a state, and let e, b and a be an SSL expression, a Boolean expression, and an assertion. Then:*

1. $\mathtt{MinReg}(b, \sigma, h) = \varnothing$.
2. *if* $\sigma, h \models^{\Gamma}_{s} x.f \mapsto e$, *then* $\mathtt{MinReg}(x.f \mapsto e, \sigma, h) = \{(\sigma(x), f)\}$.
3. *if* $\sigma, h \models^{\Gamma}_{s} a_1 * a_2$, *then* $\mathtt{MinReg}(a_1 * a_2, \sigma, h) = \mathtt{MinReg}(a_1, \sigma, h) \cup \mathtt{MinReg}(a_2, \sigma, h)$.
4. *if* $\sigma, h \models^{\Gamma}_{s} a_1 \wedge a_2$, *then* $\mathtt{MinReg}(a_1 \wedge a_2, \sigma, h) = \mathtt{MinReg}(a_1, \sigma, h) \cup \mathtt{MinReg}(a_2, \sigma, h)$.
5. *if* $\sigma, h \models^{\Gamma}_{s} b \Rightarrow a$ *and* $\sigma, h \models^{\Gamma}_{s} b$, *then* $\mathtt{MinReg}(b \Rightarrow a, \sigma, h) = \mathtt{MinReg}(a, \sigma, h)$.
6. *if* $\sigma, h \models^{\Gamma}_{s} b \Rightarrow a$ *and* $\sigma, h \not\models^{\Gamma}_{s} b$, *then* $\mathtt{MinReg}(b \Rightarrow a, \sigma, h) = \varnothing$.
7. *if* $\sigma, h \models^{\Gamma}_{s} \exists x.(y.f = x * a)$, *then* $\mathtt{MinReg}(\exists x.(y.f = x * a), \sigma, h) = \mathtt{MinReg}(y.f \mapsto x * a, \sigma[x \mapsto h[\mathcal{E}_s[\![\Gamma \vdash y : T]\!](\sigma), f]], h)$;

Moreover, $\sigma, h \models^{\Gamma}_{s} a$ *iff* $\sigma, (h\restriction \mathtt{MinReg}(a, \sigma, h)) \models^{\Gamma}_{s} a$.

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 Γ be a well-formed type environment. An assertion a is* supported *if and only if for all Γ-states (σ, 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 (σ, 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 (σ, h), and any pair of regions r_1 and r_2 where $r_1 \subseteq \mathit{dom}(h)$ and $r_2 \subseteq \mathit{dom}(h)$, and a separation logic assertion a, such that $\sigma, (h{\restriction}r_1) \models^{\Gamma}_{s} a$ and $\sigma, (h{\restriction}r_2) \models^{\Gamma}_{s} a$. Let r be the glb of r_1 and r_2, such that such that $r \subseteq r_1 \cap r_2$. If $\sigma, (h{\restriction}r) \models^{\Gamma}_{s} a$, 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 $\mathbf{region}\{\}$, 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.*

$$
\begin{array}{lcl}
\mathit{fpt}_s(b) & = & \mathbf{region}\{\} \\
\mathit{fpt}_s(x.f \mapsto e) & = & \mathbf{region}\{x.f\} \\
\mathit{fpt}_s(b \Rightarrow a) & = & b \,?\, \mathit{fpt}_s(a) : \mathbf{region}\{\} \\
\mathit{fpt}_s(a_1 * a_2) & = & \mathit{fpt}_s(a_1) + \mathit{fpt}_s(a_2) \\
\mathit{fpt}_s(a_1 \wedge a_2) & = & \mathit{fpt}_s(a_1) + \mathit{fpt}_s(a_2) \\
\mathit{fpt}_s(\exists x.(y.f = x * a)) & = & \mathbf{region}\{y.f\} + \mathit{fpt}_s(a)[y.f/x]
\end{array}
$$

∎

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:

$$\begin{array}{lll}
\mathtt{TR}[\![x]\!] = x \qquad \mathtt{TR}[\![n]\!] = n \qquad \mathtt{TR}[\![null]\!] = null & & \\
\mathtt{TR}[\![e_1 = e_2]\!] & = & \mathtt{TR}[\![e_1]\!] = \mathtt{TR}[\![e_2]\!] \\
\mathtt{TR}[\![e_1 \neq e_2]\!] & = & \mathtt{TR}[\![e_1]\!] \neq \mathtt{TR}[\![e_2]\!] \\
\mathtt{TR}[\![x.f \mapsto e]\!] & = & \mathtt{TR}[\![x]\!].f = \mathtt{TR}[\![e]\!] \\
\mathtt{TR}[\![a_1 * a_2]\!] & = & \mathtt{TR}[\![a_1]\!] \;\&\&\; \mathtt{TR}[\![a_2]\!] \;\&\&\; (\mathit{fpt}_s(a_1) \;!!\; \mathit{fpt}_s(a_2)) \\
\mathtt{TR}[\![a_1 \wedge a_2]\!] & = & \mathtt{TR}[\![a_1]\!] \;\&\&\; \mathtt{TR}[\![a_2]\!] \\
\mathtt{TR}[\![b \Rightarrow a]\!] & = & \mathtt{TR}[\![b]\!] \Rightarrow \mathtt{TR}[\![a]\!] \\
\mathtt{TR}[\![\exists x.(y.f \mapsto x * a)]\!] & = & \exists x.(\mathtt{TR}[\![y.f \mapsto x]\!] \;\&\&\; \mathtt{TR}[\![a]\!] \;\&\&\; (\mathbf{region}\{y.f\} \;!!\; \mathit{fpt}_s(a)))\blacksquare
\end{array}$$

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 $\mathtt{TR}[\![e]\!]$, as well as b and $\mathtt{TR}[\![b]\!]$ can be used interchangeably.

Lemma 11. *Let Γ be a well-formed type environment. Let σ be a store. Let e be an expression in SSL. Then $\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash \mathtt{TR}[\![e]\!] : T]\!](\sigma)$.*

Lemma 12. *Let Γ be a well-formed type environment. Let (σ, h) be a Γ-state, and H be a heap such that $h \subseteq H$. Let b be a pure assertion in SSL. Then $\sigma, h \models_s^\Gamma b$ iff $\sigma, h \models^\Gamma \mathtt{TR}[\![b]\!]$ iff $\sigma, H \models^\Gamma \mathtt{TR}[\![b]\!]$.*

The following theorem shows that the semantics of the semantic footprint function, $\mathit{fpt}_s(a)$, 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 Γ be a well-formed type environment. Let a be an assertion in SSL, and let (σ, h) be a Γ-state. If $\sigma, h \models_s^\Gamma a$, then a has a semantic footprint in (σ, h), and this semantic footprint is $\mathtt{MinReg}(a, \sigma, h) = \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$.*

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 Γ be a well-formed type environment. Let a be an assertion in SSL. Let (σ, h) be a Γ-state. If $\sigma, h \models_s^\Gamma a$, then $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \subseteq \mathit{dom}(h)$.*

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

Corollary 5. *Let Γ be a well-formed type environment. Let a be an assertion in SSL. Let (σ, h) be a Γ-state. Then $\sigma, h \models_s^\Gamma a$ iff $\sigma, h \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$.*

The following theorem shows that 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 Γ be a well-formed type environment. Let (σ, h) be a Γ-state. Let a be an assertion in SSL. Then $\sigma, h \models_s^\Gamma a$ iff $\sigma, h \models^\Gamma \mathit{TR}[\![a]\!]$.*

Fig. 6.1 summarizes the previous results, where $h = H \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$. The r is found for the SL's semantics for Theorem 4, which is $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$; since it has been proved that $\sigma, h \models_s^\Gamma a$ if and only if $\sigma, H \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$, it

must be that $h = H \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$. In addition, by Corollary 5, it must be true that $\sigma, h \models^{\Gamma}_{s} a$ *iff* $\sigma, h \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models^{\Gamma}_{s} a$. Furthermore, by Theorem 7 on the preceding page twice, it must be true that $\sigma, h \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models^{\Gamma}_{s} a$ *iff* $\models^{\Gamma} \sigma, h \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))\mathtt{TR}[\![a]\!]$, and $\sigma, h \models^{\Gamma}_{s} a$ *iff* $\sigma, h \models^{\Gamma} \mathtt{TR}[\![a]\!]$. Therefore, by transitivity, it must be true that $\sigma, h \models^{\Gamma} \mathtt{TR}[\![a]\!]$ *iff* $\sigma, H \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models^{\Gamma} \mathtt{TR}[\![a]\!]$ *iff* $\sigma, H \restriction (\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models^{\Gamma}_{sl} a$.

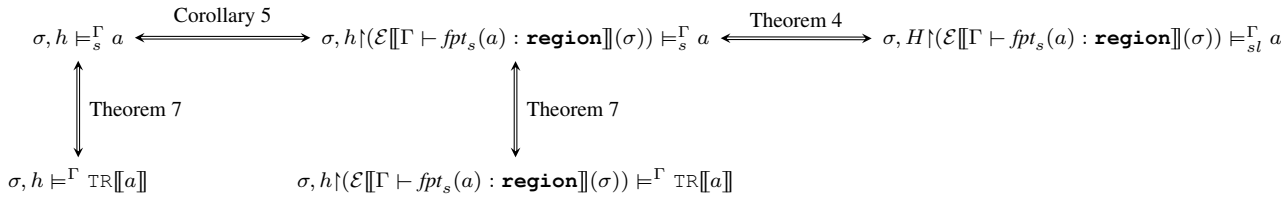

Figure 6.1: A summary of results on encoding assertions

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):

$$\begin{aligned}
&\vdash^{\Gamma}_{s} \{a\} S \{a'\} \textit{ iff} \\
&\vdash^{\Gamma}_{u} [\mathbf{reads}\ \mathit{fpt}_s(a)]\ \{\mathtt{TR}[\![a]\!]\}\ S\ \{\mathtt{TR}[\![a']\!]\}[\mathbf{modifies}\ (\mathtt{MV}(S), \mathit{fpt}_s(a)), \mathbf{fresh}(\mathit{fpt}_s(a') - r)] \\
&\textbf{where } r \text{ is a region variable, such that } \mathtt{TR}[\![a]\!] \Rightarrow r = \mathit{fpt}_s(a) \textit{ and } r \notin \mathtt{MV}(S)
\end{aligned} \tag{6.1}$$

where $\mathtt{MV}(S)$ is the set of variables that S may modify, and r snapshots the set of locations of $\mathit{fpt}_s(a)$ in the pre-state. This translation is not the only way to establish the equivalence, e.g., the

read effects can be anything from $\varnothing$ 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 Γ be a well-formed type environment. Let S be a statement. Let a and a' be assertions in SSL. Let (σ, H) be a Γ-state. Then* $\{a\}S\{a'\}$ is valid in (σ, H)*, written $\sigma, H \models_s^\Gamma \{a\}S\{a'\}$, if and only if whenever $\sigma, H \models_s^\Gamma a$, 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 $\sigma', H' \models_s^{\Gamma'} a'$.*

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

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

1. *Safety Monotonicity:* for all states (σ, H) and heaps H', such that $H \perp H'$, 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 (σ, H) and heaps H', such that $H \perp H'$, 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 (σ, H_0) and heaps H_1, such that $H_0 \perp H_1$, 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 $H_0' \perp H_1$, $H_0' \cdot H_1 = H'$, 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 $x.f_1 \mapsto$ *default*$(T_1) * \cdots * x.f_n \mapsto$ *default*(T_n), where

the $f_i : T_i$ are defined by $(f_1 : T_1, \ldots, f_n : T_n) = \mathit{fields}(T)$. 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 Γ be a well-formed type environment. Let a and a' be assertions and S be a statement, such that $\models_s^\Gamma \{a\}S\{a'\}$. Let (σ, H) be a Γ-state. If $\sigma, H \models_s^\Gamma a$ and $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$, then:*

1. *for all $x \in dom(\sigma)$, if $\sigma'(x) \neq \sigma(x)$, then $x \in \mathit{MV}(S)$.*
2. *for all $(o, f) \in dom(H)$, if $H'[o, f] \neq H[o, f]$, then $(o, f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$.*
3. *for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$, it is that $(o, f) \in (dom(H') - dom(H))$.*

There are several lemmas connecting FRL and UFRL separation operator ($\cdot/\cdot$) 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 Γ be a well-formed type environment. Let (σ, h) be a Γ-state. Let a_1 and a_2 be assertions in SSL. Then*

$$\sigma, h \models_s^\Gamma a_1 * a_2 \textit{ implies } \sigma, h \models_u^\Gamma \mathit{efs}(\mathit{TR}[\![a_2]\!]) \cdot/\cdot \mathbf{modifies}\, \mathit{fpt}_s(a_1)$$

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

$$(SKIP_s)\ \ \vdash_s^\Gamma \{true\}\mathbf{skip};\{true\}$$

$$(VAR_s)\ \ \vdash_s^\Gamma \{true\}\mathbf{var}\ x:T;\{x = default(T)\}$$

$$(ALLOC_s)\ \ \vdash_s^\Gamma \{a\}\ x := \mathbf{new}\ T;\ \{a * new_s(T,x)\},\ \mathbf{where}\ x \notin \mathtt{FV}(a)$$

$$(ASGN_s)\ \ \vdash_s^\Gamma \{true\}\ x := e;\ \{x = e\},\ \mathbf{where}\ x \notin \mathtt{FV}(e)$$

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

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

$$(IF_s)\ \frac{\vdash_s^\Gamma \{a \wedge e\}\ S_1\ \{a'\}, \qquad \vdash_s^\Gamma \{a \wedge \neg e\}\ S_2\ \{a'\}}{\vdash_s^\Gamma \{a\}\ \mathbf{if}\ e\ \{S_1\}\mathbf{else}\{S_2\}\ \{a'\}}$$

$$(WHILE_s)\ \frac{\vdash_s^\Gamma \{I \wedge e\}\ S\ \{I\}}{\vdash_s^\Gamma \{I\}\ \mathbf{while}\ e\ \{S\}\ \{I \wedge \neg e\}}$$

$$(SEQ_s)\ \frac{\vdash_s^\Gamma \{a\}\ S_1\ \{b\}, \qquad \vdash_s^{\Gamma'} \{b\}\ S_2\ \{a'\}}{\vdash_s^\Gamma \{a\}\ S_1S_2\ \{a'\}}$$

$$\begin{array}{ll}
\mathtt{FV}(x) = \{x\} & \mathtt{FV}(\mathbf{null}) = \varnothing \qquad \mathtt{FV}(n) = \varnothing \\
\mathtt{FV}(e_1 = e_2) = \mathtt{FV}(e_1) \cup \mathtt{FV}(e_2) & \mathtt{FV}(e_1 \neq e_2) = \mathtt{FV}(e_1) \cup \mathtt{FV}(e_2) \\
\mathtt{FV}(x.f \mapsto e) = \{x\} \cup \mathtt{FV}(e) & \mathtt{FV}(a_1 * a_2) = \mathtt{FV}(a_1) \cup \mathtt{FV}(a_2) \\
\mathtt{FV}(a_1 \wedge a_2) = \mathtt{FV}(a_1) \cup \mathtt{FV}(a_2) & \mathtt{FV}(b \Rightarrow a) = \mathtt{FV}(b) \cup \mathtt{FV}(a) \\
\mathtt{FV}(\exists\, x.y.f = x * a) = (\{y\} \cup \mathtt{FV}(a)) - \{x\} &
\end{array}$$

Figure 6.2: The axioms and proof rules for statements in SSL [77]

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) \subseteq \mathit{dom}(h_1)$. So the following holds.

$$\mathit{for\ all}\ \mathbf{reads}\ \mathit{RE} \leqslant \mathit{efs}(\mathtt{TR}[\![a_2]\!]) :: \mathit{RE}\ !\ !\ \mathit{fpt}_s(a_1). \tag{6.2}$$

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

$$\mathit{for\ all}\ \mathbf{reads}\ X \leqslant \mathit{efs}(\mathtt{TR}[\![a_2]\!]) :: \mathbf{reads}\ X \cdot\!\!\not\vert\ \mathbf{modifies}\ \mathit{fpt}_s(a_1). \tag{6.3}$$

Using Eq. (6.3) together with Eq. (6.2) and the definition of separator (Fig. 3.7), proves that $\mathit{efs}(\mathtt{TR}[\![a_2]\!]) \cdot\!\!\not\vert\ \mathbf{modifies}\ \mathit{fpt}_s(a_1)$.

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 $\mathtt{FV}(a) = \mathtt{FV}(\mathtt{TR}[\![a]\!])$.*

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

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

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 $\mathit{fpt}_s(a)$, write effect $(\mathit{fpt}_s(a), \mathtt{MV}(S))$ and fresh effect $(\mathit{fpt}_s(a') - r)$, where r snapshots the set of locations of $\mathit{fpt}_s(a)$ 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 Γ be a well-formed type environment. Let S be a statement, and let a and a' be assertions in SSL, such that $\models_s^\Gamma \{a\}S\{a'\}$. Let r be a region variable. Let (σ, H) be Γ-state. If $\sigma, H \models^\Gamma TR[\![a]\!] \Rightarrow r = fpt_s(a)$ and $r \notin MV(S)$, then*

$$\begin{aligned}
&\sigma, H \models_s^\Gamma \{a\}\ S\{a'\}\ \textit{iff}\\
&\qquad [\textbf{\textit{reads}}\ fpt_s(a)]\\
&\sigma, H \models_u^\Gamma \{TR[\![a]\!]\}S\{TR[\![a']\!]\}\\
&\qquad [\textbf{\textit{modifies}}\ (fpt_s(a), MV(S)), \textbf{\textit{fresh}}\ (fpt_s(a') - r)]
\end{aligned}$$

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 $ALLOC_s$ adds "**modifies alloc**" to the frame condition.

Definition 18 (Syntactic Mapping from SSL to UFRL)**.** *Let Γ be a well-formed type environment. Let a and a' be assertions in SSL. A syntactic mapping, $TR_s[\![-]\!]$, from SSL axioms and rules to those of UFRL is defined below:*

$$\begin{aligned}
&TR_s[\![\vdash_s^\Gamma \{a\}\ x\ :=\ \textbf{new}\ T; \{a * new_s(T, x)\}]\!] =\\
&\qquad [\textbf{\textit{reads}}\ fpt_s(a)]\\
&\quad \vdash_u^\Gamma \{TR[\![a]\!]\}\ x\ :=\ \textbf{new}\ T; \{TR[\![a * new_s(T, x)]\!]\}\\
&\qquad [\textbf{\textit{modifies}}\ x,\ \textbf{\textit{modifies}}\ \textbf{alloc},\ \textbf{\textit{fresh}}(fpt_s(new_s(T, x)))]\\
&TR_s[\![\vdash_s^\Gamma \{a\}\ S\ \{a'\}]\!] =\\
&\vdash_u^\Gamma [\textbf{\textit{reads}}\ fpt_s(a)]\{TR[\![a]\!]\}\ S\ \{TR[\![a']\!]\}\ [\textbf{\textit{modifies}}\ (fpt_s(a), MV(S)), \textbf{\textit{fresh}}(fpt_s(a') - r)],\\
&\qquad \textbf{\textit{where}}\ r \text{ is a region variable such that } r \notin MV(S), TR[\![a]\!] \Rightarrow r = fpt_s(a) \text{ and}\\
&\qquad\qquad S \neq x := \textbf{new}\, T;.
\end{aligned}$$

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:

$$TR_s[\![\frac{\vdash_s^\Gamma h_1, \ldots, \vdash_s^\Gamma h_n}{\vdash_s^\Gamma c}]\!] = \frac{TR_s[\![\vdash_s^\Gamma h_1]\!], \ldots, TR_s[\![\vdash_s^\Gamma h_n]\!]}{TR_s[\![\vdash_s^\Gamma c]\!]}$$

∎

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 $SEQ1_u$ 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 $y.f \mapsto 3$ before executing the first statement. In the proof, the following derivation can be achieved in SSL.

$$\frac{\begin{array}{c}\vdash_s^\Gamma \{y.f \mapsto 3\}x := y; x.f := 5; \{x = y * x.f \mapsto 5\} \\ \vdash_s^\Gamma \{x = y * x.f \mapsto 5\}x.f := 6; \{x = y * x.f \mapsto 6\}\end{array}}{\vdash_s^\Gamma \{y.f \mapsto 3\}x := y; x.f := 5; x.f := 6; \{x = y * x.f \mapsto 6\}} (SEQ_s)$$

By Def. 18, the two premises are encoded to

$$\begin{array}{l}[\mathbf{reads\ region}\{y.f\}] \\ \vdash_u^\Gamma \ \{y.f = 3\}\ x := y; x.f := 5;\ \{x = y \,\&\&\, x.f = 5\} \\ [\mathbf{modifies}\ x, \mathbf{modifies\ region}\{y.f\}]\end{array} \quad (6.4)$$

$$\begin{array}{l}[\mathbf{reads\ region}\{x.f\}] \\ \vdash_u^\Gamma \ \{x = y \,\&\&\, x.f = 5\}x.f := 6; \{x = y \,\&\&\, x.f = 6\} \\ [\mathbf{modifies\ region}\{x.f\}]\end{array} \quad (6.5)$$

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

$$\begin{array}{ll} & [\textbf{reads region}\{y.f\}] \\ \vdash_u^\Gamma & \{y.f = 3\}\ x := y; x.f := 5; x.f := 6;\ \{x = y\ \&\&\ x.f = 6\} \\ & [\textbf{modifies}\ x, \textbf{modifies region}\{y.f\}] \end{array} \tag{6.6}$$

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:

$$\begin{array}{ll} & [\textbf{reads region}\{y.f\}] \\ \vdash_u^\Gamma & \{x = y\ \&\&\ x.f = 5\}x.f := 6; \{x = y\ \&\&\ x.f = 6\} \\ & [\,\textbf{modifies region}\{y.f\}] \end{array} \tag{6.7}$$

Now the side conditions about immunity are true. Eq. (6.6) is derived by using the rule $SEQ1_u$. The proof strategy generalizes the approach used in the example. Let S_1S_2 be a sequential statement. The effects of S_2 is re-written by replacing all the variables in $\texttt{MV}(S_1)$, i.e., $\overline{x}$, with the variables $\overline{z}$, such that $a' \Rightarrow \overline{z} = \overline{x}$ and $\overline{z} \cap \texttt{MV}(S_1) = \varnothing$, 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, $x.f \mapsto e$, 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:

$$P ::= \ldots \mid P_1 * P_2$$

In the syntax, dots "..." denotes the material defined previously. Given a SSL assertion a, $\mathtt{TR}[\![a]\!]$, which replaces each occurence of $\mapsto$ 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 $P = \mathtt{TR}[\![a]\!]$.* ∎

Defining the semantics of separating conjunction, $P_1 * P_2$, in UFRL (or FRL) requires a definition of its footprint, $fpt(P_1 * P_2)$. The semantics of *fpt* is defined by using the inverse of the translation function, $\mathtt{TR}^{-1}$. This inverse exists because, by definition, $\mathtt{TR}$ 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 $\mathtt{TR}(a) = P$; thus for each supported UFRL (or FRL) assertion P, $\mathtt{TR}^{-1}[\![P]\!]$ is defined to be the SSL assertion a such that $\mathtt{TR}(a) = P$.

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

Finally, the semantics of separating conjunction is defined as follows for supported UFRL (or FRL)

assertions P_1 and P_2:

$$\sigma, H \models^{\Gamma} P_1 * P_2 \textit{ iff } \sigma, H \models^{\Gamma} P_1 \textit{ and } \sigma, H \models^{\Gamma} P_2 \textit{ and } \sigma, H \models^{\Gamma} \textit{fpt}(P_1) \,!\,!\, \textit{fpt}(P_2) \tag{6.8}$$

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

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

$$\sigma, H \models^{\Gamma} P_1 * P_2 \textit{ iff } \sigma, H \models_s^{\Gamma} \mathit{TR}^{-1}[\![P_1]\!] * \mathit{TR}^{-1}[\![P_2]\!].$$

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

$\sigma, H \models^{\Gamma} P_1 * P_2$

iff $\langle$by Eq. (6.8)$\rangle$

$\sigma, H \models^{\Gamma} P_1 \textit{ and } \sigma, H \models^{\Gamma} P_2 \textit{ and } \sigma, H \models^{\Gamma} \textit{fpt}(P_1) \,!\,!\, \textit{fpt}(P_2)$

iff $\langle$by definition $\textit{fpt}(P) = \textit{fpt}_s(\mathtt{TR}^{-1}[\![P]\!])\rangle$

$\sigma, H \models^{\Gamma} P_1 \textit{ and } \sigma, H \models^{\Gamma} P_2 \textit{ and } \sigma, H \models^{\Gamma} \textit{fpt}_s(\mathtt{TR}^{-1}[\![P_1]\!]) \,!\,!\, \textit{fpt}_s(\mathtt{TR}^{-1}[\![P_2]\!])$

iff $\langle$by semantics of assertions in Fig. 3.2$\rangle$

$\sigma, H \models^{\Gamma} P_1 \,\&\&\, P_2 \,\&\&\, \textit{fpt}_s(\mathtt{TR}^{-1}[\![P_1]\!]) \,!\,!\, \textit{fpt}_s(\mathtt{TR}^{-1}[\![P_2]\!])$

iff $\langle$by definition of the syntactical mapping from SSL to UFRL (Def. 16), as P_1 and P_2 are supported$\rangle$

$\sigma, H \models^{\Gamma} \mathtt{TR}[\![\mathtt{TR}^{-1}[\![P_1]\!] * \mathtt{TR}^{-1}[\![P_2]\!]]\!]$

iff $\langle$by Theorem 7$\rangle$

$\sigma, H \models_s^{\Gamma} \mathtt{TR}^{-1}[\![P_1]\!] * \mathtt{TR}^{-1}[\![P_2]\!]$

□

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 $P_1 * P_2$ as follows:

$$\textit{efs}(P_1 * P_2) = \textit{efs}(P_1), \textit{efs}(P_2) \tag{6.9}$$

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

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

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

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

$$\sigma, h \models^{\Gamma} P_1 * P_2 \textit{ implies } \sigma, h \models^{\Gamma} \textit{efs}(P_2) \backslash\!\!\backslash \textbf{modifies} \textit{ fpt}(P_1)$$

Note that it is not valid that $\sigma, h \models^{\Gamma} P_1 * P_2 \Rightarrow \sigma, h \models^{\Gamma}$ *efs*$(P_2) \backslash\!\!\backslash$ **modifies** $readVar(\textit{efs}(P_1)), \textit{readR}(\textit{efs}(P_1))$. For example, let P_1 be $x.f_1 = 4$ and P_2 be $x.f_2 = 5$, and $P_1 * P_2$ is valid. Because *efs*(P_2) = **reads** x, **region**$\{x.f_2\}$, and **modifies** $readVar(\textit{efs}(P_1)), \textit{readR}(\textit{efs}(P_1))$ = **modifies** x, **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:

$$(I_{sc})\ \frac{\vdash_u^{\Gamma} [\delta]\{P\}\ S\ \{Q\}[\varepsilon]}{\vdash_u^{\Gamma} [\delta]\{P * R\}\ S\ \{Q * R\}[\varepsilon]}\quad \textbf{where}\quad \begin{array}{l} P\ \&\&\ R \Rightarrow \textit{efs}(R) \backslash\!\!\backslash \textbf{modifies}\ \textit{fpt}(P), \\ P\ \&\&\ R \Rightarrow \textit{efs}(R) \backslash\!\!\backslash \varepsilon \text{ and} \\ Q\ \&\&\ R \Rightarrow \textit{efs}(R) \backslash\!\!\backslash \textbf{modifies}\ \textit{fpt}(Q) \end{array}$$

The two extra side conditions are used to conclude $P * R$ and $Q * R$, 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:

$$(FRM_u)\ \frac{\vdash_u^\Gamma [\delta]\{P\}\ S\ \{Q\}[\varepsilon]}{\vdash_u^\Gamma [\delta]\{P \mathbin{\&\&} R\}\ S\ \{Q \mathbin{\&\&} R\}[\varepsilon]}\ \textbf{\textit{where}}\ P \mathbin{\&\&} R \Rightarrow \mathit{efs}(R) \cdot\!/\cdot \varepsilon$$

$$(CONSEQ_u)\ \frac{\vdash_u^\Gamma [\delta]\{P \mathbin{\&\&} R\}\ S\ \{Q \mathbin{\&\&} R\}[\varepsilon]}{\vdash_u^\Gamma [\delta]\{P * R\}\ S\ \{Q * R\}[\varepsilon]}\ \textbf{\textit{where}}\ (*)$$

$(*)$ is $P \mathbin{\&\&} R \Rightarrow \mathit{efs}(R) \cdot\!/\cdot \texttt{modifies}\ \mathit{fpt}(P)$ and $Q \mathbin{\&\&} R \Rightarrow \mathit{efs}(R) \cdot\!/\cdot \texttt{modifies}\ \mathit{fpt}_s(Q)$

□

Lemma 21. *Let* Γ *be a well-formed type environment. Let* P_1 *and* P_2 *be supported assertions in extended UFRL proof system, and* (σ, h) *be a* Γ*-state. If* $\sigma, h \models^\Gamma P_1$ *and* $\sigma, h \models^\Gamma P_2$ *and* $\mathit{efs}(P_2) \cdot\!/\cdot \textbf{\textit{modifies}}\ \mathit{fpt}(P_1)$*, 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`**$\downarrow$. Adopting the convention of VeriFast [41], `?v` and `?vlst` declare (universally-quantified) variables `v` and `vlst` respectively that scope over the entire specification of `append`. In the body of `append`, right after the loop, the following must be true:

```
(lst(this,vlst) && (lstseg(this,curr) * (lst(curr,?cvlst) &&
                                    curr.next = null))) * lst(n,[v]),   (6.10)
```

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

```
(lst(this,vlst) && (lstseg(this,curr) * curr.val ↦ ?cv *
                                    curr.next ↦ null)) * lst(n,[v]),   (6.11)
```

```
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:

$$\begin{array}{rl} & [\textbf{reads}\ curr,\ \textbf{alloc}\downarrow] \\ \vdash_u^\Gamma & \{curr \neq null\}\ \textit{curr.next := n;}\ \{curr.next = n\} \\ & [\textbf{modifies region}\{curr.next\}] \end{array} \tag{6.12}$$

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

$$\begin{array}{rl} & [\textbf{reads}\ curr,\ \textbf{alloc}\downarrow] \\ & \{curr \neq null\ *\ lstseg(\textbf{this},\ curr) * curr.val\ \mapsto ?cv\ *\ lst(n,\ [v])\} \\ \vdash_u^\Gamma & \textit{curr.next := n;} \\ & \{curr.next = n\ *\ lstseg(\textbf{this},\ curr)\ *\ curr.val\ \mapsto ?cv\ *\ lst(n,\ [v])\} \\ & [\textbf{modifies region}\{curr.next\}] \end{array} \tag{6.13}$$

and the postcondition of Eq. (6.13) implies, by the definition of `lst`,

$$\textit{lst(curr,[curr.val]+[v])}\ \&\&\ \textit{lseg(}\textbf{\textit{this}}\textit{,n)} \tag{6.14}$$

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

$$\textbf{fpt}(\textit{lstseg(}\textbf{\textit{this}}\textit{, curr)})\ +\ \textbf{fpt}(\textit{lst(curr,[curr.val])})\ =\ \textbf{fpt}(\textit{lst(}\textbf{\textit{this}}\textit{,vlst)}), \tag{6.15}$$

Together with Eq. (6.14), at the end of the method body, it must be true that **fpt**(*lstseg(**this**,curr)*) + **fpt**(*lst(curr,[curr.val]+[v])*) = **fpt**(*lst(**this**, 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 $=$ for $\mapsto$ as follows:

$\vdash_s^\Gamma \{a\}$ `x := new T;` $\{a * new_s(T, x)$ *iff*

$[$**reads** $fpt_s(a)]$

$\vdash_u^\Gamma \{a[=/\mapsto]\}$ `x := new T;` $\{(a * new_s(T, x))[=/\mapsto]\}$

$[$ **modifies** x, **modifies alloc**, **fresh**$(fpt_s(new_s(T, x)))]$

$\vdash_s \{a\}S\{a'\}$ *iff*

$\vdash_u^\Gamma$ $[$**reads** $fpt_s(a)][a[=/\mapsto]]$ S $\{a'[=/\mapsto]\}[$**modifies**$(\mathtt{MV}(S), fpt_s(a))$, **fresh** $(fpt_s(a') - r)]$,

where $a[=/\mapsto] \Rightarrow r = fpt_s(a), r \notin \mathtt{MV}(S)$ *and* $S \neq x :=$ **new** T;

where $\mathtt{MV}(S)$ is the set of variables that S may modify, and r snapshots the set of locations of $fpt_s(a)$ in the pre-state. The encoded *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[1]

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 Predicated 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).

$$\textit{Predicate} \ ::= \ \texttt{predicate}\ p\,(\overline{x:T})\ \ \texttt{reads}\ \delta\texttt{;}\ \ [\texttt{decreases}\ G\texttt{;}]\{\ P\ \}$$

$$P \ ::= \ \ldots \ \mid \ p(\overline{G}) \ \mid \ x.p(\overline{G})$$

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].

[1]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 ($\leqslant$) 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 $\overline{x} \mapsto \overline{y}$ means pointwise mapping. A semantic function for assertion is defined below:

$$\mathcal{E}_p : \textit{Assertion Typing Judgment} \rightarrow \textit{Store} \times \textit{Heap} \rightarrow \{\textit{true}, \textit{false}\}$$

The satisfaction relation defined in Fig. 3.2 is defined by

$$\mathcal{E}_p[\![\Gamma \vdash P : \mathbf{bool}]\!](\varphi)\sigma, H \;\; \textit{iff} \;\; \sigma, H \models^{\Gamma} P$$

The semantics of a predicate call is defined as follows.

$$\begin{aligned}
&\mathcal{E}_p[\![\Gamma \vdash p(\overline{G}) : \mathbf{bool}]\!](\sigma, H) = \\
&(\mathit{fix}\underline{\lambda}(\sigma', H') \,.\, \mathcal{E}_p[\![\Gamma \vdash body(\mathbf{Object}, p) : \mathbf{bool}]\!](\sigma', H'))(\sigma(formals(\mathbf{object}, p) \mapsto \overline{v}), H) \\
&\qquad \textbf{where } \overline{v} = \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma)}
\end{aligned} \tag{7.1}$$

$$\begin{aligned}
&\mathcal{E}_p[\![\Gamma \vdash x.p(\overline{G}) : \mathbf{bool}]\!](\sigma, H) = \sigma(x) = o \; and \; o \neq null \; and \\
&(\mathit{fix}\underline{\lambda}(\sigma', H') \,.\, \mathcal{E}_p[\![\Gamma \vdash body(T, p) : \mathbf{bool}]\!](\sigma', H'))(\sigma(\mathbf{this}, formals(T, p)) \mapsto (o, \overline{v})), H) \\
&\qquad \textbf{where } T = type(o) \text{ and } \overline{v} = \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma)}
\end{aligned} \tag{7.2}$$

The read effects of predicate calls are defined as follows:

$$\begin{aligned} \mathit{efs}(p(\overline{F})) &= \mathit{efs}(F), \delta[\overline{F}/\overline{z}] \textbf{ where } \delta = \mathit{rd}(p) \mathit{ and } \overline{z} = \mathit{formals}(\texttt{Object}, p) \\ \mathit{efs}(x.p(\overline{F})) &= \texttt{reads } x, \mathit{efs}(F), \delta[(x, \overline{F})/(\texttt{this}, \overline{z})] \\ &\quad \textbf{where } \delta = \mathit{rd}(p) \mathit{ and } \overline{z} = \mathit{formals}(\mathit{type}(\sigma(x)), p) \end{aligned}$$

The rules $\mathit{LIntro1}_u$ and $\mathit{LIntro2}_u$ introduce the form of a predicate call to left-hand side of the judgment. The rules $\mathit{RIntro1}_u$ and $\mathit{RIntro2}_u$ 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.

$$(\mathit{LIntro1}_u)\ \frac{P' \vdash^{\Gamma}_u P}{p(\overline{F}) \vdash^{\Gamma}_u P} \textbf{ where } P' = \mathit{body}(\texttt{Object}, p)[\overline{F}/\mathit{formals}(\texttt{Object}, p)]$$

$$(\mathit{LIntro2}_u)\ \frac{x \neq \mathit{null}\ \&\&\ P' \vdash^{\Gamma}_u P}{x.p(\overline{F}) \vdash^{\Gamma}_u P} \textbf{ where } P' = \mathit{body}(\Gamma(x), p)[\overline{F}/\mathit{formals}(\Gamma(x), p)]$$

$$(\mathit{RIntro1}_u)\ \frac{P \vdash^{\Gamma}_u P'}{P \vdash^{\Gamma}_u p(\overline{F})} \textbf{ where } P' = \mathit{body}(\texttt{Object}, p)[\overline{F}/\mathit{formals}(\texttt{Object}, p)]$$

$$(\mathit{RIntro2}_u)\ \frac{P \vdash^{\Gamma}_u x \neq \mathit{null}\ \&\&\ P'}{P \vdash^{\Gamma}_u x.p(\overline{F})} \textbf{ where } P' = \mathit{body}(\Gamma(x), p)[\overline{F}/\mathit{formals}(\Gamma(x), p)]$$

Lemma 22. *The rules* $\mathrm{LIntro1}_u$, $\mathrm{LIntro2}_u$, $\mathrm{RIntro1}_u$ *and* $\mathrm{RIntro2}_u$ *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. □

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 $p_s(\overline{z:T})$ 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 $b_1 := (n = null)$, $a_1 := (se = [])$, $b_2 := (n \neq null)$ and $a_2 := (\exists\, m.\, n.val \mapsto se[0] * n.next \mapsto m * list(m, se[1..]))$, where se is a sequence.

$$\begin{aligned} list(n, se) \stackrel{\text{def}}{=} & (n = null \Rightarrow se = []) \wedge \\ & (n \neq null \Rightarrow (\exists\, m.\, n.val \mapsto se[0] * n.next \mapsto m * list(m, se[1..]))) \end{aligned} \tag{7.3}$$

The semantic function *idf* maps a predicate name to its induction definition, which is the conjunction of inductive cases. The semantic function $formals_s$ maps a predicate name to its formal parameters.

A semantic function for assertion is defined below:

$$\mathcal{E}_a : a \rightarrow Store \times Heap \rightarrow \{true,\ false\}$$

The satisfaction relation defined by Def. 11 is defined by

$$\mathcal{E}_a[\![\Gamma \vdash a]\!](\sigma, h) \ \ iff \ \ \sigma, h \models_s^{\Gamma} a$$

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

$$\begin{aligned} \mathcal{E}_a[\![\Gamma \vdash p_s(\overline{e})]\!](\sigma, h) = (fix\underline{\lambda}(\sigma', h')\ .\ \mathcal{E}_a[\![\Gamma \vdash idf(p_s)]\!](\sigma', h'))(\sigma(formals_s(p_s) \mapsto \overline{v}, h) \\ \textbf{where } \overline{v} = \overline{\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma)} \end{aligned} \tag{7.4}$$

Let $b \Rightarrow a$ be one of the inductive cases of predicate $p_s(\overline{z})$, then the rules $LIntro_s$ and $RIntro_s$ introduce the form of a predicate call to the left-hand side and the right-hand side of the judgment

respectively.

$$(\mathit{LIntro}_s)\ \frac{a \vdash_s^\Gamma a'}{p_s(\overline{e}) \vdash_s^\Gamma a'}\ \textbf{\textit{where}}\ a = (b \Rightarrow a)[\overline{e}/\mathit{formals}_s(p_s)]$$

$$(\mathit{RIntro}_s)\ \frac{a' \vdash_s^\Gamma a}{a' \vdash_s^\Gamma p_s(\overline{e})}\ \textbf{\textit{where}}\ a = (b \Rightarrow a)[\overline{e}/\mathit{formals}_s(p_s)]$$

7.3 Encoding

The translation of recursive predicate call is defined as follows:

$$\texttt{TR}[\![p_s(\overline{e})]\!] = p_s(\overline{\texttt{TR}[\![e]\!]}). \tag{7.5}$$

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 `!!` ... `!!` RE_n means pairwise region disjointness. For each inductive predicate $p_s : \overline{z : T} \mapsto$ **`bool`**, there is a region function with the signature $\mathit{region\_p}_s : \overline{z : T} \mapsto$ **`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 **`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 $\texttt{region}_{p_s}$ and the results in Section 6.2.3, it is known that $\texttt{TR}[\![p_s]\!] \vdash_u^\Gamma \mathit{region\_p}_s\ \mathit{frm}\ \texttt{TR}[\![p_s]\!]$.

Lemma 23. *Let Γ be a well-formed type environment. Let (σ, h) be a Γ-state, and p_s be an inductive predicate in SSL. Then*

$$\mathcal{E}_a[\![\Gamma \vdash p_s(\overline{e}) : \textbf{\texttt{bool}}]\!](\sigma, h) = \mathcal{E}_p[\![\Gamma \vdash \mathit{TR}[\![p_s(\overline{e})]\!] : \textbf{\texttt{bool}}]\!](\sigma, h).$$

$\mathtt{TR}[\![p_s]\!]$ =

```
predicate p_s(z : T)
  reads region_p_s(z);
  decreases region_p_s(z);
{   && ⋀_{i=1}^{n} TR[[b_i ⇒ a_i]] }
```

```
function region_p_s(z : T) : region
  reads region_p_s(z);
  decreases region_p_s(z));
{
  ret := fpt(⋀_{i=1}^{n} b_i ⇒ a_i);
}
```

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

```
predicate list(n : Node<T>, se: sequence<T>)
  reads region_list(n, se);
  decreases region_list(n, se);
{
    (n = null ⇒ se = [])  &&
    (n ≠ null ⇒ ( ∃ m. n.val = se[0] && n.next = m && list(m, se[1..])
        && region{n.val} !! region{n.next} !! region_list(m, se[1..])))
}
function region_list(n : Node<T>, se: sequence<T>)
  reads region_list(n, se);
  decreases region_list(n, se);
{
  ret :=
    if (n = null) then region{} +
    if (n ≠ null) then
       region{n.val} + region{n.next} + region_list(n.next, se[1..]);
}
```

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 (*LIntro*$_s$ and *RIntro*$_s$) is encoded to the followings:

$$(\texttt{TR}[\![\textit{LIntro}_s]\!])\ \frac{\texttt{TR}[\![a]\!] \vdash_u^\Gamma \texttt{TR}[\![a']\!]}{\texttt{TR}[\![p_s(e)]\!] \vdash_u^\Gamma \texttt{TR}[\![a']\!]}\ \textbf{\textit{where}}\ a = (b \Rightarrow a)[\overline{e}/\textit{formals}_s(p_s)]$$

$$(\texttt{TR}[\![\textit{RIntro}_s]\!])\ \frac{\texttt{TR}[\![a']\!] \vdash_u^\Gamma \texttt{TR}[\![a]\!]}{\texttt{TR}[\![a']\!] \vdash_u^\Gamma \texttt{TR}[\![p_s(\overline{e})]\!]}\ \textbf{\textit{where}}\ a = (b \Rightarrow a)[\overline{e}/\textit{formals}_s(p_s)]$$

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.

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \overline{\mathit{Interface}}\ \ \overline{\mathit{Class}}\ \ S \\
\mathit{Class} & ::= & \ldots \ \mid \texttt{class}\ C\ [\texttt{extends}\ C']\ [\texttt{implements}\ \overline{I}]\ \{\ \overline{\mathit{Member}}\ \} \\
\mathit{Interface} & ::= & \texttt{interface}\ I\ [\texttt{extends}\ \overline{I'}]\ \{\ \overline{\mathit{MHead}}\ \} \\
\mathit{Member} & ::= & \ldots \ \mid \mathit{MHead} \\
\mathit{MHead} & ::= & \texttt{method}\ m(\overline{x:T})\ \ [:T'] \\
\mathit{Expr} & ::= & \ldots \ \mid x\ \texttt{is}\ T \\
S & ::= & \ldots \ \mid x := (T)\,y; \ \mid x := y.m(\overline{G})\,; \\
P & ::= & \ldots \ \mid x{:}T
\end{array}
$$

Figure 8.1: The extended syntax with OO features

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

Notation	Description
implements(T)	The interface names that are directly and transitively implemented by T, where T is a class name or an interface name, where $T \notin$ *implements*(T)
super(C)	The class name that C directly extends (if any)
supers(C)	The class names and interface names that are directly and transitively extended and implemented by class C, where $C \notin$ *supers*(C)
formals(T, m)	The method $T.m$'s formal parameters, where T is a class name or an interface name
body(C, m)	The method $C.m$'s body, where C is a class name

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 **`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 Γ, maps from identifiers to types:

$$\Gamma \in \mathit{VarTypeEnv} = \mathit{Ids} \xrightarrow{\text{fin}} T$$

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 $\overline{T} \to T'$. To streamline the presentation, the auxiliary functions defined in Table. 8.1 are used.

The subtype relation, $\leqslant$:, is a partial, reflexive and transitive relation on types. **`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:

$$C \leqslant: \texttt{Object} \qquad C \leqslant: C \qquad \frac{T \in \mathit{supers}(C)}{C \leqslant: T} \qquad \frac{I' \in \mathit{implements}(I)}{I \leqslant: I'}$$

$$\frac{T_1 \leqslant: T_2 \qquad T_2 \leqslant: T_3}{T_1 \leqslant: T_3} \qquad \frac{\overline{T_2} \leqslant: \overline{T_1} \qquad T_1' \leqslant: T_2'}{\overline{T_1} \to T_1' \leqslant: \overline{T_2} \to T_2'}$$

The functions $\mathit{dfields}(C)$ and $\mathit{dmeths}(C)$ define the type of fields and methods that the class C defines respectively. The functions $\mathit{dfields}(I)$ and $\mathit{dmeths}(I)$ define the type of fields and methods that the interface I defines, respectively. The functions $\mathit{ifields}(T)$ and $\mathit{imeths}(T)$ define the type of fields and methods that the type T inherits, respectively. The function $\mathit{fields}(T)$ 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 $\mathit{meths}(T)$ defines the type of methods that the type T defines or inherits.

$$\mathit{dfields}(T) = \{(f : T') | \ (\texttt{var}\ f : T') \text{ is declared in the reference type} T\}$$

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

$$\mathit{fields}(T) = \mathit{dfields}(T) \cup \mathit{ifields}(T)$$

$$\mathit{dmeths}(T) = \{(T, m) \mapsto (\overline{T} \to T') | \ m(\overline{T}) : T' \text{ is declared in the reference type } T\}$$

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

$$\mathit{meths}(T) = \mathit{dmeths}(T) \cup \mathit{imeths}(T)$$

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

$$\frac{\Gamma \vdash x : T'}{\Gamma \vdash x\ \texttt{is}\ T : \texttt{bool}} \quad \textbf{where } \mathit{isRef}(T') \text{ and } T \leqslant: T'$$

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.

$$\frac{\Gamma \vdash x : T' \qquad \Gamma \vdash G : T}{\Gamma \vdash x := G; : ok(\Gamma)}$$

where $x \neq$ **this** and $T \leqslant: T'$

$$\frac{\Gamma \vdash x : T'}{\Gamma \vdash x := \textbf{new}\ T; : ok(\Gamma)}$$

where $x \neq$ **this** and $T \leqslant: T'$

$$\frac{\Gamma \vdash x : T'}{\Gamma \vdash x := y.f; : ok(\Gamma)}$$

where $x \neq$ **this**, $(f : T) \in \mathit{fields}(\Gamma(y))$ and $T \leqslant: T'$

$$\frac{\Gamma \vdash y : T}{\Gamma \vdash x.f := y; : ok(\Gamma)}$$

where $x \neq$ **this**, $(f : T') \in \mathit{fields}(\Gamma(x))$ and $T \leqslant: T'$

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

$$\frac{\Gamma \vdash x : T'}{\Gamma \vdash x := (T)y; : ok(\Gamma)}$$

where $x \neq$ **this**, $\mathit{isRef}(T')$ and $T \leqslant: T'$

$$\frac{\Gamma \vdash \overline{E} : \overline{T_1} \qquad \Gamma \vdash x : T''}{\Gamma \vdash x := y.m(\overline{E}) : ok(\Gamma)}$$

where $((\Gamma(y), m) \mapsto (\overline{T} \to T')) \in \mathit{meths}(\Gamma(y))$, $T' \leqslant: T''$ and $\overline{T_1} \leqslant: \overline{T}$

$$\frac{\Gamma \vdash \overline{E} : \overline{T_1} \qquad \Gamma \vdash x : T''}{\Gamma \vdash x := \textbf{super}.m(\overline{E}) : ok(\Gamma)}$$

where **this** $\in \mathit{dom}(\Gamma), C = \mathit{super}(\Gamma(\textbf{this}))$, $(C, m) \mapsto (\overline{T} \to T')) \in \mathit{meths}(C), T' \leqslant: T''$ and $\overline{T_1} \leqslant: \overline{T}$

The typing rules for assertions are adjusted as follows:

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 = E_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 \neq E_2 : \texttt{bool}}$$

where $T_1 \leqslant: T_2$ or $T_2 \leqslant: T_1$ **where** $T_1 \leqslant: T_2$ or $T_2 \leqslant: T_1$

$$\frac{\Gamma \vdash x : T' \qquad \Gamma \vdash E : T_2}{\Gamma \vdash x.f = E : \texttt{bool}}$$

where *isRef*(T'), $(f : T_1) \in$ *fields*(T') and $(T_1 \leqslant: T_2$ or $T_2 \leqslant: T_1)$

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

$$\frac{\Gamma \vdash x : T'}{\Gamma \vdash x : T : \texttt{bool}} \quad \textbf{where } \textit{isRef}(T') \text{ and } T \leqslant: T'$$

8.2 Semantics

The function, *RefCtx*, is used to denote sets of reference contexts that map references, *Ref* , to class names:

$$\rho \in \textit{RefCtx} = \textit{Ref} \xrightarrow{\text{fin}} \textit{ClassName}$$

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

$$s \in \textit{State} = \textit{Store} \times \textit{RefCtx} \times \textit{Heap} \cup \{\textit{err}\}$$

A Γ-*state* (ρ, σ, H) is a state such that $\textit{dom}(\Gamma) = \textit{dom}(\sigma)$.

There is a method environment, θ, 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

$$\theta = \textit{fix}(\lambda g \,.\, \lambda(C, m) \,.\, \underline{\lambda} s \,.\, \mathcal{MS}[\![\Gamma \vdash \textit{body}(C, m) : \textit{ok}(\Gamma')]\!](g)(s))$$

The underlined lambda ($\underline{\lambda}$) denotes a strict function that cannot recover form a nonterminating computation [81]. The semantic functions are re-defined as follows.

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

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

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

$$\mathcal{E}[\![\Gamma \vdash x \texttt{ is } T : \texttt{bool}]\!](\sigma, \rho) = \sigma(x) \neq null \ and \ \rho(\sigma(x)) \leqslant: T$$

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

$$\begin{aligned}
&\mathcal{MS}[\![\Gamma \vdash x := \texttt{new}\, T; : ok(\Gamma)]\!](\theta)(\sigma, \rho, h) = \textbf{let } (l, h') \ = \ allocate(T, h) \textbf{ in} \\
&\quad \textbf{let } (f_1 : T_1, \ldots, f_n : T_n) \ = \ \textit{fields}(T) \textbf{ in} \\
&\quad\quad \textbf{let } \sigma' \ = \ \sigma[x \mapsto l] \textbf{ in} \\
&\quad\quad\quad \textbf{let } \rho' \ = \ \textit{Extend}(\rho, l, T) \textbf{ in} \\
&\quad\quad\quad\quad (\sigma', \rho', h'[(\sigma'(x), f_1) \mapsto \textit{default}(T_1), \ldots, (\sigma'(x), f_n) \mapsto \textit{default}(T_n)])
\end{aligned}$$

The semantics for a type cast statement is defined below:

$$\begin{aligned}
&\mathcal{MS}[\![\Gamma \vdash x := (T)y; : ok(\Gamma)]\!](\theta)(\sigma, \rho, h) = \\
&\quad \textbf{if } \sigma(y) = null \text{ or } \rho(\sigma(y)) \leqslant: T \textbf{ then } \ (\sigma[x \mapsto \sigma(y)], \rho, h) \textbf{ else } \ err
\end{aligned}$$

The semantics for a dynamic method call is defined below:

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

if $\sigma(y) \neq null$ **then**

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

let σ'' = *Extend*$(\sigma, (\texttt{this}, \overline{z}), (\sigma(y), \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma,\rho)}))$**in**

if $(\rho(\sigma(y)), m) \in \theta$ **then**

let v = $(\theta(\rho(\sigma(y), m))(\sigma'', \rho, h)$**in**

if $v = (\sigma', \rho', h')$ **then** $(\sigma[x \mapsto \sigma'(\texttt{ret})], \rho', h')$

else if $v =$ *err* **then** *err* **else** $\perp$

else *err*

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.**$\mathtt{m}\,(\overline{G})$.

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

let C = *super*$(\Gamma(\texttt{this}))$ **in**

let $\overline{z}$ = *formals*(C, m) **in**

let σ'' = *Extend*$(\sigma, \overline{z}, \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma,\rho)})$**in**

if $(C, m) \in \theta$ **then**

let v = $(\theta(C, m))(\sigma'', \rho, h)$**in**

if $v = (\sigma', \rho', h')$ **then** $(\sigma[x \mapsto \sigma'(\texttt{ret})], \rho', h')$

else if $v =$ *err* **then** *err* **else** $\perp$

else *err*

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

Lemma 1A. *Let* Γ *and* Γ' *be two well-formed type environments. Let S be a statement, such that* $\Gamma \vdash S : ok(\Gamma')$. *Let* Γ'' *be a well-formed type environment, such that* $dom(\Gamma) \cap dom(\Gamma'') = \varnothing$ *and* $dom(\Gamma') \cap dom(\Gamma'') = \varnothing$. *Then*

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

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

The semantics of assertions is re-defined by using ρ in place of the function *type*. By the definition of ρ and *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:

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

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

Lemma 2A. *Let* Γ *and* Γ' *be two well-formed type environments such that* $dom(\Gamma) \cap dom(\Gamma') = \varnothing$. *Let* (σ, ρ, h) *be a* Γ*-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, δ, then the two states must agree on its dynamic types of objects as well.

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

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

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

The read effects of the type test expression is defined as $efs(x \texttt{ is } T) = \texttt{reads } x$. 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 **`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 `valid` is an invariant of the class `Node<T>`. The field `fpt` is a ghost field that computes the regions that frame the predicate `valid`. The method `add` prepends the node `n` to the linked-list. At the last line of its implementation, the ghost field `fpt` of the object **`ret`** is updated to frame its predicate `valid`. It is said that **`reads`** `fpt` is the read effects of the class `Node<T>`. For ease of the discussion, it is assumed that there is a function *bnd* that maps a class name to its read effects.

8.4 Supertype Abstraction and Local Reasoning

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 *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 $\{P_T\}\ T.m(\overline{x : T'})\ \{Q_T\}[\varepsilon_T]$ in FRL. Let R be a predicate whose read effects are separate from the write effects of ε_T. In the spirit of

supertype abstraction, it would be ideal if the following were valid:

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

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 $\texttt{old}(P_T) \wedge Q_S \Rightarrow Q_T$.

Because their work ignores frame conditions with the assumption that they could be encoded to postconditions, the relation between ε_T and ε_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 δ be the read effect of R in a state where P_T holds, i.e., such that $P_T \vdash^{\Gamma} \delta \textit{ frm } R$. As R is preserved during the execution of the method $T.m$, then it must be true that $P_T \;\&\&\; R \Rightarrow \delta \cdot\!/\!\cdot \varepsilon_T$. As $P_T \Rightarrow P_S$, using the rule ***FrmProjCtx*** in Fig. 3.6, it must be that $P_S \vdash^{\Gamma} \delta \textit{ frm } R$.

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 $P_T \;\&\&\; R \;\&\&\; \delta \cdot\!/\!\cdot \varepsilon_S$ is true. Consider different cases of the relation between ε_S and ε_T.

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

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

3. Suppose $\varepsilon_S \nleqslant \varepsilon_T$. In this case, ε_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 ε_S into two parts: ε_{S_1} and ε_{S_2}, where $\varepsilon_{S_1} \leqslant \varepsilon_T$ and $\varepsilon_{S_2} \cap \varepsilon_T = \varnothing$. Following the previous two cases, validity requires that $P_S \;\&\&\; R \;\&\&\; \delta \cdot\!/\!\cdot \varepsilon_{S_1}$.

Since reasoning at the level of the supertype knows nothing about ε_{S_2}, the regions in ε_{S_2} must be such that R cannot possibly depend on them. Following standard software engineering practice, it is called lack of dependency on ε_{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: $R_{\mathtt{c}}$!! $R_{\mathtt{rc}}$!! R_{fc},[2]

[2]The formula R_1 !! R_2 !! R_3 means that the three regions are pointwise disjoint.

```
class Cell {
  var val : int;

  method Cell(){
    val := 0;
  }

  method set(v : int}{
    this.val := v;
  }

  method get() : int{
    ret := this.val;
  }
}
```

```
class ReCell extends Cell {
  val bak : int;

  method ReCell(){
    super();  bak := 0;
  }

  method set(v : int}{
    this.bak := super.get();
    super.set(v);
  }
}

class FCell extends Cell {
  var fcc : Cell;

  method FCell(){
    super();
    fcc := new Cell();
  }
}
```

Figure 8.3: Classes `Cell`, `ReCell` and `FCell`

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* Γ *be a well-formed type environment, and* C *be a class. Let RE be a subregion of bnd*(C)*, i.e.,* $RE < bnd(C)$*. Then the class* C encapsulates *RE only if for all* Γ*-states* (σ, ρ, h) *and for all* $x : C$*,* $x' : C'$*,* $x \neq x'$ $(\sigma, \rho, h) \models^{\Gamma} RE[x/\textbf{\textit{this}}]$ *!!* $bnd(C')[x'/\textbf{\textit{this}}]$. ∎

Definition 22 (Encapsulation)**.** *Let* Γ *be a well-formed type environment, and* C *be a class. Then*

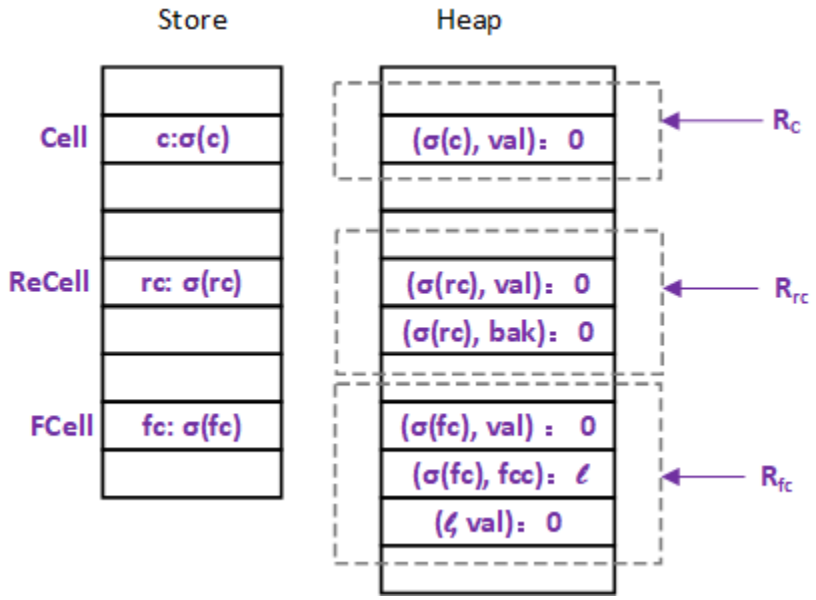

Figure 8.4: Encapsulation example

the class C is encapsulated only if for all $x : C$, $x' : C'$, Γ-states (σ, ρ, h), such that $x \neq x'$, either $(\sigma, \rho, h) \models^{\Gamma}$ bnd$(C)[x/\mathbf{this}] <$ bnd$(C')[x'/\mathbf{this}]$ or for all subregions of bnd(C), RE, C encapsulates RE. ∎

In Def. 22, the case where *bnd*$(C)[x/\mathbf{this}] <$ *bnd*$(C')[x'/\mathbf{this}]$ 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$[x/\mathbf{this}]$ is $P/\mathbf{modifies}$ bnd$(C')[x'/\mathbf{this}]$-immune,*

Proof. Let $\mathbf{region}\{x.f_1.\ldots.f_n\}$ in *RE* be arbitrary. By the definition of read effect in Fig. 3.5, *efs*$(\mathbf{region}\{x.f_1.\ldots.f_n\}) = \mathbf{reads}\ x, \mathbf{region}\{x.f_1\}.\ldots.\mathbf{region}\{x.f_1.\ldots.f_{n-1}\}$. Let R be the union of all regions in *efs*$(\mathbf{region}\{x.f_1.\ldots.f_n\})$. Let $r' = \mathbf{region}\{x'.g_1.\ldots.g_m\}$ in

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 **region**$\{x.f_1.\ldots.f_i\} \in R$, where $1 < i \leqslant n-1$, 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.

```
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 *bnd*$(Cell)[c/$**this**$]$ = **reads region**$\{c.val\}$, and the read effect of the object e is *bnd*$(ECell)[e/$**this**$]$ = (**reads region**$\{e.val\}$ + **region**$\{e.ecc\}$ + **region**$\{e.ecc.val\}$). Let P be $c \neq null$ && $e \neq null$ && $e.ecc \neq null$. 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:

```
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`

.

```
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, ε_S in a subtype S is divided into two parts, ε_{S_1} and ε_{S_2}, where $\varepsilon_{S_1} \leqslant \varepsilon_{S_2}$, $\varepsilon_{S_2} \cap \varepsilon_T = \varnothing$ and $\mathit{writeR}(\varepsilon_{S_2})$, 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, $\mathit{writeR}(\varepsilon_{S_2}) \leqslant$

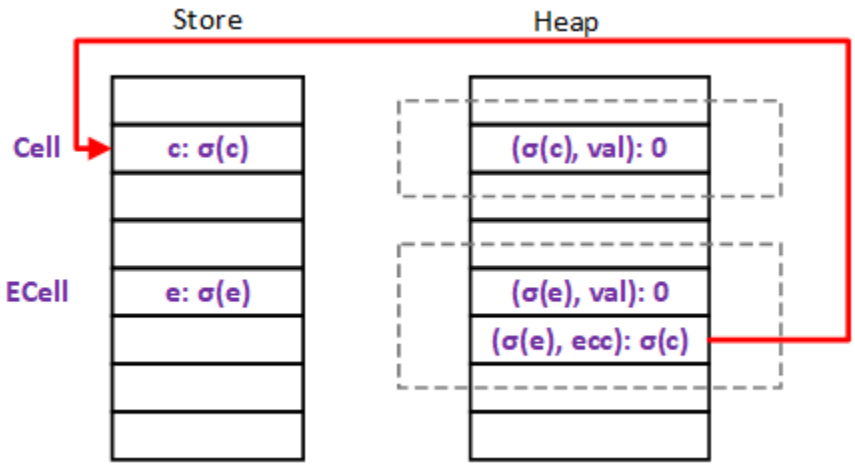

Figure 8.6: Argument exposure example

$(bnd(S) - bnd(T))$.

The previous analysis leads to the following definition of specification refinement.

Definition 23 (Specification Refinement)**.** *Let S and T be two types, such that $S \leqslant: T$. Let $\{P_T\}\ T.m(\overline{x : T'})\ \{Q_T\}[\varepsilon_T]$ and $\{P_S\}\ S.m(\overline{x : T'})\ \{Q_S\}[\varepsilon_S, \textbf{\texttt{modifies}}\ RE]$ be specifications of a method m in T and in S respectively, where $\overline{x : T'}$ are its formal parameters. Then the specification of $S.m$ refines the specification of $T.m$ if $P_T \Rightarrow P_S$, $\textbf{\texttt{old}}(P_T) \wedge Q_S \Rightarrow Q_T$, $\varepsilon_S \leqslant \varepsilon_T$, $\textbf{\texttt{modifies}}\ RE \notin \varepsilon_T$ and RE is P_T/ε_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 $\{P_T\}\_\{Q_T\}[\varepsilon_T]$ **`also`** $\{P_S\}\_\{Q_S\}[\varepsilon_S, RE]$ is defined as follows, where $\varepsilon_S \leqslant \varepsilon_T$, **`modifies`** $RE \notin \varepsilon_T$ and RE is P_T/ε_T-immune:

$$\{P_T \mid\mid P_S\}\_\{(\textbf{\texttt{old}}(P_T) \Rightarrow Q_T)\ \&\&\ (\textbf{\texttt{old}}(P_S) \Rightarrow Q_S)\}[\ \varepsilon_T, \varepsilon_S, RE], \tag{8.2}$$

The following lemma justifies the definition of the semantics of **`also`**.

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

$\{P_S\}S.m(\overline{x:T'})\{Q_S\}[\varepsilon_S, \mathit{RE}]$, *where* $\varepsilon_S \leqslant \varepsilon_T$, **`modifies`** *RE* $\notin \varepsilon_T$ *and RE is* P_T/ε_T*-immune. Then the specification of* $T.m$ *is refined by the specification of* $S.m$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash_r^\Gamma \{P_T \mid\mid P_S\}\_\{(\texttt{old}(P_T) \Rightarrow Q_T) \mathbin{\&\&} (\texttt{old}(P_S) \Rightarrow Q_S)\}[\varepsilon_T, \varepsilon_S, \mathit{RE}]}{\textbf{where } P_T \Rightarrow P_T \mid\mid P_S \text{ and } P_T \mathbin{\&\&} (\texttt{old}(P_T) \Rightarrow Q_T) \Rightarrow Q_T}(\mathit{CONSEQ}_r)}{\vdash_r^\Gamma \{P_T\}\_\{Q_T \mathbin{\&\&} (\texttt{old}(P_S) \Rightarrow Q_S)\}[\varepsilon_T, \varepsilon_S, \mathit{RE}]}}{\textbf{where } Q_T \mathbin{\&\&} (\texttt{old}(P_S) \Rightarrow Q_S) \Rightarrow Q_T}(\mathit{CONSEQ}_r)}{\vdash_r^\Gamma \{P_T\}\_\{Q_T\}[\varepsilon_T, \varepsilon_S, \mathit{RE}]}}{\vdash^\Gamma (\varepsilon_T, \varepsilon_S, \mathit{RE}) \leqslant (\varepsilon_T, \mathit{RE}) \qquad \textbf{where } \varepsilon_s \leqslant \varepsilon_T}(\mathit{SUBEFF}_r)}{\vdash_r^\Gamma \{P_T\}\_\{Q_T\}[\varepsilon_T, \mathit{RE}] \qquad \textbf{where } P_T \Leftrightarrow (P_T \mathbin{\&\&} R)}(\mathit{CONSEQ}_r)}{\vdash_r^\Gamma \{P_T \mathbin{\&\&} R\}\_\{Q_T\}[\varepsilon_T, \textbf{`modifies`}\ \mathit{RE}]}$$

$$\dfrac{\dfrac{\dfrac{\textbf{where `modifies` } \mathit{RE} \notin \varepsilon_T \text{ and } \mathit{RE} \text{ is } P_T/\varepsilon_T\text{-immune}}{\vdash_r^\Gamma \{P_T \mathbin{\&\&} R\}\_\{Q_T\}[\varepsilon_T]}(\mathit{FieldMask}_r)}{\textbf{where } P_T \Leftrightarrow (P_T \mathbin{\&\&} R)}(\mathit{CONSEQ}_r)}{\vdash_r^\Gamma \{P_T\}\_\{Q_T\}[\varepsilon_T]}$$

□

8.5.1 Correctness Judgment

A judgment with hypothesis is written as follows:

$$\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$$

where Δ 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 $\{P\}T.m(\overline{x:T'})\{Q\}[\varepsilon]$.

The specifications are obtained from declared specifications and combinations of specification from supertypes, e.g., a specification inheritance in JML [45]. A specification context Δ is Γ-valid if all the specifications in Δ are well-typed under Γ. A Δ-method environment θ means $dom(\Delta) = dom(\theta)$.

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

Def. 25 defines the meaning of a specification context.

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

1. $(\theta(T, m))(\sigma, \rho, h) \neq err \iff (\sigma, \rho, h) \vDash^{\Gamma} P$
2. *if $(\sigma, \rho, h) \vDash^{\Gamma} P$ and $(\sigma', \rho', h') = (\theta(T, m))(\sigma, \rho, h)$, then $(\sigma', \rho', h') \vDash^{\Gamma'} Q$ and*
   (a) *for all $x \in dom(\sigma)$, if $\sigma(x) \neq \sigma'(x)$, then* **`modifies`** $x \in \varepsilon$
   (b) *for all $(o, f) \in dom(h)$, if $h'(o, f) \neq h(o, f)$, then $(o, f) \in \mathcal{E}[\![\Gamma \vdash writeR(\varepsilon) :$* **`region`** $]\!](\sigma, \rho)$
   (c) *for all $(o, f) \in \mathcal{E}[\![\Gamma' \vdash freshR(\varepsilon) :$* **`region`** $]\!](\sigma', \rho')$, $(o, f) \in (dom(h') - dom(h))$ ∎

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.

$\mathcal{MS}$: *Typing Judgment* → *S* → *MethEnv* → *Specification Context* → *State* → $State_{\perp} + \{p\text{-}fault\}$

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:

$$
\begin{array}{l}
\mathcal{MS}[\![\Gamma \vdash x := y.m(\overline{G}); : ok(\Gamma)]\!](\theta)(\sigma,\rho,h) = \\
\quad \textbf{if } \sigma(y) \neq null \textbf{ then} \\
\quad\quad \textbf{let } \overline{z} \;=\; \mathit{formals}(\rho(\sigma(y)), m) \textbf{ in} \\
\quad\quad\quad \textbf{let } \overline{v} \;=\; \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma,\rho)} \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } \sigma'' \;=\; \mathit{Extend}(\sigma, (\texttt{this}, \overline{z}), (\sigma(y), \overline{v})) \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{let } \{P\}\_\{Q\}[\varepsilon] \;=\; \Delta(\rho(\sigma(y)), m) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad \textbf{if } \sigma,\rho,h \models^{\Gamma} P[\overline{v}/\overline{z}] \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{if } (\rho(\sigma(y)), m) \in \theta \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let } v \;=\; (\theta(\rho(\sigma(y), m))(\sigma'', \rho, h) \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } v = (\sigma', \rho', h') \textbf{ then } \; (\sigma[x \mapsto \sigma'(\texttt{ret})], \rho', h') \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } v = \mathit{err} \textbf{ then } \; \mathit{err} \textbf{ else } \bot \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{err} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{p\text{-}fault} \\
\quad \textbf{else } \mathit{err}
\end{array}
$$

and a static method call:

$$
\begin{array}{l}
\mathcal{MS}[\![\Gamma \vdash x := \texttt{super}.m(\overline{G});: ok(\Gamma)]\!](\theta)(\sigma,\rho,h) = \\
\quad \textbf{let } C \;=\; \mathit{super}(\Gamma(\texttt{this})) \textbf{ in} \\
\quad\quad \textbf{let } \overline{z} \;=\; \mathit{formals}(C,m) \textbf{ in} \\
\quad\quad\quad \textbf{let } \overline{v} \;=\; \overline{\mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma,\rho)} \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } \sigma'' \;=\; \mathit{Extend}(\sigma,\overline{z},\overline{v}) \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{let } \{P\}\_\{Q\}[\varepsilon] \;=\; \Delta(C,m) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad \textbf{if } \sigma,\rho,h \models^{\Gamma} P[\overline{v}/\overline{z}] \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{if } (C,m) \in \theta \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let } v \;=\; (\theta(C,m))(\sigma'',\rho,h) \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } v = (\sigma',\rho',h') \textbf{ then } \; (\sigma[x \mapsto \sigma'(\texttt{ret})],\rho',h') \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } v = \mathit{err} \textbf{ then } \; \mathit{err} \textbf{ else } \bot \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{err} \\
\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{p\text{-}fault} \\
\quad\quad\quad\quad\quad \textbf{else } \mathit{err}
\end{array}
$$

A valid FRL Hoare-Formula is defined as follows.

Definition 26 (Valid FRL Hoare-Formula with Hypothesis)**.** *Let Γ be a well-formed type environment, and Δ be a Γ-valid behaviorally-subtyped specification context, and θ be a Δ-interpretation. Let S be a statement. Let P and Q be assertions, ε be effects, and (σ,ρ,h) be a Γ-state. Then* $\{P\}\ S\ \{Q\}[\varepsilon]$ is valid *in (σ,ρ,h) under Γ and θ written $(\sigma,\rho,h);\theta \models_r^{\Gamma} \{P\}\ S\ \{Q\}[\varepsilon]$, if and only if whenever $(\sigma,\rho,h) \models^{\Gamma} P$, then*

1. $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\theta)(\sigma,\rho,h) \neq \mathit{err}$,
2. $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\theta)(\sigma,\rho,h) \neq \mathit{p\text{-}fault}$,
3. *if* $(\sigma',\rho',h') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\theta)(\sigma,\rho,h)$, *then* $(\sigma',\rho',h') \models^{\Gamma'} Q$ *and*

(a) for all $x \in dom(\sigma)$*:* $\sigma'(x) \neq \sigma(x)$: **modifies** $x \in \varepsilon$

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

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

A FRL judgment $\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$ is valid *if and only if for all* Δ*-interpretation* θ*, and all* Γ*-states* $s :: s; \theta \models_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$. ∎

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:

$(\textit{TypeCast}_r)$ $\vdash_r^\Gamma \{y = null \mid\mid y : T'\}\ x := (T)y;\ \{x = y\}\ [\textbf{modifies}\ x]$ **where** $T' \leqslant: T$

The axiom for dynamic method calls that is adapted from Banerjee and Naumann's work [2] as follows:

$(\textit{Dcall}_r)$ $\Delta, \{P\}\ T.m(\overline{z : T})\ \{Q\}[\varepsilon] \vdash_r^\Gamma \{x \neq null\ \&\&\ P[\overline{G}/\overline{z}]\}\ x.m(\overline{G})\ \{Q[\overline{G}/\overline{z}]\}[\varepsilon[\overline{G}/\overline{z}]]$

where $\Gamma(x) = T$

where $P[\overline{G}/\overline{z}]$ simultaneously substitutes $\overline{G}$ for $\overline{z}$ in P. The axiom for static method calls is:

$$(\textit{SCall}_r)\ \Delta, \{P\}\ C.m(\overline{z : T})\ Q\}[\varepsilon] \vdash_r^\Gamma \{P[\overline{G}/\overline{z}]\}\ C.m(\overline{G})\ \{Q[\overline{G}/\overline{z}]\}[\varepsilon[\overline{G}/\overline{z}]]$$

The following two structural rules show that additional type declarations and method declarations do not invalidate proved statements.

$$(\textit{TypeExt}_r)\ \frac{\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]}{\Delta \vdash_r^{\Gamma,\,\Gamma'} \{P\}\ S\ \{Q\}[\varepsilon]} \qquad (\textit{MethExt}_r)\ \frac{\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]}{\Delta, \Delta' \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]}$$

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 (σ, ρ, h) and (σ', ρ', h') be pre- and post-state respectively.

TypeCast$_r$: In this case, the precondition, $y = null \mid\mid y : T'$, is assumed. There are two cases:

- $y = null$: 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, $T' \leqslant: T$, it is true that $\rho(\sigma(y)) \leqslant: T$. 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, $x \neq null \;\&\&\; P[\overline{G}/\overline{z}]$, is assumed. By the side condition, the static type of x is T. Because Δ is behaviorally-subtyped, it is sound to use the specification of $T.m$ as hypothesis. Let θ be its interpretation, such that $(T, m) \in \mathit{dom}(\theta)$. By the definition of specification context interpretation (Def. 25) and the assumption that the precondition is true, it is true $(\theta(T, m))(\sigma, \rho, h) \neq \mathit{err}$, $(\sigma', \rho', h') = (\theta(T, m))(\sigma, \rho, h)$ and $(\sigma', \rho', h') \models^{\Gamma} Q[\overline{G}/\overline{z}]$. And the effect, $\varepsilon[\overline{G}/\overline{z}]$, 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, $\mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma') = \varnothing$. 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, $\mathcal{MS}[\![\Gamma, \Gamma' \vdash S : ok(\Gamma'')]\!](\sigma, \rho, h) = (\sigma', \rho', h')$. By Lemma 2, the postcondition follows as well. As the original specification is well-typed, $\mathit{dom}(\Gamma') \cap \mathtt{FV}(S) = \varnothing$. Thus, the effect is just ε as well.

MethExt$_r$ Let $\Gamma' = \{P_{T'}\}T'.m(\overline{x : T})\{Q_{T'}\}[\varepsilon_{T'}]$. There are two cases.

1. the method $T'.m$ inherits from $T.m$, where $S \leqslant: T$. 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 Γ be a type environment. Let Δ be a Γ-valid behaviorally-subtyped specification context. Let P and Q be assertions, S be a statement, and ε be effects. The judgment, $\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$, is modularly sound if and only if for all $\Gamma' \geqslant \Gamma$, for all Γ'-valid behaviorally-subtyped specification contexts, Δ', such that $\Delta' \geqslant \Delta$, and for all Δ'-interpretations, θ',*

$$\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon] \Rightarrow \theta' \vDash_r^{\Gamma'} \{P\}\ S\ \{Q\}[\varepsilon].$$ ∎

Theorem 10. *Let Γ be a well-formed type environment. Let Δ be a Γ-valid behaviorally-subtyped specification context. Let P and Q be assertions, S be a statement, and ε be effects. The judgment, $\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$, is modularly sound.* ∎

Proof. Assume $\Delta \vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$. By Lemma 26, it must be true that $\Delta, \Delta' \vdash_r^{\Gamma,\Gamma'} \{P\}\ S\ \{Q\}[\varepsilon]$ Then the theorem is proved by the soundness theorem (Theorem 1).

□

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, `df`, which stores an object's frame, e.g., `df` stores the dynamic frames of a linked-list. Another field `exposed` is used to store the locations that may be shared with other objects. It has default value **`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 **`Object`** such that:

```
class Object{  var df : region;  var exposed : region;  }
```

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 $\varepsilon_S \leqslant \varepsilon_T$, 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

```
var spec_public val : int;
```

is a shorthand for the declaration

```
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`.

```
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 **this**.ecc, when it is not null. This is specified by the first assertion in the body of the predicate inv. The assertion **this**.ecc $\neq$ null ? ecc.df : **region**{} is a conditional region expression, which abbreviates the assertion **this**.ecc = t && t $\neq$ null ? ecc.df : **region**{}, where t is fresh.

The method ECell.set updates both val and ecc.val with the new value. Its write effects is specified by the function wf_set. To make sure an incremental definition of the function, **super**.wf_set is forced to be called in its body. This is enforced by a syntactic check. Its body adds the region **region**{ecc.val} as well, which belongs to the extended state of the class ECell. By the definition of specification refinement (Def. 23), the proof obligation is to show that the region **region**{ecc.val} is encapsulated, i.e., it is disjoint from what may be exposed, i.e., **this**.exposed. This disjointness condition has to be true for all the states of the class ECell's objects. Therefore, this condition is defined in the predicate 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 **this**.exposed = ecc.df, and by the specification of the class Cell, 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[1]

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 $\texttt{TR}[\![a]\!]$ have the same semantic footprints, i.e., $fpt_s(a) = ftpt(\texttt{TR}[\![a]\!])$, where *ftpt* is the semantic footprint function for the UFRL (or FRL) assertions. The specification of the method `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.

[1]Part of the content in this chapter is submitted to *Formal Aspect of Computing*.

(a) If the write effect of the method is specified, or if the method is decorated by the keyword **pure** (that is a shorthand for a frame of **modifies** $\varnothing$), then verifiers consider that the specification is written in the style of FRL, set the read effects to the default value, **reads alloc**$\downarrow$, 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 $\Delta_u \vdash_u^\Gamma [\delta]\{P_1\}\, S\, \{P_2\}[\varepsilon]$, which states that S satisfies its Hoare-formula under certain hypotheses, Δ_u, which map pairs of a class and a method name to the corresponding method's specification. Hypotheses are given by the grammar:

$$\Delta_u \ ::= \ \varnothing \ \mid \ \Delta_{u_1}, \Delta_{u_2} \ \mid \ [\delta]\,\{P\}T.m\,(\overline{x:T})\,\{Q\}\,[\varepsilon]$$

Recall the hypotheses defined in Section 8.5 as follows:

$$\Delta_r \ ::= \ \varnothing \ \mid \ \Delta_{r_1}, \Delta_{r_2} \ \mid \ \{P\}T.m\,(\overline{x:T})\,\{Q\}\,[\varepsilon]$$

Hypotheses specified by SSL are given by the following grammar [73]:

$$\Delta_s \ ::= \ \varnothing \ \mid \ \Delta s_1, \Delta s_2 \ \mid \ ssl\{a\}T.m\,(\overline{x:T})\,\{a'\}\,[X]$$

where $X = \mathtt{MV}(S)$ and $S = \mathit{body}(C, m)$.

There is a syntactic translation function, $\mathtt{TR}_\Delta$, that maps the hypotheses in FRL and SSL to those in UFRL as follows:

$$
\begin{array}{lcl}
\mathtt{TR}_\Delta[\![\varnothing]\!] & = & \varnothing \\
\mathtt{TR}_\Delta[\![\Delta_{r_1}, \Delta_{r_2}]\!] & = & \mathtt{TR}_\Delta[\![\Delta_{r_1}]\!], \mathtt{TR}_\Delta[\![\Delta_{r_2}]\!] \\
\mathtt{TR}_\Delta[\![\{P\}T.m(\overline{x:T})\{Q\}[\varepsilon]]\!] & = & [\mathbf{reads\ alloc}\downarrow]\{P\}T.m(\overline{x:T})\{Q\}[\varepsilon] \\
\mathtt{TR}_\Delta[\![\Delta_{s_1}, \Delta_{s_2}]\!] & = & \mathtt{TR}_\Delta[\![\Delta_{s_1}]\!], \mathtt{TR}_\Delta[\![\Delta_{s_2}]\!] \\
\mathtt{TR}_\Delta[\![ssl\{a\}T.m(\overline{x:T})\{a'\}[X]]\!] & = & [\mathbf{reads}\ r\downarrow]\{\mathtt{TR}[\![a]\!]\}T.m(\overline{x:T})\{\mathtt{TR}[\![a']\!]\} \\
 & & [\mathbf{modifies}\ (r\downarrow, X), \mathbf{fresh}(\mathit{fpt}_s(a') - r)] \\
 & & \mathbf{where}\ \mathtt{TR}[\![a]\!] \Rightarrow r = \mathit{fpt}_s(a)\ \mathit{and}\ r \notin X
\end{array}
$$

Another way to encode SSL hypotheses would be

$$
\begin{array}{l}
\mathtt{TR}_\Delta[\![ssl\{a\}T.m(\overline{x:T})\{a'\}[X]]\!] = \\
\quad [\mathbf{reads}\ r\downarrow]\{\mathtt{TR}[\![a]\!]\ \&\&\ r = \mathit{fpt}_s(a)\}T.m(\overline{x:T})\{\mathtt{TR}[\![a']\!]\} \\
\quad [\mathbf{modifies}\ (r\downarrow, X), \mathbf{fresh}(\mathit{fpt}_s(a') - r)] \\
\quad \mathbf{where}\ r \notin X
\end{array}
$$

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 $\mathbf{this} \neq \mathit{null}$ is implicit by the program's semantics. However, the precondition of the method `setX` declared in the class `NumberS` is needed, because the location $\mathbf{region}\{\mathbf{this}.x\}$ 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 $\mathbf{region}\{n.x\}$ and assigns to $\boldsymbol{this}.x$. Thus, its write effect is just $\mathbf{region}\{\mathbf{this}.x\}$. Consider the following client code.

```
class NumberS{
  var x : int;

  method NumberS()
    ensures this.x ↦ 0;
  { this.x := 0; }

  method setX(v: int)
    requires ∃ y.this.x ↦ y;
    ensures this.x ↦ v;
  { this.x := v; }

  method getX() : int
    requires ∃ v.this.x ↦ v;
    ensures this.x ↦ 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 ≠ null;
    modifies region{this.x};
    ensures this.x = n.x + 1;
  { this.x := n.getX() + 1; }

  method getX() : int
    modifies ∅;
    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 Δ_{u_1} and Δ_{u_2} can be extended to **alloc** $\downarrow$ by using the rule *SubEff*$_u$.

$$
\begin{array}{ll}
\Delta_{u_1} = & [\textbf{reads region}\{\textbf{this}.x\}] \\
& \{\exists\, y.\textbf{this}.x = y\}\ \textit{NumberS.setX(v:}\textbf{\textit{int}}\textit{);}\ \{\textbf{this}.x = v\} \\
& [\,\textbf{modifies region}\{\textbf{this}.x\}] \\
\\
\Delta_{u_2} = & [\textbf{reads region}\{\textbf{this}.x\}] \\
& \{\exists\, v.\textbf{this}.x = v\}\ \textit{NumberS.getX();}\ \{\textbf{this}.x = v\ \ \&\&\ \ \textbf{ret} = v\} \\
& [\textbf{modifies region}\{\textbf{this}.x\}] \\
\\
\Delta_{u_3} = & [\textbf{reads alloc}\downarrow] \\
& \{true\}\ \textit{NumberR.setX(v:}\textbf{\textit{int}}\textit{);}\ \{\textbf{this}.x = v\} \\
& [\,\textbf{modifies region}\{\textbf{this}.x\}] \\
\\
\Delta_{u_4} = & [\textbf{reads alloc}\downarrow] \\
& \{n \neq \textbf{null}\}\ \textit{NumberR.addOne(n:NumberS);}\ \{\textbf{this}.x = n.x + 1\} \\
& [\,\textbf{modifies region}\{\textbf{this}.x\}]
\end{array}
$$

$$\Delta_{u_5} = [\textbf{reads alloc}\downarrow]\ \{true\}\ \textit{getX();}\ \{\textbf{ret} = \textbf{this}.x\}\ [\varnothing]$$

Figure 9.3: Translating method specifications in the class *NumberS* and *NumberR*

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* $\mapsto$ 0/**modifies region**$\{$*sNumber.x*$\}$-immune. (9.1)

By the definition of immune (Def. 6), the proof obligation is to show:

$$efs(\mathbf{region}\{rNumber.x\}) \not\cdot \mathbf{modifies}\ \mathbf{region}\{sNumber.x\}, \tag{9.2}$$

which is true. Then the two statements' write effects are accumulated by using $CONSEQ_u$ and $SEQ1_u$:

$$\Delta \vdash_u^{\Gamma} \begin{array}{l} [\mathbf{alloc}\downarrow] \\ \{sNumber.x = 0\} \\ \texttt{sNumber.setX(5);} \quad \texttt{rNumber.addOne(}sNumber\texttt{);} \\ \{sNumber.x = 5 \ \&\& \ rNumber.x = sNumber.x + 1\} \\ [\,\mathbf{modifies}\ \mathbf{region}\{sNumber.x\}, \mathbf{region}\{rNumber.x\}] \end{array} \tag{9.3}$$

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 `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 *mark* satisfies its specification under the hypothesis that recursive calls satisfy the specification being proved. Another method hypothesis is the specification of `unmarked`. When reasoning about *mark*, the specification of the method *unmarked* is used, instead of its body, i.e., if *unmarked*'s precondition is satisfied, its postcondition is assumed after calling it. Because *mark*'s precondition implies the one of *unmarked*, the function *unmarked* can be used to specify the write effect of the method *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**$\downarrow$ 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. $d = null$: 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. $dag(d) \wedge d \neq null \wedge d.mark \mapsto 1$: 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. $dag(d) \wedge d \neq null \wedge\ d.mark \mapsto 0$: 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 `dag`. The following must be true:

$$d.mark \mapsto 0 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r)), \tag{9.4}$$

which implies the precondition of the rule UPD_u, the following is derived:

$$\begin{array}{ll} & [\,\textbf{reads}\ d] \\ \vdash_u^{\Gamma} & \{d \neq null\}\ \texttt{d.mark := 1;}\{d.mark \mapsto 1\} \\ & [\,\textbf{modifies region}\{d.mark\}] \end{array} \tag{9.5}$$

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:

$$d \neq null \vdash^{\Gamma}_{u} \begin{array}{l} (\mathbf{reads}\ d, \mathbf{region}\{d.l\}, \mathbf{region}\{d.r\}, \mathbf{fpt}(dag(d.l) \wedge dag(d.r))) \\ \mathit{frm}\ (d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))) \end{array} \tag{9.6}$$

Thus, the read effects are separate from the write effects, $\mathbf{region}\{d.mark\}$. Using the rule I_{sc}, the following is derived:

$$\vdash^{\Gamma}_{u} \begin{array}{l} [\mathbf{reads}\ d] \\ \{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\ \texttt{d.mark := 1;} \\ \{d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\ [\mathbf{modifies}\ \mathbf{region}\{d.mark\}] \end{array} \tag{9.7}$$

which implies the precondition of `mark(d.l)`. Thus the following must be true (noting that preconditions, or postconditions, written on different lines of a method specification are conjoined):

$$\vdash^{\Gamma}_{u} \begin{array}{l} [\mathbf{reads}\ \mathbf{alloc}\downarrow] \\ \left\{ \begin{array}{l} dag(d.l) \wedge (d \neq null \wedge d.mark \mapsto 1 \Rightarrow \\ \qquad \forall n : Node.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d)) \Rightarrow n.mark \mapsto 1)) \end{array} \right\} \\ \texttt{mark(d.l);} \\ \{d.l \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)\} \\ [\mathbf{modifies}\ unmarked(d.l)] \end{array} \tag{9.8}$$

The rule SubEff_u is used on Eq. (9.7) and Eq. (9.8) to match up the effects for the rule $\mathit{SEQ1}_u$. Then the rule CONSEQ_u is used on Eq. (9.8) to match up the postcondition of `d.mark := 1` and the precondition of `mark(d.l)` and to get rid of the implication in the precondition.

Thus, the following is derived:

$$\begin{array}{l} [\textbf{reads alloc}\downarrow, d] \\ \{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r)\} \\ \vdash_u^\Gamma \ \texttt{d.mark := 1;} \\ \{d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\ [\textbf{modifies region}\{d.mark\}] \end{array} \tag{9.9}$$

and

$$\begin{array}{l} [\textbf{reads alloc}\downarrow, d] \\ \{dag(d.l)\} \\ \vdash_u^\Gamma \ \texttt{mark(d.l);} \\ \{d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)\} \\ [\textbf{modifies } unmarked(d.l)] \end{array} \tag{9.10}$$

By using the rule I_{sc}, FRM_u and $CONSEQ_u$ for Eq. (9.10), the following is derived:

$$\begin{array}{l} [\textbf{reads alloc}\downarrow, d] \\ \{d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\ \vdash_u^\Gamma \ \texttt{mark(d.l);} \\ \left\{ \begin{array}{l} (d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow \\ n.mark \mapsto 1)) \wedge (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))) \end{array} \right\} \\ [\textbf{modifies } unmarked(d.l)] \end{array} \tag{9.11}$$

In order to use the rule $SEQ1_u$, the following side condition has to be true:

$$unmarked(d.l) \text{ is } (d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))/ \\ \textbf{region}\{d.mark\}\texttt{-immune} \tag{9.12}$$

By the definition of immune (Def. 6), the proof obligation is to show that *for all* **modifies** *RE* in `unmarked(d.l)`:

$(d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))$ *implies efs*$(RE) \cdot / \cdot$ **region**$\{d.mark\}$.

Here shows the above is true by contradiction. Suppose that there is some *RE*, such that *efs*(RE) contains the location **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 **int**, not a reference, this is impossible.

Now the two statements' write effects can be accumulated. And the following is derived:

$$
\begin{array}{l}
[\,\textbf{reads alloc}\downarrow, d] \\
\{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\
\vdash_u^\Gamma \texttt{d.mark := 1;}\quad \texttt{mark(d.l);} \\
\left\{ \begin{array}{l} (d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow \\ n.mark \mapsto 1)) \wedge (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))) \end{array} \right\} \\
[\,\textbf{modifies region}\{d.mark\}, unmarked(d.l)]
\end{array}
\tag{9.13}
$$

The postcondition of the above implies the precondition of the method `mark(d.r)`. Using the rule $CONSEQ_u$ (getting rid of the implication in the precondition), the following must be true:

$$
\begin{array}{l}
[\textbf{reads alloc}\downarrow] \\
\{dag(d.r)\} \\
\vdash_u^\Gamma \texttt{mark(d.r);} \\
\{d.r \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.r)) \Rightarrow n.mark \mapsto 1)\} \\
[\,\textbf{modifies}\ unmarked(d.r)]
\end{array}
\tag{9.14}
$$

As the function `unmarked` only collects unmarked locations, the following must be true:

$$(d.l \neq null \Rightarrow (\forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)))$$
$$\Rightarrow \mathbf{reads}\ \mathbf{fpt}(dag(d.l)) \not/ \mathbf{modifies}\ unmarked(d.r) \tag{9.15}$$

By using the rules I_{sc}, FRM_u, $CONSEQ_u$ and $SubEff_u$, the following is derived.

$$\begin{array}{l}
[\,\mathbf{reads}\ \mathbf{alloc}{\downarrow}, d] \\
\left\{ \begin{array}{l} (d.l \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.l)) \Rightarrow \\ n.mark \mapsto 1)) \wedge\ (d.mark \mapsto 1 * d.l \mapsto\ i\ *\ d.r \mapsto j * (dag(d.l) \wedge dag(d.r))) \end{array} \right\} \\
\texttt{mark(d.r);} \\
\vdash_u^{\Gamma} \left\{ \begin{array}{l} (d.r \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.r)) \Rightarrow \\ n.mark \mapsto 1)) \wedge \\ (d.l \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.l)) \Rightarrow \\ n.mark \mapsto 1))\ \wedge\ (d.mark \mapsto 1 * d.l \mapsto\ i\ *\ d.r \mapsto j * (dag(d.l) \wedge dag(d.r))) \end{array} \right\} \\
[\,\mathbf{modifies}\ unmarked(d.r)]
\end{array} \tag{9.16}$$

Again, the side condition has to be true in order to use the rule $SEQ1_u$, i.e., $unmarked(d.r)$ is

$$(d\ \neq\ null * d.l \mapsto\ i\ *\ d.r \mapsto j * (dag(d.l) \wedge dag(d.r)))/$$
$$(\mathbf{region}\{d.mark\}, unmarked(d.l))\texttt{-immune} \tag{9.17}$$

By the definition of immune (Def. 6), the proof obligation is to show:

- *for all* **modifies** *RE* $\in$ $\mathbf{region}\{d.mark\}$:

$(d\ \neq\ null * d.l \mapsto\ i\ *\ d.r \mapsto j * (dag(d.l) \wedge dag(d.r)))$ ***implies efs***$(RE) \not/ unmarked(d.r)$.

In this case, *RE* is just $\mathbf{region}\{d.mark\}$, by the assumption, which is disjoint with $\mathbf{fpt}(dag(d.l) \wedge dag(d.r))$ that is $unmarked(d.r)$'s superset.

• *for all* **modifies** *RE* $\in unmarked(d.l)$:

$(d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r)))$ ***implies*** ***efs*****(*****RE*****)** $\not\cdot unmarked(d.r)$.

It is proved by contradiction. Suppose under the assumption, there is some *RE* that has the form $\mathbf{region}\{d.f_1.....f_n.mark\}$, then *efs*(*RE*) is $\mathbf{region}\{d.f_1.....f_n\}$, where $f_i \in \{l, r\}$, and $1 \leqslant i \leqslant n$. Moreover $d.f_1.....f_n$ has the type `Dag`. However, all the regions in $unmarked(d.r)$ has the form $\mathbf{region}\{d.f_1.....f_m.mark\}$, and $d.f_1.....f_m.mark$ has the type **int**, where $f_j \in \{l, r\}$, and $1 \leqslant j \leqslant m$. Thus, there is no overlapping between the two sets of locations.

Now by using the rule $SEQ1_u$, the following is derived:

$$
\begin{array}{l}
[\mathbf{reads\ alloc}\downarrow, d] \\
\{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r))\} \\
\texttt{d.mark := 1; mark(d.l); mark(d.r);} \\
\vdash_u^{\Gamma} \left\{ \begin{array}{l}
(d.r \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.r)) \Rightarrow \\
n.mark \mapsto 1)) \wedge \\
(d.l \neq null \Rightarrow \forall n : Dag.(\mathbf{region}\{n.mark\} \leqslant \mathbf{fpt}(dag(d.l)) \Rightarrow \\
n.mark \mapsto 1)) \ \wedge \ (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \wedge dag(d.r)))
\end{array} \right\} \\
[\mathbf{modifies\ region}\{d.mark\}, unmarked(d.l), unmarked(d.r)]
\end{array}
\tag{9.18}
$$

The postcondition above can imply the one for `mark`, thus the program is verified.

Remark: in this example, the write effects of $mark$ are not necessarily precise. Let ε_l and ε_r be the write effects of $mark(d.l)$ and $mark(d.r)$ respectively. Suppose the location $\mathbf{region}\{x.mark\}$ is contained in both write effects. To use the sequence rule, the proof obligation is to show that ε_r is $dag(d)/\varepsilon_l$-immune. By the definition of immune (Def. 6), the proof obligation is to show that for all **modifies** *RE* $\in \varepsilon_r$ :: *RE* is $dag(d)/\varepsilon_l$-immune. In this case, the proof obligation is to show that $dag(d)$ *implies* *efs*$(\mathbf{region}\{x.mark\}) \not\cdot \varepsilon_1$, by Def. 6. By the definition of read effects, *efs*$(\mathbf{region}\{x.mark\}) = \mathbf{reads}\ x$. There are two cases.

1. $x = d$. In this case, the proof obligation is to show that $dag(d)$ implies **reads** d $\not\cdot$ **region**$\{d.mark\}$, 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 $dag(d)$ implies (**reads region**$\{d.f_1.\ldots.f_n\}$, *efs*$(d.f_1.\ldots.f_{n-1}))$ $\not\cdot$ **region**$\{d.f_1.\ldots.f_n.mark\}$, which is true because, the field f_n has type `Dag`, the field $mark$ has the type **bool**, thus **region**$\{d.f_1.\ldots.f_n\}$ `!!` **region**$\{d.f_1.\ldots.f_n.mark\}$. Similarly, the regions contained in the read effect *efs*$(d.f_1.\ldots.f_{n-1})$ are all disjoint with the region **region**$\{d.f_1.\ldots.f_n.mark\}$.

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.

```
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>`

Method	**Precondition**	**Postcondition**	**Write effects**
Node<T>(v)	true	lst(**this**, [v])	**region**{**this**.∗}
get()	true	**ret** = **this**.val	∅
append(n)	lst(n, [?v]) ∗ lst(**this**, ?vlst)	lst(**this**, vlst + [v])	**region**{last().next}

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:

$$\forall i.0 <= i \;\&\&\; i < |lseq| \Rightarrow lseq[i].vOrder(oseq[3 * i..3 * i + 2]),$$

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 $0 <= i <= j <= |oseq|$. The sequence `oseq` is the

```
predicate dic(p : Pair<Key, Value>, m : map<Key, Value>)
  reads fpt(dic(p, m));
  decreases |m|;
{
  (p = null ⇒ |m| = 0) ∧
   p ≠ null ⇒ p.key ∈ m * p.val ↦ m[p.key] *
               dic(p.next,(map i | i ∈ m ∧ i ≠ p.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 vDic(m: map<Key, Value>)
    reads vDic(m);
  {
    dic(head, m)
  }

  method lookup(k: Key) : Value
    requires vDic(?m) ∧ k ∈ m;
    ensures vDic(m) ∧ 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 ∅;
    decreases |se|;
  {
     if se = [] then ret := [];
     else ret := se[2 := 1] + serverd_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

Table 9.2: The predicates that are used by clients

Class Name	Predicate Name
List<T>	vList(se : **seq**<T>)
ListIterator<T>	vLIter(vlst :**seq**<T>)
Dictionary<Key, Value>	vDic(m : **map**<Key, Value>)

explored. Model variables are used to define abstract values. For example, the specifications of `ListIterator<T>` can be revised by declaring

```
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:

```
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:

$$\begin{array}{l}[\textbf{reads fpt}(vDic(m) \ \wedge \ k \in m)] \\ \{vDic(?m) \ \wedge \ k \in m\}\,\texttt{v := lookup(k:Key);}\{vDic(m) \ \wedge \ v = m[k]\} \\ [\,\textbf{modifies fpt}(vDic(m) \ \wedge \ k \in m)]\end{array} \tag{9.19}$$

By using the rule *SubEff*$_u$, the following is derived:

$$\begin{array}{rl} & [\textbf{reads alloc}\downarrow] \\ & \{vDic(?m) \ \wedge \ k \in m\} \\ \vdash_u^\Gamma & lookup(k : Key)\ \textbf{returns}\ (v : Value) \\ & \{vDic(m) \ \wedge \ v = m[k]\} \\ & [\,\textbf{modifies fpt}(vDic(m) \ \wedge \ k \in m)]\end{array} \tag{9.20}$$

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:

$$\begin{array}{rl} & [\textbf{reads alloc}\downarrow] \\ & \{cshop(?lseq, ?oseq, ?m) \ \&\& \ item \in m\} \\ \vdash_u^\Gamma & takeOrder(item : \textbf{int}, table : \textbf{int})\textbf{returns}(ret : Order) \\ & \{cshop(lseq + [ret], oseq + [table, item, m.[item]], m)\} \\ & [\,\textbf{modifies fpt}(cshop(lseq, oseq, m) \ \&\& \ item \in m)]\end{array} \tag{9.21}$$

Proceeding to the verification of the body of `takeOrder`, its precondition is assumed:

$$cshop(lseq, oseq, m) \ \&\& \ item \in m, \tag{9.22}$$

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 $\textbf{fpt}(vDic(m)) \leqslant \textbf{fpt}(cshop(lseq, oseq, m) \ \&\& \ item \in m)$.

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: $menu.vDis(m) \wedge itemId = m[item]$.

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 $(menu.vDis(m) \wedge itemId = m[item]) * \mathbf{ret}.vOrder([table, itemId, 0]) * orders.vList(lseq)$, which implies the precondition of the method `orders.add(`**`ret`**`)`. For the write effects, according to Section 9.2, its specification is encoded as:

$$[\mathbf{reads\ alloc}\downarrow]\{vList(?vlst)\}\ \texttt{add(t);}\ \{vList(vlst + [t])\}[\mathbf{modifies\ fpt}(vList(vlst))]. \tag{9.23}$$

Together with the definition of the predicate `cshop` in Fig. 9.10, the following must be true:

$$\mathbf{fpt}(orders.vlst(lseq)) \leqslant \mathbf{fpt}(cshop(lseq, oseq, m)\ \&\&\ item \in m), \tag{9.24}$$

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 $(menu.vDis(m) \wedge itemId = m[item]) * \mathbf{ret}.vOrder([table, itemId, 0]) * orders.vList(lseq + [\mathbf{ret}])$, 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:

$$\mathbf{reads\ fpt}(shop1.cshop(?l, ?o, m))\cdot/\cdot$$
$$\mathbf{modifies\ filter}(\mathbf{fpt}(shop.cshop(lseq, oseq, m)), Order, served). \tag{9.25}$$

```
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:

$\{$**this**.$inv()\}$

`DCell.set(v :` **int**`)`

$\{$**this**.`val` $= v$ && **this**.`inv()` && **this**.`val` $= 2 * v\}$

[**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`.

$\{$**super**`.inv()` $\|$ $(\exists\, d.$**this**`.val` $= d)$ && **this**`.inv()` $\}$

`ReCell.set(v :int)`

$\{($**old**$($**super**`.inv()` $) \Rightarrow$ **this**`.val` $= v$ && **super**`.inv()` $)$ &&

old$((\exists d.$**this**`.val` $= d)$ && **this**`.inv()` $) \Rightarrow$ **this**`.val` $= v$ && **this**`.bak` $= d\}$

[**modifies this**`.wf_set()`]

By the program semantics, the assertion $\exists\, d.$**this**`.val` $= d$ is implicit. Thus, by the rule *CONSEQ*$_r$, the above specification is derived to

$\{\exists\, d.$**this**`.val` $= d$ && **this**`.inv()` $\}$

`ReCell.set(v :int)`

$\{$**this**`.val` $= v$ && **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 `TCell.set` is shown below:

$$\{\textbf{super}.\texttt{inv}() \mid\mid \textbf{this}.\texttt{inv}()\}$$
$$\texttt{TCell.set(v:}\textbf{int}\texttt{)}$$
$$\{(\textbf{old}(\textbf{super}.inv()) \Rightarrow \textbf{this}.\texttt{val} = v \;\&\&\; \textbf{super}.\texttt{inv}()) \;\&\&$$
$$(\textbf{old}(\textbf{this}.\texttt{inv}()) \Rightarrow \textbf{this}.val = v \;\&\&\; \textbf{this}.\texttt{inv}())\}$$
$$[\,\textbf{modifies this}.\texttt{wf\_set}()\,]$$

By the rule $CONSEQ_r$, the above specification is derived to

$$\{\textbf{this}.\texttt{inv}()\}$$
$$\texttt{TCell.set(v:}\textbf{int}\texttt{)}$$
$$\{(\textbf{old}(\textbf{this}.\texttt{inv}()) \Rightarrow \textbf{this}.val = v \;\&\&\; \textbf{this}.\texttt{inv}())\}$$
$$[\,\textbf{modifies this}.\texttt{wf\_set}()\,]$$

It can be proved that the implementation of the method `TCell.set` satisfies the above specification.

Fig. 9.15 declares the class `OCell` that inherits from the class `Cell`, and declares two additional fields, `c` and `o`. Together with the inherited field `val`, they compose `OCell`'s data representation. The field c is created by `OCell`'s constructor and is encapsulated. But the field o is initialized by the constructor's parameter. That causes argument exposure. Therefore, the last line of the constructor updates the field `exposed` by `o.df`.

The method `OCell.set` updates both **`this`**`.val` and **`this`**`.c.val` with the new value. Its write effects is specified by the function `wf_set`. Its definition returns regions that come from two parts: one part is from its supertype, i.e., **`super`**`.wf_set();` the other is from the extended state, i.e., `c.df`. Thus, the definition of the predicate `inv` describes the disjointness, i.e., **`this`**`.exposed!!c.df`, which means that `c.df` is encapsulated by `OCell`. Therefore, the specification of `OCell.set` refines the specification of `Cell.set`.

```
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.

$$\Gamma \vdash n : \texttt{int} \qquad \frac{}{\Gamma \vdash x : T} \ \textbf{where}\ \Gamma(x) = T \qquad \frac{}{\Gamma \vdash null : T} \ \textbf{where}\ \textit{isRef}(T)$$

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2 \qquad \Gamma \vdash \oplus : T_1 \to T_2 \to T}{\Gamma \vdash E_1 \oplus E_2 : T} \qquad \Gamma \vdash \texttt{region}\{\} : \texttt{region}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \texttt{region}\{x.f\} : \texttt{region}} \ \textbf{where}\ (f : T') \in \textit{fields}(T) \text{ and } \textit{isRef}(T)$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \texttt{region}\{x.*\} : \texttt{region}} \ \textbf{where}\ \textit{isRef}(T) \qquad \frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash RE_1 : \texttt{region} \qquad \Gamma \vdash RE_2 : \texttt{region}}{\Gamma \vdash E \ ? \ RE_1 \ : \ RE_2 : \texttt{region}}$$

$$\frac{\Gamma \vdash RE : \texttt{region}}{\Gamma \vdash \texttt{filter}\{RE, T, f\} : \texttt{region}} \ \textbf{where}\ \textit{isRef}(T) \qquad \frac{\Gamma \vdash RE : \texttt{region}}{\Gamma \vdash \texttt{filter}\{RE, T\} : \texttt{region}} \ \textbf{where}\ \textit{isRef}(T)$$

$$\frac{\Gamma \vdash RE_1 : \texttt{region} \qquad \Gamma \vdash RE_2 : \texttt{region}}{\Gamma \vdash RE_1 \otimes RE_2 : \texttt{region}}$$

Figure A.1: The typing rules for pure expressions and region expressions

$$\Gamma \vdash \texttt{skip}; : ok(\Gamma)$$

$$\Gamma \vdash \texttt{var}\ x : T; : ok(\Gamma, x : T) \quad \textbf{where } x \notin dom(\Gamma)$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash G : T}{\Gamma \vdash x := G; : ok(\Gamma)} \quad \textbf{where } x \neq \texttt{this}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash y : T'}{\Gamma \vdash x := y.f; : ok(\Gamma)} \quad \textbf{where } x \neq \texttt{this}, \mathit{isRef}(T') \text{ and } (f : T) \in \mathit{fields}(T')$$

$$\frac{\Gamma \vdash x : T' \qquad \Gamma \vdash G : T}{\Gamma \vdash x.f := G; : ok(\Gamma)} \quad \textbf{where } \mathit{isRef}(T') \text{ and } (f : T) \in \mathit{fields}(T')$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x := \texttt{new}\ T; : ok(\Gamma)} \quad \textbf{where } x \neq \texttt{this} \text{ and } \mathit{isRef}(T)$$

$$\frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash S_1 : ok(\Gamma_1) \qquad \Gamma \vdash S_2 : ok(\Gamma_2)}{\Gamma \vdash \texttt{if}\ E\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\} : ok(\Gamma)} \qquad \frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash S : ok(\Gamma')}{\Gamma \vdash \texttt{while}\ E\ \{S\} : ok(\Gamma)}$$

$$\frac{\Gamma \vdash S_1 : ok(\Gamma'') \qquad \Gamma'' \vdash S_2 : ok(\Gamma')}{\Gamma \vdash S_1 S_2 : ok(\Gamma')}$$

Figure A.2: The typing rules for statements

$$\frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 = E_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \neq E_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash x : T' \qquad \Gamma \vdash E : T}{\Gamma \vdash x.f = E : \mathbf{bool}}$$
where *isRef*(T') and $(f : T) \in$ *fields*(T')

$$\frac{\Gamma \vdash RE_1 : \mathbf{region} \qquad \Gamma \vdash RE_2 : \mathbf{region}}{\Gamma \vdash RE_1 \leqslant RE_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash RE_1 : \mathbf{region} \qquad \Gamma \vdash RE_2 : \mathbf{region}}{\Gamma \vdash RE_1 \,!!\, RE_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash P_1 : \mathbf{bool} \qquad \Gamma \vdash P_2 : \mathbf{bool}}{\Gamma \vdash P_1 \,\&\&\, P_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash P_1 : \mathbf{bool} \qquad \Gamma \vdash P_2 : \mathbf{bool}}{\Gamma \vdash P_1 \,||\, P_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash P : \mathbf{bool}}{\Gamma \vdash \neg P : \mathbf{bool}} \qquad \frac{\Gamma, x : \mathbf{int} \vdash P : \mathbf{bool}}{\Gamma \vdash \forall x : \mathbf{int} :: P : \mathbf{bool}}$$

$$\frac{\Gamma \vdash RE : \mathbf{region} \qquad \Gamma, x : T \vdash P : \mathbf{bool}}{\Gamma \vdash \forall x : T : \mathbf{region}\{x.f\} \leqslant RE : P : \mathbf{bool}}$$
where *isRef*(T) and $(f : T') \in$ *fields*(T)

$$\frac{\Gamma, x : \mathbf{int} \vdash P : \mathbf{bool}}{\Gamma \vdash \exists x : \mathbf{int} :: P : \mathbf{bool}}$$

$$\frac{\Gamma \vdash RE : \mathbf{region} \qquad \Gamma, x : T \vdash P : \mathbf{bool}}{\Gamma \vdash \exists x : T : \mathbf{region}\{x.f\} \leqslant RE : P : \mathbf{bool}}$$
where *isRef*(T) and $(f : T') \in$ *fields*(T)

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 (σ, h) be an arbitrary state, and without loss of generality, let $(\sigma', h') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, h)$. Assume $\vdash_r^\Gamma \{P\}\ S\ \{Q\}[\varepsilon]$, and $\sigma, h \models^\Gamma P$. Then the proof obligation is to prove $\sigma', h' \models^{\Gamma'} Q$, and that all the changed locations are in ε. There are 6 base cases.

1. (*SKIP*$_r$) In this case, *S* is **skip**;, P is $true$, Q is $true$, ε is $\varnothing$. By the program semantics Fig. 4.1, $\sigma' = \sigma$, $h' = h$ and $\Gamma' = \Gamma$ Thus, $\sigma', h' \models^{\Gamma'} true$. For the frame condition, *S* does not change anything, thus, it is $\varnothing$.

2. (*VAR*$_r$) In this case, *S* is **var**$x : T$;, P is $true$, Q is $x = \mathit{default}(T)$ and ε is $\varnothing$. By the program semantics Fig. 4.1, $\Gamma' = \Gamma, (x : T)$, $\sigma' = \mathit{Extend}(\sigma, x, \mathit{default}(T))$ and $h' = h$. Thus (σ', h') entails Q. For the frame condition, as the statement does not change anything existing in the prestate, thus, it is $\varnothing$.

3. (*ALLOC*$_r$) In this case, *S* is $x :=$ **new** T;, P is $true$, Q is $\overline{x.f = \mathit{default}(T)}$ and $\varepsilon =$ **modifies** x, **alloc**, **fresh**(**region**$\{x.*\}$). By the program semantics Fig. 4.1, $\Gamma' = \Gamma$, $\sigma' = \sigma(x \mapsto l)$ and $h' = h''[\overline{(l, f) \mapsto \mathit{default}(T)}]$, where $(l, h'') = allocate(T, h)$. Thus, (σ', h') entails Q.

   For the frame condition, *S* only updates the variable x and **alloc**. By the semantics, the function $allocate$ returns a new heap. So **fresh**(**region**$\{x.*\}$) is the fresh effect.

4. (*ASSGN*$_r$) In this case, *S* is $x := G$;, P is $x = x'$, Q is $\{x = G/(x \mapsto x')\}$ and $\varepsilon =$ **modifies** x, where $x \notin \mathrm{FV}(G)$. By the program semantics Fig. 4.1, $\Gamma' = \Gamma$, $(\sigma', h') =$

$(\sigma[x \mapsto \mathcal{E}[\![\Gamma \vdash G : T]\!](\sigma)], h)$, which entails Q.

For the frame condition, this statement only updates variable x. Therefore, ε is **modifies** x is correct.

5. (UPD_r) In this case, S is $x.f := G$;, P is $x \neq null$, Q is $x.f = G$ and ε is **modifies region**$\{x.f\}$. By the program semantics Fig. 4.1, $\Gamma' = \Gamma$, $(\sigma', h') = (\sigma, h[(\mathcal{E}[\![\Gamma \vdash x : T]\!](\sigma), f) \mapsto \mathcal{E}[\![\Gamma \vdash G : T']\!](\sigma)])$, which entails Q.

   For the frame condition, this statement changes the singleton heap location $(\sigma(x), f)$. Therefore, ε is **modifies region**$\{x.f\}$ is correct.

6. (ACC_r) In this case, S is $x := x'.f$;, P is $x' \neq null$, Q is $x = x_1.f$, and ε is **modifies** x, where $x \neq x'$. By the program semantics Fig. 4.1, $\Gamma' = \Gamma$, $(\sigma', h') = (\sigma[x \mapsto h[(\mathcal{E}[\![\Gamma \vdash x' : T]\!](\sigma), f)]], h)$, which entails Q.

   For the frame condition, this statement only updates variable x. Therefore, $\varepsilon =$ **modifies** x is correct.

The inductive hypothesis is that for all substatements S_i, if $\vdash_r^{\Gamma_i} \{P_i\}\, S_i\, \{Q_i\}[\varepsilon_i]$, and $\sigma_i, h_i \vDash^{\Gamma_i} P_i$, then $\sigma_i', h_i' \vDash^{\Gamma_i'} Q_i$.

1. (IF_r) In this case, S is **if** E **then** $\{S_1\}$ **else**$\{S_2\}$. There are two cases:

   - E. By the inductive hypothesis, it must be true that $\sigma, h \vDash^{\Gamma} P \,\&\&\, E$, $(\sigma', h') = \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma_1)]\!](\sigma, h)$, which entails Q. And the frame condition is correct.
   - $\neg E$. By the inductive hypothesis, it must be true that $\sigma, h \vDash^{\Gamma} P \,\&\&\, \neg E$, $(\sigma'', h'') = \mathcal{MS}[\![\Gamma \vdash S_2 : ok(\Gamma_2)]\!](\sigma, h)$, 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*$_r$) In this case, S is **`while`** E **`do`** $\{S\}$. $P = I$, $Q = I \,\&\&\, E$ and the frame conditions is ε. The premise is $\vdash_r^\Gamma \{I \,\&\&\, E\}\ \{S\}\ \{I\}[\varepsilon]$.

   By the program semantics Fig. 4.1, let g be a recursive point function, such that

   $$g = \underline{\lambda} s.\textbf{if } \mathcal{E}[\![\Gamma \vdash E : \texttt{bool}]\!](\sigma) \neq 0 \textbf{then let} s' = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) \textbf{in} g \circ s' \textbf{else } s$$

   By definition, *fix* is a fixed point function, so $\mathit{fix}(g) = g$. Then the following proves $\mathit{fix}(g)(\sigma, h) \models^{\Gamma'} I$ by fixed-point induction.

   Base Case: $\perp \models^{\Gamma'} I$ holds vacuously. It requires to prove all members in $\perp$ implies I, but there is nothing in $\perp$. Hence it is vacuously true.

   Inductive Case: Let $(\sigma'', h'') \models^{\Gamma'} I$ hold for an arbitrary iteration of g, and ε is the frame condition. Then the proof obligation is to show that $\mathit{fix}(g)(\sigma'', h'') \models^{\Gamma'} I$ holds, and the changed locations on the heap is ε.

   There are two cases:

   - E. By the semantics, $\mathit{fix}(g)(\sigma'', h'') = g(\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma'', h''))$. By the inductive hypothesis, $g(\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma'', h'')) \models^{\Gamma'} I$ holds. Hence $\mathit{fix}(g)(\sigma'', h'') \models^{\Gamma'} I$ 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.
   - $\neg E$. By the semantics, $\mathit{fix}(g)(\sigma'', h'') = (\sigma'', h'')$. Therefore, by the inductive hypothesis, $\mathit{fix}(g)(\sigma'', h'') \models^{\Gamma'} I$ holds. For the frame condition, since the state does not change, the frame is **`region`**$\{\}$, which is the subset of ε.

   Now it has been shown that if the loop exits, which means that $\neg E$ 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 $S_1 \neq$ **var** $x : T$;. Let (σ, h) be a state, such that $(\sigma, h) \models^{\Gamma} P$. By the inductive hypothesis for S_1 and S_2, $(\sigma'', h'') = \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma'')]\!](\sigma, h)$, and $(\sigma'', h'') \models^{\Gamma''} P_1$. By the second premise and the semantics, $(\sigma', h') = \mathcal{MS}[\![\Gamma'' \vdash S_2 : ok(\Gamma')]\!](\sigma'', h'')$. Hence $(\sigma', h') \models^{\Gamma'} P'$.

   For the frame condition, the proof obligation is to show $(\sigma, h) \rightarrow (\sigma', h') \models^{\Gamma} (\varepsilon_1, \varepsilon_2, \textbf{fresh}(RE))$, which is proved by Lemma 7. It is instantiated with $\Gamma_0 := \Gamma, \Gamma_1 := \Gamma''$, $\Gamma_2 := \Gamma'$, $RE_1 := RE_1$, $(\sigma_0, h_0) := (\sigma, h)$, $(\sigma_1, h_1) := (\sigma'', h'')$, $(\sigma_2, h_2) := (\sigma', h')$ and $\varepsilon_1 := (\varepsilon_1, \textbf{fresh}(RE))$. The following conditions, which are required by the Lemma, are satisifed:

   - $(\sigma, h) \models^{\Gamma} P$ and $(\sigma'', h'') \models^{\Gamma''} P_1$ from the above;
   - $(\sigma, h) \rightarrow (\sigma'', h'') \models^{\Gamma} (\varepsilon_1, \textbf{fresh}(RE))$ by the inductive hypothesis;
   - $(\sigma'', h'') \rightarrow (\sigma', h') \models^{\Gamma''} (\varepsilon_2, \textbf{modifies}\ RE_1)$ by the inductive hypothesis.
   - ε_2 is P/ε_1-immune by the given side condition;
   - for all **fresh**(*RE*) $\in \varepsilon_1$ :: *RE* is $P/(\varepsilon_2, \textbf{modifies}\ RE_1)$-immune by the given side condition.
   - $\mathcal{E}[\![\Gamma'' \vdash RE_1 : \textbf{region}]\!](\sigma'') \cap \sigma(\textbf{alloc}) = \varnothing$, *RE* are freshly allocated regions by S_1, i.e., $\mathcal{E}[\![\Gamma'' \vdash RE_1 : \textbf{region}]\!](\sigma'') \subseteq (\sigma''(\textbf{alloc}) - \sigma(\textbf{alloc}))$.

4. (*SEQ2*$_r$) In this case, S is **var** $x : T; S_2$. This case follows the inductive hypothesis and the program semantics.

5. (*SUBEFF*$_r$) By the inductive hypothesis, $\models_r^{\Gamma} \{P\}S\{Q\}[\varepsilon]$. Hence when applying the frame condition $\varepsilon' \geqslant \varepsilon$, the locations that may be changed are also contained in ε'. Therefore ε' is a correct frame.

6. (*FRM*$_r$) In this case, by the inductive hypothesis, it must be true that $\models_r^{\Gamma} \{P\}S\{Q\}[\varepsilon]$. And

by the assumption, it must be true that $P \models^{\Gamma} \delta \mathit{frm}\, Q$ and $P \,\&\&\, R \Rightarrow \delta \cdot\!/\!\cdot \varepsilon$. The proof obligation is to show $\models_r^{\Gamma} \{P \,\&\&\, R\} S \{Q \,\&\&\, R\}[\varepsilon]$. Because $P \,\&\&\, R$ implies P, Thus, it must be true that $\models_r^{\Gamma} \{P \,\&\&\, R\} S \{Q\}[\varepsilon]$. Let $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$. The proof obligation is to show that $(\sigma', H') \models^{\Gamma'} R$. By $(\sigma, H) \models^{\Gamma} P \,\&\&\, R$ and the side condition $P \,\&\&\, R \Rightarrow \delta \cdot\!/\!\cdot \varepsilon$, it must be true that $(\sigma, H) \models^{\Gamma} \delta \cdot\!/\!\cdot \varepsilon$. As the write effect is $(\sigma, H) \rightarrow (\sigma', H') \models^{\Gamma} \varepsilon$, it must be true that $(\sigma, H) \overset{\delta}{\equiv} (\sigma', H')$. By the definition of framing (Def. 5) and $(\sigma, H) \models^{\Gamma} P \,\&\&\, R$, it must be true that $(\sigma', H') \models^{\Gamma'} R$.

7. (*CONSEQ*$_r$) In this case, by the inductive hypothesis, it must be true that $\models_r^{\Gamma} \{P'\} S \{Q'\}[\varepsilon]$. By the premise, $P \Rightarrow P'$ and $Q' \Rightarrow Q$. Hence $\models_r^{\Gamma} \{P\} S \{Q\}[\varepsilon]$ is valid.

□

APPENDIX C: PROOF OF THEOREM 3

Theorem 3: Let Γ be a well-formed type environment. Let S be a statement. Let P_1 and P_2 be assertions. Let ε be effects. Then

$$\vdash_r^\Gamma \{P_1\}S\{P_2\}[\varepsilon] \ \textit{iff} \ \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_1\}S\{P_2\}[\varepsilon]$$
$$\textbf{where } P_1 \Rightarrow r = \mathbf{alloc} \text{ and } \mathbf{modifies}\ r \notin \varepsilon$$

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 *SKIP*$_r$, which is $\vdash_r^\Gamma \{true\}\mathbf{skip};\{true\}[\varnothing]$. Then, the proof obligation is to show that the judgment $\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{true\}\mathbf{skip};\{true\}[\varnothing]$, where $true \Rightarrow r = \mathbf{alloc}$, is derivable in UFRL. It can be done by using the axiom *SKIP*$_u$ and the structural rule *SubEff*$_u$.

2. *VAR:* In this case, suppose that the FRL proof consists of the axiom *VAR*$_r$, which is $\vdash_r^\Gamma \{true\}\mathbf{var}\ x : T;\{x = \textit{default}(T)\}[\varnothing]$. Then, the proof obligation is to show that the judgment $\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{true\}\mathbf{var}\ x : T;\{x = \textit{default}(T)\}[\varnothing]$, where $true \Rightarrow r = \mathbf{alloc}$, is derived in UFRL. It can be done by using the axiom *VAR*$_u$ and the structural rule *SubEff*$_u$.

3. *ALLOC:* In this case, suppose that the FRL proof consists of the axiom *ALLOC*$_r$, which is $\vdash_r^\Gamma \{true\}\ x := \mathbf{new}\ T;\ \{new(T,x)\}[\mathbf{modifies}\ x, \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})]$. Then, the proof obligation is to show the following judgment is derivable in UFRL.

$$\vdash_u^\Gamma \begin{array}{l}[\mathbf{reads}\ r\downarrow] \\ \{true\}\ x := \mathbf{new}\ T;\ \{new(T,x)\ [\ \mathbf{modifies}\ x, \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})]\end{array}$$
$$\textbf{where } true \Rightarrow r = \mathbf{alloc}$$

   It can be done by using the axiom *ALLOC*$_u$ and the structural rule *SubEff*$_u$.

4. *UPD:* Suppose that the FRL proof consists of the axiom *UPD*$_r$, which is $\vdash_r^\Gamma \{x \neq null\}\ x.f := G;\ \{x.f = G\}[\mathbf{modifies}\ \mathbf{region}\{x.f\}]$, **where** $x \notin \mathrm{FV}(G)$. Then, the

proof obligation is to show that the following judgment is derivable.

$$\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{x \neq null\}\ x.f := G;\ \{x.f = G\}[\mathbf{modifies}\ \mathbf{region}\{x.f\}]$$
$$\mathbf{where}\ x \notin \mathtt{FV}(G) \text{ and } x \neq null \Rightarrow r = \mathbf{alloc}$$

It can be done by using the axiom UPD_u and the structural rule $SubEff_u$.

5. *ASGN:* Suppose that the FRL proof consists of the axiom $ASGN_r$: $\vdash_r^\Gamma \{true\}\, x := G;\, \{x = G\}\, [\mathbf{modifies}\ x]$, where $x \notin \mathtt{FV}(G)$. Then, the proof obligation is to show that the following judgment is derivable.

$$\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{true\}\ x := G;\ \{x = G\}\ [\mathbf{modifies}\ x]$$
$$\mathbf{where}\ x \notin \mathtt{FV}(G) \text{ and } true \Rightarrow r = \mathbf{alloc}$$

It can be done by using the axiom $ASGN_u$ and the structural rule $SubEff_u$.

6. *ACC:* Suppose that the FRL proof consists of the axiom ACC_r, which is $\vdash_r^\Gamma \{x' \neq null\}x := x'.f;\ \{x = x'.f\}\ [\mathbf{modifies}\ x]$, where $x \neq x'$. Then, the proof obligation is to show that the following judgment is derivable.

$$\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{x' \neq null\}\ x := x'.f;\ \{x = x'.f\}\ [\mathbf{modifies}\ x]$$
$$\mathbf{where}\ x \neq x' \text{ and } x' \neq null \Rightarrow r = \mathbf{alloc}$$

It can be done by using the axiom ACC_u and the structural rule $SubEff_u$.

The inductive hypothesis is that for all substatements S_i, it is true that $\vdash_r^\Gamma \{P_i\}S_i\{Q_i\}[\varepsilon_i]$ *iff* $\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_i\}S_i\{Q_i\}[\varepsilon_i]$, where $P_i \Rightarrow r = \mathbf{alloc}$ and and $\mathbf{modifies}\ r \notin \varepsilon$.

1. *IF:* In this case, suppose that the FRL proof consists of the rule IF_r, which is

$$\frac{\vdash_r^\Gamma \{P \mathbin{\&\&} E\}\ S_1\ \{P'\}[\varepsilon] \qquad \vdash_r^\Gamma \{P \mathbin{\&\&} \neg E\}\ S_2\ \{P'\}[\varepsilon]}{\vdash_r^\Gamma \{P\}\mathbf{if}\ E\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}\{P'\}[\varepsilon]}$$

Then the proof obligation is to show that the following is derivable.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} E\}\ S_1\ \{P'\}[\varepsilon] \qquad \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} \neg E\}\ S_2\ \{P'\}[\varepsilon]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\mathbf{if}\ E\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}\{P'\}[\varepsilon]}$$

where $P \Rightarrow r = \mathbf{alloc}$

By the inductive hypothesis, the two premises are assumed. Using the rule IF_u, the following is derived

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} E\}\ S_1\{Q\}\ [\varepsilon] \qquad \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} \neg E\}\ S_2\{Q\}\ [\varepsilon]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow, \delta_E]\{P\}\ \mathbf{if}\ E\ \{S_1\}\mathbf{else}\{S_2\}\{Q\}\ [\varepsilon]}$$

where $\delta_E = \mathit{efs}(E), P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{modifies}\ r \notin \varepsilon$

Then, using the rule SubEff_u, the conclusion of is derived.

2. *WHILE:* Suppose that the FRL proof consists of the rule WHILE_r, which is

$$\frac{\vdash_r^\Gamma \{P \mathbin{\&\&} E\}\ S\{P\}\ [\varepsilon, \mathbf{fresh}(RE)]}{\vdash_r^\Gamma \{P \mathbin{\&\&} r = \mathbf{alloc}\}\ \mathbf{while}\ E\ \{S\}\ \{P \mathbin{\&\&} \neg E\}\ [\varepsilon]},$$

where $P \Rightarrow RE\ !!\ r, \varepsilon$ is fresh-free, ε is P/ε-immune and $\mathbf{modifies}\ r \notin \varepsilon$

where r snapshots the domain of the heap in the pre-state. Then the proof obligation is to show the following is derivable.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} E\}\ S\{P\}\ [\varepsilon, \mathbf{fresh}(RE)]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P \mathbin{\&\&} r = \mathbf{alloc}\}\ \mathbf{while}\ E\ \{S\}\ \{P \mathbin{\&\&} \neg E\}\ [\varepsilon]}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{modifies}\ r \notin \varepsilon$

By the inductive hypothesis, the premises is assumed. To use the rule 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 $\mathbf{reads}\ r$ is P/ε-immune, which is true because $\mathbf{modifies} r \notin \varepsilon$.

Then, using the rule $WHILE_u$, the following is derived

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\ \&\&\ E\}\ S\{P\}\ [\varepsilon, \mathbf{fresh}(RE)]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow, \delta_E]\{P\ \&\&\ r = \mathbf{alloc}\}\ \mathbf{while}\ E\ \{S\}\ \{P\ \&\&\ \neg E\}\ [\varepsilon]},$$

where $P \Rightarrow r = \mathbf{alloc}$, $P \Rightarrow RE\ !!\ r$, ε is fresh-free, ε is P/ε-immune and $\mathbf{modifies}\ \mathbf{r} \notin \varepsilon$,

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

$$\frac{\vdash_r^\Gamma \{P\}\ S_1\ \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \qquad \vdash_r^\Gamma \{P_1\}\ S_2\ \{P'\}[\varepsilon_2, RE]}{\vdash_r^\Gamma \{P\}\ S_1S_2\ \{P'\}[\ \varepsilon_1, \varepsilon_2, \mathbf{fresh}(RE)]}$$

where $S_1 \neq \mathbf{var}\ x : T;$, ε_1 is fresh-free, ε_2 is P/ε_1-immune, and RE is $P_1/(\mathbf{modifies}\ \ RE, \varepsilon_2)$-immune

Then the proof obligation is to show that the following is derivable.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S_1\ \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \qquad \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_1\}\ S_2\ \{P'\}[\varepsilon_2, RE]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S_1S_2\ \{P'\}[\ \varepsilon_1, \varepsilon_2, \mathbf{fresh}(RE)]}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{modifies}\ r \notin (\varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the two premises are assumed. To use the rule $SEQ1_u$, check the side condition $\mathbf{reads}\ r$ is P/ε_1-immune, which is true because $\mathbf{modifies} r \notin (\varepsilon_1, \varepsilon_2)$ 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 $SEQ1_r$:

$$\frac{\vdash_r^\Gamma \{P\ \&\&\ x = default(T)\} : S\ \{Q\}[\mathbf{modifies}\ x, \varepsilon]}{\vdash_r^\Gamma \{P\}\ \mathbf{var}\ x : T; S\ \{P'\}[\ \varepsilon]}$$

Then, the proof obligation is to show that the following is derivable.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\ \&\&\ x = \mathit{default}(T)\}\ S\ \{Q\}[\ \mathbf{modifies}\ x, \varepsilon]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ \mathbf{var}\ x : T; S\ \{P'\}[\ \varepsilon]}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $r \neq x$;

By the inductive hypothesis, the assumption is assumed. Then the conclusion is derived by using the rule $\mathit{SEQ2}_u$.

5. *FRM:* Suppose that the FRL proof consists of the rule FRM_r:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \delta\ \mathit{frm}\ Q}{\vdash_r^\Gamma \{P\ \&\&\ Q\}\ S\ \{P'\ \&\&\ Q\}[\varepsilon]}\ \textbf{where}\ P\ \&\&\ Q \Rightarrow \delta \diagup\!\!\diagup \varepsilon$$

Then the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \delta\ \mathit{frm}\ Q}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\ \&\&\ Q\}\ S\ \{P'\ \&\&\ Q\}[\varepsilon]}$$

where $P\ \&\&\ Q \Rightarrow \delta \diagup\!\!\diagup \varepsilon$, $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

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:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \varepsilon \leqslant \varepsilon'}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon']}$$

Then the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon] \qquad P \vdash^\Gamma \varepsilon \leqslant \varepsilon'}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon']}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

By the inductive hypothesis, the two premises are assumed. Then, the conclusion can be derived by using the rule *SubEff*$_u$, because the side condition *regRW*$(\varepsilon, \mathbf{reads}\ r) \leqslant$ *regRW*$(\varepsilon', \mathbf{reads}\ r)$ is true.

7. *CONSEQ:* suppose that the FRL proof consists of the rule *CONSEQ*$_r$:

$$\frac{P_2 \Rightarrow P_1 \qquad \vdash_r^\Gamma \{P_1\}\ S\ \{P_1'\}[\varepsilon] \qquad P_1' \Rightarrow P_2'}{\vdash_r^\Gamma \{P_2\}\ S\{P_2'\}[\varepsilon]}$$

Then, the proof obligation is to show that the following is derivable.

$$\frac{P_2 \Rightarrow P_1 \qquad \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_1\}\ S\ \{P_1'\}[\varepsilon] \qquad P_1' \Rightarrow P_2'}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_2\}\ S\{P_2'\}[\varepsilon]}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

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*$_r$:

$$\frac{\vdash_r^\Gamma \{P\ \&\&\ E\}\ S\ \{P'\}[\varepsilon_1] \qquad \vdash_r^\Gamma \{P\ \&\&\ \neg E\}\ S\ \{P'\}[\varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\ \&\&\ E\}\ S\ \{P'\}[\varepsilon_1] \qquad \vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\ \&\&\ \neg E\}\ S\ \{P'\}[\varepsilon_2]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

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*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_1]}\ \textbf{where}\ P \Rightarrow E$$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_1]}$$

where $P \Rightarrow E, P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *ConMask1*$_u$

10. *ConMask2:* suppose that the FRL proof consists of the rule *ConMask2*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_2]}\ \textbf{where}\ P \Rightarrow \neg E$$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon, \varepsilon_2]}$$

where $P \Rightarrow \neg E, P \Rightarrow r = \mathbf{alloc}$ and $\mathbf{reads}\ r \notin \varepsilon$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *ConMask2*$_u$.

11. *PostToFr:* Suppose that the FRL proof consists of the rule *PostToFr*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}$$

where $P \Rightarrow (E\ \&\&\ RE_1\ !!\ \mathbf{alloc})$ and $P \Rightarrow (\neg E\ \&\&\ RE_2\ !!\ \mathbf{alloc})$

Then, the proof obligation is to show the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}$$

where $P \Rightarrow r = \mathbf{alloc}, \mathbf{reads}\ r \notin \varepsilon, P \Rightarrow (E\ \&\&\ RE_1\ !!\ \mathbf{alloc})$

and $P \Rightarrow (\neg E\ \&\&\ RE_2\ !!\ \mathbf{alloc})$

By the inductive hypothesis, the premise is assumed. Then, the conclusion is derived by using the rule $\mathit{PostToFr}_u$.

12. *FrToPost:* Suppose that the FRL proof consists of the rule $\mathit{FrToPost}_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}{\vdash_r^\Gamma \begin{array}{l}\{P\}\ S\ \{P'\ \&\&\ (b \Rightarrow RE_1\ !!\ r)\ \&\&\ (\neg b \Rightarrow RE_2\ !!\ r)\} \\ [\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]\end{array}}$$

where $P \Rightarrow r = \mathbf{alloc}, P \Rightarrow b = E, \mathbf{modifies}\ b \notin \varepsilon$ and $\mathbf{modifies}\ r \notin \varepsilon$

Then, the proof obligation is to show that the following is derivable

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]}{\vdash_u^\Gamma \begin{array}{l}[\mathbf{reads}\ r\downarrow] \\ \{P\}\ S\ \{P'\ \&\&\ (b \Rightarrow RE_1\ !!\ r)\ \&\&\ (\neg b \Rightarrow RE_2\ !!\ r)\} \\ [\varepsilon,\ E\ ?\ \mathbf{fresh}(RE_1)\ :\ \mathbf{fresh}(RE_2)]\end{array}}$$

where $P \Rightarrow r = \mathbf{alloc}, P \Rightarrow b = E, \mathbf{modifies}\ b \notin \varepsilon$ and $\mathbf{modifies}\ r \notin \varepsilon$

By the inductive hypothesis, the premise is assumed. Then, the conclusion can be derived by using the rule $\mathit{FrToPost}_u$.

13. *VarMask1*: suppose that the FRL proof consists of the rule $\mathit{VarMask1}_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ (\mathbf{modifies}\ x, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow b = E, P \Rightarrow b, P \mid\mid P' \Rightarrow x = y, P\ \&\&\ b \Rightarrow \mathbf{reads}\ y \diagup (x, \varepsilon)$

and $\mathbf{modifies}\ b \notin (\varepsilon_1, \varepsilon_2)$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\ E\ ?\ (\textbf{modifies}\ x, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow r = \textbf{alloc}, P \Rightarrow b = E, P \Rightarrow b, P \mid\mid P' \Rightarrow x = y,$

$P \mathbin{\&\&} b \Rightarrow \textbf{reads}\ y \not\cdot (x, \varepsilon), \textbf{modifies}\ r \notin (\varepsilon_1, \varepsilon_2)$ and $\textbf{modifies}\ b \notin (\varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *VarMask1*$_u$.

14. *VarMask2*: Suppose that the FRL proof consists of the rule *VarMask2*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ (\textbf{modifies}\ x, \varepsilon_2)]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow b = E, P \Rightarrow \neg b, P \mid\mid P' \Rightarrow x = y, P \mathbin{\&\&} \neg b \Rightarrow \textbf{reads}\ y \not\cdot (x, \varepsilon)$

and $\textbf{modifies}\ b \notin (\varepsilon_1, \varepsilon_2)$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ (\textbf{modifies}\ x, \varepsilon_2)]}{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow r = \textbf{alloc}, P \Rightarrow b = E, P \Rightarrow \neg b, P \mid\mid P' \Rightarrow x = y,$

$P \mathbin{\&\&} \neg b \Rightarrow \textbf{reads}\ y \not\cdot (x, \varepsilon), \textbf{reads}\ r \notin (\varepsilon_1, \varepsilon_2)$ and $\textbf{modifies}\ b \notin (\varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *VarMask2*$_u$.

15. *FieldMask1*: Suppose that the FRL proof consists of the rule *FieldMask1*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ (\textbf{modifies region}\{x.f\}, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

$$\textbf{where } P \Rightarrow b = E, P \Rightarrow b, P \mid\mid P' \Rightarrow x.f = y,$$

$$P' \mathbin{\&\&} b \Rightarrow \textbf{reads}\ x \textbf{/} \cdot\ \textbf{modifies}\ \varepsilon,$$

$$P' \mathbin{\&\&} b \Rightarrow \textbf{reads}\ y \textbf{/} \cdot \textbf{modifies}\ \varepsilon \text{ and } \textbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$$

Then, the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ (\textbf{modifies region}\{x.f\}, \varepsilon_1)\ :\ \varepsilon_2]}{\vdash_u^\Gamma [\textbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

$$\textbf{where } P \Rightarrow r = \textbf{alloc}, P \Rightarrow b = E, P \Rightarrow b, P \mid\mid P' \Rightarrow x.f = y,$$

$$P' \mathbin{\&\&} b \Rightarrow \textbf{reads}\ x \textbf{/} \cdot\ \textbf{modifies}\ \varepsilon, P' \mathbin{\&\&} b \Rightarrow \textbf{reads}\ y \textbf{/} \cdot \textbf{modifies}\ \varepsilon$$

$$\textbf{modifies}\ r \notin (\varepsilon, \varepsilon_1, \varepsilon_2) \text{ and } \textbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2) \tag{C.1}$$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *FieldMask1*$_u$.

16. *FieldMask2*: Suppose that the FRL proof consists of the rule *FieldMask2*$_r$:

$$\frac{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ (\textbf{modifies region}\{x.f\}, \varepsilon_2)]}{\vdash_r^\Gamma \{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

$$\textbf{where } P \Rightarrow b = E, P \Rightarrow \neg b, P \mid\mid P' \Rightarrow x.f = y,$$

$$P' \mathbin{\&\&} \neg b \Rightarrow \textbf{reads}\ x \textbf{/} \cdot\ \textbf{modifies}\ \varepsilon,$$

$$P' \mathbin{\&\&} \neg b \Rightarrow \textbf{reads}\ y \textbf{/} \cdot \textbf{modifies}\ \varepsilon \text{ and } \textbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$$

Then the proof obligation is to show that the following is derivable in UFRL.

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ (\mathbf{modifies}\ \mathbf{region}\{x.f\}, \varepsilon_2)]}{\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P\}\ S\ \{P'\}[\varepsilon,\ E\ ?\ \varepsilon_1\ :\ \varepsilon_2]}$$

where $P \Rightarrow r = \mathbf{alloc}, P \Rightarrow b = E, P \Rightarrow \neg b, P \mid\mid P' \Rightarrow x.f = y,$

$P' \mathbin{\&\&} \neg b \Rightarrow \mathbf{reads}\ x \diagup \mathbf{modifies}\ \varepsilon, P' \mathbin{\&\&} \neg b \Rightarrow \mathbf{reads}\ y \diagup \mathbf{modifies}\ \varepsilon$

$\mathbf{modifies}\ r \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$ and $\mathbf{modifies}\ b \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

By the inductive hypothesis, the premise is assumed. Then, its conclusion is derived by using the rule *FieldMask2*$_u$.

Next, it is proved from the right side of the left side. It means that if there is a proof $\vdash_u^\Gamma [\mathbf{reads}\ r\downarrow]\{P_1\}S\{P_2\}[\varepsilon]$ in UFRL, where $P_1 \Rightarrow r = \mathbf{alloc}$ and $\mathbf{modifies}\ r \notin \varepsilon$, then there is a proof $\vdash_r^\Gamma \{P_1\}S\{P_2\}[\varepsilon]$ in FRL. The proof is done by the induction on the UFRL derivation and by cases on the last rule used, and is omitted. □

APPENDIX D: PROOF OF THEOREM 5

Theorem 5: *An assertion in SL is* supported *if and only if it has semantic footprint.*

Proof. Let Γ be a well-formed type environment. Let (σ, h) be a state and a be an assertion in SL, such that $\sigma, h \models_s^\Gamma a$. Let $\mathcal{H} \stackrel{\text{def}}{=} \{h' | h' \subseteq h \wedge \sigma, h' \models_s^\Gamma a\}$. Any subset of $\mathcal{H}$ defines a partial order, i.e., $\mathcal{H}_1 \leqslant \mathcal{H}_2$ *iff* $\mathcal{H}_1, \mathcal{H}_2, \in \mathcal{P}(\mathcal{H})$ *and* $\mathcal{H}_1 \subseteq \mathcal{H}_2$. Define $(\downarrow \mathcal{H}_i) \stackrel{\text{def}}{=} \{\mathcal{H}' | \mathcal{H}' \leqslant \mathcal{H}_i \wedge \mathcal{H}' \in \mathcal{P}(\mathcal{H})\}$, 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 $\bigsqcap_{\mathcal{H}}$ define the greatest lower bound of any subset of the intersection. Let $\bigsqcap_{\mathcal{H}}$ 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

$$\mathcal{H}_a \leqslant (\mathcal{H}_1 \bigsqcap_{\mathcal{H}} \mathcal{H}_2) \textit{ iff } (\mathcal{H}_a \leqslant \mathcal{H}_1 \textit{ and } \mathcal{H}_a \leqslant \mathcal{H}_2).$$

Thus, H_a is the least subheap for an assertion a in Definition 14. Next, it is shown that $dom(H_a)$ is a's semantic footprint. Let $\mathcal{R} \stackrel{\text{def}}{=} \{r | \sigma, h\restriction r \models_s^\Gamma a\}$. Any subset of $\mathcal{R}$ defines a partial order in a way similar to $\mathcal{H}$. Let $\bigsqcap_{\mathcal{R}}$ define the greatest lower bound of any subset of $\mathcal{R}$. Let $\mathcal{DOM}$ be a functor from $\mathcal{P}(\mathcal{H})$ to $\mathcal{P}(\mathcal{R})$, such that $\mathcal{DOM}(\{h_1, h_2, \ldots, h_n\}) = \{dom(h_1), dom(h_2), \ldots, dom(h_n)\}$. If $(\mathcal{H}_1) \leqslant (\mathcal{H}_2)$, then $\mathcal{DOM}(\mathcal{H}_1) \leqslant \mathcal{DOM}(\mathcal{H}_2)$. Thus $\mathcal{H}_p \leqslant (\mathcal{H}_1 \bigsqcap_{\mathcal{H}} \mathcal{H}_2)$ *iff* $\mathcal{DOM}(\mathcal{H}_p) \leqslant \mathcal{DOM}(\mathcal{H}_1) \bigsqcap_{\mathcal{R}} \mathcal{DOM}(\mathcal{H}_2)$. □

APPENDIX E: PROOF OF THEOREM 7

Theorem 7: Let Γ be a well-formed type environment. Let a be an assertion in SSL. Then $\sigma, h \models_s^\Gamma a$ *iff* $\sigma, h \models_u^\Gamma \mathtt{TR}[\![a]\!]$.

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 $a_1 * a_2$. The inductive hypothesis is that for all subassertions a_i, $\sigma, h \models_s^\Gamma a_i$ *iff* $\sigma, h \models_u^\Gamma \mathtt{TR}[\![a_i]\!]$.

From the left side to the right side is firstly proved. Assume $\sigma, h \models_s^\Gamma a_1 * a_2$. The proof obligation is to show that $\sigma, h \models_u^\Gamma \mathtt{TR}[\![a_1]\!]$ && $\mathtt{TR}[\![a_2]\!]$ && $(\mathit{fpt}_s(a_1)\ !!\ \mathit{fpt}_s(a_2))$.

$\sigma, h \models_s^\Gamma a_1 * a_2$

iff ⟨by the semantics of separation logic (Def. 11)⟩

$\mathit{exists}\ h_1, h_2 :: (h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2)$

iff ⟨by let fresh variables, h_1 and h_2, be the witnesses of the existential variables.⟩

$h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2$

implies⟨by truth of assertions is preserved under heap extension (Lemma 9)⟩

$h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2\ \mathit{and}\ \sigma, h \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h \models_s^\Gamma a_2$

implies⟨by let r_1 and r_2 be fresh, and by Theorem 6⟩

$h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2\ \mathit{and}\ \sigma, h \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h \models_s^\Gamma a_2\ \mathit{and}$

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1\ \mathit{and}\ \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2$

implies⟨by Corollary 4 and $h_1 \bot h_2$⟩

$h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2\ \mathit{and}\ \sigma, h \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h \models_s^\Gamma a_2\ \mathit{and}$

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1\ \mathit{and}\ \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2\ \mathit{and}\ r_1\ !!\ r_2$

iff ⟨by inductive hypothesis⟩

$h_1 \bot h_2\ \mathit{and}\ h = h_1 \cdot h_2\ \mathit{and}\ \sigma, h_1 \models_s^\Gamma a_1\ \mathit{and}\ \sigma, h_2 \models_s^\Gamma a_2\ \mathit{and}\ \sigma, h \models_s^\Gamma a_1\ \mathit{and}\sigma, h \models_s^\Gamma a_2\ \mathit{and}$

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1\ \mathit{and}\ \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2\ \mathit{and}\ r_1\ !!\ r_2$

$\mathit{and}\ \sigma, h \models_u^\Gamma \mathtt{TR}[\![a_1]\!]\ \mathit{and}\ \sigma, h \models_u^\Gamma \mathtt{TR}[\![a_2]\!]$

iff $\langle$by the semantics of assertions (Fig. 3.2)$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!] \;\&\&\; \texttt{TR}[\![a_2]\!] \;\&\&\; (\mathit{fpt}_s(a_1) \;!\,!\; \mathit{fpt}_s(a_2))$

iff $\langle$by Mapping from SSL to UFRL (Def. 16)$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1 * a_2]\!]$

Next, it is proved it from the right side to the left side. Assume

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!] \;\&\&\; \texttt{TR}[\![a_2]\!] \;\&\&\; (\mathit{fpt}_s(a_1) \;!\,!\; \mathit{fpt}_s(a_2))$. The proof obligation is to show that $\sigma, h \models_s^\Gamma a_1 * a_2$.

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!] \;\&\&\; \texttt{TR}[\![a_2]\!] \;\&\&\; (\mathit{fpt}_s(a_1) \;!\,!\; \mathit{fpt}_s(a_2))$

iff $\langle$by the semantics of assertions (Fig. 3.2)$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!]$ *and* $\sigma, h \models_u^\Gamma \texttt{TR}[\![a_2]\!]$ *and* $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1$ *and*

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2$ *and* $r_1 \;!\,!\; r_2$

iff $\langle$by inductive hypothesis$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!]$ *and* $\sigma, h \models_u^\Gamma \texttt{TR}[\![a_2]\!]$ *and* $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1$ *and*

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2$ *and* $r_1 \;!\,!\; r_2$ *and* $\sigma, h \models_u^\Gamma a_1$ *and* $\sigma, h \models_u^\Gamma a_2$

iff $\langle$by Corollary 5$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!]$ *and* $\sigma, h \models_u^\Gamma \texttt{TR}[\![a_2]\!]$ *and* $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1$ *and*

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{regoin}]\!](\sigma) = r_2$ *and* $r_1 \;!\,!\; r_2$ *and* $\sigma, h{\restriction}r_1 \models_u^\Gamma a_1$ *and* $\sigma, h{\restriction}r_2 \models_u^\Gamma a_2$

implies $\langle$by $h{\restriction}r_2 \subseteq h{\restriction}(\mathit{dom}(h) - r_1)$ and truth of assertions is closed under heap extension (Lemma 9)$\rangle$

$\sigma, h \models_u^\Gamma \texttt{TR}[\![a_1]\!]$ *and* $\sigma, h \models_u^\Gamma \texttt{TR}[\![a_2]\!]$ *and* $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_1) : \mathbf{region}]\!](\sigma) = r_1$ *and*

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a_2) : \mathbf{region}]\!](\sigma) = r_2$ *and* $r_1 \;!\,!\; r_2$

and $\sigma, h{\restriction}r_1 \models_u^\Gamma a_1$ *and* $\sigma, h{\restriction}r_2 \models_u^\Gamma a_2$ *and* $\sigma, h{\restriction}(\mathit{dom}(h) - r_1) \models_u^\Gamma a_2$

implies $\langle$by Corollary 4, $r_1 \cup (\mathit{dom}h - r_1) = \mathit{dom}(h)$, and $h_1 = h \restriction r_1$ and $h_2 = h \restriction (\mathit{dom}(h) - r_1)$$\rangle$

exists $h_1, h_2 :: (h_1 \perp h_2$ *and* $h = h_1 \cdot h_2$ *and* $\sigma, h_1 \models_s^\Gamma a_1$ *and* $\sigma, h_2 \models_s^\Gamma a_2)$

iff $\langle$by the semantics of separation logic (Def. 11)$\rangle$

$\sigma, h \models_s^{\Gamma} a_1 * a_2$

□

APPENDIX F: PROOF OF LEMMA 13

Lemma 13: Let Γ be a well-formed type environment. Let a and a' be assertions and S be a statement, such that $\models_s^\Gamma \{a\}S\{a'\}$. Let (σ, H) be an arbitrary state. If $\sigma, H \models_s^\Gamma a$ and $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$, then:

1. *for all* $x \in dom(\sigma)$:: $\sigma'(x) \neq \sigma(x)$ *implies* $x \in \texttt{MV}(S)$.
2. *for all* $(o, f) \in dom(H)$:: $H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \textbf{region}]\!](\sigma)$.
3. *for all* $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \textbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \textbf{region}]\!](\sigma))$:: $(o, f) \in (dom(H') - dom(H))$.

Proof. Let a, a', S, (σ, H) be given, such that $\models_s^\Gamma \{a\}S\{a'\}$. Let (σ', H') be such that $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$.

For property 1,it must be shown that *for all* $x \in dom(\sigma)$:: $\sigma'(x) \neq \sigma(x)$*implies*$x \in \texttt{MV}(S)$. The proof is by induction on the structure of the statement S and the definition of $\texttt{MV}(S)$. 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, $\sigma' = Extend(\sigma, x, default(T))$. Thus, it is vacuously true, as *Extend* only extends σ by definition.
3. (*ALLOC*) In this case, S has the form $y :=$ **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 σ'. For y, it must be true that $\sigma'(y) \neq \sigma(y)$, and $y \in \texttt{MV}(y :=$ **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 $y \in \texttt{MV}(y := e) = \{y\}$, according to Fig. 6.2.

5. (*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. (*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 $y \in \texttt{MV}(y := x'.f) = \{y\}$, according to Fig. 6.2.

The inductive hypothesis is that for all substatements S_i, (σ_i, H_i), and (σ_i', H_i'), *for all* $x \in dom(\sigma_i) :: \sigma_i'(x) \neq \sigma_i(x)$ *implies* $x \in \texttt{MV}(S_i)$. There are 3 inductive cases.

1. (*IF*) In this case, S has the form **if** $e\ \{S_1\}$ **else** $\{S_2\}$. According to its semantics Fig. 2.4, if $\mathcal{E}[\![\Gamma \vdash e : \textbf{bool}]\!](\sigma) \neq 0$, then the result follows from the inductive hypothesis applied to S_1. Similarly if $\neg\mathcal{E}[\![\Gamma \vdash e : \textbf{bool}]\!](\sigma)$, the result also follows similarly.

2. (*WHILE*) In this case, S has the form **while** $e\ \{S\}$. According to its semantics Fig. 2.4 on page 23, there exists $n \geqslant 0$, such that $\sigma' = \sigma_n$ and $\neg\mathcal{E}[\![\Gamma \vdash e : \textbf{bool}]\!](\sigma_n)$. 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 *for all* $x \in dom(\sigma) :: \sigma_{n-1}(x) \neq \sigma(x)$ *implies* $x \in \texttt{MV}(S)$. And by the inductive hypothesis, *for all* $x \in dom(\sigma_{n-1}) :: \sigma_n(x) \neq \sigma_{n-1}$ *implies* $x \in \texttt{MV}(S)$. Thus, *for all* $x \in dom(\sigma) :: \sigma(x) \neq \sigma'(x)$ *implies* $x \in \texttt{MV}(S)$.

3. (*SEQ*) In this case, S has the form S_1S_2. By definition, $\texttt{MV}(S_1S_2) = \texttt{MV}(S_1) \cup \texttt{MV}(S_2)$. According to the statement's semantics Fig. 2.4, assume σ_1 is the post-states of S_1. By the inductive hypothesis, *for all* $x \in dom(\sigma) :: \sigma_1(x) \neq \sigma(x)$ *implies* $x \in \texttt{MV}(S_1)$, and *for all* $x \in dom(\sigma_1) :: \sigma'(x) \neq \sigma_1(x)$ *implies* $x \in \texttt{MV}(S_2)$. Thus, *for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* $(x \in \texttt{MV}(S_1S_2))$.

For property 2, it must be shown that *for all* $(o, f) \in dom(H)$:: $H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$. Assume that $\sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\}$, $\sigma, H \models_s^\Gamma a$ and $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$. The proof is done in calculational style, starting from the assumptions.

$\sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and* $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$

iff $\left\langle \begin{array}{l} \text{by assumption } \models_s^\Gamma \{a\}S\{a'\}, \\ \text{thus } (\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\} \textit{ iff } \sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\} \end{array} \right\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by Corollary 5: $\sigma, H \models_s^\Gamma a$ *iff* $(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$

iff $\langle$by the definition of SSL valid Hoare-formula (Def. 17)$\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \neq err$ and if

$((\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)),$ then $\sigma', H'' \models_s^{\Gamma'} a')$.

iff $\langle$by frame property of SL$\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \neq err$ *and*

$((\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)),$ *and* $\sigma', H'' \models_s^{\Gamma'} a')$

and $H''\perp H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$

implies$\langle$by A and B implies $B\rangle$

$H'' \perp H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$

iff $\left\langle \text{by} \begin{array}{l} \textit{for all } (o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) :: \ldots \textit{ implies} \\ (o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) \text{ is a tautology} \end{array} \right\rangle$

$H'' \perp H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) :: H''[o,f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)[o,f]$ *implies* $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$

implies $\left\langle \text{by} \begin{array}{l} H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \textit{ and } (dom(H) - \\ \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) \cap \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) = \varnothing \end{array} \right\rangle$

$H'' \perp H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$

and for all $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) :: H'[o,f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)[o,f]$ *implies* $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$

implies$\langle$by Corollary 4, $\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) \subseteq dom(H)$, twice$\rangle$

$H'' \perp H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $(o,f) \in dom(H) :: H'[o,f] \neq H[o,f]$ *implies* $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$

implies$\langle$by A and B *implies*$B\rangle$

for all $(o,f) \in dom(H) :: H'[o,f] \neq H[o,f]$ *implies* $(o,f) \in \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)$

For property 3, it must be shown that *for all* $(o,f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) :: (o,f) \in (dom(H') - dom(H))$. Assume that $\sigma, H \models_s^\Gamma \{a\}\, S\, \{a'\}$, $\sigma, H \models_s^\Gamma a$ and $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$. The proof is done in calculational style, starting from the assumptions.

$\sigma, H \models_s^\Gamma \{a\}\, S\, \{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and* $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$

iff $\left\langle \begin{array}{l} \text{by assumption } \models_s^\Gamma \{a\}S\{a'\}, \\ \text{thus } (\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\} \textit{ iff } \sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\} \end{array} \right\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by Corollary 5: $\sigma, H \models_s^\Gamma a$ *iff* $\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma a\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma a$

iff $\langle$by the definition of SSL valid Hoare-formula (Def. 17)$\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ and if

$((\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)),$ then $\sigma', H'' \models_s^{\Gamma'} a').$

iff $\langle$by frame property of SL$\rangle$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *and* $\sigma, H \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H) = (\sigma', H')$ *and* $\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma a$ *and*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

$((\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)),$ *and* $\sigma', H'' \models_s^{\Gamma'} a')$

and $H''\perp H\restriction(\mathit{dom}(H) - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(\mathit{dom}(H) - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$

implies $\langle$by $A \wedge B$ *implies* $B\rangle$

$\sigma', H'' \models_s^{\Gamma'} a'$ *and* $H''\perp H\restriction(\mathit{dom}(H) - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H\restriction(\mathit{dom}(H) - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$

iff $\langle$by (*for all* $(o, f) \in (r' - r) :: (o, f) \in (r' - r)$) is a tautology$\rangle$

$\sigma', H'' \models_s^{\Gamma'} a'$ *and* $H'' \perp H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$

implies$\langle$by Corollary 4, $\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma) \subseteq dom(H)\rangle$

$\sigma', H'' \models_s^{\Gamma'} a'$ *and* $H'' \perp H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') -$

$\mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) :: (o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - dom(H))$

implies$\langle$by $H' = H'' \cdot H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ and Corollary 4$\rangle$

$\sigma', H'' \models_s^{\Gamma'} a'$ *and* $H'' \perp H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' = H'' \cdot H \upharpoonright (dom(H) - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (dom(H') - dom(H))$

implies$\langle$by A and B implies $B\rangle$

(*for all* $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash fpt_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash fpt_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (dom(H') - dom(H)))$

□

APPENDIX G: PROOF OF THEOREM 8

Theorem 8: Let Γ be a well-formed type environment. Let S be a statement, and let a and a' be assertions in SSL, such that $\models_s^\Gamma \{a\}S\{a'\}$. Let r be a region variable. Let (σ, H) be a Γ-state. If $\sigma, H \models^\Gamma \mathtt{TR}[\![a]\!] \Rightarrow r = \mathit{fpt}_s(a)$ and $r \notin \mathtt{MV}(S)$, then

$\sigma, H \models_s^\Gamma \{a\}\ S\{a'\}$ *iff*

$\sigma, H \models_u^\Gamma [\mathit{fpt}_s(a)]\{\mathtt{TR}[\![a]\!]\}S\{\mathtt{TR}[\![a']\!]\}[\,\mathbf{modifies}\,\mathit{fpt}_s(a), \mathtt{MV}(S), \mathbf{fresh}\,(\mathit{fpt}_s(a') - r)]$

Proof. Assume that $\sigma, H \models_s^\Gamma \{a\}S\{a'\}$. The lemma is proved by mutual implication. First it is proved that the left side implies the right side.

$\sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\}$

iff $\left\langle \begin{array}{l} \text{by assumption } \models_s^\Gamma \{a\}S\{a'\}, \\ \text{thus } (\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\} \textit{ iff } \sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\} \end{array} \right\rangle$

$(\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$

iff $\langle$by the definition of SSL valid Hoare-formula (Def. 17)$\rangle$

$(\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$ implies

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then $\sigma', H' \models_s^{\Gamma'} a'$

implies$\langle$by Lemma 13$\rangle$

$(\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma a$ implies

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then $\sigma', H' \models_s^{\Gamma'} a'$

and for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$:

$H'[o, f] \neq H{\restriction}\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ ::

$(o, f) \in (\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)))$

$$
\textit{implies}\left\langle
\begin{array}{ll}
\text{by} & \text{termination monotonicity as } (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))\bot \\
& H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \textit{ and } H = \\
& (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \cdot H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)
\end{array}
\right\rangle
$$

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^{\Gamma} a$ implies

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then

$\sigma', H' \models_s^{\Gamma'} a'$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) : H'[o, f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)))$

implies⟨by the frame property of SL⟩

$(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^{\Gamma} a$ implies

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then

$\sigma', H' \models_s^{\Gamma'} a'$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

$H'' = H' \cdot (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H'\bot(H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) : H'[o, f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)))$

implies⟨by Corollary 5⟩

$\sigma, H \models_s^\Gamma a$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then $\sigma', H' \models_s^{\Gamma'} a'$

and $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

$H'' = H' \cdot (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' \perp (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) : H'[o, f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)))$

iff ⟨by Theorem 7, twice⟩

$\sigma, H \models_u^\Gamma \mathtt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \mathtt{TR}[\![a']\!]$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

$H'' = H' \cdot (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

$H' \perp (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) : H'[o, f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \mathbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$

implies ⟨ by $\mathit{dom}(H') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) = \mathit{dom}(H') - \mathit{dom}(H)$, because

$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \subseteq \mathit{dom}(H)$ by Corollary 4, and

$H' \perp (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$ ⟩

$\sigma, H \models^{\Gamma}_{u} \texttt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \texttt{TR}[\![a']\!]$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and* $H'' =$

$H' \cdot (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$ *and* $H' \perp (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$

and *for all* $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \texttt{MV}(S)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma) : H'[o, f] \neq$

$H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \textbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))$

implies ⟨ by $\{(o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma) ::$
$H'[o, f] \neq H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)[o, f]\} =$
$\{(o, f) \in \mathit{dom}(H) :: H'[o, f] \neq H[o, f]\}$, because
$\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma) \subseteq \mathit{dom}(H)$
by Corollary 4, *and* $H' \perp (H - H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$ ⟩

$\sigma, H \models^{\Gamma}_{u} \texttt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \texttt{TR}[\![a']\!]$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \texttt{MV}(S)$ *and*

for all $(o, f) \in \mathit{dom}(H) : H'[o, f] \neq H[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \textbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))$

iff ⟨by assumption $\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash r : \textbf{region}]\!](\sigma)$⟩

$\sigma, H \models_u^{\Gamma} \texttt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \texttt{TR}[\![a']\!]$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \texttt{MV}(S)$ *and*

for all $(o, f) \in \mathit{dom}(H) : H'[o, f] \neq H[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in (\mathcal{E}[\![\Gamma' \vdash \mathit{fpt}_s(a') : \textbf{region}]\!](\sigma') - \mathcal{E}[\![\Gamma \vdash r : \textbf{region}]\!](\sigma)) ::$

$(o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))$

iff $\langle$by $\mathcal{E}[\![\Gamma \vdash r : \textbf{region}]\!](\sigma) = \mathcal{E}[\![\Gamma' \vdash r : \textbf{region}]\!](\sigma')$ because $r \notin \texttt{MV}(S)\rangle$

$\sigma, H \models_u^{\Gamma} \texttt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \texttt{TR}[\![a']\!]$ *and* $(\sigma', H'') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H)$ *and*

for all $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \texttt{MV}(S)$ *and*

for all $(o, f) \in \mathit{dom}(H) : H'[o, f] \neq H[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \textbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma' \vdash (\mathit{fpt}_s(a') - r) : \textbf{region}]\!](\sigma') :: (o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))$

implies $\left\langle \begin{array}{l} \text{by the definition of UFRL valid Hoare-formula (Def. 8),} \\ \text{as } \mathit{freshR}(\textbf{modifies}\ \mathit{fpt}_s(a), \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r)) \text{ is } (\mathit{fpt}_s(a') - r) \end{array} \right\rangle$

$\sigma, H \models_u^{\Gamma} [\mathit{fpt}_s(a)]\{\texttt{TR}[\![a]\!]\}S\{\texttt{TR}[\![a']\!]\}[\,\textbf{modifies}\ \mathit{fpt}_s(a), \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r))]$

where $\texttt{TR}[\![a]\!]$ *implies* $r = \mathit{fpt}_s(a)$

Next, let $r = \mathit{fpt}_s(a)$. The proof goes from the right side to the left side.

$\sigma, H \models_u^{\Gamma} [\mathit{fpt}_s(a)]\{\texttt{TR}[\![a]\!]\}S\{\texttt{TR}[\![a']\!]\}[\,\textbf{modifies}\ \mathit{fpt}_s(a), \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r))]$

implies $\left\langle \begin{array}{l} \text{by the definition of UFRL valid Hoare-formula (Def. 8),} \\ \text{as } \mathit{freshR}(\textbf{modifies}\ \mathit{fpt}_s(a), \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r)) \text{ is } (\mathit{fpt}_s(a') - r) \end{array} \right\rangle$

$\sigma, H \models_u^\Gamma \mathtt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then

$\sigma', H' \models_u^\Gamma \mathtt{TR}[\![a']\!]$ *and for all* $x \in \mathit{dom}(\sigma) : \sigma'(x) \neq \sigma(x) : x \in \mathtt{MV}(S)$ *and*

for all $(o, f) \in \mathit{dom}(H) : H'[o, f] \neq H[o, f] : (o, f) \in \mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)$ *and*

for all $(o, f) \in \mathcal{E}[\![\Gamma' \vdash (\mathit{fpt}_s(a') - r) : \mathbf{region}]\!](\sigma') :: (o, f) \in (\mathit{dom}(H') - \mathit{dom}(H))$

implies ⟨by A *and* B *implies* A⟩

$\sigma, H \models_u^\Gamma \mathtt{TR}[\![a]\!]$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then

$\sigma', H' \models^{\Gamma'} \mathtt{TR}[\![a']\!]$

iff ⟨by Theorem 7, twice⟩

$\sigma, H \models_s^\Gamma a$ implies $\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then $\sigma', H' \models_s^{\Gamma'} a'$

implies ⟨by Corollary 5⟩

$\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma a$ *implies*

$\mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \neq \mathit{err}$ *and*

if $(\sigma', H') = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma')]\!](\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma))$, then $\sigma', H' \models_s^{\Gamma'} a'$

iff ⟨by the definition of SL validity Hoare-formula (Def. 17)⟩

$\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma) \models_s^\Gamma \{a\}\ S\ \{a'\}$

iff ⟨by assumption $\models_s^\Gamma \{a\}S\{a'\}$, thus $(\sigma, H\restriction\mathcal{E}[\![\Gamma \vdash \mathit{fpt}_s(a) : \mathbf{region}]\!](\sigma)) \models_s^\Gamma \{a\}S\{a'\}$ *iff* $\sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\}$⟩

$\sigma, H \models_s^\Gamma \{a\}\ S\ \{a'\}$

□

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 $SKIP_u$.

2. *VAR:* by Def. 18, the encoded axiom is the axiom VAR_u.

3. *ALLOC:* by the rule $ALLOC_s$ and Def. 18, the translated rule is is shown below:

$$\begin{array}{ll} & [\mathbf{reads}\ fpt_s(a)] \\ \vdash_u^{\Gamma} & \{\mathtt{TR}[\![a]\!]\}x := \mathbf{new}\ T;\ \{\mathtt{TR}[\![a * new_s(T,x)]\!]\} \\ & [\,\mathbf{modifies}\ fpt_s(a), x, \mathbf{alloc}, \mathbf{fresh}(fpt_s(new_s(T,x)))] \\ & \mathbf{where}\ x \notin \mathtt{FV}(a) \end{array} \tag{H.1}$$

   By definition of the predicate $new_s(T,x)$, it is known that $fpt_s(new_s(T,x)) = \mathbf{region}\{x.*\}$. Given the axiom $ALLOC_u$, the translated rule is derived by using the rules FRM_u and $SubEff_u$. 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 $new(T,x)$ *iff* $new_s(T,x)$.

4. *ACC:* by the rule ACC_s and Def. 18, the translated rule is shown below:

$$\begin{array}{ll} & [\mathbf{region}\{x'.f\}] \\ \vdash_u^{\Gamma} & \{x'.f = z\}\ \mathtt{x\ :=\ x'\ f;}\ \{x'.f = z\ \&\&\ x = z\} \\ & [\,\mathbf{modifies}\ \mathbf{region}\{x'.f\}, x] \\ & \mathbf{where}\ x \neq x', x' \neq z\ and\ x \neq z \end{array} \tag{H.2}$$

   where the fresh effect is empty, thus, it is omitted; it is true that $fpt_s(x = z)\ !!\ fpt_s(x'.f \mapsto z)$, thus, it is omitted. Given the axiom ACC_u, by definition of read effects for assertions in Fig. 3.5, it must be true that $(x'.f = z) \vdash^{\Gamma} (\mathbf{reads}\ x', \mathbf{region}\{x'.f\}, z)\ frm\ x'.f = z$. By the side

conditions and the definition of separator, it must be true that $(\mathbf{reads}\ (x', \mathbf{region}\{x'.f\}, z) \not\perp \mathbf{modifies}\ x$. 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:

$$\vdash_u^\Gamma \begin{array}{l} [\mathbf{reads}\ \mathbf{region}\{x.f\}] \\ \{\exists\, z.x.f = z\}x.f := E; \{x.f = E\}[\,\mathbf{modifies}\ \mathbf{region}\{x.f\}] \\ \quad \mathbf{where}\ x \notin \mathtt{FV}(E) \end{array} \tag{H.3}$$

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:

$$\frac{\begin{array}{l} \vdash_u^\Gamma \begin{array}{l} [\mathbf{reads}\ r_1\downarrow] \\ \{\mathtt{TR}[\![a]\!]\}\ S_1\ \{\mathtt{TR}[\![b]\!]\}[\,\mathbf{modifies}\ r_1\downarrow, \mathtt{MV}(S_1), \mathbf{fresh}\ (r_2 - r_1)] \end{array} \\ \vdash_u^\Gamma \begin{array}{l} [\mathbf{reads}\ r_2\downarrow] \\ \{\mathtt{TR}[\![b]\!]\}\ S_2\ \{\mathtt{TR}[\![a']\!]\}[\,\mathbf{modifies}\ r_2\downarrow, \mathtt{MV}(S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_2)] \end{array} \end{array}}{\vdash_u^\Gamma \begin{array}{l} [\mathbf{reads}\ r_1\downarrow] \\ \{\mathtt{TR}[\![a]\!]\}\ S_1S_2\ \{\mathtt{TR}[\![a']\!]\}[\,\mathbf{modifies}\ r_1\downarrow, \mathtt{MV}(S_1S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \\ \quad \mathbf{where}\ \mathtt{TR}[\![a]\!] \Rightarrow r_1 = \mathit{fpt}_s(a), r_1 \notin \mathtt{MV}(S_1), \mathtt{TR}[\![b]\!] \Rightarrow r_2 = \mathit{fpt}_s(b)\ \mathit{and}\ r_2 \notin \mathtt{MV}(S_2) \end{array}} \tag{H.4}$$

There are two cases:

(a) $S_1 = \mathbf{var}\ x\ T;$: In this case, by the rule VAR_s, it must be true that $b = a * \mathit{default}(T)$,

$\mathtt{MV}(\mathbf{var}\ x : T) = \varnothing$ and $r_1 = r_2 = \mathit{fpt}_s(a)$. Then the proof obligation is to show

$$\frac{\begin{array}{rl} \vdash_u^\Gamma & [\mathbf{reads}\ r_1\downarrow]\{\mathtt{TR}[\![a]\!]\}\ \mathbf{var}\ x : T;\ \{\mathtt{TR}[\![a * \mathit{default}(T)]\!]\}[\ \mathbf{modifies}\ r_1\downarrow] \\ & [\mathbf{reads}\ r_1\downarrow] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a * \mathit{default}(T)]\!]\}\ S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & [\ \mathbf{modifies}\ r_1\downarrow, \mathtt{MV}(S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array}}{\begin{array}{rl} & [\mathbf{reads}\ r_1\downarrow] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a]\!]\}\ \mathbf{var}\ x : T; S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & [\ \mathbf{modifies}\ r_1\downarrow, \mathtt{MV}(S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array}} \quad \text{(H.5)}$$

where $\mathtt{TR}[\![a * \mathit{default}(T)]\!] \Rightarrow r_2 = \mathit{fpt}_s(a)\ \mathit{and}\ r_2 \notin \mathtt{MV}(S_2)$

Using the rule *SubEff*$_u$ on the second premise, the following is derived

$$\begin{array}{rl} & [\mathbf{reads}\ r_1\downarrow\ ] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a * \mathit{default}(T)]\!]\}\ S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & [\ \mathbf{modifies}\ r_1\downarrow, x, \mathtt{MV}(S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array} \quad \text{(H.6)}$$

Using the rule *SEQ2*$_u$, the conclusion of Eq. (H.5) is derived.

(b) $S_1 \neq \mathbf{var}\ x : T$;: consider the following cases:

• S_1 does not allocate new locations, i.e., $r_1 = r_2$. The rule *SEQ1*$_u$ is instantiated with *RE* := **region**{}, RE_1 := **region**{} and RE_2 := **region**{}. If the immunity side conditions are satisfied, then the conclusion of (H.4) is derived by using the rule *SEQ1*$_u$. Otherwise, *for all* $x \in \mathtt{MV}(S_1)$ and x in $\mathtt{FV}(b)$, there exists z, such that b implies $x = z$ and $z \notin \mathtt{MV}(S_1)$. The variable z is substituted for x in $\mathit{fpt}_s(b)$. Then the second premise of Eq. (H.4) is re-written as:

$$\begin{array}{rl} & [\mathbf{reads}\ r_1\downarrow\ [\overline{z}/\mathtt{MV}(S_1)]] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![b]\!]\}\ S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & [\ \mathbf{modifies}\ r_1\downarrow\ [\overline{z}/\mathtt{MV}(S_1)], \mathtt{MV}(S_2), \mathbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array} \quad \text{(H.7)}$$

where $r_1\downarrow\ [\overline{z}/\mathtt{MV}(S_1)]$ means that *for all RE* $\in r_1\downarrow$:: $\mathit{RE}[\mathtt{MV}(S_1)/\overline{z}]$. From the first premise of Eq. (H.4) and Eq. (H.7), the immunity side conditions are satisfied. After using the rule *SEQ1*$_u$,

the following is derived.

$$\begin{array}{ll} & [\textbf{reads}\ r_1{\downarrow}, r_1{\downarrow}\ [\overline{z}/\mathtt{MV}(S_1)]] \\ \vdash_u^{\Gamma} & \{\mathtt{TR}[\![a]\!]\}S_1S_2\{\mathtt{TR}[\![a']\!]\} \\ & [\,\textbf{modifies}\ r_1{\downarrow}, r_2{\downarrow}\ [\mathtt{MV}(S_1)/\overline{z}], \mathtt{MV}(S_1S_2), \textbf{fresh}(\mathit{fpt}_s(a') - r_1{\downarrow}\ [\mathtt{MV}(S_1)/\overline{z}])] \end{array} \tag{H.8}$$

Because *for all RE* $\in r_1$ :: *RE* in $r_2{\downarrow}\ [\overline{y/x}]$, Eq. (H.8) can be simplified to

$$\begin{array}{ll} & [\textbf{reads}\ r_1{\downarrow}] \\ \vdash_u^{\Gamma} & \{\mathtt{TR}[\![a]\!]\}S_1S_2\{\mathtt{TR}[\![a']\!]\} \\ & [\,\textbf{modifies}\ r_1{\downarrow}, \mathtt{MV}(S_1S_2), \textbf{fresh}(\mathit{fpt}_s(a') - r_1)] \end{array}$$

- S_1 allocates some new locations. Then the second premise of Eq. (H.4) can be re-written as:

$$\begin{array}{ll} & [\textbf{reads}\ r_1{\downarrow}, (r_2 - r_1)] \\ \vdash_u^{\Gamma} & \{\mathtt{TR}[\![a * \mathit{default}(T)]\!]\}\ S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & [\,\textbf{modifies}\ r_1{\downarrow}, (r_2 - r_1), \mathtt{MV}(S_2), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_2)] \end{array}$$

The rule $\mathit{SEQ1}_u$ is instantiated with $\mathit{RE} := r_2 - r_1$, $\mathit{RE}_1 := r_2 - r_1$ and $\mathit{RE}_2 := r_2 - r_1$. If the immunity side conditions are satisfied, then union the fresh effects of the two statements and get $\mathit{fpt}_s(a') - r_1$. Hence, the conclusion of Eq. (H.4) is derived by using the rule $\mathit{SEQ1}_u$. 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:

$$\frac{\begin{array}{ll} & [\mathbf{reads}\ fpt_s(a)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a]\!]\ \&\&\ E\}\ S_1\ \{\mathtt{TR}[\![a']\!]\} \\ & [\mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_1), \mathbf{fresh}\ (fpt_s(a') - r)] \\ & \quad [\mathbf{reads}\ fpt_s(a)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a]\!]\ \&\&\ \neg E\}\ S_2\ \{\mathtt{TR}[\![a']\!]\} \\ & \quad [\mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_2), \mathbf{fresh}\ (fpt_s(a') - r)] \end{array}}{\begin{array}{ll} & [\mathbf{reads}\ fpt_s(a)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a]\!]\}\ \mathbf{if}\ E\ \mathbf{then}\{S_1\}\mathbf{else}\{S_2\}\ \{\mathtt{TR}[\![a']\!]\} \\ & [\mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_1), \mathtt{MV}(S_2), \mathbf{fresh}(fpt_s(a') - r)] \\ & \mathbf{where}\ \mathtt{TR}[\![a]\!] \Rightarrow r = fpt_s(a)\ and\ r \notin \mathtt{MV}(S_1) \cup \mathtt{MV}(S_2) \end{array}} \quad \text{(H.9)}$$

Note that $fpt_s(E)$ and $fpt_s(\neg E)$ are both $\mathbf{region}\{\}$, 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.

$$\begin{array}{ll} & [\mathbf{reads}\ fpt_s(a), \mathbf{reads}\ efs(E)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![a]\!]\}\ \mathbf{if}\ E\ \mathbf{then}\{S_1\}\mathbf{else}\{S_2\}\ \{\mathtt{TR}[\![a']\!]\} \\ & [\mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_1), \mathtt{MV}(S_2), \mathbf{fresh}(fpt_s(a') - r)] \end{array} \quad \text{(H.10)}$$

Now, consider to use the rule $SubEff_u$. Because $regRW(\mathbf{reads}\ fpt_s(a),\ \mathbf{reads}\ efs(E),\ \mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_1), \mathtt{MV}(S_2), \mathbf{fresh}(fpt_s(a') - r)) = fpt_s(a)$, the following side condition is true:

$$fpt_s(a) \leqslant regRW(\mathbf{reads}\ fpt_s(a),\ \mathbf{modifies}\ fpt_s(a), \mathtt{MV}(S_1), \mathtt{MV}(S_2), \mathbf{fresh}(fpt_s(a') - r))$$

Therefore, after using the rule $SubEff_u$, the conclusion of Eq. (H.9) is derived.

8. *WHILE:* by the rule $WHILE_s$ and Def. 18, the translated rule is shown below:

$$\frac{\vdash_u^\Gamma [\mathbf{reads}\, fpt_s(I)]\{\mathtt{TR}[\![I]\!] \,\&\&\, \}S\{\mathtt{TR}[\![I]\!]\}[\,\mathbf{modifies}\, fpt_s(I), \mathtt{MV}(S)]}{\begin{array}{ll} & [\mathbf{reads}\, fpt_s(I)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![I]\!]\}\mathbf{while}\, E\, \{S\}\{\mathtt{TR}[\![I]\!] \,\&\&\, \neg E\} \\ & [\,\mathbf{modifies}\, fpt_s(I), \mathtt{MV}(S)] \end{array}} \quad \text{(H.11)}$$

The rule $WHILE_u$ is instantiated with $RE := \mathbf{region}\{\}$. The treatment about the immunity side condition is similar to that of the sequence rule. If it is satisfied, then the rule *WHILE*u is used and the following is derived.

$$\begin{array}{ll} & [\mathbf{reads}\, fpt_s(I), efs(E)] \\ \vdash_u^\Gamma & \{\mathtt{TR}[\![I]\!]\}\, \mathbf{while}\;\; E\{S\}\, \{\mathtt{TR}[\![I]\!] \,\&\&\, \neg E\} \\ & [\,\mathbf{modifies}\, fpt_s(I), \mathtt{MV}(S)] \end{array} \quad \text{(H.12)}$$

Similarly to the case of the rule IF_u, using the rule $SubEff_u$, the conclusion of Eq. (H.11) is derived.

If the immunity side condition is not satisfied, *for all* $x \in \mathtt{MV}(S)$ and $x \in \mathtt{FV}(I)$, there exists z, such that I implies $x = z$ and $z \notin \mathtt{MV}(S)$. The variable z is substituted for x in $fpt_s(I)$. Then the immunity side condition is satisfied. the rules $WHILE_u$ and $SubEff_u$ are used and the conclusion is derived.

9. *FRM:* by the rule FRM_s and Def. 18, the translated rule is shown as follows:

$$\frac{\begin{array}{ll} & [\textbf{reads}\ r_1\!\downarrow] \\ \vdash_u^{\Gamma} & \{\texttt{TR}[\![a]\!]\}\ S\ \{\texttt{TR}[\![a']\!]\} \\ & [\,\textbf{modifies}\ r_1\!\downarrow, \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_1)]\end{array}}{\begin{array}{ll} & [\textbf{reads}\ r_1 + r_2] \\ \vdash_u^{\Gamma} & \{\texttt{TR}[\![a]\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\}\ S\ \{\texttt{TR}[\![a']\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (\mathit{fpt}_s(a')\ !!\ r_2)\} \\ & [\,\textbf{modifies}\ \mathit{fpt}_s(a) + r, \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') + r_2 - r')]\end{array}}$$

where $\texttt{TR}[\![a]\!] \Rightarrow r_1 = \mathit{fpt}_s(a), \texttt{TR}[\![c]\!] \Rightarrow r_2 = \mathit{fpt}_s(c), r_1 \notin \texttt{MV}(S), r_2 \notin \texttt{MV}(S),$

$$\texttt{TR}[\![a]\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (\mathit{fpt}_s(a')\ !!\ r_2) \Rightarrow r' = r_1 + r_2, \text{and } \texttt{MV}(S)\ \cap\ \texttt{FV}(c) = \varnothing \tag{H.13}$$

By the inductive hypothesis, the premise of Eq. (H.13) is assumed. The rule FRM_u is instantiated with $Q := \texttt{TR}[\![c]\!]$ and $\eta := \mathit{efs}(\texttt{TR}[\![c]\!])$. The proof obligation is to show the side condition, which is:

$$\texttt{TR}[\![a]\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\ \mathit{implies}\ \mathit{efs}(\texttt{TR}[\![c]\!])\textcolor{gray}{\cdot/\cdot}(\textbf{modifies}\ \texttt{MV}(S), \mathit{fpt}_s(a)) \tag{H.14}$$

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.

$$\begin{array}{ll} & [\textbf{reads}\ r_1\ \downarrow] \\ \vdash_u^{\Gamma} & \{\texttt{TR}[\![a]\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\}\ S\ \{\texttt{TR}[\![a']\!]\ \&\&\ \texttt{TR}[\![c]\!]\} \\ & [\,\textbf{modifies}\ r_1\ \downarrow, \texttt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_1)]\end{array} \tag{H.15}$$

Now, consider to use the rule FRM_u again. It is instantiated with $Q := r_1\ !!\ r_2$ and $\eta := \textbf{reads}\ r_1, \textbf{reads}\ r_2$. The proof obligation is to show the side condition is true, which is:

$$(\texttt{TR}[\![a]\!]\ \&\&\ \texttt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2))\ \mathit{implies}\ (\textbf{reads}\ r_1, \textbf{reads}\ r_2)\textcolor{gray}{\cdot/\cdot}(\textbf{modifies}\ r_1\ \downarrow, \texttt{MV}(S)) \tag{H.16}$$

By $r_1 \notin \texttt{MV}(S)$ and $r_2 \notin \texttt{MV}(S)$, Eq. (H.16) is true. Note that $\textbf{modifies}\ r_1\ \downarrow$ 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.

$$\begin{array}{l} [\textbf{reads}\ r_1\downarrow] \\ \vdash_u^\Gamma\ \{\mathtt{TR}[\![a]\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\}\ S\ \{\mathtt{TR}[\![a']\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\} \\ [\textbf{modifies}\ r_1\downarrow, \mathtt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array} \tag{H.17}$$

Because $\mathtt{TR}[\![c]\!]$ is preserved by S, $r_2 = \mathit{fpt}_s(c)$ in the post-state. Thus, after using the rule $CONSEQ_u$, the following is derived.

$$\begin{array}{l} [\textbf{reads}\ r_1\downarrow] \\ \vdash_u^\Gamma\ \{\mathtt{TR}[\![a]\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\}\ S\ \{\mathtt{TR}[\![a']\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ \mathit{fpt}_s(c))\} \\ [\textbf{modifies}\ r_1\downarrow, \mathtt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array} \tag{H.18}$$

Now the proof obligation is to show that $\mathit{fpt}_s(a')\ !!\ \mathit{fpt}_s(c)$ in the poststate. By the definition of SSL Hoare-formula, it is known that $\mathit{fpt}_s(a') = r_1 + RE$, where RE are possibly empty regions that do not exist in the pre-state, hence $RE\ !!\ \mathit{fpt}_s(c)$. Hence, $\mathit{fpt}_s(a')\ !!\ \mathit{fpt}_s(c)$ is true. Then, after using the rule $CONSEQ_u$, the following is derived.

$$\begin{array}{l} [\textbf{reads}\ r_1\downarrow] \\ \vdash_u^\Gamma\ \{\mathtt{TR}[\![a]\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (r_1\ !!\ r_2)\}\ S\ \{\mathtt{TR}[\![a']\!]\ \&\&\ \mathtt{TR}[\![c]\!]\ \&\&\ (\mathit{fpt}_s(a')\ !!\ \mathit{fpt}_s(c))\} \\ [\textbf{modifies}\ r_1\downarrow, \mathtt{MV}(S), \textbf{fresh}\ (\mathit{fpt}_s(a') - r_1)] \end{array} \tag{H.19}$$

Now, consider the fresh effects. By the side condition $r' = r_1 + r_2$, it must be true that

$$\mathit{fpt}_s(a') - r_1 = \mathit{fpt}_s(a') + r_2 - r_1 - r_2 = \mathit{fpt}_s(a') + r_2 - r'.$$

Finally, using the rule $SubEff_u$ to loosen the read effects, the conclusion of Eq. (H.13) is derived.

□

$$
\begin{array}{l}
(\mathit{ALLOC}_u)\ \vdash_u^\Gamma\ [\varnothing]\ \{\mathit{true}\}\ \texttt{x := new T;}\ \{\mathit{new}(T,x)\}\ [\,\textbf{modifies}\ x, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \\
\qquad\qquad \mathit{true} \vdash_u^\Gamma \mathit{efs}(\texttt{TR}[\![a]\!])\ \mathit{frm}\ \texttt{TR}[\![a]\!] \\
\qquad\qquad \textbf{where}\ \texttt{TR}[\![a]\!] \Rightarrow \mathit{efs}(\texttt{TR}[\![a]\!]) \cdot\!/\!\cdot (x, \textbf{alloc})\ \text{and} \\
\qquad\qquad \texttt{FV}(a) \cap \{x\} = \varnothing \Rightarrow \texttt{FV}(\texttt{TR}[\![a]\!]) \cap \{x\} = \varnothing (\text{ Lemma 15}) \\
\hline
(\mathit{FRM}_u)\ [\varnothing]\ \vdash_u^\Gamma\ \{\texttt{TR}[\![a]\!]\}\ \texttt{x := new T;}\ \{\texttt{TR}[\![a]\!]\ \&\&\ \mathit{new}(T,x)\}\ [\,\textbf{modifies}\ x, \textbf{alloc}, \textbf{fresh region}\{x.*\}] \\
\qquad\qquad (\mathit{Subeffect}) \vdash^\Gamma \varnothing \leqslant \textbf{reads}\ \mathit{fpt}_s(a) \\
\hline
(\mathit{SubEff}_u)\ [\textbf{reads}\ \mathit{fpt}_s(a)]\ \vdash_u^\Gamma\ \{\texttt{TR}[\![a]\!]\}\ \texttt{x := new T;}\ \{\texttt{TR}[\![a]\!]\ \&\&\ \mathit{new}(T,x)\}\ [\,\textbf{modifies}\ x, \textbf{alloc}, \textbf{fresh region}\{x.*\}] \\
\qquad\qquad (\mathit{Subeffect})\ \texttt{TR}[\![a]\!] \vdash^\Gamma (\textbf{modifies}\ x, \textbf{alloc}) \leqslant (\textbf{modifies}\ x, \textbf{alloc}, \mathit{fpt}_s(a)) \\
\hline
(\mathit{SubEff}_u)\ [\textbf{reads}\ \mathit{fpt}_s(a)]\ \vdash_u^\Gamma\ \{\texttt{TR}[\![a]\!]\}\ \texttt{x := new T;}\ \{\texttt{TR}[\![a]\!]\ \&\&\ \mathit{new}(T,x)\}\ [\,\textbf{modifies}\ x, \textbf{alloc}, \mathit{fpt}_s(a), \textbf{fresh region}\{x.*\}] \\
\qquad\qquad \textbf{where}\ \texttt{TR}[\![a]\!] \Rightarrow r = \textbf{alloc}\ \mathit{and}\ \mathit{fpt}_s(a) \leqslant r \\
\hline
(\mathit{SubEff}_u)\ [\textbf{reads}\ \mathit{fpt}_s(a)]\ \{\texttt{TR}[\![a]\!]\}\ (\mathit{FrToPost}_u)\ \vdash_u^\Gamma\ \texttt{x := new T;}\ \{\texttt{TR}[\![a]\!]\ \&\&\ \mathit{new}(T,x)\ \&\&\ (\mathit{fpt}_s(a)\ \texttt{!!}\ \textbf{region}\{x.*\})\}\ [\,\textbf{modifies}\ x, \textbf{alloc}, \mathit{fpt}_s(a), \textbf{fresh}(\textbf{region}\{x.*\})]
\end{array}
$$

Figure H.1: The derivation of rule $\texttt{TR}_R[\![\mathit{ALLOC}_s]\!]$

$$
\dfrac{
\begin{array}{c}
(ACC_u)\ \vdash_u^\Gamma [\mathit{efs}(x'.f)]\{x' \neq \mathit{null}\}\ \texttt{x := x'.f;}\ \{x = x'.f\}[\,\textbf{modifies}\ x] \\
(x'.f = z) \vdash_u^\Gamma (\textbf{region}\{x'.f\}, x', z)\ \mathit{frm}\ (x'.f = z) \qquad \textbf{where}\ (1)
\end{array}
}{(FRM_u)\ \vdash_u^\Gamma [\mathit{efs}(x'f)]\{x'.f = z\}\ \texttt{x := x'.f;}\ \{x = z\ \&\&\ x'.f = z\}[\,\textbf{modifies}\ x]}
$$

$$
\dfrac{
\begin{array}{c}
(\mathit{Subeffect})\ \vdash^\Gamma \textbf{modifies}\ x \leqslant \textbf{modifies}\ x, \textbf{region}\{x'.f\} \\
[\mathit{efs}(x'.f)] \\
\vdash_u^\Gamma\ \{x'.f = z\}\ \texttt{x := x'.f;}\ \{x = z\ \&\&\ x'.f = z\} \qquad \textbf{where}\ (2) \\
[\,\textbf{modifies}\ x, \textbf{region}\{x'.f\}]
\end{array}
}{(SubEff_u)\ [\textbf{reads region}\{x'.f\}]\ \vdash_u^\Gamma\ \{x'.f = z\}\ \texttt{x := x'.f;}\ \{x = z\ \&\&\ x'.f = z\}\ [\,\textbf{modifies}\ x, \textbf{region}\{x'.f\}]}
$$

(1) is $x'.f = z\ \&\&\ x \neq y\ \&\&\ x' \neq y\ \Rightarrow ((\textbf{region}\{x'.f\}, x', y) \cdot\!/\cdot x)$ and
$x' \neq \mathit{null} \Rightarrow \exists z.(x'.f = z)$

(2) is $x' \neq \mathit{null} \Rightarrow \mathit{regRW}(\mathit{efs}(x'.f), x, \textbf{region}\{x'.f\}) \leqslant \mathit{regRW}(\textbf{region}\{x'.f\}, x)$

Figure H.2: The derivation of rule $\texttt{TR}_R[\![\mathit{ACC}_s]\!]$

$$
\dfrac{
\begin{array}{c}
(UPD_u)\ \vdash_u^\Gamma\ [\textbf{reads}\ x, \mathit{efs}(E)]\ \{x \neq \mathit{null}\}\ \texttt{x.f := E;}\ \{x.f = E\}\ [\textbf{region}\{x.f\}] \qquad \textbf{where}\ (1) \\
[\textbf{reads}\ x, \mathit{efs}(E)] \\
\vdash_u^\Gamma\ \{\exists\, z.x.f = z\}\ x.f := E; \{x.f = E\} \\
[\,\textbf{modifies region}\{x.f\}]
\end{array}
}{
\begin{array}{l}
(CONSEQ_u) \\
(\mathit{Subeffect}) \\
(\mathit{SubEff}_u)
\end{array}
\begin{array}{l}
\vdash_u^\Gamma\ \textbf{reads}\ \mathit{readR}(\textbf{reads}\ x, \mathit{efs}(E)) \leqslant \textbf{reads region}\{x.f\} \\
\textbf{where}\ \mathit{readR}(\textbf{reads}\ x, \mathit{efs}(E)) \leqslant \mathit{readR}(\textbf{reads region}\{x.f\})
\end{array}
}
$$

$$
\begin{array}{c}
[\textbf{reads region}\{x.f\}] \\
\vdash_u^\Gamma\ \{\exists\, z.x.f = z\}\ \texttt{x.f := E;}\ \{x.f = E\} \\
[\,\textbf{modifies region}\{x.f\}]
\end{array}
$$

(1) is $x \neq \mathit{null} \Rightarrow \exists\, z.x.f = z$

Figure H.3: The derivation of rule $\texttt{TR}_R[\![\mathit{UPD}_s]\!]$

APPENDIX I: PROOF OF LEMMA 23

Lemma 23:Let (σ, h) be a state, and p_s be an inductive predicate in SSL. Then

$$\mathcal{E}_a[\![\Gamma \vdash p_s(\overline{e}) : \mathbf{bool}]\!](\sigma, h) = \mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![p_s(\overline{e})]\!] : \mathbf{bool}]\!](\sigma, h).$$

Proof. The proof is an inductive case of the proof of Theorem 7. The inductive hypothesis is that for all subassertions a_i, $\mathcal{E}_a[\![\Gamma \vdash a_i : \mathbf{bool}]\!](\sigma, h) = \mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![a_i]\!] : \mathbf{bool}]\!](\sigma, h)$. Let $b_1 \Rightarrow a_1 \cdots b_n \Rightarrow a_n$ be inductive cases for p_s. We prove it as follows.

$\mathcal{E}_a[\![\Gamma \vdash p_s(\overline{e}) : \mathbf{bool}]\!](\sigma, h)$

iff $\langle$by semantics of inductive predicates Eq. (7.4)$\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, \mathcal{E}_a[\![\Gamma \vdash (b_1 \Rightarrow a_1) \wedge \ldots \wedge (b_n \Rightarrow a_n) : \mathbf{bool}]\!](\sigma', h'))$

$(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma)}), h)$

iff $\langle$by semantics of assertions Def. 11$\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_a[\![\Gamma \vdash b_1 \Rightarrow a_1 : \mathbf{bool}]\!](\sigma', h')\ \mathit{and}\ \ldots\ \mathit{and}$

$\mathcal{E}_a[\![\Gamma \vdash b_n \Rightarrow a_n : \mathbf{bool}]\!](\sigma', h')))(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma)}), h)$

iff $\langle$by inductive hypothesis$\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![b_1 \Rightarrow a_1]\!] : \mathbf{bool}]\!](\sigma', h')\ \mathit{and}\ \ldots\ \mathit{and}$

$\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![b_n \Rightarrow a_n]\!] : \mathbf{bool}]\!](\sigma', h')))(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma)}), h)$

iff $\langle$by Lemma 11: $\mathcal{E}_s[\![\Gamma \vdash e : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash \mathtt{TR}[\![e]\!] : T]\!](\sigma)\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![b_1 \Rightarrow a_1]\!] : \mathbf{bool}]\!](\sigma', h')\ \mathit{and}\ \ldots\ \mathit{and}$

$\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![b_n \Rightarrow a_n]\!] : \mathbf{bool}]\!](\sigma', h')))(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}[\![\Gamma \vdash \mathtt{TR}[\![e : T]\!]]\!](\sigma)}), h)$

iff $\langle$by the semantics of assertions (Fig. 3.2)$\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![b_1 \Rightarrow a_1 : \mathbf{bool}]\!]\ \&\&\ \ldots\ \&\&\ \mathtt{TR}[\![b_n \Rightarrow a_n]\!] : \mathbf{bool}]\!](\sigma', h')))$

$(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}[\![\Gamma \vdash \mathtt{TR}[\![e]\!] : T]\!](\sigma)}), h)$

iff $\langle$by the definition of encoding inductive predicates in Fig. 7.1$\rangle$

$(\mathit{fix}\underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\Gamma \vdash \mathtt{TR}[\![\mathit{idf}(p_s)]\!] : \mathbf{bool}]\!](\sigma', h')))$

$(\sigma(\mathit{formals}_s(p_s) \mapsto \overline{\mathcal{E}[\![\Gamma \vdash \mathtt{TR}[\![e : T]\!]]\!](\sigma)}), h)$

iff $\langle$by semantics of recursive predicate.$\rangle$

$\mathcal{E}_p[\![\Gamma \vdash \texttt{TR}[\![p_s(\overline{e})]\!] : \mathbf{bool}]\!](\sigma, h)$

□

LIST OF REFERENCES

[1] P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., Apr. 1989. Revised from the January 1989 version.

[2] A. Banerjee and D. A. Naumann. Local Reasoning for Global Invariants, Part II: Dynamic Boundaries. *Journal of the ACM*, 60(3):19:1–19:73, June 2013.

[3] A. Banerjee, D. A. Naumann, and M. Nikouei. A logical analysis of framing for specifications with pure method calls. *ACM Trans. Prog. Lang. Syst.*, under review. `https://www.cs.stevens.edu/~naumann/publications/lafsp2.pdf`.

[4] A. Banerjee, D. A. Naumann, and S. Rosenberg. Local Reasoning for Global Invariants, Part I: Region Logic. *Journal of the ACM*, 60(3):18:1–18:56, June 2013.

[5] Y. Bao and G. Ernst. A KIV project for defining semantics for intuitionistic separation logic. `http://www.eecs.ucf.edu/~ybao/project/sl-semantics/index.xml`, 2016.

[6] Y. Bao and G. Ernst. A KIV project for proving encoding supported separation logic into unified fine-grained region logic. `http://www.eecs.ucf.edu/~ybao/project/frl-sep-expr/index.xml`, 2016.

[7] Y. Bao, G. T. Leavens, and G. Ernst. Conditional effects in fine-grained region logic. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, FTfJP '15, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Ob-*

jects (FMCO) 2005, Revised Lectures, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, New York, NY, 2006. Springer-Verlag.

[9] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[10] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, New York, NY, 2005. Springer-Verlag.

[11] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction (MPC)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, July 2004.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007.

[14] J. Berdine, C. Calcagno, and P. OHearn. A decidable fragment of separation logic. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer Berlin Heidelberg, 2005.

[15] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever,

editors, *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137, Berlin, 2005. Springer-Verlag.

[16] J. Berdine, C. Calcagno, P. W. Ohearn, and Q. Mary. Symbolic execution with separation logic. In *In APLAS*, pages 52–68. Springer, 2005.

[17] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. *SIGSOFT Softw. Eng. Notes*, 30(5):217–226, Sept. 2005.

[18] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM.

[19] F. Bobot and J.-C. Filliâtre. *Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, chapter Separation Predicates: A Taste of Separation Logic in First-Order Logic, pages 167–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[20] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, Oct. 1995.

[21] J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 87–103, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, Berlin, 2006. Springer-Verlag.

[23] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005.

[24] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. *CONCUR 2011 – Concurrency Theory: 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, chapter Tractable Reasoning in a Fragment of Separation Logic, pages 235–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[25] B. J. Cox. *Object Oriented Programming: an Evolutionary Approach*. Addison-Wesley Publishing Co., Reading, Mass., 1986.

[26] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.

[27] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Verlag, June 2004.

[28] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Dec. 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[29] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[30] G. Ernst, J. Pfhler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: overview and verifythis competition. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2014.

[31] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

[32] R. L. Ford and K. R. M. Leino. Dafny reference manual (draft). https://github.com/Microsoft/dafny/blob/master/Docs/DafnyRef/out/DafnyRef.pdf.

[33] J.-Y. Girard. Linear logic: A survey. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series. Series F : Computer and System Sciences*, pages 63–112. Springer-Verlag, New York, NY, 1993.

[34] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, Sept. 1985.

[35] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *European Conference on Object-Oriented Programming*, pages 451–476. Springer, 2013.

[36] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[37] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[38] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355, 1973.

[39] A. Hobor and J. Villard. The ramifications of sharing in data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 523–536, New York, NY, USA, 2013. ACM.

[40] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[41] B. Jacobs, J. Smans, and F. Piessens. *A Quick Tour of the VeriFast Program Verifier*, pages 304–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[42] C. B. Jones. *Systematic software development using VDM*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

[43] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In E. S. J. Misra, T. Nipkow, editor, *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Berlin, 2006. Springer-Verlag.

[44] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.

[45] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, Dec. 2001. This is an obsolete version.

[46] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.*, 37(4):13:1–13:88, Aug. 2015.

[47] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *TOPLAS*, 37(4):13:1–13:88, Aug. 2015.

[48] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[49] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[50] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, New York, NY, Oct. 1998. ACM.

[51] K. R. M. Leino. Specification and verification of object-oriented software. Lecture notes from Marktoberdorf Internation Summer School, available at `http://research.microsoft.com/en-us/um/people/leino/papers/krml190.pdf`, 2008.

[52] K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://research.microsoft.com/~leino/papers.html`.

[53] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer-Verlag, 2010.

[54] K. R. M. Leino. *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers*, chapter Dafny: An Automatic Program Verifier for Functional Correctness, pages 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[55] K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the Third international conference on Verified software: theories, tools, ex-*

periments, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126, Berlin, 2010. Springer-Verlag.

[56] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516, Berlin, June 2004. Springer-Verlag.

[57] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393, Berlin, Mar. 2009. Springer-Verlag.

[58] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, Sept. 2002.

[59] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5) of *ACM SIGPLAN Notices*, pages 246–257, New York, NY, June 2002. ACM.

[60] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[61] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[62] W. Mostowski and M. Ulbrich. Dynamic dispatch for method contracts through abstract predicates. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 109–116, New York, NY, USA, 2015. ACM.

[63] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2002.

[64] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[65] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, Feb. 2003.

[66] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, Oct. 2006.

[67] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symposium on Logic in Computer Science*, pages 313–323, 2004.

[68] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 365:143–168, 2006. Extended version of [67].

[69] L. Nistor, J. Aldrich, S. Balzer, and H. Mehnert. Object propositions. In *FM 2014: Formal Methods*, pages 497–513. Springer, 2014.

[70] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[71] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2001. Springer-Verlag.

[72] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM.

[73] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):11:1–11:50, Apr. 2009.

[74] M. Parkinson. Class invariants: The end of the road? *Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)*, page 9, 2007.

[75] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *ACM Symposium on Principles of Programming Languages*, pages 247–258, New York, NY, Jan. 2005. ACM.

[76] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In P. Wadler, editor, *ACM Symposium on Principles of Programming Languages*, pages 75–86, New York, NY, Jan. 2008. ACM.

[77] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. The author's Ph.D. dissertation.

[78] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.

[79] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society Press.

[80] S. Rosenberg, A. Banerjee, and D. A. Naumann. Decision procedures for region logic. In *Verification, Model Checking, and Abstract Interpretation*, pages 379–395. Springer, 2012.

[81] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Boston, Mass., 1986.

[82] J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In *Proceedings of the 12th IFIP WG 6.1 International Conference and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'10/FORTE'10, pages 170–185, Berlin, Heidelberg, 2010. Springer-Verlag.

[83] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

[84] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. Automatic verification of java programs with dynamic frames. *Formal Aspects of Computing*, 22(3):423–457, 2010.

[85] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliabiity. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, Jan. 1986.

[86] B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[87] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 402–416, London, UK, UK, 2002. Springer-Verlag.