Unifying Separation Logic and Region Logic to Allow Interoperability

Yuyan Bao, Gary T. Leavens, and Gidon Ernst

Submitted for publication.

Unifying Separation Logic and Region Logic to Allow Interoperability

Yuyan Bao[1], Gary T. Leavens[1], and Gidon Ernst[2]

[1] University of Central Florida, Orlando, FL 32816 USA
[2] Universität Augsburg, D-86135 Augsburg, Germany

Abstract. Framing is important for specification and verification, especially in programs that mutate data structures with shared data, such as DAGs. Both separation logic and region logic are successful approaches to framing, with separation logic providing a concise way to reason about data structures that are disjoint, and region logic providing the ability to reason about framing for shared mutable data. In order to obtain the benefits of both logics for programs with shared mutable data, this paper unifies them into a single logic, which can encode both of them and allows them to interoperate. The new logic thus provides a way to reason about program modules specified in a mix of styles.

Keywords: Separation logic, fine-grained region logic, framing, shared mutable data, formal specification, formal verification, Hoare logic, unified fine-grained region logic (UFRL)

1 Introduction

In Hoare-style reasoning framing is important for verification. A method's frame indirectly describes the locations that the method may not change [16]. Framing allows verification to carry properties past statements such as method calls, since properties true for unchanged locations will remain valid.

Two approaches to framing are more central to the problem addressed in this paper. Region logic (RL) [4,1] uses ideas that are similar to dynamic frames [29,30], allowing region variables, but restricts specifications so that only first-order logic is needed for verification. Separation logic (SL) [52] features primitives (notably separating conjunction) that make specification of disjoint regions easy and concise.

However, SL does not make specification of unrestricted sharing among mutable data structures easy. Framing in SL depends on disjointness (the use of separating conjunction); the frame rule preserves the value of predicates that are separated from assertions in preconditions. Data structures with deep sharing, such as directed acyclic graphs and unrestricted graphs, complicate local reasoning because one has to figure out that how the local changes affect other parts of the program. For example, changes to the left descendants of a dag may affect its right descendants.

In this paper, we limit our study to sequential imperative programs. The goal of our work is to show how to combine the advantages of RL and SL into one logic for reasoning about framing. We consider mutable data structures that are defined by classes, but we do not consider the object-oriented features of subtyping and dynamic dispatch. In our previous work we defined fine-grained region logic (FRL) [7], a logic adapted from RL. It specifies frames for both methods (procedures) and predicates (like JML [32] model methods) as write effects and read effects. Write effects of methods are specified using expressions that denote variables and sets of heap locations, called *regions*. Regions are also used to specify the read effects of predicates, which are the variables and locations that the predicate depends on. The logic has ways to specify disjointness; if the write effects of a method are disjoint from the read effects of a predicate, then the validity of the predicate is preserved by a call to the method. Moreover, inductive data types can be reasoned about using the dynamic frames approach [29,30], i.e., by writing the effect specifications using the values of dynamically-updated ghost fields of type `region`. (A ghost field is a specification-only field that can be manipulated by the specification as the program runs.)

One drawback of specifications written in FRL is that they are not as concise as those in SL, particularly when expressing disjoint data structures, e.g., binary trees or disjoint linked-lists, for which SL is very concise. For example, when specifying a binary tree, in SL one need only specify the abstract values contained in the data structure; the use of points-to predicates and separating conjunction implicitly describes the shape of the data

structure and its disjointness properties. By contrast, when specifying a binary tree in FRL one needs to both specify the abstract values and separately specify the dynamic frame of the data structure, which is less concise than the corresponding SL specification. Another drawback of FRL is that the specification must be written to explicitly update the ghost field(s) that specify its dynamic frame; this increases the annotation burden and is error-prone.

The problem we address in this paper is how to provide a formal technique for combining specifications in both FRL and SL, which would allow a single mechanism to reason about specifications written in both logics or in a mixed combination of these logics, which also supports the specification of framing for shared mutable data. FRL and SL represent different methodologies, and we know of no previous work connecting them formally.

The approach we use to overcome the differences between FRL and SL is to define a semantics for intuitionistic SL in terms of the program's heap and a region, as these are the core concepts in FRL, and we show that this semantics is equivalent to SL's original semantics. Intuitively, the region used to determine the validity of an SL assertion is its semantic footprint, which is the greatest lower bound of the set of heap locations that determine the assertion's validity; to make this possible, we limit our study to a commonly-used subset of SL which has such lower bounds; these are the supported assertions from the work of O'Hearn et al. [47]. We call the resulting subset of SL Supported Separation Logic (SSL). An important result in making this connection is Theorem 37, which says that there is a way to compute semantic footprints syntactically and express them as region expressions in FRL. With those footprint expressions we are able to: encode SSL assertions, define read effects for them, and use FRL's framing judgment to verify programs specified with SSL.

In order to unify FRL and SSL's specification languages, we define Unified Fine-Grained Region Logic (UFRL), which merges region expressions, read and write effects, and separation. UFRL is a generalization of FRL, thus FRL is trivially embedded into UFRL. To show that UFRL also captures the meaning of SSL specifications, we map SSL's axioms and proof rules into those of UFRL, and show that they are derivable in the UFRL proof system.

We use our results to make specifications both more concise and more expressive, by combining the advantages of FRL and SL. For example, when specifying data structures whose parts are disjoint, one can use SL idioms. To specify data structures with shared parts, one can use FRL-style specifications and dynamic frames. When it is convenient, dynamic frames can be computed by footprint functions. Moreover, our results allow specifications written in FRL and SL to interoperate with each other.

1.1 Contributions

This paper provides a way to locally reason about framing for shared mutable data structures in sequential object-based programs. It does this by combining two successful logics for framing: a commonly used subset of SL and a fine-grained variant of region logic, FRL. The combined logic, UFRL, is enriched by features of both SL and FRL: separating conjunction is combined with explicit write and read effects specified by region expressions. Specifications written in these two styles can interoperate with each other as they are both encoded into UFRL. Thus, 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 [23]. Lemmas and theorems that are not formally proved in the paper have been proved in KIV. These machine-checked proofs have been exported and are available online [5,6]. Table 1 on the following page shows the correspondence between the theorems and lemmas defined in the paper and those defined in the KIV projects [5,6].

1.2 Outline

The next section motivates this work with examples. Background on the semantics of heaps in Section 3. Section 4 presents the programming language for which we formalize the programming logic. Section 5 presents the assertion language, semantics of effects, immune and separator. Section 6 presents the program correctness in FRL, i.e., proof axioms and proof rules. Section 7 presents the program correctness in UFRL, i.e., proof axioms and proof rules. Section 8 shows that FRL is an instance of UFRL. Section 9 shows the semantic connection between UFRL and SL. Section 10 and Section 11 encode SSL assertions and specifications into UFRL assertions and specifications, and shows that SSL proofs are preserved by the encoding. Section 12 extends our results with SSL inductive predicates. Section 13 extends the UFRL proof system with separating conjunction. Section 14 presents potential applications of UFRL. Section 15 discusses the idea of encoding magic wand. Section 16 discusses related work. Finally, Section 17 gives conclusions and future work.

in the paper	in the KIV project
Theorem 26	sl-semantics/equivalence/sem-equivalence
Lemma 29	frl-sep-expr-proof/expr-rsl-semantics/rsla-stable
Lemma 30	frl-sep-expr-proof/semantic-footprint-rsl/*
Corollary 39	frl-sep-expr-proof/translation/fpt-a-restrictive
Lemma 35, Lemma 36, Theorem 40	frl-sep-expr-proof/translation/sem-preservation
Theorem 37	frl-sep-expr-proof/translation/sfpt-fpt
Theorem 41	frl-sep-expr-proof/translation/sem-preservation-ex
Lemma 44	frl-sep-expr-proof/effects-rsl/sep-tr-effs
Lemma 45	frl-sep-expr-proof/translation/fv-preservation
Lemma 46	frl-sep-expr-proof/effects-rsl/efs-fv

Fig. 1: The correspondence between the theorems and lemmas defined in the paper and those defined in the KIV projects [5,6].

2   Motivation

This section sketches some examples to illustrate problems with separation logic and fine-grained region logic and our approach to solving them.

2.1   Data Structures with Unrestricted Sharing

Fig. 2 on the next 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.

In SL, the introduction of separating conjunction leads to its frame rule:

$$(FRM_s) \frac{\vdash_s \{a\}\, S\, \{a'\}}{\vdash_s \{a * c\}\, S\, \{a' * c\}} \quad \textbf{where } \mathrm{mods}(S) \cap \mathrm{FV}(c) = \varnothing$$

where $\mathrm{FV}(c)$ returns the set of free variables in c. Local reasoning can be achieved by this frame rule, since it allows a specification to solely describe the partial state that programs use. Other disjoint states are untouched and 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 SL's frame rule cannot be directly used when verifying data structures with unrestricted sharing [25] because of the use of conjunctions, e.g., the definition of the predicate dag in Fig. 2 on the next page. The left and the right descendants of a DAG may not be disjoint. Thus, changes in the left descendants may affect the value of the right descendants.

FRL supports local reasoning by the means of effects. The effects may be variables in stores or regions in heaps. FRL's frame rule uses effects to distinguish what is preserved, shown as follows:

$$(FRM_r) \frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash_r \delta\, frm\, Q}{\vdash_r \{P \,\&\&\, Q\}\, S\, \{P' \,\&\&\, Q\}[\varepsilon]} \quad \textit{where } P \,\&\&\, Q \Rightarrow \delta \cdot /\cdot \varepsilon$$

The formula ε is the *write effect* that denotes the set of locations (variables and regions) that may be modified by S. The formula δ is the *read effect* that denotes the set of locations that the assertion Q replies on. The formula $\delta \cdot /\cdot \varepsilon$ denotes the disjointness of the two sets of 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 locations in S's write effects are disjoint with

the locations that Q depends on. In the conclusion of the frame rule, P is connected with Q by the conjunction that allows one to use the frame rule directly when reasoning with unrestricted data structures.

For example, in Fig. 2, a recursive procedure, `mark`, marks all reachable nodes in a DAG, starting from a given node d. In the body of the procedure, the field `d.mark` is updated to 1 if it is not a *null* reference, and is not already marked, i.e., `d.mark`↦0, then `mark` is recursively invoked on its left sub-DAGs and right sub-DAGs, e.g., `mark(d.l)` and `mark(d.r)`. Its write effect is specified by a helping function `unmarked` that collects un-marked locations of a DAG whose root is d, and returns them through the variable **ret** with type **region**. The body of `unmarked` collects locations of the form **region**$\{d.mark\}$, where d is not *null* and not marked. Then it recursively collects un-marked locations on d's left sub-DAGs and right sub-DAGs. The **decreases** clause specifies a termination condition. It requires that the footprint of the DAG argument becomes strictly smaller each time there is a recursive call. The footprints of a DAG are also bounded so that the sequence of arguments in such recursive calls cannot decrease forever. In our example, the bound is the empty set that is the footprint of `dag(null)`.

```
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(d.l) && dag(d.r)))}

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) then { ret := region{}; }
  else {
    ret := region{};
    if (d.mark = 0) then {
      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 DRAND 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) then {
    d.mark := 1; mark(d.l); mark(d.r);
  }
}
```

Fig. 2: Specification of marking a DAG written in UFRL.

When proving the body of the procedure `mark`, right after the procedure call `mark(d.l)`, one proof obligation is to show that `d.r` is still a DAG, i.e., the structure of the right descendants is preserved. The SL's frame rule cannot be directly used as the left and the right descendants may not be disjoint. But the FRL's frame rule can be applied as long as its side condition satisfied, which is true. We show a proof later.

2.2 Separation

One of the scenarios in which separation has been used is describing non-shared data structures, e.g., acyclic linked-lists and binary trees. For a methodology with explicit frames like FRL, separation is described by set theoretic operations. For example, suppose that we want to prove that the node, **this**, does not share locations with the following node. Proving this involves showing that the two regions that frame a node and its successor are disjoint. A set of locations, R, *frames* an object o if the o's observable behavior only depends on the values in R. For example, **region**$\{n.*\}$ frames a `Node<T>` object n. The first region is **region**$\{$**this**$.*\}$, which frames the object **this**. The second region is **region**$\{$**this**$.$next$.$repr$\}$, which frames the following node. The disjointness of these two regions, specified in the predicate `valid`, is written **region**$\{$**this**$.*\}$!!**this**$.$next$.$repr, in UFRL.

Separation can be concisely expressed by separating conjunction ($*$) in SL. As we show later, UFRL can support separating conjunction either directly or by treating it as a syntactic sugar that is desugared into assertions with region expressions. As an example of the latter approach, the assertion $x.f \mapsto 5 * y.g \mapsto 6$ in SL asserts that the two points-to assertions hold on the two disjoint heaps. It can be encoded into UFRL as follows $x.f = 5 \&\& y.g = 6 \&\& ($**region**$\{x.f\}$!!**region**$\{y.g\})$, since **region**$\{x.f\}$ frames $x.f = 5$ and **region**$\{y.g\}$ frames $y.g = 6$.

With UFRL one can specify the linked-list with an SL-style inductive predicate, as shown in Fig. 3 on the next page. Adopting the convention of VeriFast [27], `?v` and `?vlst` declare (universally-quantified) variables `v` and `vlst` respectively that scope over the entire specification of `append`. Its precondition specifies the separating conjunction `lst(n,[?v]) * lst(`**this**`,?vlst)`, which can be desugared into UFRL as:

   `lst(n,[?v])` $\&\&$ `lst(`**this**`,?vlst)` $\&\&$ (**fpt**`(lst(n,[v]))`!!**fpt**`(lst(`**this**`,vlst)))`.

As can be seen, the use of separating conjunction greatly simplifies the specifications. A node's dynamic frame need not be stored in a ghost fields, but can computed by the built-in **fpt** function. The semantics of **fpt**(a) is a's *semantic footprint*, which is the smallest set of locations on which a depends. (This is defined when a is a supported separation logic (SSL) assertion.) In the example a node `n`'s dynamic frame is the set of locations that stores the list starting with the node `n`, and can be computed by a **fpt** function, e.g., **fpt**`(lst(n,[v]))`. No flexibility in specification is lost, because dynamic frames can still be stored in a region variable, e.g., `repr := `**fpt**`(lst(`**this**`))`, and manipulated as needed. However, in most cases, as in our example, ghost fields and ghost statements can be avoided, as in Fig. 3 on the next page.

2.3 Mixed styles of specifications

In software engineering, modularization allows large projects to be decomposed into smaller components. Another contribution of our work is that UFRL allows one to specify and verify programs from components specified in different logics, i.e., FRL and SSL (SSL being the supported variant of SL that we consider in this paper); as we show later, both of them can be encoded into UFRL. Consider the example in Fig. 4 on page 8. Module `CellS` is specified by SSL: method `setSX` and `getSX` may only access (write or read) a memory location **this**`.x` if it has been requested by a points-to assertion in their preconditions. Adopting the convention of VeriFast [27], the expression **this**`.x`$\mapsto$_ is an abbreviation of $\exists y.$ (**this**`.x`$\mapsto$y); again `?v` in `getSX` declares a (universally-quantified) variable `v` that scopes over the entire specification of `getSX`. Module `CellR` is specified by FRL. Method `addOne` may only write a memory location **region**$\{$**this**$.x\}$, as specified in its frame. According to our results, the specification of `setSX` can be encoded into UFRL as:

$$[\textbf{reads region}\{\textbf{this}.x\}]$$
$$\{\exists\, y.\textbf{this}.x = y\}\texttt{setSX(v)};\{\textbf{this}.x = v\}$$
$$[\textbf{modifies region}\{\textbf{this}.x\}]$$

```
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 */

}
```

Fig. 3: A linked-list example written in UFRL with separating conjunction.

where [**reads region**{**this**.x}] specifies read effects, which is followed by a Hoare triple giving the method's pre- and postconditions, and [**modifies region**{**this**.x}] specifies write effects. The union of these two effects specifies the locations that the program can access. It is also valid to encode the specification of setSX as the following in UFRL:

[∅]{∃ y. **this**.x = y} setSX(v);{**this**.x = v}[**modifies region**{**this**.x}],

as the union is still **region**{**this**.x}. And the specification of addOne(sCell) can be encoded into UFRL as:

$$[\textbf{reads alloc}\!\downarrow]$$
$$\{c \neq null\} \text{ addOne(c : CellS); } \{\textbf{this}.x = c.getSX() + 1\}$$
$$[\textbf{modifies region}\{\textbf{this}.x\}].$$

Here **alloc↓** means all the locations that are contained in the domain of the heap. Since the union of **alloc↓** and [**region**{**this**.x}] is **alloc↓**, the encoded specification allows the program to access the whole heap; that is consistent with the semantics of a Hoare-formula in FRL. Then, because the write effects of addOne separate from the read effects of getSX's postcondition, by using UFRL's frame rule (FRM_u), we can prove that the assertions are true in the following example.

```
var sCell; sCell := new CellS; var rCell; rCell := new CellR;
sCell.setSX(5);  rCell.addOne(sCell);
assert sCell.getSX() = 5; assert rCell.getRX() = 6;
```

The theoretical foundation for this proof technique has been provided by the work of Banerjee et al. [4], from which FRL is adapted.

```
class CellS{                          class CellR{
  var x : int;                          var x : int;

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

  method setSX(v: int)                  method addOne(c : CellS)
    requires this.x↦_;                    requires c ≠ null;
    ensures this.x↦v;                     modifies region{this.x};
  { this.x := v; }                        ensures this.x = c.getSX()+1;
                                        { this.x := c.getSX()+1; }
  method getSX() : int
    requires this.x↦?v;                 method getRX() : int
    ensures this.x↦v * ret = v;           modifies ∅;
  { ret := this.x; }                      ensures ret = this.x;
                                        { ret := this.x; }
}                                     }
```

Fig. 4: The class CellS is specified in the style of separation logic. The class CellR is specified in the style of UFRL.

3 Background

This section provides some background on the semantics of heaps, which are used in later sections.

3.1 Store, Heaps and Regions

Some common semantic functions are defined in this subsection. A program state is a pair of a store and a heap. A store, σ, is a partial function that maps each 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* $(o, f) \in dom(H)$; $H[o, f]$ *is the value that H associates to* (o, f).

H_1 *is extended by* H_2, *written* $H_1 \subseteq H_2$, *means: for all* $(o, f) \in dom(H_1) :: (o, f) \in dom(H_2)$ *and* $(H_1[o, f] = H_2[o, f])$.

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

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

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

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 for all* $(o, f) \in dom(H \restriction R) :: (H \restriction R)[o, f] = H[o, f]$. ∎

4 Programming Language

This section presents the programming language for which we formalize the programming logic.

4.1 Syntax

Fig. 5 defines the syntax of the language. There are two distinguished variable names. One is **this** that is the receiver object; another is **ret** that stores the return value of a method if the method one.

The syntactic category E describes expressions, RE describes region expressions, and S describes statements. A class consists of fields and methods. A field is declared with type integer, a user-defined datatype, or **region**. A

$$
\begin{aligned}
Prog &::= \overline{Class} \ \ S \\
Class &::= \textbf{class} \ \mathtt{C} \ \{ \ \overline{Member} \ \} \\
Member &::= Field \ | \ Method \\
Field &::= \textbf{var} \ f : T; \\
Method &::= \textbf{method} \ m \, (\overline{x : T}) \ [\,\textbf{returns} \ (x':T')\,] \ \{ \ \overline{S} \ \} \\
T &::= \textbf{int} \ | \ \textbf{region} \ | \ C \ | \ C{<}\overline{T}{>} \\
E &::= n \ | \ x \ | \ \textbf{null} \ | \ E_1 \oplus E_2 \\
RE &::= x \ | \ \textbf{region}\{\} \ | \ \textbf{region}\{x.f\} \ | \ \textbf{region}\{x.*\} \ |\textbf{if} \ E \ \textbf{then} \ RE_1 \ \textbf{else} \ RE_2 \\
&\quad | \ \textbf{filter}\{RE, T, f\} \ | \ \textbf{filter}\{RE, T\} \ | \ RE_1 \otimes RE_2 \\
F &::= E \ | \ RE \\
S &::= \textbf{skip}; \ | \ \textbf{var} \ x{:}T; \ | \ x{:=}F; \ | \ x_1{:=}x_2.f; \ | \ x.f{:=}F; \ | \ x{:=}\textbf{new} \ C; \\
&\quad | \textbf{if} \ E \ \textbf{then} \ \{S_1\} \ \textbf{else} \ \{S_2\} \ | \ \textbf{while} \ E \ \{S\} \ | \ S_1 S_2 \\
\oplus &::= = \ | \ + \ | \ - \ | \ * \ | \ \leqslant \ \ldots \\
\otimes &::= + \ | \ - \ | \ *
\end{aligned}
$$

Fig. 5: The syntax of the programming language, where n is a numeral, x is a variable name (or a pseudo-variable, such as **alloc**), and f is a field name. We use $\overline{T}$ to indicate a non-empty sequence of types.

method is declared in a class. In a method implementation, there are local variable declarations, update statements, condition statements, and loop 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. [3] The conditional region expression, **if** E **then** RE_1 **else** 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**$\{RE, T\} = \{o_1.f_1, o_1.f_2\}$. The operators $+$, $-$, and $*$ denote union, difference and intersection respectively. The statement for garbage collection or deallocation are excluded in our statements.

Fig. 6 shows semantic notations that are used in this paper but not formalized here; some are used in examples or are used only at the meta level. The generic mathematical types **seq** and **map** are adopted from Dafny [35,36].

Notation	Meaning
alloc	the domain of the heap
this $: C$	assert that the variable **this** has type C
seq$<$T$>$	sequence 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 starting at index i, if $0 \leqslant i < |s|$
$s[i..j]$	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 $i < |s|$ and $0 \leqslant i \leqslant j \leqslant |s|$
$s_1 + s_2$	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 domain of the map m
$k \notin m$	test whether the key k is not in the domain of the map m
$m[k := v]$	generate a new map that adds k to the domain of map m and associates the key k with the value v
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.

Fig. 6: Some features that are not formalized in this paper.

For simplicity, we do not formalize functions and pure methods, but rely on the formalization in Banerjee et al.'s work [2]. We use Γ for type environments, which map variables to types:

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

For brevity we omit a boolean type: the guards for if-statements and while-statements has type **int**, zero value is interpreted as false and non-zero value is interpreted as true. The auxiliary function *fields* takes a class name and returns a list of its declared field names and their types. The predicate *isClass* returns true just when T is a reference type in the program. The typing rules for expressions is defined in Fig. 7, for region expressions are defined in Fig. 8 on the next page, and for statements are defined in Fig. 9 on the next page.

---

[3] 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.

$$\Gamma \vdash x : T \text{ where } (\Gamma\, x) = T \qquad \Gamma \vdash \mathbf{null} : T \text{ where } isClass(T) \qquad \Gamma \vdash n : \mathbf{int}$$

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

Fig. 7: Typing rules for expressions.

$$\Gamma \vdash \mathbf{region}\{\} : \mathbf{region} \qquad \frac{\Gamma \vdash x : T \qquad isClass(T) \qquad (f : T') \in fields(T)}{\Gamma \vdash \mathbf{region}\{x.f\} : \mathbf{region}} \qquad \frac{\Gamma \vdash x : T \qquad isClass(T)}{\Gamma \vdash \mathbf{region}\{x.*\} : \mathbf{region}}$$

$$\frac{\Gamma \vdash E : \mathbf{int} \qquad \Gamma \vdash RE_1 : \mathbf{region} \qquad \Gamma \vdash RE_2 : \mathbf{region}}{\Gamma \vdash \mathbf{if}\, E\, \mathbf{then}\, RE_1\, \mathbf{else}\, RE_2 : \mathbf{region}}$$

$$\frac{\Gamma \vdash RE : \mathbf{region} \qquad isClass(T) \qquad (f : T) \in fields(T)}{\Gamma \vdash \mathbf{filter}\{RE, T, f\} : \mathbf{region}} \qquad \frac{\Gamma \vdash RE : \mathbf{region} \qquad isClass(T)}{\Gamma \vdash \mathbf{filter}\{RE, T\} : \mathbf{region}}$$

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

Fig. 8: Typing rules for region expressions.

$$\Gamma \vdash \mathbf{skip}; : ok(\Gamma) \qquad \Gamma \vdash \mathbf{var}\, x : T; : ok(\Gamma, x : T) \qquad \frac{\Gamma \vdash x : T,\ \Gamma \vdash F : T}{\Gamma \vdash x := F; : ok(\Gamma)}$$

$$\frac{\Gamma \vdash x_1 : T,\ \Gamma \vdash x_2 : T' \qquad (f : T) \in fields(T')}{\Gamma \vdash x_1 := x_2.f; : ok(\Gamma)} \qquad \frac{\Gamma \vdash x : T' \qquad (f : T) \in fields(T') \qquad \Gamma \vdash F : T}{\Gamma \vdash x.f := F; : ok(\Gamma)}$$

$$\frac{\Gamma \vdash x : C,\ \Gamma \vdash \mathbf{new}\, C : C}{\Gamma \vdash x := \mathbf{new}\, C; : ok(\Gamma)} \qquad \frac{\Gamma \vdash E : \mathbf{int},\ \Gamma \vdash S_1 : ok(\Gamma'),\ \Gamma \vdash S_2 : ok(\Gamma'')}{\Gamma \vdash \mathbf{if}\, E\, \mathbf{then}\, \{S_1\}\, \mathbf{else}\, \{S_2\} : ok(\Gamma)} \qquad \frac{\Gamma \vdash E : \mathbf{int},\ \Gamma \vdash S : ok(\Gamma')}{\Gamma \vdash \mathbf{while}\, E\, \{S\} : ok(\Gamma)}$$

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

Fig. 9: Typing rules for statements.

4.2 Semantics

Fig. 10 shows the semantics of expressions and region expressions. The set *Loc* represents locations in a heap. Each field's location is represented by a pair of an allocated reference and a field name. The semantics uses a store σ, which is a partial function that maps a variable to its value, and a heap H, which maps from an object reference and a field name to that location's value. A *Value* is either a Boolean, an object reference (which may be $null$), an integer or a set of locations: $Value = Boolean + Object + Int + PowerSet(Loc)$.

Pure expressions evaluate to Values; thus the semantics of $E_1 = E_2$ and $E_1 \neq E_2$ have no need to check for errors. Region expressions evaluate to regions, i.e., sets of locations, and may also fault. The pair $(\textbf{null}, f)$ is not allowed in the regions of our language's semantics. The value of $\textbf{region}\{x.f\}$ is an empty set in a state (σ, H), where $\sigma(x) = null$.

The special variable, **alloc**, is treated in the semantics as a variable that contains the domain of the heap; it is only updated whenever the storage is allocated (by the **new** statement). The program semantics maintains the invariant: $\sigma(\textbf{alloc}) = dom(H)$. So the value of **alloc** can be looked up in the store.

A Γ-state contains a store and a heap: Γ-*State=Store×Heap*, where $dom(\sigma) = dom(\Gamma)$, and for all $x : T \in \Gamma$, $\sigma(x)$ agrees with the type T. *Type* is a function that takes a reference and a store and returns the type of the reference. The semantics of statements is standard.

$$\mathcal{E} : F \rightarrow Store \rightarrow Value$$
$$\mathcal{E}[\![x]\!](\sigma) = \sigma(x) \qquad \mathcal{E}[\![\textbf{null}]\!](\sigma) = null \qquad \mathcal{E}[\![n]\!](\sigma) = \mathcal{N}[\![n]\!]$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!](\sigma) = \mathcal{E}[\![E_1]\!](\sigma)\, \mathcal{MO}[\![\oplus]\!]\, \mathcal{E}[\![E_2]\!](\sigma)$$

$$\mathcal{E}[\![r]\!](\sigma) = \sigma(r)$$
$$\mathcal{E}[\![\textbf{region}\{\}]\!](\sigma) = \varnothing$$
$$\mathcal{E}[\![\textbf{region}\{x.f\}]\!](\sigma) = \textbf{if } \sigma(x) = null \textbf{ then } \varnothing \textbf{ else } \{(\sigma(x), f)\}$$
$$\mathcal{E}[\![\textbf{region}\{x.*\}]\!](\sigma) = \textbf{if } \sigma(x) = null \textbf{ then } \varnothing$$
$$\textbf{else } \{(o, f) \mid o = \sigma(x) \text{ and } T = Type(o, \sigma) \text{ and } (f : T) \in fields(T)\}$$
$$\mathcal{E}[\![\textbf{if } E \textbf{ then } RE_1 \textbf{ else } RE_2]\!](\sigma) = \textbf{if } \mathcal{E}[\![E]\!](\sigma) \textbf{ then } \mathcal{E}[\![RE_1]\!](\sigma) \textbf{ else } \mathcal{E}[\![RE_2]\!](\sigma)$$
$$\mathcal{E}[\![\textbf{filter}\{RE, T\}]\!](\sigma) = \{(o, f) \mid (o, f) \in \mathcal{E}[\![RE]\!](\sigma) \wedge Type(o, \sigma) = T\}$$
$$\mathcal{E}[\![\textbf{filter}\{RE, T, f\}]\!](\sigma) = \{(o, f') \mid (o, f') \in \mathcal{E}[\![RE]\!](\sigma) \wedge f' = f \wedge Type(o, \sigma) = T\}$$
$$\mathcal{E}[\![RE_1 \otimes RE_2]\!](\sigma) = \mathcal{E}[\![RE]\!](\sigma)\, \mathcal{MO}[\![\otimes]\!]\, \mathcal{E}[\![RE]\!](\sigma)$$

Fig. 10: Semantics of properly typed expressions. $\mathcal{N}$ is the standard meaning function for numeric literals. The function MO gives the semantics of operators.

Assume a semantic function, $\mathcal{MS}$, for a statement that relates an input state to possible output states, or an error state, $\bot$. An error happens when statements attempt to access of a location not in the domain of the heap. We consider the form **skip**; S to be identical to S. We assume a function $Extend(\sigma, x, v)$ that extends σ to map x to value v if $x \notin dom(\sigma)$. The semantics of statements are defined in Fig. 11 on the next page.

5 Assertion Language

In this section, an assertion language is formalized.

5.1 Syntax and semantics of assertions

The syntax of assertions is shown in Fig. 12 on the next page. We call the first three *atomic assertions*. Quantification is restricted in the syntax. Quantified variables may denote an **int**, or a location drawn from a region. The typing rules for assertions are in Fig. 13 on page 14. The semantics of assertions is shown in Fig. 14 on page 14.

$\mathcal{MS} : S \to State \to State \cup \{\bot, err\}$

$\mathcal{MS}[\![\mathbf{skip};]\!](\sigma, H) = (\sigma, H)$

$\mathcal{MS}[\![\mathbf{var}\ x : T;]\!](\sigma, H) = \mathbf{if}\ x \notin dom(\sigma)\ \mathbf{then}\ (\mathrm{Extend}(\sigma, x, default(T)), H)\ \mathbf{else}\ err$

$\mathcal{MS}[\![x := F;]\!](\sigma, H) = (\sigma[x \mapsto \mathcal{E}[\![F]\!](\sigma)], H)$

$\mathcal{MS}[\![x.f := F;]\!](\sigma, H) = \mathbf{if}\ \sigma(x) \neq null\ \mathbf{then}\ (\sigma, H[(\sigma(x), f) \mapsto \mathcal{E}[\![F]\!](\sigma)])\ \mathbf{else}\ err$

$\mathcal{MS}[\![x_1 := x_2.f;]\!](\sigma, H) = \mathbf{if}\ \sigma(x_2) \neq null\ \mathbf{then}\ (\sigma[x_1 \mapsto H[(\sigma(x_2), f)]], H)\ \mathbf{else}\ err$

$\mathcal{MS}[\![x := \mathbf{new}\ C;]\!](\sigma, H) =$
$\quad \mathbf{let}\ (l, H') = allocate(C, H)\ \mathbf{in}$
$\quad\quad \mathbf{let}\ (f_1 : T_1, \dots, f_n : T_n) = fields(C, CT)\ \mathbf{in}$
$\quad\quad\quad \mathbf{let}\ \sigma' = \sigma[x \mapsto l][\mathbf{alloc} \mapsto dom(H')]\ \mathbf{in}$
$\quad\quad\quad\quad (\sigma', H'[(\sigma'(x), f_1) \mapsto default(T_1), \dots, (\sigma'(x), f_n) \mapsto default(T_n)])$

$\mathcal{MS}[\![\mathbf{if}\ E\ \mathbf{then}\{S_1\}\mathbf{else}\{S_2\};]\!](\sigma, H) = \mathbf{if}\ \mathcal{E}[\![E]\!](\sigma)\ \mathbf{then}\ \mathcal{MS}[\![S_1]\!](\sigma, H)\ \mathbf{else}\ \mathcal{MS}[\![S_2]\!](\sigma, H)$

$\mathcal{MS}[\![\mathbf{while}\ E\ \{S\};]\!](\sigma, H) =$
$\quad fix(\lambda g . \lambda s . \mathbf{let}\ v = \mathcal{E}[\![E]\!](\sigma)\ \mathbf{in}$
$\quad\quad \mathbf{if}\ v \neq 0\ \mathbf{then}\ \mathbf{let}\ s' = \mathcal{MS}[\![S]\!](s)\ \mathbf{in}\ g \circ s'$
$\quad\quad \mathbf{else}\ \mathbf{if}\ v = 0\ \mathbf{then}\ s\ \mathbf{else}\ err)(\sigma, H)$

$\mathcal{MS}[\![S_1 S_2]\!](\sigma, H) = \mathbf{let}\ s' = \mathcal{MS}[\![S_1]\!](\sigma, H)\ \mathbf{in}\ \mathbf{if}\ s' \neq err\ \mathbf{then}\ \mathcal{MS}[\![S_2]\!](s')\ \mathbf{else}\ err$

Fig. 11: The semantics of statements. The *allocate* function takes a class name and a heap, and returns a location and a new heap.

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*.

$P ::= E_1 = E_2 \mid x.f = E \mid RE_1 \leqslant RE_2 \mid RE_1\,!\,!\,RE_2 \mid \neg P \mid P_1\ \&\&P_2 \mid P_1 \mid\mid P_2 \mid \forall x{:}\mathbf{int}{::}P$
$\quad\mid \forall x{:}T{:}\mathbf{region}\{x.f\} \leqslant RE{:}P \mid \exists x{:}\mathbf{int}{::}P \mid \exists x{:}T{:}\mathbf{region}\{x.f\} \leqslant RE{:}P$

Fig. 12: The syntax of assertions

5.2 EFFECTS

FRL uses effects to specify frame conditions and to frame formulas. The grammar for effects is given in Fig. 15 on page 15. 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. We introduce a conditional effect: **if** E **then** ε_1 **else** ε_2; it denotes that if $E \neq 0$, the effect is ε_1, otherwise the effect is ε_2.

The latter five forms are called atomic effects. We omit **modifies** and **reads** when the context is obvious. For example, we write **if** E **then** RE **else** x instead of **if** E **then modifies** RE **else modifies** x. And the effect, **if** E **then** ε, is an abbreviation of **if** E **then** ε **else** $\varnothing$.

Effects must be well-formed (wf) for the 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 type environment, and δ be an effect. The effect δ is* well-formed *in Γ if,*

1. *for all $(M\ x) \in \delta :: x \in dom(\Gamma)$,*

$$\frac{\Gamma \vdash E_1 : T, \quad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 = E_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash x : T', \quad (f : T) \in \mathit{fields}(T'), \quad \Gamma \vdash E : T}{\Gamma \vdash x.f = E : \mathbf{bool}}$$

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

$$\frac{\Gamma \vdash P_1 : \mathbf{bool}, \quad \Gamma \vdash P_2 : \mathbf{bool}}{\Gamma \vdash P_1 \,\&\&\, P_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash P_1 : \mathbf{bool}, \quad \Gamma \vdash P_2 : \mathbf{bool}}{\Gamma \vdash P_1 \,||\, P_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash P : \mathbf{bool}}{\Gamma \vdash \neg P : \mathbf{bool}}$$

$$\frac{\Gamma, x : \mathbf{int} \vdash P : \mathbf{bool}}{\Gamma \vdash \forall x : \mathbf{int} :: P : \mathbf{bool}} \qquad \frac{\Gamma \vdash RE : \mathbf{region}, \quad \Gamma, x : T \vdash P : \mathbf{bool}}{\Gamma \vdash \forall x : T : \mathbf{region}\{x, f\} \leqslant RE : P : \mathbf{bool}} \qquad \frac{\Gamma, x : \mathbf{int} \vdash P : \mathbf{bool}}{\Gamma \vdash \exists x : \mathbf{int} :: P : \mathbf{bool}}$$

$$\frac{\Gamma \vdash RE : \mathbf{region}, \quad \Gamma, x : T \vdash P : \mathbf{bool}}{\Gamma \vdash \exists x : T : \mathbf{region}\{x, f\} \leqslant RE : P : \mathbf{bool}}$$

Fig. 13: Typing rules for assertions.

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

Fig. 14: Semantics of assertions. Note that assertions with errors are false, so this is two-valued logic.

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

*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.[4] We use a subeffect rule defined in Fig. 17 on page 16 to reason about such cases; it encodes the standard properties of sets.

To streamline explanations, we define the following functions on effects in Fig. 16.

– *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*(**modifies region**$\{x.f\}$, **fresh region**$\{y.*\}$) = **region**$\{y.*\}$.

---

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

$$\varepsilon, \delta \ ::= \ \varnothing \ | \ \varepsilon_1, \varepsilon_2 \ | \ \texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2 \ | \ \texttt{reads } RE \ | \ \texttt{reads } x \ | \ \texttt{modifies } RE$$
$$| \ \texttt{modifies } x \ | \ \texttt{fresh}(RE)$$

Fig. 15: The grammar of effects.

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

$writeR : \varepsilon \rightarrow RE$ $readR : \varepsilon \rightarrow RE$
$writeR(\varnothing) = \varnothing$ $readR(\varnothing) = \varnothing$
$writeR(\varepsilon_1, \varepsilon_2) = writeR(\varepsilon_1) + writeR(\varepsilon_2)$ $readR(\varepsilon_1, \varepsilon_2) = readR(\varepsilon_1) + readR(\varepsilon_2)$
$writeR(\texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2) =$ $readR(\texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2) =$
  $\texttt{if } E \texttt{ then } writeR(\varepsilon_1) \texttt{ else } writeR(\varepsilon_2)$ $\texttt{if } E \texttt{ then } readR(\varepsilon_1) \texttt{ else } readR(\varepsilon_2)$
$writeR(\texttt{modifies } RE) = RE$ $readR(\texttt{modifies } RE) = \texttt{region}\{\}$
$writeR(\texttt{reads } RE) = \texttt{region}\{\}$ $readR(\texttt{reads } RE) = RE$
$writeR(\texttt{reads } x) = \texttt{region}\{\}$ $readR(\texttt{reads } x) = \texttt{region}\{\}$
$writeR(\texttt{modifies } x) = \texttt{region}\{\}$ $readR(\texttt{modifies } x) = \texttt{region}\{\}$
$writeR(\texttt{fresh } RE) = \texttt{region}\{\}$ $readR(\texttt{fresh } RE) = \texttt{region}\{\}$

$freshR : \varepsilon \rightarrow RE$ $readVar : \varepsilon \rightarrow var$
$freshR(\varnothing) = \varnothing$ $readVar(\varnothing) = \varnothing$
$freshR(\varepsilon_1, \varepsilon_2) = freshR(\varepsilon_1) + freshR(\varepsilon_2)$ $readVar(\varepsilon_1, \varepsilon_2) = readVar(\varepsilon_1) \cup readVar(\varepsilon_2)$
$freshR(\texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2) =$ $readVar(\texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2) =$
  $\texttt{if } E \texttt{ then } freshR(\varepsilon_1) \texttt{ else } freshR(\varepsilon_2)$ $\texttt{if } E \texttt{ then } readVar(\varepsilon_1) \texttt{ else } readVar(\varepsilon_2)$
$freshR(\texttt{modifies } RE) = \texttt{region}\{\}$ $readVar(\texttt{modifies } RE) = \varnothing$
$freshR(\texttt{reads } RE) = \texttt{region}\{\}$ $readVar(\texttt{reads } RE) = \varnothing$
$freshR(\texttt{reads } x) = \texttt{region}\{\}$ $readVar(\texttt{reads } x) = \{x\}$
$freshR(\texttt{modifies } x) = \texttt{region}\{\}$ $readVar(\texttt{modifies } x) = \varnothing$
$freshR(\texttt{fresh } RE) = RE$ $readVar(\texttt{fresh } RE) = \varnothing$

$rwR : \varepsilon \rightarrow RE$
$rwR(\varnothing) = \varnothing$ $rwR(\varepsilon_1, \varepsilon_2) = rwR(\varepsilon_1) + rwR(\varepsilon_2)$
$rwR(\texttt{if } E \texttt{ then } \varepsilon_1 \texttt{ else } \varepsilon_2) = \texttt{if } E \texttt{ then } rwR(\varepsilon_1) \texttt{ else } rwR(\varepsilon_2)$
$rwR(\texttt{modifies } RE) = RE$ $rwR(\texttt{reads } RE) = RE$
$rwR(\texttt{reads } x) = \texttt{region}\{\}$ $rwR(\texttt{modifies } x) = \texttt{region}\{\}$
$rwR(\texttt{fresh } RE) = \texttt{region}\{\}$

Fig. 16: The definitions of the functions on effects.

Definition 3 (Changes allowed by write and freshness effects)**.** *Let ε be effects in Γ, and (σ, h) and (σ', h') be Γ'-states for some $\Gamma' \supseteq \Gamma$. ε allows change from (σ, h) to (σ', h'), written $(\sigma, h) \rightarrow (\sigma', h') \models \varepsilon$ if and only if:*

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

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

$$\frac{P \vdash \varepsilon_1 \leqslant \varepsilon_2 \quad P \vdash \varepsilon_2 \leqslant \varepsilon_3}{P \vdash \varepsilon_1 \leqslant \varepsilon_3} \qquad \frac{P' \Rightarrow P \quad P \vdash \varepsilon_1 \leqslant \varepsilon_2}{P' \vdash \varepsilon_1 \leqslant \varepsilon_2} \qquad \frac{P \vdash \varepsilon_1 \leqslant \varepsilon_2}{P \vdash \varepsilon_1, \varepsilon \leqslant \varepsilon_2, \varepsilon}$$

$$\vdash \mathbf{modifies}\, RE_1, RE_2 \;\overset{\scriptstyle\leqslant}{}\; \mathbf{modifies}\, RE_1 + RE_2 \qquad \vdash \mathbf{reads}\, RE_1, RE_2 \;\overset{\scriptstyle\leqslant}{}\; \mathbf{reads}\, RE_1 + RE_2$$

$$\vdash \mathbf{modifies}\, \mathbf{filter}\{RE, T, f\} \leqslant \mathbf{modifies}\, RE \qquad \vdash \mathbf{modifies}\, \mathbf{filter}\{RE, T\} \leqslant \mathbf{modifies}\, RE$$

$$RE_1 \leqslant RE_2 \vdash \mathbf{modifies}\, RE_1 \leqslant \mathbf{modifies}\, RE_2 \qquad RE_1 \leqslant RE_2 \vdash \mathbf{reads}\, RE_1 \leqslant \mathbf{reads}\, RE_2$$

$$\vdash \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2 \leqslant \varepsilon_1, \varepsilon_2 \qquad \frac{\begin{array}{cc} P \&\& E_1 \neq 0 \&\& E_2 \neq 0 \vdash \varepsilon_1 \leqslant \varepsilon_3 & P \&\& E_1 = 0 \&\& E_2 \neq 0 \vdash \varepsilon_2 \leqslant \varepsilon_3 \\ P \&\& E_1 \neq 0 \&\& E_2 = 0 \vdash \varepsilon_1 \leqslant \varepsilon_4 & P \&\& E_1 = 0 \&\& E_2 = 0 \vdash \varepsilon_2 \leqslant \varepsilon_4 \end{array}}{P \vdash \mathbf{if}\, E_1\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2 \leqslant \mathbf{if}\, E_2\, \mathbf{then}\, \varepsilon_3\, \mathbf{else}\, \varepsilon_4}$$

Fig. 17: Subeffect rules.

5.3   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 $\mathbf{reads}\, R'\, frm\, P$. If $R' \leqslant \bar{R}$, i.e., $R' \,!\,! R$, then P's validity is preserved after m is called. We use $efs(-)$ in Fig. 18 to define R' for expressions, region expressions, and atomic assertions in a given state. The frame judgment, $P \vdash^{\Gamma} \delta\, frm\, Q$, means that in the type context Γ, δ contains the locations that are needed to evaluate Q in a Γ-state that satisfies P. Note that we use δ to denote read effects and ε to denote write effects, and Γ is omitted when the type context is the same in the judgment. Fig. 19 on the next page shows the judgment for non-atomic assertions.

$$
\begin{aligned}
efs(x) &= \mathbf{reads}\, x \\
efs(n) &= \varnothing \\
efs(null) &= \varnothing \\
efs(E_1 \oplus E_2) &= efs(E_1), efs(E_2) \\
efs(\mathbf{region}\{\}) &= \varnothing \\
efs(\mathbf{region}\{x.f\}) &= \mathbf{reads}\, x \\
efs(\mathbf{region}\{x.*\}) &= \mathbf{reads}\, x \\
efs(\mathbf{if}\, E\, \mathbf{then}\, RE_1\, \mathbf{else}\, RE_2) &= efs(E) + \mathbf{if}\, E\, \mathbf{then}\, efs(RE_1)\, \mathbf{else}\, efs(RE_2) \\
efs(\mathbf{filter}\{RE, f\}) &= efs(RE) \\
efs(\mathbf{filter}\{RE, T, f\}) &= efs(RE) \\
efs(RE_1 \otimes RE_2) &= efs(RE_1), efs(RE_2) \\
efs(E_1 = E_2) &= efs(E_1), efs(E_2) \\
efs(x.f = E) &= \mathbf{reads}\, x, \mathbf{region}\{x.f\}, efs(E) \\
efs(RE_1 \leqslant RE_2) &= efs(RE_1), efs(RE_2) \\
efs(RE_1 \,!\,! RE_2) &= efs(RE_1), efs(RE_2)
\end{aligned}
$$

Fig. 18: Read effects of expressions, region expressions and atomic assertions.

Definition 4 says that if two states agree on a read effect, δ, then the values of the expressions that depend on δ are identical.

FRMFTPT
$$\frac{P \text{ is atomic}}{true \vdash efs(P)\,frm\,P}$$

FRMSUB
$$\frac{R \vdash \delta_1\,frm\,Q \qquad Q \vdash \delta_1 \leqslant \delta_2}{P \vdash \delta_2\,frm\,Q} \text{ where } P \Rightarrow R$$

FRMCONJ
$$\frac{P \vdash \delta\,frm\,Q_1 \qquad P\&\&Q_1 \vdash \delta\,frm\,Q_2}{P \vdash \delta\,frm\,Q_1\&\&Q_2}$$

FRMDISJ
$$\frac{P \vdash \delta\,frm\,Q_1 \qquad P \vdash \delta\,frm\,Q_2}{P \vdash \delta\,frm\,Q_1||Q_2}$$

FRMFTPTNEG
$$\frac{P \text{ is atomic}}{true \vdash efs(P)\,frm\,\neg P}$$

FRM∀₁
$$\frac{P \vdash^{\Gamma,x:\mathbf{int}} \delta,\mathbf{reads}\,x\,frm\,Q}{P \vdash^{\Gamma} \delta\,frm\,\forall\,x:\mathbf{int}::Q}$$

FRM∀₂
$$\frac{P \vdash \mathbf{reads}\,efs(RE) \leqslant \delta \qquad P\&\&\mathbf{region}\{x.f\} \leqslant RE \vdash^{\Gamma,x:T} (\varepsilon,\,x)\,frm\,Q}{P \vdash \delta\,frm\,\forall\,x:T:\mathbf{region}\{x.f\} \leqslant RE:Q}$$

FRM∃₁
$$\frac{P \vdash^{\Gamma,x:\mathbf{int}} \delta,\mathbf{reads}\,x\,frm\,Q}{P \vdash^{\Gamma} \delta\,frm\,\exists\,x:\mathbf{int}::Q}$$

FRM∃₂
$$\frac{P \vdash \mathbf{reads}\,efs(RE) \leqslant \delta \qquad P\&\&\mathbf{region}\{x.f\} \leqslant RE \vdash^{\Gamma,x:T} (\delta,x)\,frm\,Q}{P \vdash \delta\,frm\,\exists\,x:T:\mathbf{region}\{x.f\} \leqslant RE:Q}$$

Fig. 19: Rules for the framing judgment. Γ is omitted when it is the same in the judgment.

$$\delta \,\dot{/}\, \varnothing = true$$
$$\varnothing \,\dot{/}\, \varepsilon = true$$
$$\mathbf{reads}\,y \,\dot{/}\, \mathbf{modifies}\,x = y \not\equiv x$$
$$\mathbf{reads}\,y \,\dot{/}\, \mathbf{modifies}\,RE = true$$
$$\mathbf{reads}\,RE_1 \,\dot{/}\, \mathbf{modifies}\,x = true$$
$$\mathbf{reads}\,RE_1 \,\dot{/}\, \mathbf{modifies}\,RE_2 = RE_1\,!\,!\,RE_2$$
$$\delta \,\dot{/}\, (\varepsilon,\varepsilon') = (\delta \,\dot{/}\, \varepsilon)\&\&(\delta \,\dot{/}\, \varepsilon')$$
$$(\delta,\delta') \,\dot{/}\, \varepsilon = (\delta \,\dot{/}\, \varepsilon)\&\&(\delta' \,\dot{/}\, \varepsilon)$$
$$\delta \,\dot{/}\, (\mathbf{if}\,E\,\mathbf{then}\,\varepsilon_1\,\mathbf{else}\,\varepsilon_2) = \mathbf{if}\,E\,\mathbf{then}\,(\delta \,\dot{/}\, \varepsilon_1)\,\mathbf{else}\,(\delta \,\dot{/}\, \varepsilon_2)$$
$$(\mathbf{if}\,E\,\mathbf{then}\,\delta_1\,\mathbf{else}\,\delta_2) \,\dot{/}\, \varepsilon = \mathbf{if}\,E\,\mathbf{then}\,(\delta_1 \,\dot{/}\, \varepsilon)\,\mathbf{else}\,(\delta_2 \,\dot{/}\, \varepsilon)$$

Fig. 20: The definition of separator. In this figure δ is a read effect and ε is a write effect.

Definition 4 (Agreement on read effects). *Let δ be an effect that is type-checked in Γ. Let $\Gamma' \supseteq \Gamma$ and $\Gamma'' \supseteq \Gamma$. Let (σ',h') and (σ'',h'') be a Γ'-state and a Γ''-state respectively. Then (σ',h') and (σ'',h'') agree on δ, $(\sigma',h') \stackrel{\delta}{\equiv} (\sigma'',h'')$, if and only if:*

1. *for all $\mathbf{reads}\,x \in \delta :: \sigma'(x) = \sigma''(x)$*
2. *for all $\mathbf{reads}\,\mathbf{region}\{x.f\} \in \delta: o = \sigma(x)$ and $o = \sigma''(x)$ and $f \in dom(fields(Type(o),CT)) : H'[o,f] = H''[o,f]$.*

Agreement is used to define when a read effect frames an assertion, under some condition, in the following definition. For example, using the notation described in the following definition, $x = y \vdash^{\Gamma} (\mathbf{reads}\,y,\mathbf{region}\{x.f\})\,frm\,(x.f = 5)$ is valid.

Definition 5 (Frame Validity). *$P \vdash^{\Gamma} \delta\,frm\,Q$ is valid, written $P \models^{\Gamma} \delta\,frm\,Q$, if and only if for all Γ-states (σ,h), (σ',h'), if $(\sigma,h) \stackrel{\delta}{\equiv} (\sigma',h')$ and $\sigma,h \models^{\Gamma} P\&\&Q$, then $\sigma',h' \models^{\Gamma} Q$.*

Lemma 6 (Frame Soundness of Expressions). *Let (σ,h) and (σ',h') be arbitrary Γ-states. Let E be an expression. If $(\sigma,h) \stackrel{efs(E)}{\equiv} (\sigma',h')$, then $\mathcal{E}[\![E]\!](\sigma) = \mathcal{E}[\![E]\!](\sigma')$.*

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

Lemma 7 (Frame Soundness of Region expressions). *Let (σ, h) and (σ', h') be arbitrary Γ-states. Let RE be a region expression. If $(\sigma, h) \overset{efs(RE)}{\equiv} (\sigma', h')$, then $\mathcal{E}[\![RE]\!](\sigma) = \mathcal{E}[\![RE]\!](\sigma')$*

Proof. The proof is straightforward by structural induction on atomic region expressions.   □

Lemma 8 (Frame Soundness of Assertions). *Every derivable framing judgment is valid.*

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

5.4   Separator and Immune

We use $\cdot/\cdot$ to define the disjointness on effects in Fig. 20 on the previous page. $\delta \cdot/\cdot \varepsilon$ means that the read effects in δ are disjoint with the write effects in ε. We treat **reads** δ, where δ is not a conditional effect, 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**$\{y.f\}$ $\cdot/\cdot$ **modifies region**$\{\}$ by the rule ConMask introduced in the next section.

Lemma 9. *Let RE_1 and RE_2 be two regions. Let (σ, h) be a state. If $\sigma, h \models^\Gamma RE_1\ !\ !RE_2$, then* **reads** $RE_1 \cdot/\cdot$ **modifies** RE_2 *and* **reads** $RE_2 \cdot/\cdot$ **modifies** RE_1.

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

Lemma 10 (Separator Agreement). *Let ε and δ be effects that are type-checked in Γ. Let $\Gamma' \supseteq \Gamma$. Let (σ, h) and (σ', h') be Γ' states, such that $(\sigma', h') = \mathcal{MS}[\![S]\!](\sigma, h)$. Let ε be the write effect of executing S, and $(\sigma, h) \models^\Gamma \delta \cdot/\cdot \varepsilon$, then $(\sigma, h) \overset{\delta}{\equiv} (\sigma', h')$.*

Proof. According to Definition 4, there are two cases.

1. Let **reads** $x \in \delta$ be arbitrary. Since $(\sigma, H) \models^\Gamma \delta \cdot/\cdot \varepsilon$, **modifies** $x \notin \varepsilon$. So we have $\sigma(x) = \sigma'(x)$.
2. Let **reads** $RE \in \delta$ be arbitrary. Since $(\sigma, H) \models^\Gamma \delta \cdot/\cdot \varepsilon$, for all **modifies** $RE' \in \varepsilon$, we have $RE\ !\ !\ RE'$. So we have for all $(o.f) \in \mathcal{E}[\![RE]\!](\sigma)$, we have $H[o, f] = H'[o, f]$

□

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** $(x, \mathbf{region}\{x, f\})$, 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 post-condition 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 15 on page 25 .

Definition 11 (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 $\delta :: RE$ 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**$\{x.f\}$. We now 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. We can calculate a proof of this as follows.

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

iff   ⟨by the definition of Immune (Def. 11 on the previous page)⟩

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

iff   ⟨by RE is **region**$\{x.f\}$⟩

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

iff   ⟨by the definition of Immune (Def. 11 on the previous page)⟩

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

iff   ⟨by the definition of read effects (Fig. 18 on page 16)⟩

$x \neq null$ *implies* **reads** x⸳$/$⸳**modifies region**$\{x.f\}$

iff   ⟨by the definition of separator (Fig. 20 on page 17)⟩

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   ⟨by the definition of Immune (Def. 11 on the previous page)⟩

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

iff   ⟨by RE is **region**$\{x.f.g\}$⟩

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

iff   ⟨by the definition of Immune (Def. 11 on the previous page)⟩

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

iff   ⟨by the definition of read effects (Fig. 18 on page 16)⟩

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

iff   ⟨by the definition of separator (Fig. 20 on page 17)⟩

false

Lemma 12. *Let ε an effect, RE be a region expression, and P be an assertion, such that RE is P/ε-immune. Then $\Sigma(RE) = \Sigma'(RE)$ for any Σ, Σ' such that $\Sigma \rightarrow \Sigma' \models \varepsilon$ and $\Sigma \models P$.*

The following lemma is used in proving Theorem 15.

Lemma 13 (Effect Transfer). *Let Σ_0, Σ_1, Σ_2 be states. Let ε_1 and ε_2 be two effects, and P and P' be two assertions. Suppose the following hold:*

1. $\Sigma_0 \models P$ and $\Sigma_1 \models P'$;
2. $\Sigma_0 \rightarrow \Sigma_1 \models \varepsilon_1$ and $\Sigma_1 \rightarrow \Sigma_2 \models \varepsilon_2, \textbf{modifies}\ RE_1$;
3. ε_2 is P/ε_1-immune;
4. for all $\textbf{fresh}(RE) \in \varepsilon_1$ :: RE is $P/(\varepsilon_2, \textbf{modifies}\ RE_1)$-immune;
5. $\Sigma_1(RE_1) \cap \Sigma_0(\textbf{alloc}) = \varnothing$.

Then $\Sigma_0 \rightarrow \Sigma_2 \models \varepsilon_1, \varepsilon_2$.

Proof. We need to prove that $\Sigma_0 \rightarrow \Sigma_2 \models \varepsilon_1, \varepsilon_2$ satisfies the conditions defined in Definition 3.

For condition (1) in Definition 3, let x be a variable, such that $\Sigma_0(x) \neq \Sigma_2(x)$. It is the case such that either $\Sigma_0(x) \neq \Sigma_1(x)$ or $\Sigma_1(x) \neq \Sigma_2(x)$ or both. By assumption 2, **modifies** x is either in ε_1 or in ε_2 or both.

For condition (2) in Definition 3, let $(o, f) \in \Sigma_0(\textbf{alloc})$, such that $\Sigma_0(o, f) \neq \Sigma_2(o, f)$. There are two cases:

1. $\Sigma_0(o.f) \neq \Sigma_1(o.f)$: By assumption 2, there is some RE, such that **modifies** $RE \in \varepsilon_1$ and $\Sigma_0(RE) = \{(o, f)\}$. By assumption 3 and Lemma 12, $\Sigma_1(RE) = \Sigma_2(RE)$. So $\Sigma_0 \to \Sigma_2 \models \varepsilon_1, \varepsilon_2$
2. $\Sigma_1(o.f) \neq \Sigma_2(o.f)$: By assumption 2, there is some RE, such that **modifies** $RE \in \varepsilon_2$ and $\Sigma_1(RE) = \{(o, f)\}$. So $\Sigma_0 \to \Sigma_2 \models \varepsilon_1, \varepsilon_2$

For condition (3) in Definition 3, there are two cases:

1. Suppose **fresh**$(RE) \in \varepsilon_1$. By assumption 2, $\Sigma_1(RE) \subseteq \Sigma_1(\mathbf{alloc}) - \Sigma_0(\mathbf{alloc})$, and $\Sigma_1(\mathbf{alloc}) \subseteq \Sigma_2(\mathbf{alloc})$. By assumption 3, $\Sigma_1(RE) = \Sigma_2(RE)$. So $\Sigma_2(RE) \subseteq \Sigma_2(RE) - \Sigma_0(\mathbf{alloc})$. So $\Sigma_0 \to \Sigma_2 \models \varepsilon_1, \varepsilon_2$.
2. Suppose **fresh**$(RE) \in \varepsilon_2$. By assumption 2, $\Sigma_2(RE) \subseteq \Sigma_2(\mathbf{alloc}) - \Sigma_1(\mathbf{alloc})$, and $\Sigma_1(\mathbf{alloc}) \subseteq \Sigma_0(\mathbf{alloc})$. So $\Sigma_2(RE) \subseteq \Sigma_2(RE) - \Sigma_0(\mathbf{alloc})$. So $\Sigma_0 \to \Sigma_2 \models \varepsilon_1, \varepsilon_2$.

$\square$

6   Program Correctness In FRL

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 [4,1] 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 14 (Valid FRL Hoare-Formula). *Let S be a statement, let P_1 and P_2 be assertions, let ε be effects, and let (σ, H) be a state. Then $\{P_1\}\, S\, \{P_2\}[\varepsilon]$ is valid in (σ, H), written $\sigma, H \models_r^\Gamma \{P_1\}\, S\, \{P_2\}[\varepsilon]$, if and only if whenever $\sigma, H \models^\Gamma P_1$, then*

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

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 states (σ, H) :: $\sigma, H \models_r^\Gamma \{P_1\}\, S\, \{P_2\}[\varepsilon]$. $\blacksquare$

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.

From now on, Γ is omitted in the judgment when it is clear in the context. Fig. 21 shows the axioms and rules The axioms for variable assignment, field access, field update and allocation are "small" [45] 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 objects.

6.1   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 effects of $S_1 S_2$ are collected from those of S_1 and S_2. To understand $SEQ1_r$, it may be helpful to consider two cases:

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

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

$(ALLOC_r)\ \vdash_r \{true\}\ x := \mathbf{new}\ C;\ \{new_r(C,x)\}[\,\mathbf{modifies}\ x, \mathbf{modifies\ alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})]$

$(ASGN_r)\ \vdash_r \{true\}\ x := F;\ \{x = F\}\,[\,\mathbf{modifies}\ x]\ \mathbf{where}\ x \notin \mathrm{FV}(F)$

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

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

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

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

$(IF_r)\ \dfrac{\vdash_r \{P\&\&E \neq 0\}\ S_1\ \{P'\}[\varepsilon] \qquad \vdash_r \{P\&\&E = 0\}\ S_2\ \{P'\}[\varepsilon]}{\vdash_r \{P\}\mathbf{if}\ (E)\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}\{P'\}[\varepsilon]}$

$(WHILE_r)\ \dfrac{\vdash_r \{P\ \&\&\ E \neq 0\}\ S\ \{P\}\,[\varepsilon, \mathbf{modifies}\ RE]}{\vdash_r \{P\&\&r = \mathbf{alloc}\}\ \mathbf{while}\ (E)\ \{S\}\ \{P\ \&\&\ E = 0\}\,[\varepsilon]},\ \begin{array}{l}\mathbf{where}\ P \Rightarrow RE\,!\,!\,r, \varepsilon\ \text{is fresh-free},\\ \varepsilon\ \text{is}\ P/\varepsilon\text{-immune, and}\ \mathbf{modifies}\ r \notin \varepsilon\end{array}$

Fig. 21: Correctness axioms and rules for statements in FRL.

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 $S_1 S_2$ can drop RE from the write effects of S_2. For example, consider the sequence: $x := \textbf{new } C; x.f := 5;$. We assume f is the only field of the class C for simplicity. Using the axioms $ALLOC_r$ and UPD_r, we have

$$\vdash_r \{true\} x := \textbf{new } C; \{new_u(C,x)\}[\, \textbf{modifies } x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \quad (1)$$

$$\vdash_r \{x \neq null\} x.f := 5; \{x.f = 5\}[\, \textbf{modifies region}\{x.f\}] \quad (2)$$

Then, we use the $SubEff_r$ rule to loosen the write effect of Eq. (2), and get

$$\vdash_r \{x \neq null\} x.f := 5; \{x.f = 5\}[\, \textbf{modifies region}\{x.*\}] \quad (3)$$

Then, we use the $CONSEQ_r$ rule (in Fig. 22 on page 27) on Eq. (1), and get

$$\vdash_r \{true\} x := \textbf{new } C; \{x \neq null\}[\, \textbf{modifies } x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})] \quad (4)$$

In order to use the $SEQ1_r$ rule on Eq. (4) and Eq. (3), it is instantiated with $RE := \textbf{region}\{x.*\}$, $RE_1 := \textbf{region}\{x.*\}$, $\varepsilon_1 := \textbf{modifies } x, \textbf{modifies alloc}$ and $\varepsilon_2 := \varnothing$. Then, we check the immune side conditions, which are:

$$\textbf{modifies } x \text{ is } true/\textbf{modifies } x, \textbf{modifies alloc}\text{-immune} \quad (5)$$

and

$$\varnothing \text{ is } true/\textbf{modifies } x, \textbf{modifies alloc}\text{-immune} \quad (6)$$

Eq. (6) is obviously true. By the definition of immune (Def. 11 on page 18), to prove Eq. (5) is to show

$$\textit{for all } \textbf{modifies } RE \in (\textbf{modifies } x) :: RE \text{ is } true/\textbf{modifies } x, \textbf{modifies alloc}\text{-immune} \quad (7)$$

Eq. (7) is vacuously true, since no region expression RE can be a variable x. Now we can use the rule $SEQ1_r$ and get

$$\vdash_r \frac{\{true\} x := \textbf{new } C; x.f := 5; \{x.f = 5\}}{[\, \textbf{modifies } x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})]}$$

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

$$\frac{\vdash_r \{P\} S_1 \{P_1\}[\varepsilon_1] \qquad \vdash_r \{P_1\} S_2 \{P'\}[\varepsilon_2]}{\vdash_r \{P\} S_1 S_2 \{P'\}[\varepsilon_1, \varepsilon_2]}$$
$$\textbf{where } \varepsilon_1 \text{ is fresh-free and } \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune}$$

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 5.4 on page 18. 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 $\textbf{reads } z$ and $\textbf{modifies } y$ respectively, and the read effect of the second statement is $\textbf{reads } y, \textbf{reads region}\{y.f\}$. The read effects of their composition may not be ($\textbf{reads } z, \textbf{reads } y, \textbf{reads region}\{y.f\}$), as $\textbf{region}\{y.f\}$ 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 $\textbf{reads region}\{z.f\}$, which is immune from updating y.

Consider again the example in Section 5.4, $x.f := x; x.f := x;$. There, we have shown ε_2 is P/ε_1-immune, where ε_1 and ε_2 are both $\textbf{modifies region}\{x.f\}$, and P is $x \neq null$. Here, we show δ_2 is P/ε_1-immune as follows, where δ_2 is $\textbf{reads } x$.

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

iff   ⟨by the definition of Immune (Def. 11 on page 18)⟩
   for all **reads** $RE \in$ **reads** $x :: RE$ is $x \neq null/$**modifies region**$\{x.f\}$-immune
iff   ⟨by there does not exist such RE⟩
   true

The following example shows the use of the rule *SEQ2$_r$*. Consider the program **var** *y* : **int***; y := 5;*. After using the axiom *VAR$_r$*, we get

$$\vdash_r \{true\}\textbf{var}\, y : \textbf{int}; \{y = 0\}[\varnothing] \tag{8}$$

After using the axiom *ASGN$_r$*, we get:

$$\vdash_r \{true\}\, y := 5;\, \{y = 5\}\,[\,\textbf{modifies}\, y] \tag{9}$$

By the rule *CONSEQ$_r$* on Eq. (9), we get

$$\vdash_r \{y = 0\}y := 5;\, \{y = 5\}\,[\,\textbf{modifies}\, y] \tag{10}$$

Using the rule *SEQ2$_r$* on Eq. (8) and Eq. (10), we get $\vdash_r \{true\}\textbf{var}\, y : \textbf{int}; y := 5; \{y = 5\}\,[\varnothing]$

6.2   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. We use an example to show how to instantiate r in the rule *WHILE$_r$*. Consider the following program in program context $\Gamma = \textbf{alloc} : \textbf{region}, f : \textbf{region}, y : \textbf{int}, x : C$:

$$B \overset{\text{def}}{=} x := \textbf{new}\, C;\, f := f + \textbf{region}\{x.*\};\, y := y - 1;$$
$$S \overset{\text{def}}{=} f := \textbf{region}\{\};\, y := 5;\, \textbf{while}\,(y)\,\{B\}$$

We want to prove

$$\vdash_r \{true\}\, S\, \{y = 0\}[\,\textbf{modifies}\, f, \textbf{modifies alloc}, \textbf{modifies}\, x, \textbf{modifies}\, y, \textbf{fresh}(f)] \tag{11}$$

After using the axiom *ASGN$_r$*, once for each of the following, we get

$$\vdash_r \{true\}\, f := \textbf{region}\{\};\{f = \textbf{region}\{\}\}[\,\textbf{modifies}\, f] \tag{12}$$

$$\vdash_r \{true\}\, y := 5;\{y = 5\}[\,\textbf{modifies}\, y] \tag{13}$$

After using the rule *FRM$_r$* on Eq. (13), we get

$$\vdash_r \{f = \textbf{region}\{\}\}\, y := 5;\{f = \textbf{region}\{\}\&\&y = 5\}[\,\textbf{modifies}\, y] \tag{14}$$

From Eq. (12) and Eq. (14), the rule *SEQ1$_r$* is instantiated with $RE := \textbf{region}\{\}$. As the immunity conditions are vacuously true, we can get

$$\vdash_r \{true\}\, f := \textbf{region}\{\};\, y := 5;\{f = \textbf{region}\{\}\&\&y = 5\}[\,\textbf{modifies}\, f, \textbf{modifies}\, y] \tag{15}$$

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

$$\vdash_r \{g\,!\,!\,f\}B\{g\,!\,!\,f\}[\,\textbf{modifies}\, f, \textbf{modifies}\, x, \textbf{modifies}\, y, \textbf{modifies alloc}] \tag{16}$$

From Eq. (16), the rule $WHILE_r$ is instantiated with $r := g$ and $RE = \mathbf{region}\{\}$. Because the immunity conditions are vacuously true, we can get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f \&\& g = \mathbf{alloc}\}\ \mathbf{while}\ (y)\ \{B\}\ \{g\,!\,!\,f \&\& y = 0\} \\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ y, \mathbf{modifies}\ f, \mathbf{modifies}\ \mathbf{alloc}\,] \end{array} \tag{17}$$

The rule $PostToFr_r$ is instantiated with $r := g$ and $RE := f$. We get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f \&\& g = \mathbf{alloc}\}\ \mathbf{while}\ (y)\ \{B\}\ \{g\,!\,!\,f \&\& y = 0\} \\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ y, \mathbf{modifies}\ f, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(f)] \end{array} \tag{18}$$

After using the rule $CONSEQ_r$ from the above, we get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f \&\& g = \mathbf{alloc}\}\ \mathbf{while}\ (y)\ \{B\}\ \{y = 0\} \\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ y, \mathbf{modifies}\ f, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(f)] \end{array} \tag{19}$$

The postcondition of Eq. (15) implies the precondition of Eq. (19). After using the rule $CONSEQ_r$ on Eq. (15), we get

$$\vdash_r \{true\}\ f := \mathbf{region}\{\};\ y := 5;\ \{g\,!\,!\,f \&\& g = \mathbf{alloc}\}[\,\mathbf{modifies}\ f, \mathbf{modifies}\ y\,] \tag{20}$$

From Eq. (20) and Eq. (19), the rule $SEQ1_r$ is instantiated with $RE = \mathbf{region}\{\}$. As the immunity conditions are vacuously true, we can get Eq. (11).

Now we show the proof of Eq. (16). After using the axiom $ALLOC_r$, we get:

$$\vdash_r \{true\}\ x := \mathbf{new}\ C;\ \{new_u(C, x)\}[\,\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \tag{21}$$

Then by the rule FRM_r from the above, we get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f\}\ x := \mathbf{new}\ C;\ \{new_r(C, x) \&\& g\,!\,!\,f\} \\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{22}$$

The rule $FrToPost_r$ is instantiated with $r := g$ and $RE = \mathbf{region}\{x.*\}$. And $\mathbf{reads}\ g\ /\!\!\!\cdot\ (\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc})$ is true. After applying the rule, we get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f\}\ x := \mathbf{new}\ C;\ \{new_u(C, x) \&\& g\,!\,!\,f \&\& g\,!\,!\,\mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{23}$$

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 $ASGN_r$, we get:

$$\vdash_r \{true\}\ f := f' + \mathbf{region}\{x.*\};\ \{f = f' + \mathbf{region}\{x.*\}\}\,[\,\mathbf{modifies}\ f\,] \tag{24}$$

After using the rule FRM_r, we get

$$\vdash_r \begin{array}{l} \{new_u(C, x) \&\& g\,!\,!\,f \&\& g\,!\,!\,\mathbf{region}\{x.*\}\} \\ f := f' + \mathbf{region}\{x.*\}; \\ \{f = f' + \mathbf{region}\{x.*\} \&\& new_u(C, x) \&\& g\,!\,!\,f \&\& g\,!\,!\,\mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\ f\,] \end{array} \tag{25}$$

From Eq. (23) and Eq. (25), the rule $SEQ1_r$ is instantiated with $RE = \mathbf{region}\{\}$. As the immunity conditions are vacuously true, we can get

$$\vdash_r \begin{array}{l} \{g\,!\,!\,f\} \\ x := \mathbf{new}\ C;\ f := f' + \mathbf{region}\{x.*\}; \\ \{f = f' + \mathbf{region}\{x.*\} \&\& new_u(C, x) \&\& g\,!\,!\,f \&\& g\,!\,!\,\mathbf{region}\{x.*\}\} \\ [\,\mathbf{modifies}\ f, \mathbf{modifies}\ x, \mathbf{modifies}\ \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \end{array} \tag{26}$$

Then by the rules $CONSEQ_r$ and $SubEff_r$, we get

$$\vdash_r \frac{\{g\,!\,!\,f\}\; x := \textbf{new}\; C;\; f := f' + \textbf{region}\{x.*\};\; \{g\,!\,!\,f\}}{[\,\textbf{modifies}\; f, \textbf{modifies}\; x, \textbf{modifies}\; \textbf{alloc}\,]} \qquad (27)$$

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 $ASGN_r$, we get $\vdash_r \{true\}\; y := y' - 1;\; \{y = y' - 1\}[\,\textbf{modifies}\; f\,]$ Then, by the rules FRM_r, $SEQ1_r$ and $CONSEQ_r$, we get Eq. (16).

6.3   Soundndess

Theorem 15. *Let S be a statement, P and Q be assertions, and ε be effects. If $\vdash_r \{P\}S\{Q\}[\varepsilon]$, then $\models_r$ $\{P\}\; S\; \{Q\}[\varepsilon]$.*

Proof. The proof is done by induction on the derivation and by cases on the last rule used. In each axiom, we show the judgment is valid according to the statement's semantics. In each inference rule, we show 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 $(\sigma', h') = \mathcal{MS}[\![S]\!](\sigma, h)$. We assume $\vdash_r^\Gamma \{P\}\; S\; \{Q\}[\varepsilon]$, and $\sigma, h \models^\Gamma P$. Then we must 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 $\textbf{skip}$;, P is $true$, Q is $true$, ε is $\varnothing$. By the program semantics Fig. 21 on page 21, $\sigma' = \sigma$, $h' = h$ 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 $\textbf{var}\, x : T;$, P is $true$, Q is $x = default(T)$ and ε is $\varnothing$. By the program semantics Fig. 21, $\Gamma' = \Gamma, (x : T)$, $\sigma' = \texttt{Extend}(\sigma, x, default(T))$ 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 $x := \textbf{new}\; C;$, P is $true$, Q is $x.f = default(T)$ and $\varepsilon = \textbf{modifies}\; x$, $\textbf{alloc}$, $\textbf{fresh}(\textbf{region}\{x.*\})$. By the program semantics Fig. 21, $\Gamma' = \Gamma$, $\sigma' = \sigma(x \mapsto l)$ and $h' = h''[(l, f) \mapsto default(T)]$, where $(l, h'') = allocate(C, h)$. Thus, (σ', h') entails Q.
   For the frame condition, S only updates the variable x and $\textbf{alloc}$. By the semantics, the function $allocate$ returns a new heap. So $\textbf{fresh}(\textbf{region}\{x.*\})$ is the fresh effect.
4. ($ASSGN_r$) In this case, S is $x := F;$, P is $x = x'$, Q is $\{x = F/(x \mapsto x')\}$ and $\varepsilon = \textbf{modifies}\; x$, where $x \notin \text{FV}(F)$. By the program semantics Fig. 21 on page 21, $\Gamma' = \Gamma$, $(\sigma', H') = (\sigma[x \mapsto \mathcal{E}[\![F]\!](\sigma)], H)$, which entails Q.
   For the frame condition, this statement only updates variable x. Therefore, ε is $\textbf{modifies}\; x$ is correct.
5. (UPD_r) In this case, S is $x.f := F;$, P is $x \neq null$, Q is $x.f = F$ and ε is $\textbf{modifies}\; \textbf{region}\{x.f\}$. By the program semantics Fig. 21, $\Gamma' = \Gamma$, $(\sigma', H') = (\sigma, H[(\mathcal{E}[\![x]\!](\sigma), f) \mapsto \mathcal{E}[\![F]\!](\sigma)])$, which entails Q.
   For the frame condition, this statement changes the singleton heap location $(\sigma(x), f)$. Thus, ε is $\textbf{modifies}\; \textbf{region}\{x.f\}$ 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 $\textbf{modifies}\; x$, where $x \neq x'$. By the program semantics Fig. 21, $\Gamma' = \Gamma$, $(\sigma', H') = (\sigma[x \mapsto H[(\mathcal{E}[\![x']\!](\sigma), f)]], H)$, which entails Q.
   For the frame condition, this statement only updates variable x. Therefore, $\varepsilon = \textbf{modifies}\; x$ 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 $\sigma_i, h_i \models^{\Gamma_i} P_i$, then $\sigma_i', h_i' \models^{\Gamma_i'} Q_i$.

1. (IF_r) In this case, S is $\textbf{if}\; (E)\; \textbf{then}\; \{S_1\}\; \textbf{else}\{S_2\}$. We consider are two cases:
   - $E \neq 0$. By the inductive hypothesis, we have $\sigma, H \models^\Gamma P \&\& E \neq 0$, $(\sigma', H') = \mathcal{MS}[\![S_1]\!](\sigma, H)$, which entails Q. And the frame condition is correct.
   - $E = 0$. By the inductive hypothesis, we have $\sigma, H \models^\Gamma P \&\& E = 0$, $(\sigma'', H'') = \mathcal{MS}[\![S_2]\!](\sigma, H)$, which entails Q. And the frame condition is correct.
   By the program semantics Fig. 21, 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** $\{S\}$. $P = I$, $Q = I \,\&\&\, E \neq 0$ and the frame conditions is ε. The premise is $\vdash_r \{I \,\&\&\, E \neq 0\} \{S\} \{I\}[\varepsilon]$.

   By the program semantics Fig. 21, let g be a recursive point function, such that

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

   By definition, fix is a fixed point function, so $fix(g) = g$. Then we prove $fix(g)(\sigma, H) \models^{\Gamma} I$ by fixed-point induction.

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

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

   There are two cases:
   - $E \neq 0$. By the semantics, $fix(g)(\sigma'', H'') = g(\mathcal{MS}[\![S]\!](\sigma'', H''))$. By the inductive hypothesis, we know that $g(\mathcal{MS}[\![S]\!](\sigma'', H'')) \models^{\Gamma} I$ holds. Hence $fix(g)(\sigma'', H'') \models^{\Gamma} I$ 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.
   - $E = 0$. By the semantics, $fix(g)(\sigma'', H'') = (\sigma'', H'')$. Therefore, by the inductive hypothesis, we know that $fix(g)(\sigma'', H'') \models^{\Gamma} I$ holds. For the frame condition, since the state does not change, the frame is **region**$\{\}$, which is the subset of ε.

   Now we conclude that if the loop exits, which means that $E = 0$ holds, the loop invariant I holds. Therefore, Q holds and ε is its frame condition.

3. ($SEQ1_r$) In this case, S is $S_1 S_2$, where $S_1 \neq \textbf{var } x : T;$. Let (σ, H) be a state, such that $(\sigma, H) \models^{\Gamma} P$. By the inductive hypothesis for S_1 and S_2, $(\sigma'', H'') = \mathcal{MS}[\![S_1]\!](\sigma, H)$, and $(\sigma'', H'') \models^{\Gamma''} P_1$. By the second premise and the semantics, $(\sigma', H') = \mathcal{MS}[\![S_2]\!](\sigma'')$. Hence $(\sigma', H') \models^{\Gamma'} P'$.

   For the frame condition, we must show $(\sigma, H) \to (\sigma', H') \models_r \varepsilon_1, \varepsilon_2, \textbf{fresh}(RE)$, which is proved by Lemma 13. It is instantiated with $RE_1 := RE_1$, $\Sigma_0 := (\sigma, H)$, $\Sigma_1 := (\sigma'', H'')$, $\varepsilon_1 := (\varepsilon_1, \textbf{fresh}(RE))$. 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;
   - $(\sigma, H) \to (\sigma'', H'') \models (\varepsilon_1, \textbf{fresh}(RE))$ by the inductive hypothesis;
   - $(\sigma'', H'') \to (\sigma', H') \models (\varepsilon_2, \textbf{modifies } RE_1)$ by the inductive hypothesis.
   - ε_2 is P/ε_1-immune by the given side condition;
   - for all $\textbf{fresh}(RE) \in \varepsilon_1 :: RE$ is $P/(\varepsilon_2, \textbf{modifies } RE_1)$-immune by the given side condition.
   - $\mathcal{E}[\![RE_1]\!](\sigma'') \cap \sigma(\textbf{alloc}) = \varnothing$, RE are freshly allocated regions by S_1, such that $\mathcal{E}[\![RE_1]\!](\sigma'') \subseteq (\sigma''(\textbf{alloc}) - \sigma(\textbf{alloc}))$.

4. ($SEQ2_r$) In this case, S is $\textbf{var } x : T; S_2$. 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, we have $\models_r^{\Gamma} \{P\}S\{Q\}[\varepsilon]$. And by the assumption, we have $P \models^{\Gamma} \delta \, frm \, Q$ and $P \,\&\&\, R \Rightarrow \delta \cdot / \varepsilon$. We must show that $\models_r^{\Gamma} \{P \,\&\&\, R\}S\{Q \,\&\&\, R\}[\varepsilon]$. Because $P \,\&\&\, R$ implies P, Thus, we have $\models_r^{\Gamma} \{P \,\&\&\, R\}S\{Q\}[\varepsilon]$. Let $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H)$. We must show that $(\sigma', H') \models^{\Gamma'} R$. By $(\sigma, H) \models^{\Gamma} P \,\&\&\, R$ and the side condition $P \,\&\&\, R \Rightarrow \delta \cdot / \varepsilon$, we have $(\sigma, H) \models^{\Gamma} \delta \cdot / \varepsilon$. As the write effect is $(\sigma, H) \to (\sigma', H') \models \varepsilon$, we have $(\sigma, H) \stackrel{\delta}{\equiv} (\sigma', H')$. By the definition of framing (Def. 5 on page 17) and $(\sigma, H) \models^{\Gamma} P \,\&\&\, R$, we conclude that $(\sigma', H') \models^{\Gamma'} R$.

7. ($CONSEQ_r$) In this case, by the inductive hypothesis, we have $\models_r^{\Gamma} \{P'\}S\{Q'\}[\varepsilon]$. By the premise, $P \Rightarrow P'$ and $Q' \Rightarrow Q$. Hence $\models_r^{\Gamma} \{P\}S\{Q\}[\varepsilon]$ is valid.

$\square$

$$(FRM_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon] \qquad P \vdash \delta \, frm \, Q}{\vdash_r \{P \&\& Q\} \, S \, \{P' \&\& Q\}[\varepsilon]} \quad \textbf{where } P \&\& Q \Rightarrow \delta \cdot / \cdot \varepsilon$$

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

$$(CONSEQ_r) \quad \frac{\vdash_r \{P_1\} \, S \, \{P_1'\}[\varepsilon]}{\vdash_r \{P_2\} \, S\{P_2'\}[\varepsilon]} \quad \textbf{where } P_2 \Rightarrow P_1 \text{ and } P_1' \Rightarrow P_2'$$

$$(ConEff_r) \quad \frac{\vdash_r \{P \&\& E \neq 0\} \, S \, \{P'\}[\varepsilon_1] \qquad \vdash_r \{P \&\& E = 0\} \, S \, \{P'\}[\varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$

$$(ConMask1_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{modifies if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \varepsilon_1]} \quad \textbf{where } P \Rightarrow E \neq 0$$

$$(ConMask2_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{modifies if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \varepsilon_2]} \quad \textbf{where } P \Rightarrow E = 0$$

$$(PostToFr_r) \quad \frac{\vdash_r \{P \&\& r = \texttt{alloc}\} \, S \, \{P'\}[\varepsilon]}{\vdash_r \{P \&\& r = \texttt{alloc}\} \, S \, \{P'\}[\varepsilon, \textbf{fresh}(RE)]}$$
$$\textbf{where } r \text{ is fresh and } P' \Rightarrow RE \,!! \, r \text{ and } \textbf{reads } \, r \cdot / \cdot \varepsilon$$

$$(FrToPost_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{fresh}(RE)]}{\vdash_r \{P\} \, S \, \{P' \&\& r \,!! \, RE\}[\varepsilon, \textbf{fresh}(RE)]}$$
$$\textbf{where } \textbf{reads } \, r \cdot / \cdot \varepsilon$$

$$(VarMask1_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\textbf{if } E \textbf{ then } x, \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow z = E, P \Rightarrow z \neq 0, P \,\|\, P' \Rightarrow x = y, P \&\& z \neq 0 \Rightarrow \textbf{reads } \, y \cdot / \cdot (x, \varepsilon),$$
$$\text{and } \textbf{modifies } z \notin (\varepsilon_1, \varepsilon_2)$$

$$(VarMask2_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } x, \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow z = E, P \Rightarrow z = 0, P \,\|\, P' \Rightarrow x = y, P \wedge z = 0 \Rightarrow \textbf{reads } \, y \cdot / \cdot (x, \varepsilon)$$
$$\text{and } \textbf{modifies } z \notin (\varepsilon_1, \varepsilon_2)$$

$$(FieldMask1_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then modifies region}\{x.f\}, \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow z = E, P \Rightarrow z \neq 0, P \,\|\, P' \Rightarrow x.f = y, P' \&\& z \neq 0 \Rightarrow \textbf{reads } \, x \cdot / \cdot \textbf{modifies } \varepsilon$$
$$P' \&\& z \neq 0 \Rightarrow \textbf{reads } \, y \cdot / \cdot \textbf{modifies } \varepsilon \text{ and } \text{ and } \textbf{modifies } z \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$$

$$(FieldMask2_r) \quad \frac{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else modifies region}\{x.f\}, \varepsilon_2]}{\vdash_r \{P\} \, S \, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow z = E, P \Rightarrow z = 0, P \,\|\, P' \Rightarrow x.f = y, P' \&\& z = 0 \Rightarrow \textbf{reads } \, x \cdot / \cdot \textbf{modifies } \varepsilon$$
$$\text{and } P' \&\& z = 0 \Rightarrow \textbf{reads } \, y \cdot / \cdot \textbf{modifies } \varepsilon \text{ and } \textbf{modifies } z \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$$

Fig. 22: Structural rules in FRL.

7   Program Correctness In UFRL

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 are the same as those in FRL. In particular, it inherits the special variable **alloc** as well.

However, Hoare-formulas in UFRL are different. The correctness judgment 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 16 (Validity of UFRL Hoare-formula)**.** *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 state. Then $[\delta]\{P_1\}S\{P_2\}[\varepsilon]$ is valid in (σ, H), written $\sigma, H \vDash_u \{P_1\}S\{P_2\}[\varepsilon][\delta]$, if and only if whenever $\sigma, H \vDash_u P_1$, then:*

1. *$\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma)) \neq err$, and*
2. *if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma))$, then the following all hold:*
   - *$\sigma', H' \vDash_u P_2$,*
   - *for all $x \in dom(\sigma){::}\,\sigma'(x) \neq \sigma(x)$ implies* **modifies** $x \in \varepsilon$*,*
   - *for all $(o, f) \in dom(H){::}\,H'[o, f] \neq H[o, f]$ implies $(o, f) \in \mathcal{E}[\![writeR(\varepsilon)]\!](\sigma)$, and*
   - *for all $(o, f) \in \mathcal{E}[\![freshR(\varepsilon)]\!](\sigma'){::}\,(o, f) \in (dom(H') - dom(H))$.*

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

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

$$[\textbf{reads region}\{x.f\}]\{x \neq null\}y := x.f; \{y = x.f\}[\textbf{modifies } y]. \tag{28}$$

Eq. (28) is a valid UFRL Hoare-formula, because $rwR(\textbf{reads region}\{x.f\}, \textbf{modifies } y) = \textbf{region}\{x.f\}$. 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, because $rwR(\textbf{reads } \varnothing, \textbf{modifies } y) = \textbf{region}\{\}$. As another example, consider the following formula:

$$[\varnothing]\{x \neq null\}x.f := y; \{x.f = y\}[\textbf{modifies region}\{x.f\}]. \tag{29}$$

Eq. (29) is a valid UFRL Hoare-formula, because $rwR(\varnothing, \textbf{modifies region}\{x.f\}) = \textbf{region}\{x.f\}$.

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 what the statement can access on the heap. However, read effects (on the heap) are needed for future work; e.g., for framing of specifications with pure method calls [2].

7.1   Proof rules for UFRL

In this section we describe the proof rules for proving statements correct in UFRL. Fig. 23 on the next page and Fig. 24 on page 31 show the axioms and rules; these are based on FRL, but with read effects (δ and η) specified.

In the rules, we use the shorthand $new_u(C, x)$ to mean $x.f_1 = default(T_1)\&\&\ldots\&\&x.f_n = default(T_n)$, where the $f_i : T_i$ are defined by $(f_1 : T_1, \ldots, f_n : T_n) = fields(C, CT)$. The function, *default*, takes a type and returns its default value. The predicate *true* is syntactic sugar for $1 = 1$.

The axioms for variable declaration, variable assignment, field access, field update and allocation are "small" [45] 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 rwR (Def. 16). The fresh effects in the rule of the new statement account for the newly allocated locations.

$(SKIP_u)$ $\vdash_u [\varnothing]\{true\}\textbf{skip};\{true\}[\varnothing]$

(VAR_u) $\vdash_u [\varnothing]\{true\}\textbf{var}\ x:T;\{x = default(T)\}[\varnothing]$

$(ALLOC_u)$ $\vdash_u [\varnothing]\{true\}\ x := \textbf{new}\ C;\ \{new_u(C,x)\}[\textbf{modifies}\ x, \textbf{modifies alloc}, \textbf{fresh}(\textbf{region}\{x.*\})]$

$(ASGN_u)$ $\vdash_u [\eta]\{true\}\ x := F;\ \{x = F\}\ [\textbf{modifies}\ x]\ \text{where}\ x \notin \text{FV}(F)\ \text{and}\ \eta = efs(F)$

(UPD_u) $\vdash_u [x,\eta]\ \{x \neq null\}\ x.f := F;\ \{x.f = F\}[\textbf{modifies region}\{x.f\}]\text{where}\ \eta = efs(F)$

(ACC_u) $\vdash_u [\eta]\{x' \neq null\}\ x := x'.f;\ \{x = x'.f\}\ [\textbf{modifies}\ x]\ ,\ \text{where}\ x \neq x'\ \text{and}\ \eta = efs(x'.f)$

(IF_u) $\dfrac{\vdash_u [\delta]\{P\ \&\&\ E \neq 0\}\ S_1\{Q\}\ [\varepsilon] \qquad \vdash_u [\delta]\{P\ \&\&\ E = 0\}\ S_2\{Q\}\ [\varepsilon]}{\vdash_u [\delta,\delta_E]\{P\}\ \textbf{if}\ (E)\ \{S_1\}\textbf{else}\{S_2\}\{Q\}\ [\varepsilon]}$ $\text{where}\ \delta_E = efs(E)$

$(SEQ1_u)$ $\dfrac{\vdash_u [\delta_1]\{P\}\ S_1\ \{P_1\}[\varepsilon_1, \textbf{fresh}(RE)] \qquad \vdash_u [\delta_2, \textbf{reads}\ RE_1]\{P_1\}\ S_2\ \{P'\}[\varepsilon_2, \textbf{modifies}\ RE_2]}{\vdash_u [\delta_1,\delta_2]\{P\}\ S_1 S_2\ \{P'\}[\varepsilon_1, \varepsilon_2, \textbf{fresh}(RE)]}$
$\text{where}\ S_1 \neq \textbf{var}\ x:T;, \varepsilon_1\ \text{is fresh-free},\ \delta_2\ \text{is}\ P/\varepsilon_1\text{-immune},\ \varepsilon_2\ \text{is}\ P/\varepsilon_1\text{-immune},$
$RE\ \text{is}\ P_1/(\textbf{modifies}\ RE_2, \varepsilon_2)\text{-immune}, RE_1 \leqslant RE\ \text{and}\ RE_2 \leqslant RE$

$(SEQ2_u)$ $\dfrac{\vdash_u [\delta, \textbf{reads}\ x]\{P\ \&\&\ x = default(T)\}\ S\ \{Q\}[\textbf{modifies}\ x, \varepsilon]}{\vdash_u [\delta]\{P\}\ \textbf{var}\ x:T; S\ \{Q\}[\varepsilon]}$

$(WHILE_u)$ $\dfrac{\vdash_u [\delta]\{P\ \&\&\ E \neq 0\}\ S\{P\}\ [\varepsilon, \textbf{modifies}\ RE]}{\vdash_u [\delta,\delta_E]\{P\ \&\&\ r = \textbf{alloc}\}\ \textbf{while}\ (E)\ \{S\}\ \{P\ \&\&\ E = 0\}\ [\varepsilon]}$
$\text{where}\ \delta_E = efs(E), P \Rightarrow RE\ !\ !\ r, \varepsilon\ \text{is fresh-free}, \textbf{modifies}\ r \notin \varepsilon, \delta\ \text{is}\ P/\varepsilon\text{-immune}\ and\ \varepsilon\ \text{is}\ P/\varepsilon\text{-immune}$

Fig. 23: Correctness rules and axioms for statements in UFRL.

Fig. 22 on page 27 shows structural rules. The FRM_u follows the FRM_r rule. The rule $SubEff_u$ allows approximation of effects; it can be used to match up the effects for the rule IF_u, where different branches may have different effects. $SubEff_u$ 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, we can derive the following from the rule $FrToPost_u$ as follows:

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

by using the subeffect rule, because $rwR(\delta, \mathbf{fresh}(RE), \varepsilon) \leqslant rwR(\delta, \varepsilon)$, as rwR ignores fresh effects.

The Sequence Rules We discuss the complication arouse from read effects. 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 $S_1 S_2$ can drop RE from the read effects of S_2. For example, consider the sequence: $x := \mathbf{new}\, C; y := x.f$, where $x \neq y$. We assume f is the only field of class C for simplicity. Using the rules $ALLOC_u$ and ACC_u, we have

$$\vdash_u [\varnothing]\{true\}x := \mathbf{new}\, C; \{new_u(C, x)\}[\,\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \quad (30)$$

$$\vdash_u [\mathbf{reads}\, x, \mathbf{region}\{x.f\}]\{x \neq null\}y := x.f; \{y = x.f\}[\,\mathbf{modifies}\, y] \quad (31)$$

Then, we use the $SubEff_u$ rule to loosen the read effect of Eq. (31), and get

$$\vdash_u [\mathbf{reads}\, x, \mathbf{region}\{x.*\}]\{x \neq null\}y := x.f; \{y = x.f\}[\,\mathbf{modifies}\, y] \quad (32)$$

Then, we use the $CONSEQ_u$ rule (in Fig. 22 on page 27) on Eq. (30), and get

$$\vdash_u [\varnothing]\{true\}x := \mathbf{new}\, C; \{x \neq null\}[\,\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc}, \mathbf{fresh}(\mathbf{region}\{x.*\})] \quad (33)$$

In order to use the $SEQ1_u$ rule on Eq. (33) and Eq. (32), it is instantiated with $RE := \mathbf{regoin}\{x.*\}$, $RE_1 := \mathbf{region}\{x.*\}$, $RE_2 := \mathbf{region}\{\}$, $\varepsilon_1 := \mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc}$ and $\varepsilon_2 := \mathbf{modifies}\, y$. Then, we check the immune side conditions, which are:

$$\mathbf{reads}\, x \text{ is } true/(\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc})\text{-immune} \quad (34)$$

and

$$\mathbf{modifies}\, y \text{ is } true/(\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc})\text{-immune} \quad (35)$$

By the definition of immune (Def. 11 on page 18), to prove Eq. (34) and Eq. (35) is to show

$$\textit{for all } \mathbf{reads}\, RE \in (\mathbf{reads}\, x) :: RE \text{ is } true/(\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc})\text{-immune} \quad (36)$$

and

$$\textit{for all } \mathbf{modifies}\, RE \in (\mathbf{modifies}\, y) :: RE \text{ is } true/(\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc})\text{-immune} \quad (37)$$

Eq. (36) and Eq. (37) are vacuously true. Now we can use the rule $SEQ1_u$ and get

$$\vdash_u \begin{array}{l} [\mathbf{reads}\, x] \\ \{true\}\mathtt{x := new\ C;\ y := x.f;}\{y = x.f\} \\ [\,\mathbf{modifies}\, x, \mathbf{modifies}\, \mathbf{alloc}, \mathbf{modifies}\, y, \mathbf{fresh}(\mathbf{region}\{x.*\})\} \end{array}$$

In this case, the $\mathbf{region}\{x.*\}$ 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.

(FRM_u) $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash \eta\, frm\, Q}{\vdash_u [\delta]\{P\&\&Q\}\, S\, \{P'\&\&Q\}[\varepsilon]}$ **where** $P\&\&Q \Rightarrow \eta\,/\!\cdot\varepsilon$

$(SubEff_u)$ $\dfrac{\vdash_u [\delta]\{P_1\}S\{P_2\}[\varepsilon] \qquad P_1 \vdash \varepsilon \leqslant \varepsilon'}{\vdash_u [\delta']\{P_1\}S\{P_2\}[\varepsilon']}$ **where** $P_1 \Rightarrow rwR(\varepsilon,\delta) \leqslant rwR(\varepsilon',\delta')$

$(CONSEQ_u)$ $\dfrac{\vdash_u [\delta]\{P_1\}\, S\, \{P_1'\}[\varepsilon]}{\vdash_u [\delta]\{P_2\}\, S\, \{P_2'\}[\varepsilon]}$ **where** $P_2 \Rightarrow P_1$ and $P_1' \Rightarrow P_2'$

$(ConEff_u)$ $\dfrac{\vdash_u [\delta]\{P\&\&E \neq 0\}\, S\, \{P'\}[\varepsilon_1] \qquad \vdash_u [\delta]\{P\&\&E = 0\}\, S\, \{P'\}[\varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}$

$(ConMask1_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_1]}$ **where** $P \Rightarrow E \neq 0$

$(ConMask2_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_2]}$ **where** $P \Rightarrow E = 0$

$(PostToFr_u)$ $\dfrac{\vdash_u [\delta]\{P\&\&r = \textbf{alloc}\}\, S\, \{P'\}[\varepsilon]}{\vdash_u [\delta]\{P\&\&r = \textbf{alloc}\}\, S\, \{P'\}[\varepsilon, \textbf{fresh}(RE)]}$
where r is fresh and $P' \Rightarrow RE\,!\,!\,r$ and $\textbf{reads}\ r\!\cdot\!/\!\cdot\varepsilon$

$(FrToPost_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{fresh}(RE)]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{fresh}(RE)]}$
where $\textbf{reads}\ r\!\cdot\!/\!\cdot\varepsilon$

$(VarMask1_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\textbf{if}\, E\, \textbf{then}\, \textbf{modifies}\, x, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}$
where $P \Rightarrow z = E, P \Rightarrow z \neq 0, P \,||\, P' \Rightarrow x = y, P\&\&z \neq 0 \Rightarrow \textbf{reads}\ y\!\cdot\!/\!\cdot(x, \varepsilon),$
and $\textbf{modifies}\ z \notin (\varepsilon_1, \varepsilon_2)$

$(VarMask2_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \textbf{modifies}\, x, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}$
where $P \Rightarrow z = E, P \Rightarrow z = 0, P \,||\, P' \Rightarrow x = y, P\&\&z = 0 \Rightarrow \textbf{reads}\ y\!\cdot\!/\!\cdot(x, \varepsilon)$
and $\textbf{modifies}\ z \notin (\varepsilon_1, \varepsilon_2)$

$(FieldMask1_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \textbf{modifies}\, \textbf{region}\{x.f\}, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}$
where $P \Rightarrow z = E, P \Rightarrow z \neq 0, P \,||\, P' \Rightarrow x.f = y, P'\&\&z \neq 0 \Rightarrow \textbf{reads}\ x\!\cdot\!/\!\cdot\textbf{modifies}\ \varepsilon$
$P'\&\&z \neq 0 \Rightarrow \textbf{reads}\ y\!\cdot\!/\!\cdot\textbf{modifies}\ \varepsilon$ and and $\textbf{modifies}\ z \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

$(FieldMask2_u)$ $\dfrac{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \textbf{modifies}\, \textbf{region}\{x.f\}, \varepsilon_2]}{\vdash_u [\delta]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if}\, E\, \textbf{then}\, \varepsilon_1\, \textbf{else}\, \varepsilon_2]}$
where $P \Rightarrow z = E, P \Rightarrow z = 0, P \,||\, P' \Rightarrow x.f = y, P'\&\&z = 0 \Rightarrow \textbf{reads}\ x\!\cdot\!/\!\cdot\textbf{modifies}\ \varepsilon$
and $P'\&\&z = 0 \Rightarrow \textbf{reads}\ y\!\cdot\!/\!\cdot\textbf{modifies}\ \varepsilon$ and $\textbf{modifies}\ z \notin (\varepsilon, \varepsilon_1, \varepsilon_2)$

Fig. 24: Structural rules.

7.2 Soundness

Theorem 17. *Let S be a statement. Let P and Q be assertions. Let ε be effects and δ be read effects. If $\vdash_u [\delta]\{P\}S\{Q\}[\varepsilon]$, then $\models_u \{P\}S\{Q\}[\varepsilon][\delta]$.*

Proof. Using the result of Theorem 15 on page 25, the proof only needs to check the read effects. Let S be a statement and (σ, h) be an arbitrary state. We assume $\vdash_u [\delta]\{P\} S \{Q\}[\varepsilon]$ and $\sigma, h \models^\Gamma P$. Then we must show that $\mathcal{MS}[\![S]\!](\sigma, h \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma)) \neq err$.

1. ($SKIP_u$) In this case, S is **skip;**, P is *true*, and δ and ε are $\varnothing$. As $h \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \varnothing$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![$**skip;**$]\!](\sigma, \varnothing) \neq err$.
2. (VAR_u) In this case, S is **var**$x : T$;, P is *true*, and δ and ε are $\varnothing$. As $h \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \varnothing$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![$**var**$x : T;]\!](\sigma, \varnothing) \neq err$.
3. ($ALLOC_u$) In this case, S is $x := $ **new** C;, P is *true* and $\varepsilon = \delta = \varnothing$. As $h \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \varnothing$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![x := $ **new** $C;]\!](\sigma, \varnothing) \neq err$.
4. ($ASSGN_u$) In this case, S is $x := F$;, P is $x = x'$ and δ is $efs(F)$ and ε is **modifies** x, where $x \notin \text{FV}(F)$. Since $h \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \varnothing$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![x := F;]\!](\sigma, \varnothing) \neq err$.
5. (UPD_u) In this case, S is $x.f := F$;, P is $x \neq null$ δ is (**reads** $x, efs(F)$) and ε is **modifies region**$\{x.f\}$. By the precondition, we know that $\sigma(x) \neq null$. Since $\mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \{(\sigma(x), f)\}$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![x.f := F;]\!](\sigma, h \upharpoonright \{(\sigma(x), f)\}) \neq err$.
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 $x \neq x'$. By the precondition, we know that $\sigma(x') \neq null$. As $\mathcal{E}[\![rwR(\varepsilon, \delta)]\!](\sigma) = \{(\sigma(x'), f)\}$, by the program semantics Fig. 11 on page 13, $\mathcal{MS}[\![x := x'.f;]\!](\sigma, h \upharpoonright \{(\sigma(x'), f)\}) \neq err$.

Other inductive cases follow inductive hypotheses.

$\square$

8   The Relationship between FRL and UFRL

The following lemma shows that FRL Hoare formulas can be translated into UFRL by using the read effect **reads alloc↓**.

Lemma 18. *Let S be a statement, and let P_1 and P_2 be assertions. Let ε be effects, and let (σ, H) be a state. Then*

$$\sigma, H \models_r \{P_1\}S\{P_2\}[\varepsilon] \text{ iff } \sigma, H \models_u [\textbf{reads alloc↓}]\{P_1\}S\{P_2\}[\varepsilon].$$

Proof. We prove the lemma as follows, starting from the left side.

$\sigma, H \models_r \{P_1\} S \{P_2\}[\varepsilon]$
iff $\langle$by the definition of FRL valid Hoare-formula (Def. 14).$\rangle$
  $\sigma, H \models P_1$ implies $\mathcal{MS}[\![S]\!](\sigma, H) \neq err$ and if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H)$,
  then $(\sigma', H' \models P_2$ and (*for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* **modifies** $x \in \varepsilon$) and
  for all $(o, f) \in dom(H) :: (H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![writeR(\varepsilon)]\!](\sigma)$ and
  for all $(o, f) \in \mathcal{E}[\![freshR(\varepsilon)]\!](\sigma') :: (o, f) \in (dom(H') - dom(H)))$
iff $\langle$by $H = H \upharpoonright dom(H), dom(H) = \mathcal{E}[\![rwR(\varepsilon, \textbf{alloc↓})]\!](\sigma)\rangle$
  $\sigma, H \models P_1$ implies $\mathcal{MS}[\![S]\!](\sigma, H) \neq err$ and if
  $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![rwR(\varepsilon, \textbf{alloc↓})]\!](\sigma))$,
  then $(\sigma', H' \models P_2)$ and (*for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* **modifies** $x \in \varepsilon$) and
  (*for all* $(o, f) \in dom(H) :: (H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![writeR(\varepsilon)]\!](\sigma)))$ and
  (*for all* $(o, f) \in \mathcal{E}[\![freshR(\varepsilon)]\!](\sigma') :: (o, f) \in (dom(H') - dom(H))))$
iff $\langle$by the definition of UFRL valid Hoare-formula (Def. 16)$\rangle$
  $\sigma, H \models_u [\textbf{reads alloc↓}]\{P_1\} S \{P_2\}[\varepsilon]$

$\square$

Corollary 19. *Let S be a statement, and let P_1 and P_2 be assertions. Let ε be effects, and η be read effects. Then*

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

Def. 20 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 20 (Syntactic Mapping from FRL to UFRL). *Let P_1 and P_2 be assertions in FRL. Let ε be effects. We define a syntactic mapping $TR_R[\![-]\!]$ from FRL rules to those of UFRL below:*

For the FRL axioms: $TR_R[\![\vdash_r \{P_1\} S \{P_2\}[\varepsilon]]\!] = \vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{P_1\} S \{P_2\} [\varepsilon].$

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:

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

∎

Theorem 21. *Let S be a statement. Let P_1 and P_2 be assertions. Let ε be effects. Then*

$$\vdash_r \{P_1\}S\{P_2\}[\varepsilon] \text{ iff } \vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{P_1\}S\{P_2\}[\varepsilon]$$

Proof. We first prove that the left hand side implies the right hand side; i.e., that if there is a proof in FRL, then there is a encoded proof in UFRL. The proof is by in the induction on the FRL derivation and by cases on the last rule used. There are 6 base cases.

1. *SKIP:* In this case, we suppose that the FRL proof consists of the axiom $SKIP_r$, i.e., $\vdash_r \{true\}\textbf{skip};\{true\}[\varnothing]$. Then, we must prove that

    $$\vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{true\}\textbf{skip};\{true\}[\varnothing].$$

    The conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom $SKIP_u$, which is $\vdash_u [\varnothing]\{true\}\textbf{skip};\{true\}[\varnothing]$.

2. *VAR:* In this case, we suppose that the FRL proof consists of the axiom VAR_r, which is $\vdash_r \{true\}\textbf{var } x : T;\{x = default(T)\}[\varnothing]$. Then, we must prove that

    $$\vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{true\}\textbf{var } x : T;\{x = default(T)\}[\varnothing].$$

    The conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom VAR_u, which is $\vdash_u [\varnothing]\{true\}\textbf{var } x : T;\{x = default(T)\}[\varnothing]$.

3. *ALLOC:* In this case, we suppose that the FRL proof consists of the axiom $ALLOC_r$, which is $\vdash_r \{true\} x := \textbf{new } C; \{new_r(C,x)\}[\textbf{modifies } x, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})]$. Then, we must prove that

    $$\vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{true\} x := \textbf{new } C; \{new_u(C,x)\}[\textbf{modifies } x, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})].$$

    The conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom $ALLOC_u$, which is $\vdash_u [\varnothing]\{true\} x := \textbf{new } T; \{new_u(C,x)\}[\textbf{modifies } x, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})]$.

4. *UPD:* We suppose that the FRL proof consists of the axiom UPD_r, which is $\vdash_r \{x \neq null\} x.f := E; \{x.f = E\}[\textbf{region}\{x.f\}]$, **where** $x \notin \text{FV}(E)$ Then, we must prove that

    $$\vdash_u [\textbf{\textit{reads alloc}}{\downarrow}]\{x \neq null\} x.f := E; \{x.f = E\}[\textbf{region}\{x.f\}], \textbf{where } x \notin \text{FV}(E). \quad (38)$$

    That conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom UPD_u, which is $\vdash_u [\textbf{reads } x, \eta]\{x \neq null\} x.f := E; \{x.f = E\}[\textbf{region}\{x.f\}]$, where $x \notin \text{FV}(E)$ and $\eta = efs(E)$.

5. *ASGN:* We suppose that the FRL proof consists of the axiom $ASGN_r$: $\vdash_r \{true\}\, x := E;\, \{x = E\}\, [x]$, where $x \notin \mathrm{FV}(E)$. Then, we must prove that

$$\vdash_u [\textbf{reads alloc}\!\downarrow]\{true\}\, x := E;\, \{x = E\}\, [x]\,, \textbf{where}\ x \notin \mathrm{FV}(E)$$

The conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom $ASGN_u$, which is $\vdash_u [\eta]\{true\}\, x := E;\, \{x = E\}\, [x]\,, \textbf{where}\ x \notin \mathrm{FV}(E)$ and $\eta = efs(E)$.

6. *ACC:* We suppose that the FRL proof consists of the axiom ACC_r, which is $\vdash_r \{x' \neq null\}\, x := x'.f;\, \{x = x'.f\}\, [x]$, where $x \neq x'$. Then, we must prove that

$$\vdash_u [\textbf{reads alloc}\!\downarrow]\{x' \neq null\}\, x := x'.f;\, \{x = x'.f\}\, [x]\,, \textbf{where}\ x \neq x'$$

The conclusion is derivable by using the rule $SubEff_u$, with the hypothesis of the UFRL axiom ACC_u, which is $\vdash_u [\eta]\{x' \neq null\}\, x := x'.f;\, \{x = x'.f\}\, [x]$, where $x \neq x'$ and $\eta = efs(x'.f)$.

The inductive hypothesis is that for all substatements S_i, $\vdash_r \{P_i\}S_i\{Q_i\}[\varepsilon_i]$ *iff* $\vdash_u [\textbf{reads alloc}\!\downarrow]\{P_i\}S_i\{Q_i\}[\varepsilon_i]$

1. *IF:* In this case, we suppose that the FRL proof consists of the rule IF_r, which is

$$\frac{\vdash_r \{P\ \&\&\ E \neq 0\}\, S_1\, \{P'\}[\varepsilon] \qquad \vdash_r \{P\ \&\&\ E = 0\}\, S_2\, \{P'\}[\varepsilon]}{\vdash_r \{P\}\textbf{if}\,(E)\,\textbf{then}\,\{S_1\}\,\textbf{else}\,\{S_2\}\{P'\}[\varepsilon]}$$

Then we must prove that

$$\frac{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E \neq 0\}\, S_1\, \{P'\}[\varepsilon]}{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E = 0\}\, S_2\, \{P'\}[\varepsilon]}{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\}\textbf{if}\,(E)\,\textbf{then}\,\{S_1\}\,\textbf{else}\,\{S_2\}\{P'\}[\varepsilon]} \qquad (39)$$

By inductive hypothesis, the two premises Eq. (39) are assumed. After using the rule IF_u, we get

$$\frac{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E \neq 0\}\, S_1\{Q\}\,[\varepsilon] \qquad \vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E = 0\}\, S_2\{Q\}\,[\varepsilon]}{\vdash_u [\textbf{reads alloc}\!\downarrow, \delta_E]\{P\}\,\textbf{if}\,(E)\,\{S_1\}\textbf{else}\{S_2\}\{Q\}\,[\varepsilon]},$$

where $\delta_E = efs(E)$. Then, using the rule $SubEff_u$, we can the conclusion of Eq. (39).

2. *WHILE:* Suppose that the FRL proof consists of the rule $WHILE_r$

$$\frac{\vdash_r \{P\ \&\&\ E \neq 0\}\, S\{P\}\, [\varepsilon, \textbf{fresh}(RE)]}{\vdash_r \{P\}\,\textbf{while}\,(E)\,\{S\}\,\{P\ \&\&\ E = 0\}\,[\varepsilon]},$$
$$\textbf{where}\ \varepsilon\ \text{is fresh-free},\, \varepsilon\ \text{is}\ P/\varepsilon\text{-immune, and}\ \textbf{modifies alloc} \notin \varepsilon$$

Then we must prove that

$$\frac{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E \neq 0\}\, S\{P\}\, [\varepsilon, \textbf{fresh}(RE)]}{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\}\,\textbf{while}\,(E)\,\{S\}\,\{P\ \&\&\ E = 0\}\,[\varepsilon]}, \qquad (40)$$
$$\textbf{where}\ \varepsilon\ \text{is fresh-free},\, \varepsilon\ \text{is}\ P/\varepsilon\text{-immune, and}\ \textbf{modifies alloc} \notin \varepsilon.$$

By the inductive hypothesis, the two premises of Eq. (40) are assume. After using the rule $WHILE_u$, we get

$$\frac{\vdash_u [\textbf{reads alloc}\!\downarrow]\{P\ \&\&\ E \neq 0\}\, S\{P\}\, [\varepsilon, \textbf{fresh}(RE)]}{\vdash_u [\textbf{reads alloc}\!\downarrow, \delta_E]\{P\}\,\textbf{while}\,(E)\,\{S\}\,\{P\ \&\&\ E = 0\}\,[\varepsilon]},$$
$$\textbf{where}\ \varepsilon\ \text{is fresh-free},\, \varepsilon\ \text{is}\ P/\varepsilon\text{-immune, and}\ \textbf{modifies alloc} \notin \varepsilon,$$

where $\delta_E = efs(E)$. Then, the conclusion of Eq. (40) is derivable by using the rule $SubEff_u$.

3. *SEQ1:* We suppose that the FRL proof consists of the rule *SEQ1$_r$*

$$\frac{\vdash_r \{P\}\, S_1\, \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \qquad \vdash_r \{P_1\}\, S_2\, \{P'\}[\varepsilon_2, RE]}{\vdash_r \{P\}\, S_1 S_2\, \{P'\}[\, \varepsilon_1, \varepsilon_2, \mathbf{fresh}(RE)]}$$

where $S_1 \neq \mathbf{var}\ x : T; , \varepsilon_1$ is fresh-free, ε_2 is P/ε_1-immune, and RE is $P_1/(\mathbf{modifies}\ RE, \varepsilon_2)$-immune

Then we must prove that

$$\frac{\vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P\}\, S_1\, \{P_1\}[\varepsilon_1, \mathbf{fresh}(RE)] \quad \vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P_1\}\, S_2\, \{P'\}[\varepsilon_2, RE]}{\vdash_u \{P\}\, S_1 S_2\, \{P'\}[\, \varepsilon_1, \varepsilon_2, \mathbf{fresh}(RE)][\mathbf{reads\ alloc}\!\downarrow]}$$

$$\text{where } \varepsilon_1 \text{ is fresh-free}, \mathbf{reads\ alloc}\!\downarrow \text{ is } P/\epsilon_1\text{-immune}, \varepsilon_2 \text{ is } P/\varepsilon_1\text{-immune and}$$
$$RE \text{ is } P_1/(\mathbf{modifies}\ RE, \varepsilon_2)\text{-immune} \tag{41}$$

By inductive hypothesis, the two premises of Eq. (41) are assumed. To use the rule *SEQ1$_u$*, we check the side conditions $\mathbf{reads\ alloc}\!\downarrow$ is P/ϵ_1-immune. However, it may not be true. Thus, *for all* $x \in \mathrm{mods}(S_1)$ and x in $\mathrm{FV}(P_1)$, there exists z, such that P_1 implies $x = z$ and $z \notin \mathrm{mods}(S_1)$. We substitute z for x in $\mathbf{alloc}\!\downarrow$. Then the second premise of Eq. (117) is re-written as:

$$\vdash_u [\mathbf{reads\ alloc}\!\downarrow [\overline{z}/\mathrm{mods}(S_1)]]\{P_1\}\, S_2\, \{P'[\varepsilon_2]\} \tag{42}$$

Now, we can derive the conclusion of Eq. (41) by using the rule *SEQ1$_u$*.

4. *SEQ2:* We suppose that the FRL proof consists of the rule *SEQ1$_r$*:

$$\frac{\vdash_r \{P\ \&\&\ x = \mathit{default}(T)\} : S\, \{Q\}[\mathbf{modifies}\ x, \varepsilon]}{\vdash_r \{P\}\, \mathbf{var}\ x : T; S\, \{P'\}[\, \varepsilon]}$$

Then, we must prove that

$$\frac{\vdash_u [\delta, \mathbf{reads}\ x]\{P\ \&\&\ x = \mathit{default}(T)\}\, S\, \{Q\}[\, \mathbf{modifies}\ x, \varepsilon]}{\vdash_u [\delta]\{P\}\, \mathbf{var}\ x : T; S\, \{P'\}[\, \varepsilon]} \tag{43}$$

By the inductive hypothesis, the premise of the above equation is assumed. Its conclusion can be derived by using the rule *SEQ2$_u$*.

5. *FRM:* We suppose that the FRL proof consists of the rule *FRM$_r$*:

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash \delta\, \mathit{frm}\, Q}{\vdash_r \{P\ \&\&\ Q\}\, S\, \{P'\ \&\&\ Q\}[\varepsilon]} \ \mathbf{where}\ P\ \&\&\ Q \Rightarrow \delta\,\cdot\!/\!\cdot\varepsilon$$

Then we must prove that

$$\frac{\vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash \delta\, \mathit{frm}\, Q}{\vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P\ \&\&\ Q\}\, S\, \{P'\ \&\&\ Q\}[\varepsilon]} \ \mathbf{where}\ P\ \&\&\ Q \Rightarrow \delta\,\cdot\!/\!\cdot\varepsilon. \tag{44}$$

By inductive hypothesis, the two premises of the above equations are assumed. Then, we can derive the conclusion by using the rule *FRM$_u$*.

6. *SubEff:* We suppose that the FRL proof consists of the rule *SubEff$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash \varepsilon \leqslant \varepsilon'}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon']}$$

Then we must prove that

$$\frac{\vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P\}\, S\, \{P'\}[\varepsilon] \qquad P \vdash \varepsilon \leqslant \varepsilon'}{\vdash_u [\mathbf{reads\ alloc}\!\downarrow]\{P\}\, S\, \{P'\}[\varepsilon']} \tag{45}$$

By the inductive hypothesis, the premise of Eq. (45) is assumed. Then, we can derive the conclusion by using the rule The premise of the conclusion is derivable by using the rule *SubEff$_u$*, because, the side condition $rwR(\varepsilon, \mathbf{reads\ alloc}\!\downarrow) \leqslant rwR(\varepsilon', \mathbf{reads\ alloc}\!\downarrow)$ is true.

7. *CONSEQ:* We suppose that the FRL proof consists of the rule *CONSEQ$_r$*

$$\frac{P_2 \Rightarrow P_1 \qquad \vdash_r \{P_1\}\, S\, \{P_1'\}[\varepsilon] \qquad P_1' \Rightarrow P_2'}{\vdash_r \{P_2\}\, S\{P_2'\}[\varepsilon]}$$

Then, we must prove that

$$\frac{P_2 \Rightarrow P_1 \qquad \vdash_u [\textbf{reads alloc}{\downarrow}]\{P_1\}\, S\, \{P_1'\}[\varepsilon] \qquad P_1' \Rightarrow P_2'}{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P_2\}\, S\{P_2'\}[\varepsilon]} \qquad (46)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *CONSEQ$_u$*.

8. *ConEff:* We suppose that the FRL proof consists of the rule *ConEff$_r$*

$$\frac{\vdash_r \{P \mathrel{\&\&} E \neq 0\}\, S\, \{P'\}[\varepsilon_1] \qquad \vdash_r \{P \mathrel{\&\&} E = 0\}\, S\, \{P'\}[\varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}$$

Then, we must prove that

$$\frac{\begin{array}{c}\vdash_u [\textbf{reads alloc}{\downarrow}]\{P \mathrel{\&\&} E \neq 0\}\, S\, \{P'\}[\varepsilon_1]\\ \vdash_u [\textbf{reads alloc}{\downarrow}]\{P \mathrel{\&\&} E = 0\}\, S\, \{P'\}[\varepsilon_2]\end{array}}{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]} \qquad (47)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *ConEff$_u$*

9. *ConMask1:* We suppose that the FRL proof consists of the rule *ConMask1$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_1]} \textbf{ where } P \Rightarrow \textbf{old}(E) \neq 0$$

Then, we must prove that

$$\frac{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_1]} \textbf{ where } P \Rightarrow \textbf{old}(E) \neq 0 \qquad (48)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *ConMask1$_u$*

10. *ConMask2:* We suppose that the FRL proof consists of the rule *ConMask2$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_2]} \textbf{ where } P \Rightarrow \textbf{old}(E) = 0$$

Then, we must prove that

$$\frac{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then } \varepsilon_1 \textbf{ else } \varepsilon_2]}{\vdash_u [\textbf{reads alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \varepsilon_2]} \textbf{ where } P \Rightarrow \textbf{old}(E) = 0 \qquad (49)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *ConMask2$_u$*.

11. *PostToFr:* We suppose that the FRL proof consists of the rule *PostToFr$_r$*

PostToFr

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon]}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1) \textbf{ else fresh}(RE_2)]}$$
$$\textbf{where } P \Rightarrow (E \neq 0 \mathrel{\&\&} RE_1\,!!\,\textbf{old}(\textbf{alloc})) \text{ and } P \Rightarrow (E = 0 \mathrel{\&\&} RE_2\,!!\,\textbf{old}(\textbf{alloc}))$$

Then, we must prove that

$$\frac{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\varepsilon]}{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1)\, \textbf{else fresh}(RE_2)]} \tag{50}$$
$$\textbf{where } P \Rightarrow (E \neq 0 \,\&\&\, RE_1\,!\,!\,\textbf{old}(\textbf{alloc})) \text{ and } P \Rightarrow (E = 0 \,\&\&\, RE_2\,!\,!\,\textbf{old}(\textbf{alloc}))$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *PostToFr$_u$*.

12. *FrToPost:* We suppose that the FRL proof consists of the rule *FrToPost$_r$*

$$\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1)\, \textbf{else fresh}(RE_2)]$$

$$\vdash_r \begin{array}{l} \{P\} \\ S \\ \{P' \,\&\&\, (\textbf{old}(E) \neq 0 \Rightarrow RE_1\,!\,!\,\textbf{old}(\textbf{alloc})) \,\&\&\, (\textbf{old}(E) = 0 \Rightarrow RE_2\,!\,!\,\textbf{old}(\textbf{alloc}))\} \\ [\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1)\, \textbf{else fresh}(RE_2)] \end{array}$$

Then, we must prove that

$$\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1)\, \textbf{else fresh}(RE_2)]$$

$$\vdash_u \begin{array}{l} [\textbf{reads alloc↓}] \\ \{P\} \\ S \\ \{P' \,\&\&\, (\textbf{old}(E) \neq 0 \Rightarrow RE_1\,!\,!\,\textbf{old}(\textbf{alloc})) \,\&\&\, (\textbf{old}(E) = 0 \Rightarrow RE_2\,!\,!\,\textbf{old}(\textbf{alloc}))\} \\ [\varepsilon, \textbf{if } E \textbf{ then fresh}(RE_1)\, \textbf{else fresh}(RE_2)] \end{array} \tag{51}$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *FrToPost$_u$*.

13. *VarMask1*: We suppose that the FRL proof consists of the rule *VarMask1$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } x, \varepsilon_1\, \textbf{else } \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow E \neq 0, P \vee P' \Rightarrow x = y \text{ and } P \,\&\&\, \textbf{old}(E) \neq 0 \Rightarrow \textbf{reads } y\,/\cdot(x, \varepsilon)$$

Then, we must prove that

$$\frac{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } x, \varepsilon_1\, \textbf{else } \varepsilon_2]}{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } \varepsilon_2]} \tag{52}$$
$$\textbf{where } P \Rightarrow E \neq 0, P \vee P' \Rightarrow x = y \text{ and } P \,\&\&\, \textbf{old}(E) \neq 0 \Rightarrow \textbf{reads } y\,/\cdot(x, \varepsilon)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *VarMask1$_u$*.

14. *VarMask2*: We suppose that the FRL proof consists of the rule *VarMask2$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } x, \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } \varepsilon_2]}$$
$$\textbf{where } P \Rightarrow E = 0, P \vee P' \Rightarrow x = y \text{ and } P \,\&\&\, \textbf{old}(E) = 0 \Rightarrow \textbf{reads } y\,/\cdot(x, \varepsilon)$$

Then, we must prove that

$$\frac{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } x, \varepsilon_2]}{\vdash_u [\textbf{reads alloc↓}]\{P\}\, S\, \{P'\}[\textbf{if } E \textbf{ then } \varepsilon_1\, \textbf{else } \varepsilon_2]} \tag{53}$$
$$\textbf{where } P \Rightarrow E = 0, P \vee P' \Rightarrow x = y \text{ and } P \,\&\&\, \textbf{old}(E) = 0 \Rightarrow \textbf{reads } y\,/\cdot(x, \varepsilon)$$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *VarMask2$_u$*.

15. *FieldMask1*: We suppose that the FRL proof consists of the rule *FieldMask1$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \mathbf{region}\{x.f\}, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]}$$

where $P \Rightarrow E \neq 0, P \vee P' \Rightarrow x.f = y, P'\, \&\&\, \mathbf{old}(E) \neq 0 \Rightarrow \mathbf{reads}\, x \mathbf{\cdot}\, \mathbf{modifies}\, \varepsilon$

and $P'\, \&\&\, \mathbf{old}(E) \neq 0 \Rightarrow \mathbf{reads}\, y \mathbf{\cdot} \mathbf{modifies}\, \varepsilon$

Then, we must prove that

$$\frac{\vdash_u [\mathbf{reads}\, \mathbf{alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \mathbf{region}\{x.f\}, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]}{\vdash_u [\mathbf{reads}\, \mathbf{alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]} \tag{54}$$

where $P \Rightarrow E \neq 0, P \vee P' \Rightarrow x.f = y, P'\, \&\&\, \mathbf{old}(E) \neq 0 \Rightarrow \mathbf{reads}\, x \mathbf{\cdot}\, \mathbf{modifies}\, \varepsilon$

and $P'\, \&\&\, \mathbf{old}(E) \neq 0 \Rightarrow \mathbf{reads}\, y \mathbf{\cdot} \mathbf{modifies}\, \varepsilon$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *FieldMask1$_u$*.

16. *FieldMask2*: We suppose that the FRL proof consists of the rule *FieldMask2$_r$*

$$\frac{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \mathbf{region}\{x.f\}, \varepsilon_2]}{\vdash_r \{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]}$$

where $P \Rightarrow E = 0, P \vee P' \Rightarrow x.f = y$ and $P'\, \&\&\, \mathbf{old}(E) = 0 \Rightarrow \mathbf{reads}\, x \mathbf{\cdot}\, \mathbf{modifies}\, \varepsilon$

and $P'\, \&\&\, \mathbf{old}(E) = 0 \Rightarrow \mathbf{reads}\, y \mathbf{\cdot} \mathbf{modifies}\, \varepsilon$

Then we must prove that

$$\frac{\vdash_u [\mathbf{reads}\, \mathbf{alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \mathbf{region}\{x.f\}, \varepsilon_2]}{\vdash_u [\mathbf{reads}\, \mathbf{alloc}{\downarrow}]\{P\}\, S\, \{P'\}[\varepsilon, \mathbf{if}\, E\, \mathbf{then}\, \varepsilon_1\, \mathbf{else}\, \varepsilon_2]} \tag{55}$$

where $P \Rightarrow E = 0, P \vee P' \Rightarrow x.f = y$ and $P'\, \&\&\, \mathbf{old}(E) = 0 \Rightarrow \mathbf{reads}\, x \mathbf{\cdot}\, \mathbf{modifies}\, \varepsilon$

and $P'\, \&\&\, \mathbf{old}(E) = 0 \Rightarrow \mathbf{reads}\, y \mathbf{\cdot} \mathbf{modifies}\, \varepsilon$

By the inductive hypothesis, the premise of the above equation is assumed. Then, its conclusion can be derived by using the rule *FieldMask2$_u$*.

Next, we prove it from the right side of the left side. It means that if there is a proof in UFRL with read effect **reads alloc**$\downarrow$, then there is a proof in FRL. It is true because we can always approximate the read effects of any proofs by using the rule *SubEff$_u$*:

$$\frac{\vdash_u [\eta]\{P_1\}\, S\, \{P_2\}[\varepsilon]}{\vdash_u [\mathbf{reads}\, \mathbf{alloc}{\downarrow}]\{P_1\}\, S\, \{P_2\}[\varepsilon]} \tag{56}$$

$\square$

Corollary 22. *The meaning of a FRL judgment is preserved by the syntactic mapping.*

Corollary 23. *Let S be a statement. Let P_1 and P_2 be assertions. Let ε be effects and η be read effects. Then*

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

The proof uses the subeffect rule to convert η into **reads alloc**$\downarrow$ and then applies Theorem 21 on page 33.

9   Semantic Connection Between SL and UFRL

To understand the relationship between SL and UFRL, we connect their semantics by defining the semantics of SL in terms of a heap and a region. This section is inspired by Parkinson and Summers' work [51]. They connect the semantics of separation logic and implicit dynamic frames [57] by a "total heap semantics" [51]. However, our heap is a partial function.

9.1   Separation Logic Review

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 disjoint parts of the current heap. The separating implication, $a_1 \mathrel{-\!*} a_2$, 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 24 (SL Assertions). *Let x be a variable and f be a field name. The syntax of* assertions in separation logic *is as follows:*

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

The semantics given below assumes that expressions and assertions are properly typed. Expressions are pure, meaning that they are independent of the heap. We consider intuitionistic separation logic [26,50]. Recall that its semantics [17,26,50] is as follows.

Definition 25 (SL 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:*

$$\mathcal{E}_s : e \to Store \to Value$$
$$\mathcal{E}_s[\![x]\!](\sigma) = \sigma(x) \quad \mathcal{E}_s[\![n]\!](\sigma) = \mathcal{N}[\![n]\!] \quad \mathcal{E}_s[\![\texttt{null}]\!](\sigma) = null$$

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

$$\mathcal{E}_a : a \to Store \times Heap \to \{true,\ false\}$$
$$\mathcal{E}_a[\![e = e']\!](\sigma, h) = \mathcal{E}_s[\![e]\!](\sigma) = \mathcal{E}_s[\![e']\!](\sigma)$$
$$\mathcal{E}_a[\![x.f \mapsto e]\!](\sigma, h) = (\mathcal{E}_s[\![x]\!](\sigma), f) \in dom(h) \text{ and } h[\mathcal{E}_s[\![x]\!](\sigma), f] = \mathcal{E}_s[\![e]\!](\sigma)$$
$$\mathcal{E}_a[\![a_1 * a_2]\!](\sigma, h) = exists\ h_1, h_2 :: (h_1 \bot h_2 \text{ and } h = h_1 \cdot h_2 \text{ and } \mathcal{E}_a[\![a_1]\!](\sigma, h_1) \text{ and } \mathcal{E}_a[\![a_2]\!](\sigma, h_2)$$
$$\mathcal{E}_a[\![a_1 \mathrel{-\!*} a_2]\!](\sigma, h) = for\ all\ h' :: (h' \bot h \text{ and } \mathcal{E}_a[\![a_1]\!](\sigma, h') \text{ implies } \mathcal{E}_a[\![a_2]\!](\sigma, h \cdot h'))$$
$$\mathcal{E}_a[\![a_1 \wedge a_2]\!](\sigma, h) = \mathcal{E}_a[\![a_1]\!](\sigma, h) \text{ and } \mathcal{E}_a[\![a_2]\!](\sigma, h)$$
$$\mathcal{E}_a[\![a_1 \vee a_2]\!](\sigma, h) = \mathcal{E}_a[\![a_1]\!](\sigma, h) \text{ or } \mathcal{E}_a[\![a_2]\!](\sigma, h)$$
$$\mathcal{E}_a[\![a_1 \Rightarrow a_2]\!](\sigma, h) = for\ all\ h' :: (h' \bot h \text{ and } \mathcal{E}_a[\![a_1]\!](\sigma, h \cdot h') \text{ implies } \mathcal{E}_a[\![a_2]\!](\sigma, h \cdot h'))$$
$$\mathcal{E}_a[\![\exists x.a]\!](\sigma, h) = exists\ v :: \mathcal{E}_a[\![a]\!](\sigma[x \mapsto v], h)$$

The satisfaction relation is defined by $\sigma, h \models_s a$ iff $\mathcal{E}_a[\![a]\!](\sigma, h)$. $\qquad\blacksquare$

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.

9.2   Semantic connection

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 ($\upharpoonright$) 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 $r \subseteq dom(H)$, define $\sigma, H \upharpoonright r \models_{sl} a$ if and only if $\sigma, (H \upharpoonright r) \models_s a$, however for clarity we use the following definitions of validity for separating conjunction and implications.

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.

$$\sigma, H \upharpoonright r \models_{sl} a_1 * a_2 \text{ iff } exists\ r_1, r_2 :: (r_1 \cap r_2 = \varnothing \text{ and } r = r_1 \cup r_2 \text{ and } \sigma, H \upharpoonright r_1 \models_{sl} a_1 \text{ and } \sigma, H \upharpoonright r_2 \models_{sl} a_2)$$

The semantics for the magic wand and logical implication consider all possible extensions of the partial heap $H{\upharpoonright}r$. The extensions are not necessarily disjoint with the heap H, but must be disjoint with the subheap $H{\upharpoonright}r$, so that the extended heap $h'{\upharpoonright}r'$ is disjoint; this is guaranteed when $r' \cap r = \varnothing$.

$\sigma, H{\upharpoonright}r \models_{sl} a_1 \mathbin{-\!\!*} a_2$ *iff for all* $h', r' :: (r' \cap r = \varnothing$ *and* $\sigma, h'{\upharpoonright}r' \models_{sl} a_1$ *implies* $\sigma, (H{\upharpoonright}r \cup h'{\upharpoonright}r') \models_{sl} a_2)$

$\sigma, H{\upharpoonright}r \models_{sl} a_1 \Rightarrow a_2$ *iff for all* $h', r' :: (r' \cap r = \varnothing$ *and* $\sigma, (H{\upharpoonright}r \cup h'{\upharpoonright}r') \models_{sl} a_1$ *implies* $\sigma, (H{\upharpoonright}r \cup h'{\upharpoonright}r') \models_{sl} a_2)$

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

Theorem 26. *Let σ be a store, h and H be heaps, and r be a region, such that $r \subseteq dom(H)$ and $h = H{\upharpoonright}r$, then $\sigma, h \models_s a$ iff $\sigma, H{\upharpoonright}r \models_{sl} a$.*

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 26. Therefore, in the next section we find another candidate for r that is more constructive.

10 Supported Separation Logic and Encoding Assertions

This section shows that the semantic footprint is another candidate for the region r needed in Theorem 26. 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 [47].

10.1 Semantic Footprints

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 $\{(\sigma(x), f) \mapsto \mathcal{E}_s[\![e]\!](\sigma)\}$. Since we are using intuitionistic SL, this heap is the greatest lower bound (glb) of the heaps in which the assertion holds. We now define the semantic footprint for SL assertions that capture this glb. We say 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:

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

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

Definition 27 (Semantic Footprint). *Let a be an assertion in SL, and (σ, h) be a state. Then $\texttt{MinReg}(a, \sigma, h)$ is the semantic footprint of a if and only if $\sigma, (h{\upharpoonright}\texttt{MinReg}(a, \sigma, h)) \models_s a$. In this case we say 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. [47]. We call this fragment supported separation logic (SSL). This is the biggest subset of the syntax in Def. 24, 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 [12,13,14,15,20,22,51].[5] To avoid introducing new notations, we reuse the syntax of separation logic (Def. 24). From now on, those notations mean supported separation logic.

---

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

Definition 28 (Supported Separation Logic). *The syntax of supported separation logic has expressions (e), Boolean expressions (b) and assertions (a) defined as follows:*

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

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 and r_2, must satisfy $r_1 \cup r_2 = dom(h)$, which is required by its semantics. Picking $h \upharpoonright r_1$ as the witness for h_1 pushes the proof to take $h \upharpoonright (dom(h) - r_1)$ as the witness for h_2. Lemma 29 below can be applied in this scenario as $h \upharpoonright r_2 \subseteq h \upharpoonright (dom(h) - r_1)$, as $r_1 \subseteq dom(h)$ and $r_2 \subseteq dom(h)$.

Lemma 29. *Let a be an SSL assertion, and (σ, h) be a state. Let h' be a heap, such that $h \subseteq h'$. Then $\sigma, h \models_s a \Rightarrow \sigma, h' \models_s a$.*

The semantic footprints for assertions in SSL are derived in Lemma 30 based on the SL semantics in terms of a heap and a region.

Lemma 30. *Let (σ, h) be a state, and let e, b and a be an SSL expression, a Boolean expression, and an assertion. Then:*

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

Moreover, $\sigma, h \models_s a$ iff $\sigma, (h \upharpoonright MinReg(a, \sigma, h)) \models_s a$.

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 30, and the converse can be proved using Lemma 29.

10.2 Supported Assertions in SL

O'Hearn et al. [47] note that for the soundness of proofs under hypothesis, assertions used in preconditions and resource invariants need to be *supported* (Theorem 26 [47, 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. [47].

Definition 31 (Supported). *An assertion a is supported if and only if for all states (σ, h), when h has a subheap $h_0 \subseteq h$ such that $\sigma, h_0 \models_s a$, then there is at least subheap $h_a \subseteq h$ with $\sigma, h_a \models_s a$ such that for all subheaps $h' \subseteq h$, if $\sigma, h' \models_s a$, 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 $\sigma, h_1 \models_s a$ and $\sigma, h_2 \models_s a$, if $h_a = h_1 \cap h_2$ and $\sigma, h_a \models_s a$, 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 $r_1 \subseteq dom(h)$ and $r_2 \subseteq dom(h)$, and a separation logic assertion a, such that $\sigma, (h \upharpoonright r_1) \models_s a$ and $\sigma, (h \upharpoonright r_2) \models_s a$. Let r be the glb of r_1 and r_2, such that such that $r \subseteq r_1 \cap r_2$. If $\sigma, (h \upharpoonright r) \models_s a$, then a has a semantic footprint. The following theorem summaries this. The proof is found in Appendix A.

Theorem 32. *An assertion in SL is* supported *if and only if it has semantic footprint.*

SSL assertions are supported by Theorem 32. This property provides the soundness of the hypothetical frame rule for Hoare triple judgment under certain hypothesis [46,47]. See the discussion in Section 14.2.

10.3   Encoding SSL

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

Definition 33 (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.*

$$
\begin{aligned}
fpt_s(b) &= \textbf{region}\{\} \\
fpt_s(x.f \mapsto e) &= \textbf{region}\{x.f\} \\
fpt_s(b \Rightarrow a) &= \textbf{if } b \textbf{ then } fpt_s(a) \textbf{ else region}\{\} \\
fpt_s(a_1 * a_2) &= fpt_s(a_1) + fpt_s(a_2) \\
fpt_s(a_1 \wedge a_2) &= fpt_s(a_1) + fpt_s(a_2) \\
fpt_s(\exists x.(y.f = x * a)) &= \textbf{region}\{y.f\} + fpt_s(a)[y.f/x]
\end{aligned}
$$

∎

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 34 (Mapping from SSL to UFRL). *We define a function TR that syntactically maps from SSL to UFRL as follows:*

$$
\begin{aligned}
TR[\![x]\!] = x \quad TR[\![n]\!] = n \quad & TR[\![null]\!] = null \\
TR[\![e_1 = e_2]\!] &= TR[\![e_1]\!] = TR[\![e_2]\!] \\
TR[\![e_1 \neq e_2]\!] &= TR[\![e_1]\!] \neq TR[\![e_2]\!] \\
TR[\![x.f \mapsto e]\!] &= TR[\![x]\!].f = TR[\![e]\!] \\
TR[\![a_1 * a_2]\!] &= TR[\![a_1]\!] \&\& TR[\![a_2]\!] \&\& (fpt_s(a_1) \,!\, !fpt_s(a_2)) \\
TR[\![a_1 \wedge a_2]\!] &= TR[\![a_1]\!] \&\& TR[\![a_2]\!] \\
TR[\![b \Rightarrow a]\!] &= TR[\![b]\!] \Rightarrow TR[\![a]\!] \\
TR[\![\exists x.(y.f \mapsto x * a)]\!] &= \exists x.(TR[\![y.f \mapsto x]\!] \&\& TR[\![a]\!] \&\& (\textbf{region}\{y.f\} \,!\, !fpt_s(a)))
\end{aligned}
$$

∎

Lemma 35 and Lemma 36 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 $TR[\![e]\!]$, as well as b and $TR[\![b]\!]$ can be used interchangeably.

Lemma 35. *Let σ be a store. Let e be an expression in SSL. Then $\mathcal{E}_s[\![e]\!](\sigma) = \mathcal{E}[\![TR[\![e]\!]]\!](\sigma)$.*

Lemma 36. *Let (σ, h) be a state, and H be a heap such that $h \subseteq H$. Let b be a pure assertion in SSL. Then $\sigma, h \models_s b$ iff $\sigma, h \models_u TR[\![b]\!]$ iff $\sigma, H \models_u TR[\![b]\!]$.*

The following theorem shows that the semantics of the semantic footprint function, $fpt_s(a)$, 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 we henceforth just call semantic footprint the "footprint".

Theorem 37. *Let a be an assertion in SSL, and let (σ, h) be a state. If $\sigma, h \models_s a$, then a has a semantic footprint in (σ, h), and this semantic footprint is $\texttt{MinReg}(a, \sigma, h) = \mathcal{E}[\![fpt_s(a)]\!](\sigma)$.*

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 40.

Corollary 38. *Let a be an assertion in SSL. Let (σ, h) be a state. If $\sigma, h \models_s a$, then $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(h)$.*

The following corollary shows that $\mathcal{E}[\![fpt_s(a)]\!](\sigma)$ is another candidate for the region r needed in Theorem 26. As $fpt_s(a)$ gives the semantic footprint for each a, the corollary can be proved by Lemma 30 and Theorem 37.

Corollary 39. *Let a be an assertion in SSL. Let (σ, h) be a state. Then $\sigma, h \models_s a$ iff $\sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_s a$.*

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 B. 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 40. *Let a be an assertion in SSL. Then $\sigma, h \models_s a$ iff $\sigma, h \models_u TR[\![a]\!]$.*

10.4   Summary and Conclusions

Fig. 25 summarizes our results. We find the r for the SL's semantics in Section 9.2, which is $\mathcal{E}[\![fpt_s(a)]\!](\sigma)$; since $\sigma, h \models_s a$ if and only if $\sigma, H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_s a$, it must be that $h = H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma))$. In addition, by Corollary 39, we have $\sigma, h \models_s a$ iff $\sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_s a$. Furthermore, by Theorem 40 twice, we have $\sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_s a$ iff $\sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_u TR[\![a]\!]$, and $\sigma, h \models_s a$ iff $\sigma, h \models_u TR[\![a]\!]$. Therefore, by transitivity, we have $\sigma, h \models_u TR[\![a]\!]$ iff $\sigma, H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_u TR[\![a]\!]$ iff $\sigma, H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_{sl} a$.

$$\sigma, h \models_s a \xleftrightarrow{\text{Corollary 39}} \sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_s a \xleftrightarrow{\text{Theorem 26}} \sigma, H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_{sl} a$$

$$\text{Theorem 40} \updownarrow \qquad\qquad \text{Theorem 40} \updownarrow$$

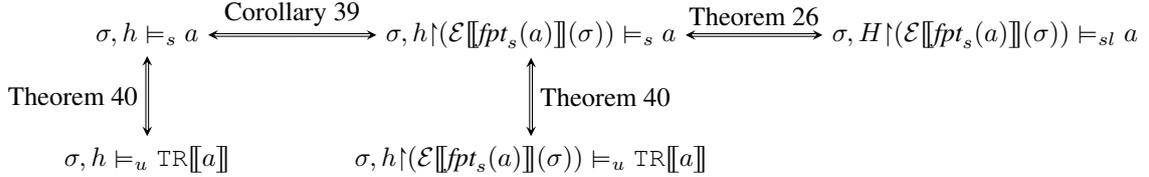$$\sigma, h \models_u TR[\![a]\!] \qquad\qquad \sigma, h{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \models_u TR[\![a]\!]$$

Fig. 25: A summary of results, where $h = H{\restriction}(\mathcal{E}[\![fpt_s(a)]\!](\sigma))$.

We need to translate each assertion in SSL into the one in UFRL that preserves its value in a heap that is an extension of the partial heap used in the semantics of SSL. That requires us to prove that assertion's value is closed under heap extension. Unfortunately, this does not hold in general. Consider $x.f \mapsto 5$ which is false in a state where $(\sigma(x), f) \notin dom(h)$. But its value is preserved if $dom(h)$ contains $(\sigma(x), f)$. So if $(\sigma(x), f) \in dom(h)$ is assumed, then also the value false is preserved under heap extension. The location $(\sigma(x), f)$ is the semantic footprint of $x.f \mapsto 5$. Thus, the necessary hypothesis of preserving the value of an assertion is the existence of the assertion's semantic footprint. The following theorem shows the hypothesis under which the value of an assertion is preserved by the translation.

Theorem 41. *Let a be an assertion in SSL. Let (σ, h) be a state, and H be a heap, such that $h \subseteq H$. If $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(h)$, then $\sigma, h \models_s a$ iff $\sigma, H \models_u TR[\![a]\!]$.*

The theorem shows that when the heap contains the locations that an assertion depends on, then the validity of the assertion is closed under heap extension.

11   Encoding SSL Proofs

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.

11.1 SSL Proofs Review and Approach

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. We consider the region guaranteed by a as its implicit frame. Thus we will prove that the following encoding into UFRL is valid (in Section 11.3):

$$
\vdash_s \{a\}S\{a'\} \text{ iff } \vdash_u \begin{array}{l} [\mathbf{reads}\, fpt_s(a)] \\ \{\mathrm{TR}[\![a]\!]\,\&\&\,r = fpt_s(a)\} \\ S \\ \{\mathrm{TR}[\![a']\!]\} \\ [\,\mathbf{modifies}\,(\mathrm{mods}(S), fpt_s(a)), \mathbf{fresh}(fpt_s(a') - r)] \end{array} \tag{57}
$$
$$\mathbf{where}\ r\ \text{is fresh and}\ r \notin \mathrm{mods}(S)$$

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

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

Definition 42 (Validity of SSL Hoare-formula). *Let S be a statement. Let a and a' be assertions in SSL. Let (σ, H) be a state. Then $\{a\}S\{a'\}$ is valid in (σ, H), written $\sigma, H \models_s \{a\}S\{a'\}$, if and only if whenever $\sigma, H \models_s a$, then $\mathcal{MS}[\![S]\!](\sigma, H) \neq err$ and if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H)$, then $\sigma', H' \models_s a'$.*
A SSL Hoare-formula $\{a\}S\{a'\}$ is valid, written $\models_s \{a\}S\{a'\}$, if and only if for all states $(\sigma, H) :: \sigma, H \models_s \{a\}S\{a'\}$. ∎

The *locality properties* [47,61] of SSL Hoare-formula are:

1. *Safety Monotonicity:* for all states (σ, H) and heaps H', such that $H \bot H'$, if $\mathcal{MS}[\![S]\!](\sigma, H) \neq err$, then $\mathcal{MS}[\![S]\!](\sigma, H \cdot H') \neq err$.
2. *Termination Monotonicity:* for all states (σ, H) and heaps H', such that $H \bot H'$, if $\mathcal{MS}[\![S]\!](\sigma, H)$ terminates normally, then $\mathcal{MS}[\![S]\!](\sigma, H \cdot H')$ terminates normally.
3. *Frame Property:* for all states (σ, H_0) and heaps H_1, such that $H_0 \bot H_1$, if $\mathcal{MS}[\![S]\!](\sigma, H_0) \neq err$ and $\mathcal{MS}[\![S]\!](\sigma, H_0 \cdot H_1) = (\sigma', H')$, then there is a subheap $H_0' \subseteq H'$ such that $H_0' \bot H_1$, $H_0' \cdot H_1 = H'$, and $\mathcal{MS}[\![S]\!](\sigma, H_0) = (\sigma', H_0')$.

Hoare-style proof rules for SSL are found in Fig. 26 on the next page, following Parkinson's work [50]. The type environment Γ is omitted. In the figure, the shorthand $new_s(C, x)$ 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 $(f_1 : T_1, \ldots, f_n : T_n) = fieldsC$. We use SSL expressions (e) instead of FRL expressions (E) in the syntax of the statements, although the statements of SSL are those of FRL, the expressions have the same syntax and meaning, by Lemma 35.

The following lemma states the frame property of SL Hoare-formulas semantically. It is used in the proof of Lemma 47 later. The proof is found in Appendix C.

Lemma 43. *Let a and a' be assertions and S be a statement, such that $\models_s \{a\}S\{a'\}$. Let (σ, H) be an arbitrary state. If $\sigma, H \models_s a$ and $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$, then:*

1. *for all $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ implies $x \in \mathrm{mods}(S)$.*
2. *for all $(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]$ implies $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$.*
3. *for all $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$.*

11.2 Some Properties

This subsection states several lemmas connecting the FRL and UFRL separation operator $(\cdot/\cdot)$ to SL's separating conjunction operator $(*)$. These lemmas are used to prove the frame rule case of our Theorem that the translation between SSL and UFRL preserves provability (Theorem 49 in Section 11.3).

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

$(SKIP_s)$ $\vdash_s \{true\}\textbf{skip};\{true\}$

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

$(ALLOC_s)$ $\vdash_s \{a\}\ x := \textbf{new}\ C;\ \{a * new_s(C, x)\}, \textbf{where}\ x \notin \mathrm{FV}(a)$

$(ASGN_s)$ $\vdash_s \{true\}\ x := e;\ \{x = e\}, \textbf{where}\ x \notin \mathrm{FV}(e)$

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

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

(IF_s) $\dfrac{\vdash_s \{a \land e \neq 0\}\ S_1\ \{a'\}, \qquad \vdash_s \{a \land e = 0\}\ S_2\ \{a'\}}{\vdash_s \{a\}\ \textbf{if}\ e\ \{S_1\}\textbf{else}\{S_2\}\ \{a'\}}$

$(WHILE_s)$ $\dfrac{\vdash_s \{I \land e \neq 0\}\ S\ \{I\}}{\vdash_s \{I\}\ \textbf{while}\ e\ \{S\}\ \{I \land e = 0\}}$

(SEQ_s) $\dfrac{\vdash_s \{a\}\ S_1\ \{b\}, \quad \vdash_s \{b\}\ S_2\ \{a'\}}{\vdash_s \{a\}\ S_1 S_2\ \{a'\}}$

$\mathrm{mods}(\textbf{skip};) = \varnothing \qquad \mathrm{mods}(\textbf{var}\ x : T;) = \varnothing \qquad \mathrm{mods}(x := \textbf{new}\ C;) = \{x\}$
$\mathrm{mods}(x := e;) = \{x\} \qquad \mathrm{mods}(x.f := e;) = \varnothing \qquad \mathrm{mods}(x := x'.f;) = \{x\}$
$\mathrm{mods}(\textbf{if}\ e\ \textbf{then}\ \{S_1\}\textbf{else}\ \{S_2\}) = \mathrm{mods}(S_1) \cup \mathrm{mods}(S_2)$
$\mathrm{mods}(\textbf{while}\ e\ \{S\}) = \mathrm{mods}(S) \qquad \mathrm{mods}(S_1 S_2) = \mathrm{mods}(S_1) \cup \mathrm{mods}(S_2)$

$\mathrm{FV}(x) = \{x\} \qquad \mathrm{FV}(\textbf{null}) = \varnothing \qquad \mathrm{FV}(n) = \varnothing \qquad \mathrm{FV}(e_1 = e_2) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)$
$\mathrm{FV}(e_1 \neq e_2) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \qquad \mathrm{FV}(x.f \mapsto e) = \{x\} \cup \mathrm{FV}(e) \qquad \mathrm{FV}(a_1 * a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$
$\mathrm{FV}(a_1 \land a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \qquad \mathrm{FV}(b \Rightarrow a) = \mathrm{FV}(b) \cup \mathrm{FV}(a)$
$\mathrm{FV}(\exists\ x.y.f = x * a) = (\{y\} \cup \mathrm{FV}(a)) - \{x\}$

Fig. 26: Correctness rules and axioms for statements in SSL [50]. The type environment Γ is omitted.

Lemma 44. *Let a_1 and a_2 be assertions in SSL. Then*

$$\sigma, h \models_s a_1 * a_2 \text{ implies } \sigma, h \models_u \text{efs}(\text{TR}[\![a_2]\!]) \cdot / \textbf{modifies } \text{fpt}_s(a_1)$$

Informally, the proof goes as follows. By the semantics of separating conjunction, we know that a_1 and a_2 hold on disjoint heaps, say h_1 and h_2, respectively. By Corollary 38, we know that $\mathcal{E}[\![\text{fpt}_s(a_1)]\!](\sigma) \subseteq \text{dom}(h_1)$. So we have

$$\text{for all } \textbf{reads } RE \leqslant \text{efs}(\text{TR}[\![a_2]\!]) :: RE\,!\,!\text{fpt}_s(a_1). \tag{58}$$

In addition, by definition of separator (Fig. 20), we have

$$\text{for all } \textbf{reads } X \leqslant \text{efs}(\text{TR}[\![a_2]\!]) :: \textbf{reads } X \cdot / \textbf{modifies } \text{fpt}_s(a_1). \tag{59}$$

Using Eq. (59) together with Eq. (58) and the definition of separator (Fig. 20), proves that $\text{efs}(\text{TR}[\![a_2]\!]) \cdot /$ **modifies** $\text{fpt}_s(a_1)$.

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

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

Lemma 46. *Let c be an assertion in SSL, then $\text{readVar}(\text{efs}(\text{TR}[\![c]\!])) \subseteq FV(\text{TR}[\![c]\!])$.*

11.3   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 $\text{fpt}_s(a)$, write effect $(\text{fpt}_s(a), \text{mods}(S))$ and fresh effect $(\text{fpt}_s(a') - r)$, where r snapshots the set of locations of $\text{fpt}_s(a)$ 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 D.

Theorem 47. *Let S be a statement, and let a and a' be assertions in SSL, such that $\models_s \{a\}S\{a'\}$. Let r be a region variable. Let (σ, H) be an arbitrary state. Then*

$$
\vdash_s \{a\}S\{a'\} \text{ iff } \vdash_u
\begin{array}{l}
[\textbf{reads } \text{fpt}_s(a)] \\
\{\text{TR}[\![a]\!] \text{ \&\& } r = \text{fpt}_s(a)\} \\
S \\
\{\text{TR}[\![a']\!]\} \\
[\textbf{modifies } (\text{mods}(S), \text{fpt}_s(a)), \textbf{fresh}(\text{fpt}_s(a') - r)]
\end{array}
$$

$$\textbf{where } r \text{ is fresh and } r \notin \text{mods}(S)$$

Def. 48 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 48 (Syntactic Mapping from SSL to UFRL)**.** *Let a and a' be assertions in SSL. We define a syntactic mapping $\text{TR}_s[\![-]\!]$ from SSL axioms and rules to those of UFRL below:*

$$TR_s[\![\vdash_s \{a\}\ x := \textbf{\textit{new}}\ C; \{a * new_s(C,x)\}]\!] =$$
$$[\textbf{reads}\ fpt_s(a)]$$
$$\vdash_u \{TR[\![a]\!]\}\ x := \textbf{\textit{new}}\ C;\ \{TR[\![a * new_s(C,x)]\!]\}$$
$$[\textbf{modifies}\ x,\ \textbf{modifies alloc},\ \textbf{fresh}(fpt_s(new_s(C,x)))]$$

$$TR_s[\![\vdash_s \{a\}\ S\ \{a'\}]\!] =$$
$$[\textbf{reads}\ fpt_s(a)]$$
$$\vdash_u \{TR[\![a]\!]\ \&\&\ r = fpt_s(a)\}\ S\ \{TR[\![a']\!]\}$$
$$[\textbf{modifies}\ (fpt_s(a), mods(S)),\ \textbf{fresh}(fpt_s(a') - r)]$$

where r *is fresh*, $r \notin mods(S)$ *and* $S \neq x := \textbf{\textit{new}}\ C;$.

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:

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

∎

Theorem 49. *Each translated SSL axiom is derivable, and each translated rule is admissible in the UFRL proof system.*

The proof is by the induction on the derivation and by cases in the last rule used, and can be found in Appendix E. The sequential case is not intuitive. We use an example to show that how to use $SEQ1_u$ to prove that the encoded sequence rule is admissible in UFRL. Particularly, we explain the proof strategy of proving the side conditions on immunity. Consider the example $x := y; x.f := 5; x.f := 6;$. We assume $y.f \mapsto 3$ before executing the first statement. In the proof, we have the following derivation in SSL.

$$\frac{\vdash_s \{y.f \mapsto 3\}x := y; x.f := 5; \{x = y * x.f \mapsto 5\} \qquad \vdash_s \{x = y * x.f \mapsto 5\}x.f := 6; \{x = y * x.f \mapsto 6\}}{\vdash_s \{y.f \mapsto 3\}x := y; x.f := 5; x.f := 6; \{x = y * x.f \mapsto 6\}} (SEQ_s)$$

By Def. 48, the two premises are encoded to

$$[\textbf{reads region}\{y.f\}]$$
$$\vdash_u \{y.f = 3\ \&\&\ r = \textbf{region}\{y.f\}\}\ x := y; x.f := 5;\ \{x = y\ \&\&\ x.f = 5\} \qquad (60)$$
$$[\textbf{modifies}\ x, \textbf{modifies region}\{y.f\}, \textbf{fresh}(\textbf{region}\{x.f\} - r)]$$

$$[\textbf{reads region}\{x.f\}]$$
$$\vdash_u \{x = y\ \&\&\ x.f = 5\ \&\&\ r' = \textbf{region}\{x.f\}\}x.f := 6; \{x = y\ \&\&\ x.f = 6\} \qquad (61)$$
$$[\textbf{modifies region}\{x.f\}, \textbf{fresh}(\textbf{region}\{x.f\} - r')]$$

And we want to show that from Eq. (60) and Eq. (61), the translated conclusion below can be derived.

$$[\textbf{reads region}\{y.f\}]$$
$$\vdash_u \{y.f = 3\ \&\&\ r = \textbf{region}\{y.f\}\}\ x := y; x.f := 5; x.f := 6;\ \{x = y\ \&\&\ x.f = 6\} \qquad (62)$$
$$[\textbf{modifies}\ x, \textbf{modifies region}\{y.f\}, \textbf{fresh}(\textbf{region}\{x.f\} - r)]$$

The immune side conditions are not satisfied. However, according to the postcondition of Eq. (60), we know that $y = x$ and y is not modified by the statement in Eq. (60). Hence we substitute y for x in the effects of Eq. (61), using the consequence rule, and get:

$$[\textbf{reads region}\{y.f\}]$$
$$\vdash_u \{x = y\ \&\&\ x.f = 5\ \&\&\ r' = \textbf{region}\{y.f\}\}x.f := 6; \{x = y\ \&\&\ x.f = 6\} \qquad (63)$$
$$[\textbf{modifies region}\{y.f\}, \textbf{fresh}(\textbf{region}\{y.f\} - r')]$$

Now the side conditions about immunity are true. Eq. (62) is derived by using the rule $SEQ1_u$. Our proof strategy generalizes the approach that we use 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 $\text{mods}(S_1)$, i.e., $\overline{x}$, with the variables $\overline{z}$, such that $a' \Rightarrow \overline{z} = \overline{x}$ and $\overline{z} \cap \text{mods}(S_1) = \varnothing$, where a' is the postcondition for S_1. The detailed proof is shown in Appendix E.

Corollary 50. *The meaning of a SSL judgment is preserved by the syntactic mapping.*

12 Recursive Predicates

Many examples in SL feature inductive predicates, as do some examples in this paper. Thus our connection between SL and UFRL needs to treat such inductively-defined predicates. As part of this treatment, we extend UFRL with a limited form of recursive predicates. We also show how to translate abstract function definitions and calls in SL to recursive predicate definitions and calls in UFRL.

12.1 Recursive predicates in UFRL

The following grammar shows the extension of the UFRL syntax from Fig. 5. It allows predicate declarations and calls to predicates in assertions (P).

$$
\begin{aligned}
&\textit{Predicate} ::= \textbf{predicate } p\,(\overline{x:T})\ \textbf{ reads }\ \delta;\ [\textbf{decreases } F;]\{\ P\ \} \\
&P ::= \ \dots\ |\ p(\overline{F})\ |\ x.p(\overline{F})
\end{aligned}
$$

where p is the predicate name and F is either an expression or a region expression (as in Fig. 5). We assume that predicate names are unique in each program. UFRL allows a restricted form of recursive definition; 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 [37,53]. 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 [53].

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 [53].

The semantic function *body* maps a pair of a class name and a predicate name to its definition. Global predicates are considered to be wrapped in a distinguished class $\texttt{Object}$. The semantic function *formals* maps a predicate name to its declared formal parameters. The semantic function *rd* maps a predicate name to its read effect. The notation $\overline{x} \mapsto \overline{y}$ means pointwise mapping. We introduce a semantic function $\mathcal{E}_p : P \to \textit{Store} \times \textit{Heap} \to \{\textit{true, false}\}$ which meaning is defined by the satisfaction relation: $\mathcal{E}_p[\![P]\!](\sigma, H)$ *iff* $\sigma, H \vDash_r^P$. The semantics of a predicate call is defined as follows, where *fix* denotes the least fixed point.

$$
\begin{aligned}
&\mathcal{E}_p[\![p(\overline{F})]\!](\sigma, H) = (\textit{fix}\underline{\lambda}(\sigma', H')\,.\,\mathcal{E}_p[\![body(Object, p)]\!](\sigma', H'))(\sigma(formals(Object, p) \mapsto \overline{\mathcal{E}_u[\![F]\!](\sigma)}), H) \\
&\mathcal{E}_p[\![x.p(\overline{F})]\!](\sigma, H) = \sigma(x) = o \text{ and } o \neq null \text{ and} \\
&\qquad\qquad (\textit{fix}\underline{\lambda}(\sigma', H')\,.\,\mathcal{E}_p[\![body(T, p)]\!](\sigma', H'))(\sigma(\textbf{this}, formals(T, p)) \mapsto (o, \overline{\mathcal{E}_u[\![F]\!](\sigma)})), H) \\
&\textbf{where } T = type(o, \sigma)
\end{aligned}
$$

The read effects of predicate calls are defined as follows, where Γ is a type environment.

$$
\begin{aligned}
&efs_u(p(\overline{F})) = efs_u(F), \delta[\overline{F}/\overline{z}] \qquad \textbf{where } \delta = rd(Object, p) \text{ and } \overline{z} = formals(Object, p) \\
&efs_u(x.p(\overline{F})) = \textbf{reads } x, efs_u(F), \delta[(x, \overline{F})/(\textbf{this}, \overline{z})] \qquad \textbf{where } \delta = rd(\Gamma(x), p) \text{ and } \overline{z} = formals(\Gamma(x), p)
\end{aligned}
$$

The rules *LIntro1*$_u$ and *LIntro2*$_u$ introduce the form of a predicate call to left-hand side of the judgment. The rules *RIntro1*$_u$ and *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.

$$(LIntro1_u) \quad \frac{P' \vdash_u P}{p(\overline{F}) \vdash_u P} \quad \textbf{where } P' = body(Object, p)[\overline{F}/formals(Object, p)]$$

$$(LIntro2_u) \quad \frac{x \neq null \&\& P' \vdash_u P}{x.p(\overline{F}) \vdash_u P} \quad \textbf{where } P' = body(\Gamma(x), p)[\overline{F}/formals(\Gamma(x), p)]$$

$$(RIntro1_u) \quad \frac{P \vdash_u P'}{P \vdash_u p(\overline{F})} \quad \textbf{where } P' = body(Object, p)[\overline{F}/formals(Object, p)]$$

$$(RIntro2_u) \quad \frac{P \vdash_u x \neq null \&\& P'}{P \vdash_u x.p(\overline{F})} \quad \textbf{where } P' = body(\Gamma(x), p)[\overline{F}/formals(\Gamma(x), p)]$$

Lemma 51. *The rules* LIntro1$_u$, LIntro2$_u$, RIntro1$_u$ *and* 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. □

12.2 Inductive Definition in SSL

The following grammar shows the extension of the SSL syntax given in Def. 28. It allows predicate calls in assertions.

$$a ::= \ldots \mid p_s(\overline{e})$$

where p_s is the predicate name and $\overline{e}$ are arguments. We apply the definition of "inductive definition set" from Brotherston's work [17] to SSL as follows:

Definition 52 (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. 52 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.

$$list(n, se) \stackrel{\text{def}}{=} (n = null \Rightarrow se = [])\&\&(n \neq null \Rightarrow (\exists\, m.\, n.val \mapsto se[0] * n.next \mapsto m * list(m, se[1..]))) \tag{64}$$

The semantic function *idf* maps a predicate name to its induction definition, which is the conjunction of inductive cases. The semantic function *formals* maps a predicate name to its formal parameters. The semantics of inductive predicate $p_s(\overline{e})$ is defined as follows:

$$\mathcal{E}_a[\![p_s(\overline{e})]\!](\sigma, h) = (fix\underline{\lambda}(\sigma', h')\,.\,\mathcal{E}_a[\![idf(p_s)]\!](\sigma', h'))(\sigma(formals(p_s) \mapsto \overline{\mathcal{E}_s[\![e]\!](\sigma)}), h) \tag{65}$$

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.

$$(LIntro_s) \quad \frac{a \vdash_s a'}{p_s(\overline{e}) \vdash_s a'} \quad \textbf{where } a = (b \Rightarrow a)[\overline{e}/formals(p_s)]$$

$$(RIntro_s) \quad \frac{a' \vdash_s a}{a' \vdash_s p_s(\overline{e})} \quad \textbf{where } a = (b \Rightarrow a)[\overline{e}/formals(p_s)]$$

12.3 Encoding

We define the translation of recursive predicate call as follows:

$$\text{TR}[\![p_s(\bar{e})]\!] = p_s(\overline{\text{TR}[\![e]\!]}). \tag{66}$$

Assume an inductive predicate p_s has n inductive cases. Fig. 27 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 $RE_1 \,!\,!\cdots\,!\,! RE_n$ means pairwise region disjointness. For each inductive predicate $p_s : \overline{z : T} \mapsto \textbf{bool}$, there is a region function with the signature $region\_p_s : \overline{z : T} \mapsto \textbf{region}$ 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. 28 shows the encoding of the inductive predicate in Eq. (64). Note that the invalid syntax can be solved by program instruments.

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

```
          predicate p_s (z : T)
            reads region_p_s (z̄);
            decreases region_p_s (z̄);
          {   &&ⁿᵢ₌₁ TR[[bᵢ ⇒ aᵢ]]  }
```

```
function region_p_s (z : T) : region
  reads region_p_s (z̄);
  decreases region_p_s (z̄));
{
  ret := fpt (⋀ⁿᵢ₌₁  bᵢ ⇒ aᵢ);
}
```

Fig. 27: 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..]);
}
```

Fig. 28: The encoding of Eq. (64).

By the definition of $region_{p_s}$ and the results in Section 10.3, we know that $\text{TR}[\![p_s]\!] \vdash_u region\_p_s \, frm \, \text{TR}[\![p_s]\!]$.

Lemma 53. *Let (σ, h) be a state, and p_s be an inductive predicate in SSL. Then $\mathcal{E}_a[\![p_s(\bar{e})]\!](\sigma, h) = \mathcal{E}_p[\![\mathit{TR}[\![p_s(\bar{e})]\!]]\!](\sigma, h)$.*

The proof is found in Appendix F. By the syntactic mapping from SSL to UFRL proofs Def. 48, the induction rule in SSL (*LIntro$_s$* and *RIntro$_s$*) is encoded to the followings:

$$(\text{TR}[\![LIntro_s]\!]) \quad \frac{\text{TR}[\![a]\!] \vdash_u \text{TR}[\![a']\!]}{\text{TR}[\![p_s(e)]\!] \vdash_u \text{TR}[\![a']\!]} \quad \textit{where } a = (b \Rightarrow a)[\bar{e}/formals(p_s)]$$

$$(\text{TR}[\![RIntro_s]\!]) \quad \frac{\text{TR}[\![a']\!] \vdash_u \text{TR}[\![a]\!]}{\text{TR}[\![a']\!] \vdash_u \text{TR}[\![p_s(\bar{e})]\!]} \quad \textit{where } a = (b \Rightarrow a)[\bar{e}/formals(p_s)]$$

The encoded rules are admissible in the UFRL proof system by Theorem 40 and Lemma 53.

13 Extending the UFRL Proof System with Separating Conjunction

To allow SL and FRL to interoperate we want to allow users to write SL style assertions directly 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 UFRL. We define the semantics of separating conjunction in UFRL and show that it is equivalent to the one in SSL. Then we define the read effects of separating conjunction, and show the soundness of the framing judgment, which is the key to the soundness of UFRL's frame rule.

13.1 Extending the Syntax and the Semantics

To have the ability to write SL style specifications in UFRL, 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 UFRL's equality assertion $x.f = e$. Thus, we only need to extend the syntax of UFRL assertions, from Fig. 5, as follows:

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

Given a SSL assertion a, $\text{TR}[\![a]\!]$, which replaces each occurence of $\mapsto$ in a with $=$, is a valid assertion in the extended UFRL system. To ease the notational burden, we sometimes use the points-to assertion and the equality assertion interchangeably in examples when the context is clear.

The separating conjunction is a supported UFRL assertion in the following sense.

Definition 54 (Supported UFRL Assertions)**.** *Let P be an assertion in UFRL. P is supported if there exists an SSL assertion, a, such that $P = \text{TR}[\![a]\!]$.*

Defining the semantics of separating conjunction, $P_1 * P_2$, in UFRL requires a definition of its footprint, $fpt_u(P_1 * P_2)$. The semantics of fpt_u is defined by using the inverse of the translation function, TR^{-1}. This inverse exists because, by definition, TR is injective, as can be shown by induction on the structure of SSL assertions (see Def. 34). So, for each supported UFRL assertion, P, by definition there is some SSL assertion a such that $\text{TR}(a) = P$; thus for each supported UFRL assertion P, we define $\text{TR}^{-1}[\![P]\!]$ to be the SSL assertion a such that $\text{TR}(a) = P$.

Using TR^{-1}, we define the semantic footprint function for supported UFRL assertions, P, by $fpt_u(P) = fpt_s(\text{TR}^{-1}[\![P]\!])$.

Finally, the semantics of separating conjunction is defined as follows for supported UFRL assertions P_1 and P_2:

$$\sigma, H \models_u P_1 * P_2 \text{ iff } \sigma, H \models_u P_1 \text{ and } \sigma, H \models_u P_2 \text{ and } \sigma, H \models_u fpt_u(P_1) \, ! \, ! \, fpt_u(P_2) \tag{67}$$

The following lemma shows that Eq. (67) is the correct semantics.

Lemma 55. *Let (σ, H) be a state. Let P_1 and P_2 be supported assertions in extended UFRL. Then*

$$\sigma, H \models_u P_1 * P_2 \text{ iff } \sigma, H \models_s \text{TR}^{-1}[\![P_1]\!] * \text{TR}^{-1}[\![P_2]\!].$$

Proof. We assume $\sigma, H \vDash_u P_1 * P_2$ and calculate it as follows.

$\sigma, H \vDash_u P_1 * P_2$

iff $\quad \langle$by Eq. (67)$\rangle$

$\sigma, H \vDash_u P_1$ *and* $\sigma, H \vDash_u P_2$ *and* $\sigma, H \vDash_u fpt_u(P_1) \text{!} \text{!} fpt_u(P_2)$

iff $\quad \langle$by definition $fpt_u(P) = fpt_s(\text{TR}^{-1}[\![P]\!])\rangle$

$\sigma, H \vDash_u P_1$ *and* $\sigma, H \vDash_u P_2$ *and* $\sigma, H \vDash_u fpt_s(\text{TR}^{-1}[\![P_1]\!]) \text{!} \text{!} fpt_s(\text{TR}^{-1}[\![P_2]\!])$

iff $\quad \langle$by semantics of UFRL in Fig. 14$\rangle$

$\sigma, H \vDash_u P_1 \text{ \&\& } P_2 \text{ \&\& } fpt_s(\text{TR}^{-1}[\![P_1]\!]) \text{!} \text{!} fpt_s(\text{TR}^{-1}[\![P_2]\!])$

iff $\quad \langle$by definition of the syntactical mapping from SSL to UFRL (Def. 34), as P_1 and P_2 are supported$\rangle$

$\sigma, H \vDash_u \text{TR}[\![\text{TR}^{-1}[\![P_1]\!] * \text{TR}^{-1}[\![P_2]\!]]\!]$

iff $\quad \langle$by Theorem 40$\rangle$

$\sigma, H \vDash_s \text{TR}^{-1}[\![P_1]\!] * \text{TR}^{-1}[\![P_2]\!]$

$\square$

13.2   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:

$$efs(P_1 * P_2) = efs(P_1), efs(P_2) \tag{68}$$

Lemma 56 shows that the soundness of the frame validity (Def. 5), i.e., $true \vdash_u efs(P_1 * P_2) \, frm \, (P_1 * P_2)$ is valid. The proof is by induction on the structure of assertions.

Lemma 56 (Frame Soundness of Extended Assertions). *Let (σ, h) and (σ', h') be arbitrary states. Let P be a supported assertion in extended UFRL. If $(\sigma, h) \stackrel{efs(\text{P})}{\equiv} (\sigma', h')$, then*

$$\mathcal{E}[\![fpt_u(P)]\!](\sigma) = \mathcal{E}[\![fpt_u(P)]\!](\sigma'), \text{ and } \sigma, h \vDash_u P \text{ iff } \sigma', h' \vDash_u P.$$

The separating conjunction proves some properties about the separator (defined in Fig. 20).

Lemma 57. *Let (σ, h) be a state. Let P_1 and P_2 be supported assertions in extended UFRL. Then*

$$\sigma, h \vDash_u P_1 * P_2 \text{ implies } \sigma, h \vDash_u efs(P_2) \cdot\!/ \textbf{\textit{modifies}} \, fpt_u(P_1)$$

Note that it is not valid that $\sigma, h \vDash_u P_1 * P_2 \Rightarrow \sigma, h \vDash_u efs(P_2) \cdot\!/ \textbf{modifies} \, readVar(efs(P_1)), readR(efs(P_1))$. For example, let P_1 be $x.f_1 = 4$ and P_2 be $x.f_2 = 5$, and $P_1 * P_2$ is valid. Because $efs(P_2) = \textbf{reads} \, x$, $\textbf{region}\{x.f_2\}$, and $\textbf{modifies} \, readVar(efs(P_1)), readR(efs(P_1)) = \textbf{modifies} \, x, \textbf{region}\{x.f_1\}$, they are not disjoint sets.

13.3   Proof Rules

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

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

The two extra side conditions are used to conclude $P * R$ and $Q * R$, which is justified by the following lemma:

Lemma 58 (Soundness). I_{sc} *is admissible in the extended UFRL proof system.*

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

$$(FRM_u) \quad \cfrac{\cfrac{\vdash_u [\delta]\{P\}\,S\,\{Q\}[\varepsilon]}{\vdash_u [\delta]\{P \text{ \&\& } R\}\,S\,\{Q \text{ \&\& } R\}[\varepsilon]} \; \textbf{where } P \text{ \&\& } R \Rightarrow efs(R)\dot{/}\varepsilon}{\vdash_u [\delta]\{P * R\}\,S\,\{Q * R\}[\varepsilon]} \quad \begin{array}{l} \textbf{where } P \text{ \&\& } R \Rightarrow efs(R)\dot{/}\textbf{modifies}\,fpt_u(P) \\ and\; Q \text{ \&\& } R \Rightarrow efs(R)\dot{/}\textbf{modifies}\,fpt_u(Q) \end{array}$$
$$(CONSEQ_u)$$

$\square$

Lemma 59. *Let P_1 and P_2 be supported assertions in extended UFRL proof system, and (σ, h) be a state. If $\sigma, h \vDash_u P_1$ and $\sigma, h \vDash_u P_2$ and $efs(P_2)\dot{/}\textbf{modifies}\,fpt_u(P_1)$, then $\sigma, h \vDash_u P_1 * P_2$.*

Consider the example in Fig. 3. As it has explicit write effects, but no specified read effect, the read effect defaults to **reads alloc**$\downarrow$ (see Section 14.2). In the body of append, right after the loop, we have:

$$(lst(\textbf{this}, vlst) \text{ \&\& } (lstseg(\textbf{this}, curr) * (lst(curr, ?cvlst) \text{ \&\& } curr.next = null))) * lst(n, [v]), \quad (69)$$

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

$$(lst(\textbf{this}, vlst) \text{ \&\& } (lstseg(\textbf{this}, curr) * curr.val \mapsto ?cv * curr.next \mapsto null)) * lst(n, [v]), \quad (70)$$

which implies the precondition of the rule UPD_u. Using the rules UPD_u and $SubEff_u$, we derive:

$$\vdash_u \begin{array}{l} [\textbf{reads}\, curr, \textbf{alloc}\downarrow] \\ \{curr \neq null\}\, curr.next := n;\, \{curr.next = n\} \\ [\textbf{modifies region}\{curr.next\}] \end{array} \quad (71)$$

By $CONSEQ_u$ and I_{sc}, we get:

$$\vdash_u \begin{array}{l} [\textbf{reads}\, curr, \textbf{alloc}\downarrow] \\ \{curr \neq null * lstseg(\textbf{this}, curr) * curr.val \mapsto ?cv * lst(n, [v])\} \\ curr.next := n; \\ \{curr.next = n * lstseg(\textbf{this}, curr) * curr.val \mapsto ?cv * lst(n, [v])\} \\ [\textbf{modifies region}\{curr.next\}] \end{array} \quad (72)$$

and the postcondition of Eq. (72) implies, by the definition of lst

$$lst(curr, [curr.val] + [v]) \text{ \&\& } lseg(\textbf{this}, n) \quad (73)$$

To prove the postcondition, consider the second loop invariant in Fig. 3:

$$\textbf{fpt}(lstseg(\textbf{this}, curr)) + \textbf{fpt}(lst(curr, [curr.val])) = \textbf{fpt}(lst(\textbf{this}, vlst)), \quad (74)$$

Together with Eq. (73), we have at the end of the method body

$$\textbf{fpt}(lstseg(\textbf{this}, curr)) + \textbf{fpt}(lst(curr, [curr.val] + [v])) = \textbf{fpt}(lst(\textbf{this}, vlst + [v])), \quad (75)$$

which implies the postcondition of the procedure.

13.4   Encoding SSL specifications

Using separating conjunctions in the extended UFRL, we can encode the SSL Hoare-formulas by substituting $=$ for $\mapsto$ as follows:

$$\vdash_s \{a\}\, x := \mathbf{new}\, C; \{a * new_s(C,x)\ iff$$
$$[\mathbf{reads}\, fpt_s(a)]$$
$$\vdash_u \{a[=/\!\!\mapsto]\}\, x := \mathbf{new}\, C;\ \{(a * new_s(C,x))[=/\!\!\mapsto]\}$$
$$[\mathbf{modifies}\, x,\ \mathbf{modifies\ alloc},\ \mathbf{fresh}(fpt_s(new_s(C,x)))]$$

$$s \vdash_s \{a\}S\{a'\}\ iff$$
$$[\mathbf{reads}\, fpt_s(a)]$$
$$\vdash_u \begin{array}{l}\{a[=/\!\!\mapsto]\ \&\&\ fpt_s(a) = r\} \\ S\,\{a'[=/\!\!\mapsto]\}\end{array}$$
$$[\mathbf{modifies}(\mathrm{mods}(S), fpt_s(a)),\ \mathbf{fresh}\,(fpt_s(a') - r)]$$
$$\mathbf{where}\ r\ \text{is fresh}, r \notin \mathrm{mods}(S)\ and\ S \neq x := \mathbf{new}\, C;$$

where $\mathrm{mods}(S)$ 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 11.3.

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, we use the rule *FRM*$_u$ 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.

13.5   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.

14   Applications

This section shows several potential applications of our results.

14.1   A Footprint Function

FRL 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 FRL 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 $\mathrm{TR}[\![a]\!]$ have the same semantic footprints, i.e., $fpt_s(a) = fpt_u(\mathrm{TR}[\![a]\!])$, where fpt_r is the semantic footprint function for FRL. The specification of the method `mark` (Fig. 2) 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. Other examples can be found in Section 14.5.

14.2   Intraoperation of FRL and SSL within Modules

This section introduces a technique for verifying methods whose specifications are written in either UFRL, FRL, or SSL.

1. If both the method's read effect and write effect are specified, then the system verifies the body directly using UFRL's rules.
2. If the method's read effects are not specified, then

(a) If its write effects are specified, or if the specification uses the keyword **pure** (a shorthand for a frame of **modifies** $\varnothing$), then the system considers that the method is specified in FRL, sets the read effects to the default value, **reads alloc**↓, and verifies the body using UFRL's rules. This is justified by Theorem 21.

(b) Otherwise the system checks that the assertions used in the method's specifications follow the restrictions for SSL, and if these checks pass, then it translates the specification into UFRL and then verifies the body using UFRL's rules. This is justified by Theorem 49.

Consider the example in Fig. 29. The class ReCell holds the value val and its backup bak. The method set updates val with the new value and stores its old value to bak. The method undo performs rollback. The method set is specified in the style of SSL. Its precondition is trivially true as the fields val and bak are of primitive type. The method undo is specified in the style of FRL. Using Theorem 21 and Theorem 49, both are translated into specifications written in UFRL. Then by using the rule *SubEff$_u$*, we derive:

$$
\begin{aligned}
&[\textbf{reads alloc↓}] \\
&\{\textbf{this}.val = ?v \ \&\& \ \textbf{this}.bak = ?bk \ \&\& \ \textbf{region}\{\textbf{this}.val\}!!\textbf{region}\{\textbf{this}.bak\} \\
\vdash_u \ & \texttt{set(x : int)}; \\
&\{\textbf{this}.val = x \ \&\& \ \textbf{this}.bak = v \ \&\& \ \textbf{region}\{\textbf{this}.val\}!!\textbf{region}\{\textbf{this}.bak\}\} \\
&[\textbf{modifies region}\{\textbf{this}.val\} + \textbf{region}\{\textbf{this}.bak\}]
\end{aligned}
\tag{76}
$$

and

$$
\vdash_u [\textbf{reads alloc↓}]\{true\}\texttt{undo()} \{\textbf{this}.bak = ?vb \ \&\& \ \textbf{this}.val = vb\}[\textbf{modifies region}\{\textbf{this}.val\}]
\tag{77}
$$

Consider this client code:

```
var rc : ReCell; rc := new ReCell;  rc.set(3);  rc.set(5); rc.undo();
assert rc.val = 3 && rc.bak = 3.
```

```
class ReCell
{                                              void undo()
  var val, bak : int;                            requires true;
                                                 modifies region{this.val};
  void set(x : int)                              ensures this.bak = ?vb && this.val = vb;
  requires this.val↦?v * this.bak↦?bk;         { var b := this.bak; this.val := b; }
  ensures this.val↦x * this.bak↦v;        }
  { bak := val; val := x; }
```

Fig. 29: An example of using specifications written in SSL and FRL. This example is adapted from Parkinson and Bierman's work [49].

To prove that the assertion is true, we firstly use the axiom *VAR$_u$* and the axiom *ALLOC$_u$* as their preconditions are *true*:

$$
\vdash_u [\varnothing]\{true\}\textbf{var} \ \texttt{rc : ReCell;} \ \{rc = null\}[\varnothing]
$$

and

$$
\vdash_u [\varnothing]\{true\}\texttt{rc := new ReCell;} \{new_u(ReCell, rc)\}[\textbf{modifies} \ rc, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{rc.*\})]
\tag{78}
$$

After using the rule *CONSEQ$_u$*, *SEQ$_u$* and *CONSEQ$_u$*, we derive:

$$
\begin{aligned}
&[\textbf{alloc↓}] \\
\vdash_u \ & \{true\}\textbf{var} \ \texttt{rc ReCell; rc := new ReCell;} \{new_u(ReCell, rc)\} \\
&[\textbf{modifies} \ rc, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{rc.*\})]
\end{aligned}
\tag{79}
$$

which implies the precondition of the procedure call set. In order to use the rule *SEQ$_u$*, we verify the following side conditions:

1. **modifies** rc, **alloc** is fresh-free;
2. $\varnothing$ is $true$/(**modifies** rc, **alloc**)-immune;
3. **region**$\{rc.*\}$ is $new_u(ReCell, rc)$/(**modifies region**$\{rc.*\}$)-immune.

(1) and (2) are trivially true. We consider (3). By the definition of immune (Def. 11), we need to verify:

$$\textbf{modifies region}\{rc.val\} \text{ is } new_u(ReCell, rc)/\textbf{modifies region}\{rc.*\}\text{-immune} \tag{80}$$

and

$$\textbf{modifies region}\{rc.bak\} \text{ is } new_u(ReCell, rc)/\textbf{modifies region}\{rc.*\}\text{-immune}, \tag{81}$$

which are

$$new_u(ReCell, rc) \text{ } implies \text{ } efs(\textbf{region}\{rc.val\})\cdot/\cdot\textbf{region}\{rc.*\} \tag{82}$$

and

$$new_u(ReCell, rc) \text{ } implies \text{ } efs(\textbf{region}\{rc.bak\})\cdot/\cdot\textbf{region}\{rc.*\} \tag{83}$$

By definition of read effects (Def. 18), we have $efs(\textbf{region}\{rc.val\}) = \textbf{reads } rc$ and $efs(\textbf{region}\{rc.bak\}) = \textbf{reads } rc$. These are separate with **region**$\{rc.*\}$. Thus Eq. (82) and Eq. (83) are true. After using the rule $CONSEQ_u$ and SEQ_u, we derive:

$$
\begin{aligned}
&[\textbf{alloc}{\downarrow}] \\
&\{true\} \\
\vdash_u \text{ } &\textbf{var } \texttt{rc : ReCell; rc := } \textbf{new } \texttt{ReCell; rc.set(3);} \\
&\{\textbf{this}.val = 3 \text{ \&\& } \textbf{this}.bak = 0 \text{ \&\& } \textbf{region}\{\textbf{this}.val\}\text{!!}\textbf{region}\{\textbf{this}.bak\}\} \\
&[rc, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{rc.*\})]
\end{aligned} \tag{84}
$$

which implies the precondition of $\texttt{set}$. Because Eq. (79) and Eq. (84) have the same effects, after another call of $\texttt{set}$, we derive the following by using the rule $CONSEQ_u$ and SEQ_u:

$$
\begin{aligned}
&[\textbf{reads alloc}{\downarrow}] \\
&\{true\} \\
\vdash_u \text{ } &\textbf{var } \texttt{rc : ReCell; rc := } \textbf{new } \texttt{ReCell; rc.set(3); rc.set(5);} \\
&\{\textbf{this}.val = 5 \text{ \&\& } \textbf{this}.bak = 3 \text{ \&\& } \textbf{region}\{\textbf{this}.val\}\text{!!}\textbf{region}\{\textbf{this}.bak\}\} \\
&[\textbf{modifies } rc, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{rc.*\})]
\end{aligned} \tag{85}
$$

which implies the precondition of $\texttt{undo}$. In order to use the rule SEQ_u, we verify the following side conditions:

1. **modifies** rc, **alloc** is fresh-free;
2. $\varnothing$ is $true$/(**modifies** rc, **alloc**)-immune;
3. **region**$\{rc.*\}$ is (**this**$.val = 5$ && **this**$.bak = 3$ && **region**$\{$**this**$.val\}$!!**region**$\{$**this**$.bak\}$) / (**modifies region**$\{rc.val\}$)-immune.

(1) and (2) are trivially true. We consider (3). Let P be (**this**$.val = 5$ && **this**$.bak = 3$ && **region**$\{$**this**$.val\}$!!**region**$\{$**this**$.bak\}$). By the definition of immune (Def. 11), we need to verify:

$$\textbf{modifies region}\{rc.val\} \text{ is } P/\textbf{modifies region}\{rc.*\}\text{-immune}, \tag{86}$$

which is

$$P \text{ } implies \text{ } efs(\textbf{region}\{rc.val\})\cdot/\cdot\textbf{modifies region}\{rc.*\} \tag{87}$$

By definition of read effects (Def. 18), we have $efs(\textbf{region}\{rc.val\}) = \textbf{reads } rc$, which is separate with **region**$\{rc.*\}$. Thus Eq. (87) is true. After using the rule $CONSEQ_u$ and SEQ_u, we derive:

$$
\begin{aligned}
&[\textbf{reads alloc}{\downarrow}] \\
&\{true\} \\
\vdash_u \text{ } &\textbf{var}rc : ReCell; rc := \textbf{new}ReCell; rc.set(3); rc.set(5)rc.undo() \\
&\{\textbf{this}.val = 3 \text{ \&\& } \textbf{this}.bak = 3 \text{ \&\& } \textbf{region}\{\textbf{this}.val\}\text{!!}\textbf{region}\{\textbf{this}.bak\}\} \\
&[\textbf{ modifies } rc, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{rc.*\})]
\end{aligned} \tag{88}
$$

which implies that the assertions are true.

14.3 Hypothetical Reasoning and Interoperation between Modules

In this section, we extend the language in Fig. 5 with method calls, and show how FRL and SSL specifications interoperate between modules.

$$S ::= \ldots \mid x.m\,(\overline{F})$$

where $\overline{F}$ indicates a possibly empty sequence of actual parameters. The statement $\texttt{y := x.m}\,(\overline{F})$ is sugar for the sequential statement $\texttt{x.m}\,(\overline{F})\,\texttt{;}\; \texttt{y := \textbf{ret};}$.

The language has a call-by-value semantics, and writes to the formal parameters are not allowed in method bodies. In a method call such as $x.m()$, the value of x must not be null and its body is looked up in the class of x's type. The formal semantics for method calls is standard, thus, it is omitted.

In order to modularly reason about programs, each method is specified and verified individually, and reasoning about method calls uses the method's specifications, instead of its body.

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 different methodologies, i.e., separation logic and dynamic frames (using FRL). Method specifications may be written in the style of either SSL or FRL. This section shows how these different styles of specifications interoperate with each other. Assume that $C.m$ denotes a method m declared in the class C. Cl is client code that just calls $C.m$. The form of program correctness judgment in UFRL is $\Phi_u \vdash_u [\delta]\{P_1\}Cl\{P_2\}[\varepsilon]$, which states that Cl satisfies its Hoare-formula under certain hypotheses, Φ, which map pairs of class and method names to the corresponding method's specification. Hypotheses are given by the grammar:

$$\Phi_u ::= \varnothing \mid \Phi_{u1}, \Phi_{u2} \mid [\delta]\{P_1\}C.m(\overline{F})\{P_2\}[\varepsilon],$$

and Φ_u contains all the methods that may be used by Cl. When Cl invokes a method $C.m$, UFRL uses the axiom for method calls that is adapted from Banerjee and Naumann's work [1] as follows:

$$(CALL_u)\; \Phi_u, [\delta]\{P\}\,C.m(\overline{x})\,Q)[\varepsilon] \vdash_u [\delta[\overline{F}/\overline{x}]]\{P[\overline{F}/\overline{x}]\}\,y.m(\overline{F})\,\{Q[\overline{F}/\overline{x}]\}[\varepsilon[\overline{F}/\overline{x}]]\;\textbf{where}\;\Gamma(y) = C,$$

where $P[\overline{F}/\overline{x}]$ simultaneously substitutes $\overline{F}$ for $\overline{x}$ in P. Note that since our work does not have subclassing or subtyping, in the rule $\Gamma(y)$ is both y's static and dynamic type.

We assume a function, $mbody$, that takes a class name and a method name, and returns a list of formal parameters and its body. To verify that each method $C.m$ satisfies its specification, UFRL uses the rule for proving a method body that is adapted from Banerjee and Naumann's work [1] as follows:

$$(METH_u)\; \frac{\Phi_u \vdash_u [\delta]\{\textbf{this} : C \;\&\&\; P\,\}S\{Q\}[\varepsilon]}{\Phi_u \vdash_u [\delta]\{P\}\,C.m(\overline{x})\,\{Q\}[\varepsilon]}$$
$$\textbf{where}\; mbody(C, m) = (\overline{x}, S), \Phi_u(C, m) = [\delta]\{P\}\,C.m(\overline{x})\,\{Q\}[\varepsilon]$$

The presence of $[\delta]\{P\}C.m(\overline{x})\{Q\}[\varepsilon]$ in the assumption, Φ_u, for the method body allows recursive calls. According to Theorem 21 and Theorem 49, we define a function TR_Φ that syntactically maps from FRL and SSL hypotheses to those in UFRL. Hypotheses specified by FRL are given by the following grammar:

$$\Phi_r ::= \varnothing \mid \Phi_{r1}, \Phi_{r2} \mid frl\{P_1\}C.m(\overline{z})\{P_2\}[\varepsilon],$$

where $mbody(C, m) = (\overline{z}, S)$. For such FRL hypotheses, TR_Φ merely adds $\textbf{alloc}{\downarrow}$ as the read effects of the non-empty hypotheses. Hypotheses specified by SSL are given by the following grammar [47]:

$$\Phi_s ::= \varnothing \mid \Phi_{s1}, \Phi_{s2} \mid ssl\{a\}C.m(\overline{z})\{a'\}[X],$$

where $X = \text{mods}(S)$ and $mbody(C, m) = (\overline{z}, S)$. These are translated into UFRL using $\text{TR}[\![\cdot]\!]$ and $fpt_s(\cdot)$ as follows:

$$
\begin{aligned}
\text{TR}_\Phi[\![\varnothing]\!] &= \varnothing \\
\text{TR}_\Phi[\![\Phi_{s1}, \Phi_{s2}]\!] &= \text{TR}_\Phi[\![\Phi_{s1}]\!], \text{TR}_\Phi[\![\Phi_{s2}]\!] \\
\text{TR}_\Phi[\![ssl\{a\}C.m(\overline{z})\{a'\}[X]]\!] &= [\textbf{reads}\, fpt_s(a)]\{\text{TR}[\![a]\!] \;\&\&\; r = fpt_s(a)\}C.m(\overline{z})\{\text{TR}[\![a']\!]\} \\
&\qquad [\textbf{modifies}\,(X, fpt_s(a)), \textbf{fresh}(fpt_s(a') - r)] \\
&\qquad \textbf{where}\; r \text{ is fresh } and\; r \notin X
\end{aligned}
$$

Consider the cell example in Fig. 4. The method specifications for this example are translated as follows:

$$\Phi_{u_1} = \begin{array}{l} [\textbf{reads region}\{\textbf{this}.x\}] \\ \{\textbf{this}.x = \_ \&\& r = \textbf{region}\{\textbf{this}.x\}\} \, \texttt{setSX(v : int)}; \{\textbf{this}.x = v\} \\ [\textbf{modifies region}\{\textbf{this}.x\}] \end{array}$$

$$\Phi_{u_2} = \begin{array}{l} [\textbf{reads region}\{\textbf{this}.x\}] \\ \{\textbf{this}.x = \_ \&\& r = \textbf{region}\{\textbf{this}.x\}\} \, \texttt{getSX()}; \{\textbf{this}.x = v \&\& \textbf{ret} = v\} \\ [\textbf{modifies region}\{\textbf{this}.x\}] \end{array}$$

$$\Phi_{u_3} = \begin{array}{l} [\textbf{reads alloc}\!\downarrow] \\ \{c \neq \textbf{null}\} \, \texttt{addOne(c : Cell)}; \{\textbf{this}.x = c.getSX() + 1\} \\ [\textbf{modifies region}\{\textbf{this}.x\}] \end{array}$$

$$\Phi_{u_4} = [\textbf{reads alloc}\!\downarrow] \, \{true\} \, \texttt{getRX()}; \{\textbf{ret} = \textbf{this}.x\} \, [\varnothing]$$

The read effects of Φ_{u_1} and Φ_{u_2} can be extended to $\textbf{alloc} \downarrow$ by using the rule $SubEff_u$. After the declaration and initialization ($\textbf{var}$ sCell; sCell := $\textbf{new}$ CellS; $\textbf{var}$ rCell; rCell := $\textbf{new}$ CellR;), we have $sCell.x = 0 \,\&\&\, rCell.x = 0$, which implies the precondition of sCell.setSX(5). Thus, its postcondition is assumed right after it. As the read effects of $rcell.x = 0$ is separate from the method's write effects, using the rule FRM_u, we have $sCell.x = 5 \,\&\&\, rCell.x = 0$, which implies the precondition of rCell.addOne(sCell). Thus its postcondition is assumed right after it. As the read effects of $sCell.x = 5$ is separate from the method's write effects, using the rule FRM_u, we have $sCell.x = 5 \,\&\&\, rCell.x = 6$. In order to use the rule $SEQ1_u$, we need to prove the side condition:

$$\textbf{region}\{rCell.x\} \text{ is } sCell.x \mapsto \_/\textbf{modifies region}\{sCell.x\}\text{-immune}. \tag{89}$$

By the definition of immune (Def. 11), we need to prove:

$$efs(\textbf{region}\{rCell.x\}) \,\cdot\!/\cdot\, \textbf{modifies region}\{sCell.x\}, \tag{90}$$

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

$$\begin{array}{l} [\textbf{alloc}\!\downarrow] \\ \{sCell.x \mapsto \_\} \\ \texttt{sCell.setSX(5);} \quad \texttt{rCell.addOne(sCell);} \\ \{sCell.x \mapsto 5 \,\&\&\, rCell.x = sCell.getSX() + 1\} \\ [\textbf{modifies region}\{sCell.x\}, \textbf{region}\{rCell.x\}] \end{array} \tag{91}$$

Thus, we can prove $sCell.getSX() = 5$ and $rCell.getRX() = 6$ is true.

14.4 The DAG Example

This section presents proofs of the DAG example in Fig. 2 that is specified by using separating conjunction for disjointness and conjunction for sharing. It illustrates proof rules, i.e., the frame rule and the immunity condition, which distinguish UFRL.

We prove 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, we use unmarked's specification, 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, we can specify mark's write effects with the function unmarked. The method mark's read effects are not specified, thus are $\textbf{alloc}\!\downarrow$ 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, we have the second precondition. We prove mark by 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. $dag(d)$ && $d \neq null$ && $d.mark \mapsto 1$: 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. $dag(d)$ && $d \neq null$ && $d.mark \mapsto 0$: 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`. We have:

$$d.mark \mapsto 0 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r)), \tag{92}$$

which implies the precondition of the rule UPD_u, we derive:

$$\vdash_u [\textbf{reads } d]\{d \neq null\} \, \texttt{d.mark := 1;} \, \{d.mark \mapsto 1\} \, [\textbf{modifies region}\{d.mark\}] \tag{93}$$

One can translate Eq. (93) into a formula in UFRL by Def. 34, or use the result in Section 13 without translation. To avoid big formulas, we explore the second approach, and have:

$$d \neq null \vdash_u \frac{(\textbf{reads } d, \textbf{region}\{d.l\}, \textbf{region}\{d.r\}, \textbf{fpt}(dag(d.l) \text{ \&\& } dag(d.r)))}{frm \, (d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r)))} \tag{94}$$

Thus, the read effects are separate from the write effects, $\textbf{region}\{d.mark\}$. Using the rule I_{sc}, we derive:

$$
\begin{aligned}
& [\textbf{reads } d] \\
& \{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r))\} \\
\vdash_u \; & \texttt{d.mark := 1;} \\
& \{d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r))\} \\
& [\textbf{modifies region}\{d.mark\}]
\end{aligned}
\tag{95}
$$

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

$$
\begin{aligned}
& [\textbf{reads alloc}\downarrow] \\
& \left\{ \begin{array}{l} dag(d.l) \text{ \&\& } (d \neq null \text{ \&\& } d.mark \mapsto 1 \Rightarrow \\ \qquad \forall n : Node.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d)) \Rightarrow n.mark \mapsto 1)) \end{array} \right\} \\
\vdash_u \; & \texttt{mark(d.l);} \\
& \{d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)\} \\
& [\textbf{modifies } unmarked(d.l)]
\end{aligned}
\tag{96}
$$

We use the rule $SubEff_u$ on Eq. (95) and Eq. (96) to match up the effects for the rule $SEQ1_u$, and use the rule $CONSEQ_u$ on Eq. (96) to match up the postcondition of $\texttt{d.mark := 1}$ and the precondition of $\texttt{mark(d.l)}$ and to get rid of the implication in the precondition. Thus, we derive:

$$
\begin{aligned}
& [\textbf{reads alloc}\downarrow, d] \\
& \{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r))\} \\
\vdash_u \; & \texttt{d.mark := 1;} \\
& \{d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \text{ \&\& } dag(d.r))\} \\
& [\textbf{modifies region}\{d.mark\}]
\end{aligned}
\tag{97}
$$

and

$$
\begin{aligned}
& [\textbf{reads alloc}\downarrow, d] \\
& \{dag(d.l)\} \\
\vdash_u \; & \texttt{mark(d.l);} \\
& \{d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)\} \\
& [\textbf{modifies } unmarked(d.l)]
\end{aligned}
\tag{98}
$$

By using the rule I_{sc}, FRM_u and $CONSEQ_u$ for Eq. (98), we derive:

$$\vdash_u \begin{array}{l} [\textbf{reads alloc}\downarrow, d] \\ \{d.mark \mapsto 1 * d.l \mapsto i \ * \ d.r \mapsto j * (dag(d.l) \ \&\& \ dag(d.r))\} \\ \texttt{mark(d.l);} \\ \left\{ \begin{array}{l} (d.l \neq null \ \Rightarrow \ \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \ \Rightarrow \ n.mark \mapsto 1)) \ \&\& \\ (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \ \&\& \ dag(d.r))) \end{array} \right\} \\ [\textbf{modifies} \ unmarked(d.l)] \end{array} \tag{99}$$

In order to use the rule $SEQ1_u$, we need to prove the side condition:

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

By the definition of immune (Def. 11), we need to prove that *for all* **modifies** *RE* in $\texttt{unmarked(d.l)}$:

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

We show 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. 18). Because the type of $mark$ is **int**, not a reference, this is impossible.

Now we can accumulate the two statements' write effects and derive:

$$\vdash_u \begin{array}{l} [\textbf{reads alloc}\downarrow, d] \\ \{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(i) \ \&\& \ dag(j))\} \\ \texttt{d.mark := 1;} \quad \texttt{mark(d.l);} \\ \left\{ \begin{array}{l} (d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(i)) \Rightarrow n.mark \mapsto 1)) \ \&\& \\ (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(i) \ \&\& \ dag(j))) \end{array} \right\} \\ [\textbf{modifies region}\{d.mark\}, unmarked(i)] \end{array} \tag{100}$$

The postcondition of the above implies the precondition of the method $\texttt{mark(d.r)}$. Using the rule $CONSEQ_u$ (getting rid of the implication in the precondition), we have:

$$\vdash_u \begin{array}{l} [\textbf{reads alloc}\downarrow] \\ \{dag(d.r)\} \\ \texttt{mark(d.r);} \\ \{d.r \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.r)) \ \Rightarrow \ n.mark \mapsto 1)\} \\ [\textbf{modifies} \ unmarked(d.r)] \end{array} \tag{101}$$

As the function $\texttt{unmarked}$ only collects unmarked locations, we have:

$$\begin{array}{l} (d.l \neq null \Rightarrow (\forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \ \Rightarrow \ n.mark \mapsto 1))) \\ \Rightarrow \ \textbf{reads fpt}(dag(d.l)) \cdot / \cdot \textbf{modifies} \ unmarked(d.r) \end{array} \tag{102}$$

By using the rules I_{sc}, FRM_u, $CONSEQ_u$ and $SubEff_u$, we derive

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

$$\tag{103}$$

Again, we need to prove the side condition in order to use the rule $SEQ1_u$, i.e., $unmarked(d.r)$ is

$(d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \ \&\& \ dag(d.r)))/(\textbf{region}\{d.mark\}, unmarked(d.l))$-immune.

By the definition of immune (Def. 11), we need to prove:

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

$$(d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \,\&\&\, dag(d.r)))\ implies\ efs(RE) \cdot / \cdot unmarked(d.r).$$

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

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

$$(d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \,\&\&\, dag(d.r)))\ implies\ efs(RE) \cdot / \cdot unmarked(d.r).$$

We show that it is true by contradiction. Suppose under the assumption, there is some *RE* that has the form **region**$\{d.f_1.\cdots.f_n.mark\}$, then *efs(RE)* is **region**$\{d.f_1.\cdots.f_n\}$, where $f_i \in \{l, r\}$, and $1 \leqslant i \leqslant n$. Moreover $d.f_1.\cdots.f_n$ has the type Dag. However, all the regions in $unmarked(d.r)$ has the form **region**$\{d.f_1.\cdots.f_m.mark\}$, and $d.f_1.\cdots.f_m.mark$ 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 $SEQ1_u$, we derive

$$
\begin{array}{l}
[\textbf{reads alloc}{\downarrow}, d] \\
\{d \neq null * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \,\&\&\, dag(d.r))\} \\
\texttt{d.mark := 1; mark(d.l); mark(d.r);} \\
\vdash_u \left\{
\begin{array}{l}
(d.r \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.r)) \Rightarrow n.mark \mapsto 1))\ \&\& \\
(d.l \neq null \Rightarrow \forall n : Dag.(\textbf{region}\{n.mark\} \leqslant \textbf{fpt}(dag(d.l)) \Rightarrow n.mark \mapsto 1)) \\
\&\& \ (d.mark \mapsto 1 * d.l \mapsto i * d.r \mapsto j * (dag(d.l) \,\&\&\, dag(d.r)))
\end{array}
\right\} \\
[\textbf{modifies region}\{d.mark\}, unmarked(d.l), unmarked(d.r)]
\end{array}
\tag{104}
$$

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 **region**$\{x.mark\}$ is contained in both write effects. To use the sequence rule, we need to show that ε_r is $dag(d)/\varepsilon_l$-immune. By the definition of immune (Def. 11), we need to show that for all **modifies** $RE \in \varepsilon_r :: RE$ is $dag(d)/\varepsilon_l$-immune. In this case, we need to show that $dag(d)\ implies\ efs(\textbf{region}\{x.mark\}) \cdot / \cdot \varepsilon_1$, by Def. 11. By the definition of read effects, $efs(\textbf{region}\{x.mark\}) = \textbf{reads}\ x$. There are two cases.

1. $x = d$. In this case, we need to show that $dag(d)$ implies **reads** $d \cdot / \cdot$ **region**$\{d.mark\}$, which is true.
2. $x = d.f_1.\cdots.f_n$, where $f_1 \cdots f_n$ are either the field name l or the field name r. In this case, we need to show that $dag(d)$ implies (**reads region**$\{d.f_1.\cdot.f_n\}, efs(d.f_1.\cdots.f_{n-1})) \cdot / \cdot$ **region**$\{d.f_1.\cdots.f_n.mark\}$, which is true because, the field f_n has type Dag, the field $mark$ has the type **bool**, thus **region**$\{d.f_1.\cdots.f_n\} !! $ **region**$\{d.f_1.\cdots.f_n.mark\}$. Similarly, the regions contained in the read effect $efs(d.f_1.\cdots.f_{n-1})$ are all disjoint with the region **region**$\{d.f_1.\cdots.f_n.mark\}$.

14.5 An Integrated Specification and Verification 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. 30 on the following page. Two shop objects share one menu object. Taking orders and performing services only read the menu. Thus, we can prove that executing shop1's method service preserves shop2's property, as the write effects of shop1.service in Fig. 36 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. 33. 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

```
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();
```

Fig. 30: The client code

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. 32. Moreover, the use of separating conjunction makes the specifications concise.

Now we explain 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, we 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. 32. The class `List<T>` is implemented by a list of `Node<T>` defined in Fig. 3 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 Fig. 31. The specifications and implementations of the class `List<T>` are shown in Fig. 32 on the next 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. 33 on the next page shows an implementation of `List<T>`'s iterator. The field `curr` denotes the cursor position.

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. 31: Selected specifications for the class `Node<T>`.

Fig. 34 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 Fig. 6. 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, we omit the details of `Menu`. 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

```
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) then { 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 */
```

Fig. 32: A generic linked-list specified 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 */
}
```

Fig. 33: The class ListIterator specified in the style of FRL.

```
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 */
}
```

Fig. 34: A generic dictionary specified in the style of SSL.

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 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 [29,30], this can be specified by declaring these three parameters as ghost fields and updating them when it is needed.

```
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 */
}
```

Fig. 35: The class Order and its specification

Abstraction Although information hiding and abstraction are not a focus of this paper, they figure prominently in other works on SL [48,49]. This technique can also be handled in our approach. In our example, we assume 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. Fig. 37 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. 33 exposes the field curr that is supposed to be a private field. This can be solved by (at least) two established methodologies: *data groups* [34,40] and *model variables* [19,33]. Following JML [31], we explore the second approach. 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;
```

```
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);
   }

  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 */
}
```

Fig. 36: The class Shop specified in a mixed style.

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:

<p align="center">modifies region{this._curr};</p>

And also the specifications that use **this**.curr can be rewritten by substituting **this**._curr for it. For simplicity, in the remainder of this paper, we just use program fields and consider that all such fields that are publicly accessible in specifications.

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

Fig. 37: Visible predicate names to clients.

Interoperation The specifications in this example are written in different styles, nevertheless, with our approach they can be combined and used in verification. As an example, we will now 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 14.2) 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. 34 is encoded in UFRL as:

$$[\textbf{reads fpt}(vDic(m)\&\&k \in m)]$$
$$\{vDic(?m)\&\&k \in m\} \text{ v } := \text{ lookup(k:Key);} \{vDic(m)\&\&v = m[k]\} \tag{105}$$
$$[\textbf{modifies fpt}(vDic(m)\&\&k \in m)]$$

By using the rule $SubEff_u$, we derive:

$$[\textbf{reads alloc}\!\downarrow]$$
$$\{vDic(?m)\&\&k \in m\}$$
$$\vdash_u lookup(k : Key) \textbf{ returns } (v : Value) \tag{106}$$
$$\{vDic(m)\&\&v = m[k]\}$$
$$[\textbf{modifies fpt}(vDic(m)\&\&k \in m)]$$

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:

$$[\textbf{reads alloc}\!\downarrow]$$
$$\{cshop(?lseq, ?oseq, ?m)\&\&item \in m\}$$
$$\vdash_u takeOrder(item : \textbf{int}, table : \textbf{int})\textbf{returns}(ret : Order) \tag{107}$$
$$\{cshop(lseq + [ret], oseq + [table, item, m.[item]], m)\}$$
$$[\textbf{modifies fpt}(cshop(lseq, oseq, m)\&\&item \in m)]$$

Proceeding to the verification of the body of takeOrder, we first assume its precondition:

$$cshop(lseq, oseq, m)\&\&item \in m, \tag{108}$$

which implies the precondition of menu.lookup by using the definition of the predicate cshop in Fig. 36. For the write effects, by the definition of the predicate cshop again, we have:

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

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:

$$menu.vDis(m)\&\&itemId = m[item]. \tag{109}$$

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}, we have:

$$(menu.vDis(m)\&\&itemId = m[item]) * \mathbf{ret}.vOrder([table, itemId, 0]) * orders.vList(lseq). \tag{110}$$

Eq. (110) implies the precondition of the method `orders.add(`**ret**`)`. For the write effects, according to Section 14.2, its specification is encoded as:

$$[\mathbf{reads\ alloc\!\downarrow}]\{vList(?vlst)\}\ \texttt{add(t)};\ \{vList(vlst + [t])\}[\mathbf{modifies\ fpt}(vList(vlst))]. \tag{111}$$

Together with the definition of the predicate `cshop` in Fig. 36, we have:

$$\mathbf{fpt}(orders.vlst(lseq)) \leqslant \mathbf{fpt}(cshop(lseq, oseq, m)\&\&item \in m), \tag{112}$$

Thus, `orders.add` is allowed in the body of `takeOrder`. After finishing executing it, its postcondition gets assumed. And using the rule I_{sc}, we have:

$$(menu.vDis(m)\&\&itemId = m[item]) * \mathbf{ret}.vOrder([table, itemId, 0]) * orders.vList(lseq + [\mathbf{ret}])$$
$$\tag{113}$$

As Eq. (113) implies the postcondition of `takeOrder`, the implementation is verified.

Verifying a Client of `CoffeeShop` For simplicity, we 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. 30. 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:

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

Then we can use the rule FRM_u and the rule $CONSEQ_u$ to prove 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.

15 Discussion

This section discusses the idea of encoding the magic wand. Fig. 38 on the next page specifies a procedure `sum_it` that calculates the sum of `val` of all the nodes along a linked-list. This example is an adaptation from the one in Schwerhoff and Summers' work [55]. The procedure is implemented by a loop statement that traverses a list and sums up values, where a local variable `curr` points to the current node which the program is visiting. A loop invariant specifies that the sum of the visited list is the subtraction the one of the to-be-visited from the one of the whole list. Thus, an additional predicate `lstseg` specifies a fragment of a list starting with the node `s` to the node `e`; and a helping function `sum_seq` recursively sums up the values in a given sequence. Fig. 39 on page 70 depicts the dynamic situation.

To avoid specifying additional predicate `lstseg`, inspired by Schwerhoff and Summers' work [55], the specification can be improved as follows.

The use of the magic wand, `lst(curr,?cvlst)*(lst(curr,cvlst)-*lst(`**this**`,vlst))`, has the same semantics as **fpt**`(lstseg(`**this**`,curr))`+**fpt**`(lst(curr,?cvlst))`=**fpt**`(lst(`**this**`,vlst))`

```
method sum_it() returns (ret: int)
  requires lst(this, ?vlst);
  modifies region{};
  ensures ret = sum_seq(vlst);
{
  ret := 0;
  curr := this;
  while (curr ≠ null)
    invariant lst(curr, ?cvlst) * (lst(curr, cvlst) -* lst(this, vlast));
    invariant curr = null ⇒ ret = sum(this);
    invariant curr ≠ null ⇒ ret = sum(vlst) - sum(cvlst);
  {
     ret := ret + curr.val;  curr := curr.next;
  }
}
predicate lst(n : Node, se : seq){
   (n = null ⇒ |se| = 0) && (n ≠ null ⇒ n.val↦se[0] * lst(n.next, se[1..])
}

predicate lstseg(s: Node, e : Node){
   (s = null ⇒ s = e) && (∃ i. s.val↦i * lst(s.next, e))
}

function sum_seq(se: seq) : returns (ret : int)
{
  if se = [] then ret := 0;
  else ret := se[0] + sum_seq(se[1..]);
}

class Node {
  var val: int;   var next: Node;

  method sum_it() returns (ret: int)
    requires lst(n, ?vlst);
    modifies ∅;
    ensures ret = sum(vlst);
  {
    var curr: Node; ret := 0; curr := this;

    while (curr ≠ null)
      invariant fpt(lstseg(this, curr))+fpt(lst(curr, ?cvlst))
              = fpt(lst(this, vlst));
      invariant curr = null ⇒ ret = sum_seq(vlst);
      invariant curr ≠ null ⇒ ret = sum_seq(vlst)-sum_rec(cvlst);
    {
      ret := ret + curr.val; curr := curr.next;
    }
  }
}
```

Fig. 38: A sum example specified by UFRL.
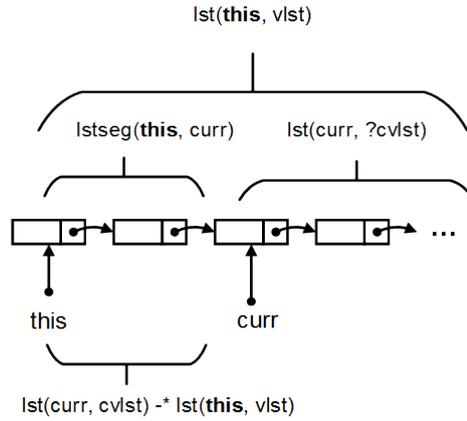
Fig. 39: The pictorial description of summing up values along a linked-list.

in its original specificaton. That specifies that `curr` points somewhere in the list, and what has been visited is: `lst(curr,?cvlst)` or **fpt**`(lstseg(`**this**`,curr))`, and what has not is: `lst(curr,cvlst)-*lst(`**this**`,vlst))` or **fpt**`(lst(curr,cvlst))`. Thus, the semantic footprint of `lst(curr,cvlst)-*lst(`**this**`,vlst)` has to be the one of subtracting $lst(curr, cvlst)$ from $lst(\mathbf{this}, vlst)$. We discuss its soundness by definition of the magic wand :

$$\sigma, H{\upharpoonright}r \models_{sl} a_1 \mathbin{-\!\!*} a_2 \iff \text{for all } h', r' :: (dom(h') \cap r = \varnothing \text{ and } \sigma, h'{\upharpoonright}r' \models_{sl} a_1 \text{ implies } \sigma, H{\upharpoonright}r \cdot h'{\upharpoonright}r' \models_{sl} a_2),$$

where $r \subseteq dom(H)$, a_1 and a_2 are SL assertions. There are three possible extensions of the partial heap $H{\upharpoonright}r$ shown in Fig. 40. The extension shown on the left is inside the global heap H which is the case in the example; on the right is outside H; and in the middle is a union of inside and outside H. We leave the encoding of the three cases as future work.

(a) inside extension (b) inside and out side extension   (c) outside extension
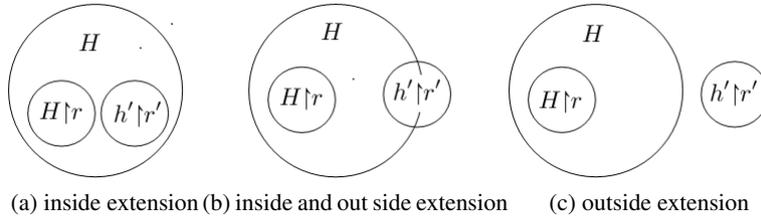
Fig. 40: All possible extensions of the partial heap $H{\upharpoonright}r$ for magic wand.

16   Related Work

16.1   Related Work on Framing

There are several approaches to framing that have been described in the formal methods literature. Historically specification languages such as VDM [28] and interface specification languages in the Larch family [24] specify frames for procedures by writing a clause in the specification that names the variables that are allowed to be changed during the procedure's execution; all other locations must be unchanged. However, such a simple approach does not easily generalize to layered structures of mutable objects. One approach that works with object structures

is ownership [44], which only allows a designated owner object to mutate the objects that make up part of a complex object structure. The universe type system [42] 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 invariants [43]. However, the universe type system and other approaches based on ownership have difficulties in specifying and verifying shared data structures.

The Boogie methodology [8,9] has a dynamic notion of ownership; each object has an "owner" field that points to that object's owner. Although the Boogie methodology eases the problem of dealing with shared mutable data structures by easing the transfer of ownership, it introduces a fair amount of overhead and complexity in writing specifications.

In the approach of dynamic frames [29,30], the program dynamically tracks sets of locations (regions) expressed by specification variables (or 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.

16.2 Related Work on Region Logic

Our work is partly based on the work of Banerjee et al. on region logic (RL) [3,4]. However, there are several key differences between FRL (and UFRL) and this work on the 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, such as the points-to assertion. Using sets of locations also matches specification languages in which frames are specified using such sets, like JML [18].
2. The meaning of the points-to predicate in RL is two-valued and classical. Our semantics if two-valued, but intuitionistic in order to support garbage collection.
3. 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 $\mathbf{rd}\{x\}`f, x$, 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 a declared field name.
4. Finally, RL does not have conditional region expressions, which make FRL more expressive for specifying the frames of SL assertions that involve implication.

FRL (UFRL) and RL also share lots of similarities.

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 $\mathbf{rd}\, G`f$ allows one to read the field of objects in $G$ [4, p.22]; e.g., the RL read effect $\mathbf{rd}\, x`f$ is equivalent to the FRL read effect **region**$\{x.f\}$.
3. Although the semantics of the points-to predicate are different, their read effects 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, are defined by $\mathtt{fptp}(x.f = E) = \mathbf{rd}\, x, x.f, \mathtt{ftpt}(E)$, where $\mathbf{rd}$ is the keyword for read effects (our work uses **reads** instead). The form $\mathbf{rd}\, x.f$ abbreviates the form $\mathbf{rd}\,\{x\}`f$ [4, p.23]. Although $x`f$ may not be the same as our **region**$\{x.f\}$ as explained previously, $\mathbf{rd}\, x`f$ is semantically equivalent to as our **reads region**$\{x.f\}$.
4. RL and FRL (and UFRL) have similar definitions of agreement, frame validity, separator, immunity, and Hoare-formula. Therefore, our proof rules are quite similar as well. In particular, we have semantically equivalent frame conditions for the proof axioms.

Rosenberg's work [54] implements a semi-decision procedure for RL as a plugin inside the SMT solver Z3. Similarly, FRL and UFRL expressions could also be encoded into SMT, but such an encoding is beyond the scope of this paper.

16.3   Related work on Separation Logic with Permissions

Our work also draws on work in separation logic (SL). There has been much work on automating SL using first-order tools [12,13,14,15,20,22,51]. Our results show another way of encoding SL into first order logic, via UFRL. An inspiration for our work was the work of Parkinson and Summers [51], who showed a relationship between SL and the methodology of Chalice [38] that combines the core of implicit dynamic frames [57][6] with fractional permissions and concurrency. They encode a separation logic fragment (similar to SSL) into the language of implicit dynamic frames by defining a total heap semantics of SL, which agrees with the weakest pre-condition semantics of the implicit dynamic frames language. While their work did not connect SL and RL, our results go further than the analogous results in their paper, as we also formalize a translation of axioms and proof rules for a Hoare logic based on SL and show that proofs can be soundly translated (Theorem 49).

16.4   Related Work based on Dynamic Frames

Leino's Dafny language [35,36] 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. 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. The dynamic frames approach used by Smans et al. [58], 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. We consider data abstraction and information hiding to be orthogonal to the problems discussed in this paper, as standard solutions can be applied [1,2,34,39]. While their language has much of the power of FRL, they do not formally connect SL with their language, and do not address the problem of allowing specifications in both SL and RL to interoperate.

The KeY tool [11,60] extends JML with dynamic frames. It introduces a type `\locset` that stands for sets of memory locations. Recently, Mostowski and Ulbrich [41] 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.

16.5   Other Works

Tuerk's work [59] presents an inference rule that allows local reasoning to verify loops. Instead of using loop invariants, the inference rule uses pre- and post-conditions. It would be interesting to introduce this inference to FRL (and UFRL). However, we leave that as future work.

17   Conclusion and Future work

This work introduces 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.

In addition to the future work discussed previously, we are planning to work on a prototype implementation of UFRL with an automatic theorem prover such as Z3 [21] or CVC4 [10]. That future work will rely on an encoding of UFRL into first-order logic, which can be accomplished by making use of the following three observations. First, the quantifiers in FRL (and UFRL) are first-order; this allows implementations to use SMT solvers. Second, the operators on regions (union, difference and intersection) can be translated into corresponding set operations. Third, UFRL can be used in automated verification tools that prove programs in a method-modular way. That is, when verifying a method call, its precondition is asserted and its frame and postcondition are assumed. Alternatively, instead of directly accumulating effects and composing each proof rule, verification tools for UFRL can be

---

[6] Implicit dynamic frames is considered as a separation logic approach by Semans et al. [56].

implemented by computing weakest preconditions or by symbolic execution. Finally, we plan to adopt Dafny's logical encoding that can reason about fixpoints automatically [53].

It would also be interesting future work to develop a formal comparison between region logic and Dafny [35,36]. Other future work includes extending UFRL by allowing mutually-recursive predicates, reasoning about subtyping and dynamic dispatch, and incorporating ideas from our work on UFRL into JML [18].

18   Acknowledgments

References

1. 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.

2. A. Banerjee and D. A. Naumann. *Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, chapter A Logical Analysis of Framing for Specifications with Pure Method Calls, pages 3–20. Springer International Publishing, Cham, 2014.

3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411, New York, NY, 2008. Springer-Verlag.

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 Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, New York, NY, 2006. Springer-Verlag.

9. 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.

10. 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.

11. 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.

12. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 115–137, Berlin, Heidelberg, 2006. Springer-Verlag.

13. 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.

14. J. Berdine, C. Calcagno, P. W. Ohearn, and Q. Mary. Symbolic execution with separation logic. In *In APLAS*, pages 52–68. Springer, 2005.

15. 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.

16. 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.

17. 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.

18. 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.

19. 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.

20. 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.

21. 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.

22. 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.

23. 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.

24. J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, Sept. 1985.

25. 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.

26. 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.

27. B. Jacobs, J. Smans, and F. Piessens. The verifast program verifier: A tutorial, 2010.

28. C. B. Jones. *Systematic software development using VDM*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

29. 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.

30. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.

31. 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.

32. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

33. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

34. 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.

35. 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.

36. 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.

37. 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, experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126, Berlin, 2010. Springer-Verlag.

38. 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.

39. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, Sept. 2002.

40. 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.

41. 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.

42. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2002.

43. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, Oct. 2006.

44. 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.

45. 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.

46. 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.

47. 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.

48. 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.

49. 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.

50. M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. The author's Ph.D. dissertation.

51. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.

52. 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.

53. K. R. M. L. Richard L. Ford. Dafny reference manual (draft).

54. 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.

55. M. Schwerhoff and A. J. Summers. Lightweight Support for Magic Wands in an Automatic Verifier. In *ECOOP*, volume 37 of *Leibniz International Proceedings in Informatics*, pages 614–638. Schloss Dagstuhl Leibniz-Zentrum fr Informatik, 2015.

56. 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.

57. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

58. 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.

59. T. Tuerk. Local reasoning about while-loops. In *In International Conference on Verified Software: Theories, Tools and Experiments - Theory Workshop (VS-Theory*, 2010.

60. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

61. 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.

A   Proof of Theorem 32

Theorem 32: *An assertion in SL is* supported *if and only if it has semantic footprint.*

Proof. Let (σ, h) be a state and a be an assertion in SL, such that $\sigma, h \models_s a$. Let $\mathcal{H} \stackrel{\text{def}}{=} \{h' | h' \subseteq h \wedge \sigma, h' \models_s a\}$. 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* $\mathcal{H}_1 \subseteq \mathcal{H}_2$. We 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 31. Next we show that $dom(H_a)$ is a's semantic footprint. Let $\mathcal{R} \stackrel{\text{def}}{=} \{r | \sigma, h \restriction r \models_s a\}$. 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 $\mathcal{DOM}(\{h_1, h_2, \cdots, h_n\}) = \{dom(h_1), dom(h_2), \cdots, dom(h_n)\}$. 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* $\mathcal{DOM}(\mathcal{H}_p) \leqslant \mathcal{DOM}(\mathcal{H}_1) \bigsqcap_{\mathcal{R}} \mathcal{DOM}(\mathcal{H}_2)$. $\qquad \square$

B Proof of Theorem 40

Theorem 40: Let a be an assertion in SSL. Then $\sigma, h \models_s a$ *iff* $\sigma, h \models_u \text{TR}[\![a]\!]$.

Proof. The proof is by the induction on the assertion's structure. Here we only show 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, $\sigma, h \models_s a_i$ *iff* $\sigma, h \models_u \text{TR}[\![a_i]\!]$. We first prove it from the left side to the right side. Assume $\sigma, h \models_s a_1 * a_2$. We need to prove $\sigma, h \models_u \text{TR}[\![a_1]\!]$ && $\text{TR}[\![a_2]\!]$ && $(fpt_s(a_1)\,!\,!fpt_s(a_2))$.

$\sigma, h \models_s a_1 * a_2$
iff $\langle$by the semantics of separation logic (Def. 25)$\rangle$
   $exists\ h_1, h_2 :: (h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2)$
iff $\langle$by let fresh variables, h_1 and h_2, be the witnesses of the existential variables.$\rangle$
   $h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2$
implies$\langle$by truth of assertions is preserved under heap extension (Lemma 29)$\rangle$
   $h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2\ and\ \sigma, h \models_s a_1\ and\ \sigma, h \models_s a_2$
implies$\langle$by let r_1 and r_2 be fresh, and by Theorem 37$\rangle$
   $h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2\ and\ \sigma, h \models_s a_1\ and\ \sigma, h \models_s a_2\ and$
   $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1\ and\ \mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2$
implies$\langle$by Corollary 38 and $h_1 \perp h_2$$\rangle$
   $h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2\ and\ \sigma, h \models_s a_1\ and\ \sigma, h \models_s a_2\ and$
   $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1\ and\ \mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2\ and\ r_1\,!\,!r_2$
iff $\langle$by inductive hypothesis$\rangle$
   $h_1 \perp h_2\ and\ h = h_1 \cdot h_2\ and\ \sigma, h_1 \models_s a_1\ and\ \sigma, h_2 \models_s a_2\ and\ \sigma, h \models_s a_1\ and \sigma, h \models_s a_2\ and$
   $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1\ and\ \mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2\ and\ r_1\,!\,!r_2\ and\ \sigma, h \models_u \text{TR}[\![a_1]\!]\ and\ \sigma, h \models_u \text{TR}[\![a_2]\!]$
iff $\langle$by the semantics of UFRL (Fig. 14)$\rangle$
   $\sigma, h \models_u \text{TR}[\![a_1]\!]$ && $\text{TR}[\![a_2]\!]$ && $(fpt_s(a_1)\,!\,!fpt_s(a_2))$
iff $\langle$by Mapping from SSL to UFRL (Def. 34)$\rangle$
   $\sigma, h \models_u \text{TR}[\![a_1 * a_2]\!]$

Next, we prove it from the right side to the left side. Assume $\sigma, h \models_u \text{TR}[\![a_1]\!]$ && $\text{TR}[\![a_2]\!]$ && $(fpt_s(a_1)\,!\,!fpt_s(a_2))$. We need to prove $\sigma, h \models_s a_1 * a_2$.

$\sigma, h \models_u \text{TR}[\![a_1]\!]$ && $\text{TR}[\![a_2]\!]$ && $(fpt_s(a_1)\,!\,!fpt_s(a_2))$
iff $\langle$by the semantics of UFRL (Fig. 14)$\rangle$
   $\sigma, h \models_u \text{TR}[\![a_1]\!]\ and\ \sigma, h \models_u \text{TR}[\![a_2]\!]\ and\ \mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1\ and\ \mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2\ and\ r_1\,!\,!r_2$
iff $\langle$by inductive hypothesis$\rangle$

$\sigma, h \vDash_u \mathrm{TR}[\![a_1]\!]$ and $\sigma, h \vDash_u \mathrm{TR}[\![a_2]\!]$ and $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1$ and $\mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2$ and $r_1 \,!\,!\, r_2$
and $\sigma, h \vDash_u a_1$ and $\sigma, h \vDash_u a_2$

iff   ⟨by Corollary 39⟩
  $\sigma, h \vDash_u \mathrm{TR}[\![a_1]\!]$ and $\sigma, h \vDash_u \mathrm{TR}[\![a_2]\!]$ and $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1$ and $\mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2$ and $r_1 \,!\,!\, r_2$
  and $\sigma, h{\restriction}r_1 \vDash_u a_1$ and $\sigma, h{\restriction}r_2 \vDash_u a_2$

implies ⟨by $h{\restriction}r_2 \subseteq h{\restriction}(dom(h) - r_1)$ and truth of assertions is closed under heap extension (Lemma 29)⟩
  $\sigma, h \vDash_u \mathrm{TR}[\![a_1]\!]$ and $\sigma, h \vDash_u \mathrm{TR}[\![a_2]\!]$ and $\mathcal{E}[\![fpt_s(a_1)]\!](\sigma) = r_1$ and $\mathcal{E}[\![fpt_s(a_2)]\!](\sigma) = r_2$ and $r_1 \,!\,!\, r_2$
  and $\sigma, h{\restriction}r_1 \vDash_u a_1$ and $\sigma, h{\restriction}r_2 \vDash_u a_2$ and $\sigma, h{\restriction}(dom(h) - r_1) \vDash_u a_2$

implies ⟨by Corollary 38, $r_1 \cup (domh - r_1) = dom(h)$, and we find $h_1 = h{\restriction}r_1$ and $h_2 = h{\restriction}(dom(h) - r_1)$⟩
  $exists\ h_1, h_2 :: (h_1 \bot h_2$ and $h = h_1 \cdot h_2$ and $\sigma, h_1 \vDash_s a_1$ and $\sigma, h_2 \vDash_s a_2)$

iff   ⟨by the semantics of separation logic (Def. 25)⟩
  $\sigma, h \vDash_s a_1 * a_2$

□

C Proof of Lemma 43

Lemma 43: Let a and a' be assertions and S be a statement, such that $\vDash_s \{a\}S\{a'\}$. Let (σ, H) be an arbitrary state. If $\sigma, H \vDash_s a$ and $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$, then:

1. *for all* $x \in dom(\sigma)::$ $\sigma'(x) \neq \sigma(x)$ *implies* $x \in \mathrm{mods}(S)$.
2. *for all* $(o, f) \in dom(H)::$ $H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$.
3. *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))::$ $(o, f) \in (dom(H') - dom(H))$.

Proof. Let a, a', S, (σ, H) be given, such that $\vDash_s \{a\}S\{a'\}$. Let (σ', H') be such that $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H)$. For property 1, we must show that *for all* $x \in dom(\sigma)::$ $\sigma'(x) \neq \sigma(x) implies x \in \mathrm{mods}(S)$. The proof is by induction on the structure of the statement S and the definition of $\mathrm{mods}(S)$. There are 6 base cases.

1. (*SKIP*) In this case, S has the form **skip**;. According to its semantics Fig. 11 on page 13, $\sigma = \sigma'$. Thus, it is vacuously true.
2. (*VAR*) In this case, S has the form **var** $x : T$;. According to its semantics Fig. 11 on page 13, $\sigma' = \mathrm{Extend}(\sigma, x, default(T))$. Thus, it is vacuously true, as Extend only extends σ by definition.
3. (*ALLOC*) In this case, S has the form $y := $ **new** C;, for some variable y. According to the semantics Fig. 11 on page 13, $\sigma' = \sigma[y \mapsto l]$, where l is some new location. Thus, no other variables are mapped to different values by σ'. For y, we have $\sigma'(y) \neq \sigma(y)$, and $y \in \mathrm{mods}(y := $ **new** $C;) = \{y\}$, according to Fig. 26.
4. (*ASGN*) In this case, S has the form $y := e$; for some variable y. According to its semantics Fig. 11 on page 13, $\sigma' = \sigma[y \mapsto v]$, where v is the value of e. For y, $\sigma'(y) \neq \sigma(y)$, and $y \in \mathrm{mods}(y := e) = \{y\}$, according to Fig. 26.
5. (*UPD*) In this case, S has the form $y.f := e$;. According to its semantics Fig. 11 on page 13, $\sigma' = \sigma$. Thus, it is vacuously true.
6. (*ACC*) In this case, S has the form $y := x'.f$;. According to its semantics Fig. 11 on page 13, $\sigma' = \sigma[y \mapsto v]$, where v is the value of $x'.f$. Thus, $\sigma'(y) \neq \sigma(y)$, and $y \in \mathrm{mods}(y := x'.f) = \{y\}$, according to Fig. 26.

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* $x \in \mathrm{mods}(S_i)$. 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. 11 on page 13, if $\mathcal{E}_u[\![e]\!](\sigma, H)$ is true, then the result follows from the inductive hypothesis applied to S_1. Similarly if $\mathcal{E}_u[\![e]\!](\sigma, H)$ is false, the result also follows similarly.
2. (*WHILE*) In this case, S has the form **while** e $\{S\}$. According to its semantics Fig. 11 on page 13, there exists $n \geq 0$, such that $\sigma' = \sigma_n$ and $\mathcal{E}_u[\![e]\!](\sigma_n, H_n) = 0$. We prove it by induction on n. The base case is $n = 0$. According to the semantics Fig. 11 on page 13, $\sigma' = \sigma$. Thus, it is vacuously true. For the inductive case, we assume *for all* $x \in dom(\sigma) :: \sigma_{n-1}(x) \neq \sigma(x)$ *implies* $x \in \mathrm{mods}(S)$. And by the inductive hypothesis, *for all* $x \in dom(\sigma_{n-1}) :: \sigma_n(x) \neq \sigma_{n-1}$ *implies* $x \in \mathrm{mods}(S)$. Thus, *for all* $x \in dom(\sigma) :: \sigma(x) \neq \sigma'(x)$ *implies* $x \in \mathrm{mods}(S)$.

3. (*SEQ*) In this case, S has the form $S_1 S_2$. By definition, $\mathrm{mods}(S_1 S_2) = \mathrm{mods}(S_1) \cup \mathrm{mods}(S_2)$. According to the statement's semantics Fig. 11 on page 13, 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* $x \in \mathrm{mods}(S_1)$, and *for all* $x \in dom(\sigma_1) :: \sigma'(x) \neq \sigma_1(x)$ *implies* $x \in \mathrm{mods}(S_2)$. Thus, *for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* $(x \in \mathrm{mods}(S_1 S_2))$.

For property 2, we must show that *for all* $(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$. We assume that $\sigma, H \vDash_s \{a\} S \{a'\}$, $\sigma, H \vDash_s a$ and $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$. The proof is done in calculational style, starting from the assumptions.

$\sigma, H \vDash_s \{a\} S \{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by assumption $\vDash_s \{a\}S\{a'\}$, thus $(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *iff* $\sigma, H \vDash_s \{a\} S \{a'\}\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by Corollary 39: $\sigma, H \vDash_s a$ *iff* $(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$

iff $\langle$by the definition of SSL valid Hoare-formula (Def. 42)$\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and if
$((\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H'' \vDash_s a')$.

iff $\langle$by frame property of SL$\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
$((\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *and* $\sigma', H'' \vDash_s a')$ *and*
$H'' \perp H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$

implies $\langle$by A and B implies $B\rangle$
$H'' \perp H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$

iff $\langle$by *for all* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: \ldots$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ is a tautology$\rangle$
$H'' \perp H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and*
for all $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H''[o, f] \neq H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$

implies $\left\langle \text{by} \begin{array}{l} H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \text{ and} \\ (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \cap \mathcal{E}[\![fpt_s(a)]\!](\sigma) = \varnothing \end{array} \right\rangle$
$H'' \perp H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and*
for all $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$

implies $\langle$by Corollary 38, $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(H)$, twice$\rangle$
$H'' \perp H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H{\restriction}(dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and*
for all $(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$

implies $\langle$by A and B *implies* $B\rangle$
for all $(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$

For property 3, we must show that *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$. We assume that $\sigma, H \vDash_s \{a\} S \{a'\}$, $\sigma, H \vDash_s a$ and $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$. The proof is done in calculational style, starting from the assumptions.

$\sigma, H \vDash_s \{a\} S \{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by assumption $\vDash_s \{a\}S\{a'\}$, thus $(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *iff* $\sigma, H \vDash_s \{a\} S \{a'\}\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$

iff $\langle$by Corollary 39: $\sigma, H \vDash_s a$ *iff* $\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s a\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s a$

iff $\langle$by the definition of SSL valid Hoare-formula (Def. 42)$\rangle$
$(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and if
$((\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H'' \vDash_s a')$.

iff $\langle$by frame property of SL$\rangle$

$(\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *and* $\sigma, H \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H) = (\sigma', H')$ *and*
$\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s a$ *and* $\mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
$((\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *and* $\sigma', H'' \vDash_s a')$ *and*
$H'' \bot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$

implies ⟨by $A \wedge B$ implies B⟩
$\sigma', H'' \vDash_s a'$ *and* $H'' \bot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$

iff   ⟨by *(for all* $(o, f) \in (r' - r) :: (o, f) \in (r' - r))$ is a tautology⟩
$\sigma', H'' \vDash_s a'$ *and* $H'' \bot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$
and *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma)) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$

implies ⟨by Corollary 38, $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(H)$⟩
$\sigma', H'' \vDash_s a'$ *and* $H'' \bot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$
and *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma)) - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma) - dom(H))$

implies ⟨by $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and Corollary 38⟩
$\sigma', H'' \vDash_s a'$ *and* $H'' \bot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ *and* $H' = H'' \cdot H \restriction (dom(H) - \mathcal{E}[\![fpt_s(a)]\!](\sigma))$
and *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$

implies ⟨by A and B implies B⟩
$(\textit{for all } (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H)))$

□

D   Proof of Theorem 47

Theorem 47: Let S be a statement, and let a and a' be assertions in SSL, such that $\vDash_s \{a\}S\{a'\}$. Let r be a region variable. Let (σ, H) be an arbitrary state. If $\sigma, H \vDash_u \text{TR}[\![a]\!]$ *implies* $r = fpt_s(a)$ and $r \notin \text{mods}(S)$, then

$\sigma, H \vDash_s \{a\} S\{a'\}$ *iff*
$\qquad\qquad \sigma, H \vDash_u \{\text{TR}[\![a]\!]\}S\{\text{TR}[\![a']\!]\}[\,\textbf{modifies}\, fpt_s(a), \text{mods}(S), \textbf{fresh}\, (fpt_s(a') - r)][fpt_s(a)].$

Proof. Assume that $\sigma, H \vDash_s \{a\}S\{a'\}$, and that r is a region variable such that $\sigma, H \vDash_u \text{TR}[\![a]\!]$ *implies* $r = fpt_s(a)$ and $r \notin \text{mods}(S)$.
We prove the lemma by mutual implication. First we prove that the left side implies the right side.

$\qquad \sigma, H \vDash_s \{a\} S \{a'\}$
iff   ⟨by assumption $\vDash_s \{a\}S\{a'\}$, thus $(\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *iff* $\sigma, H \vDash_s \{a\} S \{a'\}$⟩
$\qquad (\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$
iff   ⟨by the definition of SSL valid Hoare-formula (Def. 42)⟩
$\qquad (\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
$\qquad$ *if* $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_s a'$
implies ⟨by Lemma 43⟩
$\qquad (\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
$\qquad$ *if* $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_s a'$ *and*
$\qquad$ *for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* $x \in \text{mods}(S)$ *and*
$\qquad$ *for all* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$\qquad$ *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)))$
implies ⟨by termination monotonicity as $H = (H - H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \cdot H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ *and*
$\qquad\qquad (H - H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \bot H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)$⟩
$\qquad (\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
$\qquad$ *if* $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_s a'$ *and* $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ *and*
$\qquad$ *for all* $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* $x \in \text{mods}(S)$ *and*
$\qquad$ *for all* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \restriction \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$\qquad$ *for all* $(o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)))$
implies ⟨by the frame property of SL⟩

$(\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s a$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_s a'$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$H'' = H' \cdot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)))$

implies⟨by Corollary 39⟩

$\sigma, H \vDash_s a$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_s a'$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$H'' = H' \cdot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)))$

iff   ⟨by Theorem 40, twice⟩

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$
 and $H'' = H' \cdot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$
and $for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)))$

implies⟨by $dom(H') - \mathcal{E}[\![fpt_s(a)]\!](\sigma) = dom(H') - dom(H)$, because $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(H)$
     by Corollary 38, and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$⟩

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$H'' = H' \cdot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$
and $for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$

implies⟨by $\{(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma) :: H'[o, f] \neq H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)[o, f]\} =$
     $\{(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]\}$, because $\mathcal{E}[\![fpt_s(a)]\!](\sigma) \subseteq dom(H)$
     by Corollary 38, and $H' \bot (H - H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$⟩

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in dom(H) :: H'[o, f] \neq H[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![fpt_s(a)]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$

iff   ⟨by assumption $\mathcal{E}[\![fpt_s(a)]\!](\sigma) = \mathcal{E}[\![r]\!](\sigma)$⟩

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in dom(H) :: H'[o, f] \neq H[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$for\ all\ (o, f) \in (\mathcal{E}[\![fpt_s(a')]\!](\sigma') - \mathcal{E}[\![r]\!](\sigma)) :: (o, f) \in (dom(H') - dom(H))$

iff   ⟨by $\mathcal{E}[\![r]\!](\sigma) = \mathcal{E}[\![r]\!](\sigma')$ because $r \notin \mathrm{mods}(S)$⟩

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ implies $\mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ and
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H \upharpoonright \mathcal{E}[\![fpt_s(a)]\!](\sigma))$, then $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ and $(\sigma', H'') = \mathcal{MS}[\![S]\!](\sigma, H)$ and
$for\ all\ x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)\ implies\ x \in \mathrm{mods}(S)$ and
$for\ all\ (o, f) \in dom(H) :: H'[o, f] \neq H[o, f]\ implies\ (o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
$for\ all\ (o, f) \in \mathcal{E}[\![fpt_s(a') - r]\!](\sigma') :: (o, f) \in (dom(H') - dom(H))$

implies⟨by the definition of UFRL valid Hoare-formula (Def. 16),
     as $freshR(\mathbf{modifies}\ fpt_s(a), \mathrm{mods}(S), \mathbf{fresh}\ (fpt_s(a') - r))$ is $(fpt_s(a') - r)$⟩

$\sigma, H \vDash_u \{\mathrm{TR}[\![a]\!]\}S\{\mathrm{TR}[\![a']\!]\}[\textbf{modifies}\, fpt_s(a), \mathrm{mods}(S), \textbf{fresh}\,(fpt_s(a') - r))][fpt_s(a)]$
where $\mathrm{TR}[\![a]\!]$ *implies* $r = fpt_s(a)$

Next, let $r = fpt_s(a)$. we prove it from the right side to the left side.

$\sigma, H \vDash_u \{\mathrm{TR}[\![a]\!]\}S\{\mathrm{TR}[\![a']\!]\}[\textbf{modifies}\, fpt_s(a), \mathrm{mods}(S), \textbf{fresh}\,(fpt_s(a') - r))][fpt_s(a)]$

implies $\Big\langle$ by the definition of UFRL valid Hoare-formula (Def. 16),
    as $freshR(\textbf{modifies}\, fpt_s(a), \mathrm{mods}(S), \textbf{fresh}\,(fpt_s(a') - r))$ is $(fpt_s(a') - r)$ $\Big\rangle$

$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$ *and*
for all $x \in dom(\sigma) :: \sigma'(x) \neq \sigma(x)$ *implies* $x \in \mathrm{mods}(S)$ *and*
for all $(o, f) \in dom(H) :: H'[o, f] \neq H[o, f]$ *implies* $(o, f) \in \mathcal{E}[\![fpt_s(a)]\!](\sigma)$ and
for all $(o, f) \in \mathcal{E}[\![fpt_s(a') - r]\!](\sigma') :: (o, f) \in (dom(H') - dom(H))$

implies $\langle$by A and B implies $A\rangle$
$\sigma, H \vDash_u \mathrm{TR}[\![a]\!]$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_u \mathrm{TR}[\![a']\!]$

iff $\langle$by Theorem 40, twice$\rangle$
$\sigma, H \vDash_s a$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_s a'$

implies $\langle$by Corollary 39$\rangle$
$\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s a$ *implies* $\mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \neq err$ *and*
if $(\sigma', H') = \mathcal{MS}[\![S]\!](\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma))$, *then* $\sigma', H' \vDash_s a'$

iff $\langle$by the definition of SL validity Hoare-formula (Def. 42)$\rangle$
$\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma) \vDash_s \{a\}\,S\,\{a'\}$

iff $\langle$by assumption $\vDash_s \{a\}S\{a'\}$, thus $(\sigma, H{\restriction}\mathcal{E}[\![fpt_s(a)]\!](\sigma)) \vDash_s \{a\}S\{a'\}$ *iff* $\sigma, H \vDash_s \{a\}\,S\,\{a'\}\rangle$
$\sigma, H \vDash_s \{a\}\,S\,\{a'\}$

$\square$

E   Proof of Theorem 49

Theorem 49: 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, we show that the translated proof axioms and rules are derivable.

1. *SKIP:* by Def. 48, the encoded axiom is the axiom $SKIP_u$.
2. *VAR:* by Def. 48, the encoded axiom is the axiom VAR_u.
3. *ALLOC:* by the rule $ALLOC_s$ and Def. 48, we get the translated rule below:

$$[\textbf{reads}\, fpt_s(a)]$$
$$\vdash_u \{\mathrm{TR}[\![a]\!]\}x := \textbf{new}\, C; \{\mathrm{TR}[\![a * new_s(C, x)]\!]\} \qquad (114)$$
$$[\textbf{modifies}\, fpt_s(a), x, \textbf{alloc}, \textbf{fresh}(fpt_s(new_s(C, x)))]$$
$$\textbf{where}\, x \notin \mathrm{FV}(a)$$

By definition of the predicate $new_s(C, x)$, we know that $fpt_s(new_s(C, x)) = \textbf{region}\{x.*\}$. Given the axiom $ALLOC_u$, we derive Eq. (114) by using the rules FRM_u and $SubEff_u$. The derivation is shown in Fig. 41.

4. *ACC:* by the rule ACC_s and Def. 48, we get the translated rule below:

$$\vdash_u [\textbf{region}\{x'.f\}]\{x'.f = z\}x := x'f; \{x'.f = z\,\&\&\,x = z\}[\textbf{modifies region}\{x'.f\}, x] \qquad (115)$$
$$\textbf{where}\, x \neq x', x' \neq z\, and\, x \neq z$$

where the fresh effect is empty, thus, it is omitted; $fpt_s(x = z)\,!\,!fpt_s(x'.f \mapsto z)$ is true, thus, it is omitted. Given the axiom ACC_u, by definition of read effects for assertions in Fig. 18, we have

($\textbf{reads }x'$, $\textbf{region}\{x'.f\}, z$) *frm* $x'.f = z$. By the side conditions and the definition of separator, we have ($\textbf{reads }(x'$, $\textbf{region}\{x'.f\}, z)$ $\cdot/\cdot$ $\textbf{modifies }x$. Hence $x.'f = z$ is the frame. Using the rules FRM_u and $SubEff_u$, Eq. (115) can be derived. The derivation is shown in Fig. 42.

5. *UPD:* by the rule UPD_s and Def. 48, we get the translated rule below:

$$\vdash_u \begin{bmatrix}\textbf{reads region}\{x.f\}\end{bmatrix}\{\exists z.x.f = z\}x.f := E;\{x.f = E\}\begin{bmatrix}\textbf{modifies region}\{x.f\}\end{bmatrix} \tag{116}$$
$$\textbf{where } x \notin \mathrm{FV}(E)$$

where the fresh effect is empty, thus, it is omitted. Note that $x.f \mapsto \_$ is an abbreviation for $\exists z.x.f \mapsto \_$. Thus $x.f \mapsto \_$ is translated to $\exists z.x.f = z$. Eq. (116) can be derived by using the rules $SubEff_u$ and $CONSEQ_u$. The derivation is shown in Fig. 43.

6. *SEQ:* by the rule SEQ_s and Def. 48, we get the translated rule below:

$$\frac{\begin{array}{c}[\textbf{reads }r_1\!\downarrow] \\ \vdash_u \{\mathrm{TR}[\![a]\!]\ \&\&\ r_1 = fpt_s(a)\}\ S_1\ \{\mathrm{TR}[\![b]\!]\} \\ [\textbf{modifies }r_1\!\downarrow, \mathrm{mods}(S_1), \textbf{fresh }(r_2 - r_1)] \\ [\textbf{reads }r_2\!\downarrow] \\ \vdash_u \{\mathrm{TR}[\![b]\!]\ \&\&\ r_2 = fpt_s(b)\}\ S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_2\!\downarrow, \mathrm{mods}(S_2), \textbf{fresh }(fpt_s(a') - r_2)]\end{array}}{\begin{array}{c}[\textbf{reads }r_1\!\downarrow] \\ \vdash_u \{\mathrm{TR}[\![a]\!]\ \&\&\ r_1 = fpt_s(a)\}\ S_1 S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_1\!\downarrow, \mathrm{mods}(S_1 S_2), \textbf{fresh }(fpt_s(a') - r_1)]\end{array}} \tag{117}$$
$$\textbf{where } r_1 \text{ and } r_2 \text{ are fresh}, r_1 \notin \mathrm{mods}(S_1)\ \textit{and}\ r_2 \notin \mathrm{mods}(S_2)$$

There are two cases:

(a) $S_1 = \textbf{var }x\ T;$: In this case, by the rule VAR_s, we have $b = a * default(T)$, $\mathrm{mods}(\textbf{var }x : T) = \varnothing$ and $r_1 = r_2 = fpt_s(a)$. Then we need to show

$$\frac{\begin{array}{c}\vdash_u [\textbf{reads }r_1\!\downarrow]\{\mathrm{TR}[\![a]\!]\ \&\&\ r_1 = fpt_s(a)\}\ \textbf{var }x : T;\ \{\mathrm{TR}[\![a * default(T)]\!]\}[\textbf{modifies }r_1\!\downarrow] \\ [\textbf{reads }r_1\!\downarrow] \\ \vdash_u \{\mathrm{TR}[\![a * default(T)]\!]\ \&\&\ r_1 = fpt_s(a)\}\ S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_1\!\downarrow, \mathrm{mods}(S_2), \textbf{fresh }(fpt_s(a') - r_1)]\end{array}}{\begin{array}{c}[\textbf{reads }r_1\!\downarrow] \\ \vdash_u \{\mathrm{TR}[\![a]\!]\ \&\&\ r_1 = fpt_s(a)\}\ \textbf{var }x : T; S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_1\!\downarrow, \mathrm{mods}(S_2), \textbf{fresh }(fpt_s(a') - r_1)]\end{array}}$$
$$\textbf{where } r_1 \text{ is fresh } \textit{and}\ r_1 \notin \mathrm{mods}(S_2)$$
$$\tag{118}$$

Using the rule $SubEff_u$ on the second premise, we get

$$\vdash_u \begin{array}{c}[\textbf{reads }r_1\!\downarrow\,] \\ \{\mathrm{TR}[\![a * default(T)]\!]\ \&\&\ r_1 = fpt_s(a)\}\ S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_1\!\downarrow, x, \mathrm{mods}(S_2), \textbf{fresh }(fpt_s(a') - r_1)]\end{array} \tag{119}$$

Using the rule $SEQ2_u$, we can get the conclusion of Eq. (118).

(b) $S_1 \neq \textbf{var }x : T;$: we consider the following cases:
  – S_1 does not allocate new locations, i.e., $r_1 = r_2$. The rule $SEQ1_u$ is instantiated with $RE := \textbf{region}\{\}$, $RE_1 := \textbf{region}\{\}$ and $RE_2 := \textbf{region}\{\}$. If the immunity side conditions are satisfied, then the conclusion of (117) is derived by using the rule $SEQ1_u$. Otherwise, *for all* $x \in \mathrm{mods}(S_1)$ and x in $\mathrm{FV}(b)$, there exists z, such that b implies $x = z$ and $z \notin \mathrm{mods}(S_1)$. We substitute z for x in $fpt_s(b)$. Then the second premise of Eq. (117) is re-written as:

$$\vdash_u \begin{array}{c}[\textbf{reads }r_1\!\downarrow\ [\overline{z}/\mathrm{mods}(S_1)]] \\ \{\mathrm{TR}[\![b]\!]\ \&\&\ r_1 = fpt_s(b)\}\ S_2\ \{\mathrm{TR}[\![a']\!]\} \\ [\textbf{modifies }r_1\!\downarrow\ [\overline{z}/\mathrm{mods}(S_1)], \mathrm{mods}(S_2), \textbf{fresh }(fpt_s(a') - r_1)]\end{array} \tag{120}$$

where $r_1 \downarrow [\overline{z}/\mathrm{mods}(S_1)]$ means that *for all RE $\in r_1 \downarrow :: RE[\mathrm{mods}(S_1)/\overline{z}]$*. From the first premise of Eq. (117) and Eq. (120), the immunity side conditions are satisfied. After using the rule $SEQ1_u$, we get

$$
\begin{aligned}
&[\mathbf{reads}\, r_1\downarrow, r_1\downarrow [\overline{z}/\mathrm{mods}(S_1)]] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r_1 = fpt_s(a)\} S_1 S_2 \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, r_1\downarrow, r_2\downarrow [\mathrm{mods}(S_1)/\overline{z}], \mathrm{mods}(S_1 S_2), \mathbf{fresh}(fpt_s(a') - r_1\downarrow [\mathrm{mods}(S_1)/\overline{z}])]
\end{aligned}
\tag{121}
$$

Because *for all RE $\in r_1 :: RE$ in $r_2\downarrow [\overline{y/x}]$*, Eq. (121) can be simplified to

$$
\begin{aligned}
&[\mathbf{reads}\, r_1\downarrow] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r_1 = fpt_s(a)\} S_1 S_2 \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, r_1\downarrow, \mathrm{mods}(S_1 S_2), \mathbf{fresh}(fpt_s(a') - r_1)]
\end{aligned}
\tag{122}
$$

– S_1 allocates some new locations. Then the second premise of Eq. (117) can be re-written as:

$$
\begin{aligned}
&[\mathbf{reads}\, r_1\downarrow, (r_2 - r_1)] \\
\vdash_u &\{\mathrm{TR}[\![a * default(T)]\!]\, \&\&\, r_2 = fpt_s(a * default(T))\} S_2 \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, r_1\downarrow, (r_2 - r_1), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a') - r_2)]
\end{aligned}
\tag{123}
$$

The rule $SEQ1_u$ is instantiated with $RE := r_2 - r_1$, $RE_1 := r_2 - r_1$ and $RE_2 := r_2 - r_1$. If the immunity side conditions are satisfied, then we union the fresh effects of the two statements and get $fpt_s(a') - r_1$. Hence, the conclusion of Eq. (117) is derived by using the rule $SEQ1_u$. Otherwise, the treatment is similar to the previous case.

7. *IF:* by the rule IF_s and Def. 48, we get the translated rule below:

$$
\begin{aligned}
&[\mathbf{reads}\, fpt_s(a)] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r = fpt_s(a)\, \&\&E \neq 0\} S_1 \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, fpt_s(a), \mathrm{mods}(S_1), \mathbf{fresh}(fpt_s(a') - r)] \\
&\quad [\mathbf{reads}\, fpt_s(a)] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r = fpt_s(a)\, \&\&\, E = 0\} S_2 \{\mathrm{TR}[\![a']\!]\} \\
&\quad [\mathbf{modifies}\, fpt_s(a), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a') - r)] \\
\hline
&[\mathbf{reads}\, fpt_s(a)] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r = fpt_s(a)\}\, \mathbf{if}\, E\, \mathbf{then}\{S_1\}\mathbf{else}\{S_2\}\, \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, fpt_s(a), \mathrm{mods}(S_1), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a') - r)] \\
&\mathbf{where}\, \mathrm{TR}[\![a]\!] \Rightarrow r = fpt_s(a)\, and\, r \notin \mathrm{mods}(S_1) \cup \mathrm{mods}(S_2)
\end{aligned}
\tag{124}
$$

Note that $fpt_s(E \neq 0)$ and $fpt_s(E = 0)$ are both $\mathbf{region}\{\}$, thus are omitted. By the inductive hypothesis, the premise of Eq. (124) is assumed. Then, we use the rule IF_u and get:

$$
\begin{aligned}
&[\mathbf{reads}\, fpt_s(a), \mathbf{reads}\, efs(E)] \\
\vdash_u &\{\mathrm{TR}[\![a]\!]\, \&\&\, r = fpt_s(a)\}\, \mathbf{if}\, E\, \mathbf{then}\{S_1\}\mathbf{else}\{S_2\}\, \{\mathrm{TR}[\![a']\!]\} \\
&[\mathbf{modifies}\, fpt_s(a), \mathrm{mods}(S_1), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a') - r)]
\end{aligned}
\tag{125}
$$

Now we consider use the rule $SubEff_u$. Because

$$rwR(\mathbf{reads}fpt_s(a), \mathbf{reads}efs(E), \mathbf{modifies}fpt_s(a), \mathrm{mods}(S_1), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a')-r)) = fpt_s(a),$$

the following side condition is true:

$$fpt_s(a) \leqslant rwR(\mathbf{reads}\, fpt_s(a), \mathbf{modifies}\, fpt_s(a), \mathrm{mods}(S_1), \mathrm{mods}(S_2), \mathbf{fresh}(fpt_s(a') - r))$$

Therefore, after using the rule $SubEff_u$, we can get the conclusion of Eq. (124).

8. *WHILE:* by the rule *WHILE$_s$* and Def. 48, we get the translated rule below:

$$\frac{\vdash_u [\mathbf{reads}\,fpt_s(I)]\{\mathrm{TR}[\![I]\!] \ \&\& \ E \neq 0\}S\{\mathrm{TR}[\![I]\!]\}[\mathbf{modifies}\,fpt_s(I),\mathrm{mods}(S)]}{\vdash_u [\mathbf{reads}\,fpt_s(I)]\{\mathrm{TR}[\![I]\!]\}\mathbf{while}\,E\,\{S\}\{\mathrm{TR}[\![I]\!] \ \&\& \ E = 0\}[\mathbf{modifies}\,fpt_s(I),\mathrm{mods}(S)]} \quad (126)$$

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 we can directly use the rule *WHILEu* and get

$$\vdash_u [\mathbf{reads}\,fpt_s(I), efs(E)]\{\mathrm{TR}[\![I]\!]\}\mathbf{while}\,E\,\{S\}\{\mathrm{TR}[\![I]\!] \ \&\& \ E = 0\}[\mathbf{modifies}\,fpt_s(I),\mathrm{mods}(S)] \quad (127)$$

Similarly to the case of the rule *IF$_u$*, we can use the rule *SubEff$_u$* and get the conclusion of Eq. (126).

If the immunity side condition is not satisfied, *for all* $x \in \mathrm{mods}(S)$ and $x \in \mathrm{FV}(I)$, there exists z, such that I implies $x = z$ and $z \notin \mathrm{mods}(S)$. We substitute z for x in $fpt_s(I)$. Then the immunity side condition is satisfied. We can use the rules *WHILE$_u$* and *SubEff$_u$* and get the conclusion.

9. *FRM:* by the rule *FRM$_s$* and Def. 48, we get the translated rule below:

$$\frac{\vdash_u \begin{array}{l}[\mathbf{reads}\,r_1\downarrow]\\ \{\mathrm{TR}[\![a]\!] \ \&\& \ r_1 = fpt_s(a_1)\}\,S\,\{\mathrm{TR}[\![a']\!]\}\\ [\mathbf{modifies}\,r_1\downarrow,\mathrm{mods}(S),\mathbf{fresh}\,(fpt_s(a') - r_1)]\end{array}}{\vdash_u \begin{array}{l}[\mathbf{reads}\,r_1 + r_2]\\ \{\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2) \ \&\& \ r_1 = fpt_s(a) \ \&\& \ r_2 = fpt_s(c)\}\\ S\\ \{\mathrm{TR}[\![a']\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (fpt_s(a') \, ! \, ! \, fpt_s(c))\}\\ [\mathbf{modifies}\,fpt_s(a) + r,\mathrm{mods}(S),\mathbf{fresh}\,(fpt_s(a') + r_2 - r')]\end{array}} \quad (128)$$

$$\textbf{where } r_1 \text{ and } r_2 \text{ are fresh}, r_1 \notin \mathrm{mods}(S), r_2 \notin \mathrm{mods}(S),$$
$$\mathrm{TR}[\![a]\!] \&\& \mathrm{TR}[\![c]\!] \&\& (fpt_s(a') \, ! \, ! \, r_2) \Rightarrow r' = r_1 + r_2, \text{ and } \mathrm{mods}(S) \cap \mathrm{FV}(c) = \varnothing$$

By the inductive hypothesis, the premise of Eq. (128) is assumed. The rule *FRM$_u$* is instantiated with $Q := \mathrm{TR}[\![c]\!]$ and $\eta := efs(\mathrm{TR}[\![c]\!])$. We need to prove the side condition, which is:

$$\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2) \ \textit{implies} \ efs(\mathrm{TR}[\![c]\!]) \, \cdot\! / \cdot (\mathbf{modifies}\,\mathrm{mods}(S), fpt_s(a)) \quad (129)$$

By Lemma 44 and by the definition of separator (Fig. 20), Eq. (129) is true. After using the rule *FRM$_u$*, we obtain:

$$\vdash_u \begin{array}{l}[\mathbf{reads}\,r_1\downarrow]\\ \{\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2)\}\,S\,\{\mathrm{TR}[\![a']\!] \ \&\& \ \mathrm{TR}[\![c]\!]\}\\ [\mathbf{modifies}\,r_1\downarrow,\mathrm{mods}(S),\mathbf{fresh}\,(fpt_s(a') - r_1)]\end{array} \quad (130)$$

Now we consider to use the rule *FRM$_u$* again. It is instantiated with $Q := r_1 \, ! \, ! \, r_2$ and $\eta := \mathbf{reads}\,r_1, \mathbf{reads}\,r_2$. We need to prove the side condition, which is:

$$(\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2)) \ \textit{implies} \ (\mathbf{reads}\,r_1, \mathbf{reads}\,r_2) \, \cdot\! / \cdot (\mathbf{modifies}\,r_1\downarrow,\mathrm{mods}(S)) \quad (131)$$

By $r_1 \notin \mathrm{mods}(S)$ and $r_2 \notin \mathrm{mods}(S)$, Eq. (131) is true. Note that $\mathbf{modifies}\,r_1\downarrow$ 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$_u$*, we obtain:

$$\vdash_u \begin{array}{l}[\mathbf{reads}\,r_1\downarrow]\\ \{\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2) \ \&\& \ r_1 = fpt_s(a) \ \&\& \ r_2 = fpt_s(c)\}\\ S\\ \{\mathrm{TR}[\![a']\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2)\}\\ [\mathbf{modifies}\,r_1\downarrow,\mathrm{mods}(S),\mathbf{fresh}\,(fpt_s(a') - r_1)]\end{array} \quad (132)$$

Because $\mathrm{TR}[\![c]\!]$ is preserved by S, $r_2 = fpt_s(c)$ in the poststate. Thus, after using the rule *CONSEQ$_u$*, we get

$$\vdash_u \begin{array}{l}[\mathbf{reads}\,r_1\downarrow]\\ \{\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! \, r_2)\,r_1 = fpt_s(a) \ \&\& \ r_2 = fpt_s(c)\}\\ S\,\{\mathrm{TR}[\![a']\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1 \, ! \, ! fpt_s(c))\}\\ [\mathbf{modifies}\,r_1\downarrow,\mathrm{mods}(S),\mathbf{fresh}\,(fpt_s(a') - r_1)]\end{array} \quad (133)$$

Now we need to prove that $fpt_s(a')\,!\,!fpt_s(c)$ in the poststate. By the definition of SSL Hoare-formula, we know that $fpt_s(a') = r_1 + RE$, where RE are possibly empty regions that do not exist in the pre-state, hence $RE\,!\,!fpt_s(c)$. Hence, $fpt_s(a')\,!\,!fpt_s(c)$ is true. Then, after using the rule $CONSEQ_u$, we get

$$\vdash_u \begin{array}{l} [\mathbf{reads}\ r_1 \downarrow] \\ \{\mathrm{TR}[\![a]\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (r_1\,!\,!r_2)\ r_1 = fpt_s(a) \ \&\& \ r_2 = fpt_s(c)\} \\ S \\ \{\mathrm{TR}[\![a']\!] \ \&\& \ \mathrm{TR}[\![c]\!] \ \&\& \ (fpt_s(a')\,!\,!fpt_s(c))\} \\ [\mathbf{modifies}\ r_1 \downarrow, \mathrm{mods}(S), \mathbf{fresh}\ (fpt_s(a') - r_1)] \end{array} \tag{134}$$

Now we consider the fresh effects. By the side condition $r' = r_1 + r_2$, we have

$$fpt_s(a') - r_1 = fpt_s(a') + r_2 - r_1 - r_2 = fpt_s(a') + r_2 - r'.$$

Finally, we can use the rule $SubEff_u$ to loosen the read effects, and get the conclusion of Eq. (128).

<div align="right">□</div>

$$(ALLOC) \ \vdash_u \ [\varnothing]\{true\} \ x := \textbf{new} \ C; \{new_u(C,x)\}[\textbf{modifies} \ x, \textbf{alloc}, \textbf{fresh}(\textbf{region}\{x.*\})]$$

(FRM) $\dfrac{true \vdash_u efs(\text{TR}[\![a]\!]) \, frm \, \text{TR}[\![a]\!] \qquad \textbf{where} \ \text{TR}[\![a]\!] \Rightarrow efs(\text{TR}[\![a]\!]) \cdot / \cdot (x, \textbf{alloc}) \ \text{and} \ \text{FV}(a) \cap \{x\} = \varnothing \Rightarrow \text{FV}(\text{TR}[\![a]\!]) \cap \{x\} = \varnothing (\text{Lemma 45})}{\vdash_u [\varnothing]\{\text{TR}[\![a]\!]\} \ x := \textbf{new} \ C; \{\text{TR}[\![a]\!] \ \&\& \ new_u(C,x)\}[\textbf{modifies} \ x, \textbf{alloc}, \textbf{fresh} \ \textbf{region}\{x.*\}] \qquad (Subeffect) \vdash \varnothing \leqslant \textbf{reads} \ fpt_s(a)}$

$(SubEff)$ $\dfrac{}{\vdash_u [\textbf{reads} \, fpt_s(a)]\{\text{TR}[\![a]\!]\} \ x := \textbf{new} \ C; \{\text{TR}[\![a]\!] \ \&\& \ new_u(C,x)\}[\textbf{modifies} \ x, \textbf{alloc}, \textbf{fresh} \ \textbf{region}\{x.*\}]}$

$(SubEff)$ $\dfrac{(Subeffect) \ \text{TR}[\![a]\!] \vdash (\textbf{modifies} \ x, \textbf{alloc}) \leqslant (\textbf{modifies} \ x, \textbf{alloc}, fpt_s(a))}{\vdash_u [\textbf{reads} \, fpt_s(a)]\{\text{TR}[\![a]\!]\} \ x := \textbf{new} \ C; \{\text{TR}[\![a]\!] \ \&\& \ new_u(C,x)\}[\textbf{modifies} \ x, \textbf{alloc}, fpt_s(a), \textbf{fresh} \ \textbf{region}\{x.*\}]}$

$(SubEff)$ $\dfrac{\textbf{where} \ \text{TR}[\![a]\!] \Rightarrow r = \textbf{alloc} \ and \ fpt_s(a) \leqslant r}{}$

$$[\textbf{reads} \, fpt_s(a)]$$
$(FrToPost) \ \vdash_u \ \{\text{TR}[\![a]\!]\}x := \textbf{new} \ C; \{\text{TR}[\![a]\!] \ \&\& \ new_u(C,x) \ \&\& \ (fpt_s(a) \, !! \, \textbf{region}\{x.*\})\}$
$$[\textbf{modifies} \ x, \textbf{alloc}, fpt_s(a), \textbf{fresh}(\textbf{region}\{x.*\})]$$

Fig. 41: The derivation of rule $\text{TR}_R[\![ALLOC_s]\!]$. The subscript, u, is omitted in each rule's name. The program semantics assumes that the location for each field in a class is disjoint with each other, thus $new_u(C,x)$ iff $new_s(C,x)$.

$$(ACC) \;\; \vdash_u [efs(x'.f)]\{x' \neq null\} \; x := x'.f; \{x = x'.f\}[\textbf{ modifies } x]$$

$$(FRM) \;\; \frac{(x'.f = z) \vdash_u (\textbf{region}\{x'.f\}, x', z) \, frm \, (x'.f = z) \qquad \textbf{where } x'.f = z \wedge x \neq y \wedge x' \neq y \;\Rightarrow\; ((\textbf{region}\{x'.f\}, x', y)/x) \text{ and } x' \neq null \Rightarrow \exists z.(x'.f = z)}{\vdash_u [efs(x'f)]\{x'.f = z\} \; x := x'.f; \{x = z \;\&\&\; x'.f = z\}[\textbf{ modifies } x]}$$

$$(SubEff) \;\; \frac{(Subeffect) \vdash \textbf{modifies } x \leqslant \textbf{modifies } x, \textbf{region}\{x'.f\}}{\vdash_u [efs(x'.f)]\{x'.f = z\} \; x := x'.f; \{x = z \;\&\&\; x'.f = z\}[\textbf{ modifies } x, \textbf{region}\{x'.f\}]}$$

$$(SubEff) \;\; \frac{\textbf{where } x' \neq null \Rightarrow rwR(efs(x'.f), x, \textbf{region}\{x'.f\}) \leqslant rwR(\textbf{region}\{x'.f\}, x)}{\vdash_u [\textbf{reads region}\{x'.f\}]\{x'.f = z\} \; x := x'.f; \{x = z \;\&\&\; x'.f = z\}[\textbf{ modifies } x, \textbf{region}\{x'.f\}]}$$

Fig. 42: The derivation of rule $\text{TR}_R[\![ACC_s]\!]$. The subscript, u, is omitted in each rule's name.

$$(CONSEQ) \;\; \frac{(UPD) \;\; \vdash_u [\textbf{reads } x, efs(E)]\{x \neq null\} \; x.f := E; \{x.f = E\}[\textbf{region}\{x.f\}] \textbf{ where } x \neq null \Rightarrow \exists \, z.x.f = z}{\vdash_u [\textbf{reads } x, efs(E)]\{\exists \, z.x.f = z\} \; x.f := E; \{x.f = E\}[\textbf{ modifies region}\{x.f\}]}$$

$$(SubEff) \;\; \frac{(Subeffect) \; readR(\textbf{reads } x, efs(E)) \leqslant readR(\textbf{reads region}\{x.f\}) \vdash_u \textbf{reads } readR(\textbf{reads } x, efs(E)) \leqslant \textbf{reads region}\{x.f\}}{\vdash_u [\textbf{reads region}\{x.f\}]\{\exists \, z.x.f = z\} \; x.f := E; \{x.f = E\}[\textbf{ modifies region}\{x.f\}]}$$

Fig. 43: The derivation of rule $\text{TR}_R[\![UPD_s]\!]$. The subscript, u, is omitted in each rule's name.

F   Proof of Lemma 53

Lemma 53: Let (σ, h) be a state, and p_s be an inductive predicate in SSL. Then

$$\mathcal{E}_a[\![p_s(\overline{e})]\!](\sigma, h) = \mathcal{E}_p[\![\text{TR}[\![p_s(\overline{e})]\!]]\!](\sigma, h).$$

Proof. The proof is an inductive case of the proof of Theorem 40. The inductive hypothesis is that for all subassertions a_i, $\mathcal{E}_a[\![a_i]\!](\sigma, h) = \mathcal{E}_p[\![\text{TR}[\![a_i]\!]]\!](\sigma, h)$. Let $b_1 \Rightarrow a_1 \cdots b_n \Rightarrow a_n$ be inductive cases for p_s. We prove it as follows.

$\quad \mathcal{E}_a[\![p_s(\overline{e})]\!](\sigma, h)$

iff $\langle$by semantics of inductive predicates Eq. (65)$\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, \mathcal{E}_a[\![(b_1 \Rightarrow a_1) \wedge ... \wedge (b_n \Rightarrow a_n)]\!](\sigma', h'))(\sigma(formals(p_s) \mapsto \overline{\mathcal{E}_s[\![e]\!](\sigma)}), h)$

iff $\langle$by semantics of assertions Def. 25$\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_a[\![b_1 \Rightarrow a_1]\!](\sigma', h') \text{ and } ... \text{ and } \mathcal{E}_a[\![b_n \Rightarrow a_n]\!](\sigma', h')))(\sigma(formals(p_s) \mapsto \overline{\mathcal{E}_s[\![e]\!](\sigma)}), h)$

iff $\langle$by inductive hypothesis$\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\text{TR}[\![b_1 \Rightarrow a_1]\!]]\!](\sigma', h') \text{ and } ... \text{ and } \mathcal{E}_p[\![\text{TR}[\![b_n \Rightarrow a_n]\!]]\!](\sigma', h')))$
$\quad (\sigma(formals(p_s) \mapsto \overline{\mathcal{E}_s[\![e]\!](\sigma)}), h)$

iff $\langle$by Lemma 35: $\mathcal{E}_s[\![e]\!](\sigma) = \mathcal{E}[\![\text{TR}[\![e]\!]]\!](\sigma)\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\text{TR}[\![b_1 \Rightarrow a_1]\!]]\!](\sigma', h') \text{ and } ... \text{ and } \mathcal{E}_p[\![\text{TR}[\![b_n \Rightarrow a_n]\!]]\!](\sigma', h')))$
$\quad (\sigma(formals(p_s) \mapsto \overline{\mathcal{E}[\![\text{TR}[\![e]\!]]\!](\sigma)}), h)$

iff $\langle$by the semantics of UFRL assertions (Def. 14)$\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\text{TR}[\![b_1 \Rightarrow a_1]\!] \&\& ... \&\& \text{TR}[\![b_n \Rightarrow a_n]\!]]\!](\sigma', h')))(\sigma(formals(p_s) \mapsto \overline{\mathcal{E}[\![\text{TR}[\![e]\!]]\!](\sigma)}), h)$

iff $\langle$by the definition of encoding inductive predicates in Fig. 27$\rangle$

$\quad (fix \underline{\lambda}(\sigma', h') \,.\, (\mathcal{E}_p[\![\text{TR}[\![body(p_s)]\!]]\!](\sigma', h')))(\sigma(formals(p_s) \mapsto \overline{\mathcal{E}[\![\text{TR}[\![e]\!]]\!](\sigma)}), h)$

iff $\langle$by semantics of recursive predicate.$\rangle$

$\quad \mathcal{E}_p[\![\text{TR}[\![p_s(\overline{e})]\!]]\!](\sigma, h)$

$\square$