Fine-Grained Region Logic

Yuyan Bao¹, Gary T. Leavens¹, and Gidon Ernst²

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

Abstract. The report is about a fine-grained region logic.

Keywords: Separation logic, region logic, fine-grained region logic, framing, formal specification, formal verification, Hoare logic

1 Introduction

Fine-Grained region logic (FRL) is an adaptation of region logic (RL) [2, 3] for the purpose of encoding separation logic [7, 11]. It supports the concept of region expressions that are syntactic denotations of sets of locations on the heap. A new primitive data type region is introduced to represent sets of locations. Variables declared with type region are specification-only, but not mentioned in branch statements or in statements that mutate program state. Region expressions are commonly used to store the locations of data structures, and specify frame conditions of procedures. FRL uses region expressions to denote footprints that are locations of the values reside in. Heap structures are denoted by region operations, e.g. region unions, and region assertions, e.g. sub-regions and disjoint regions.

RL and FRL have two major differences: (1) In region logic, regions, G, are sets of references, possibly containing null [3]. For example, \{x\} is a region containing a singleton object x. Image expressions \(x^f\) denote the location if f has the type rgn or Object, otherwise, \(x^f\) is not well-typed, neither region operations, such as \(x^f \cup \{y\}^f\), where \(\cup\) denotes region union in region logic, and \(f_x \) and \(f_y\) are not type rgn or Object. It is not convenient to express footprints of separation logic assertions, e.g. \(x.f_x \rightarrow 5 \# y.f_y \rightarrow 6\). Our work uniformly uses sets of locations, which one can think of as sets of pairs of object references and field names; for this reason we call our region logic a “fine-grained” region logic. Region expression, \(\text{region}\{x.f\}\) is well-typed, where f can be any primitive types, such as integers and Booleans. This region expression can be used to directly denote footprints of the points-to assertion in separation logic. And the footprints of the example above can be denoted by \(\text{region}\{x.f_x\} + \text{region}\{y.f_y\}\), where + denotes region union in FRL. Using sets of locations is also a good match for specification languages such as JML [6], in which frames are specified in terms of sets of locations. (2) The work on region logic [2, 3] does not consider conditional region expressions. FRL enhances expressiveness by allowing conditional region expressions that can be used to reason about methods with conditional effects [4], and to denote footprint of a restricted form of implications in separation logic. The proof system of FRL and RL are no fundamental difference, except the rules of introducing and eliminating conditional effects.
2 Problems and FRL Overview

In a formal specification language, frame properties describe what is allowed to change and thus what does not change when an imperative procedure is executed [5]. There are two common ways of expressing frame properties. One is a clause in a procedure’s specification (such as modifies, assignable and wr) to specify write effects [2, 3] by listing a set of abstract locations that may be changed in a method. Another is using post conditions, such as $a = \text{old}(a)$, which says $a$’s value is the same as the value before the method call. Dynamic frames [8, 9] techniques, such as Dafny [10] and region logic [2, 3], use unconditional effects that usually over-approximate write effects. That means locations where the values are preserved in some conditions also appear in frame conditions. Consequently, one has to specify which parts of the specific objects are not changed in post-conditions.

We explain the problem of unconditional effects by a toy example written in fine-grained region logic (FRL). It is an adaption of Dafny’s solution [10] to the problem 2 in the VSI benchmark [13]. The purpose of this example is to give an overview of the FRL language and to illustrate the issues of unconditional effects. The detailed implementations of the functions and methods are omitted. And we also only show some excerpts of the specifications that are related to the issues that we are discussing in this paper.

The FRL\(^3\) programming language is a sequential programming language. A FRL program consists of a set of classes with fields and methods. The example in Fig. 1 shows a generic map implemented by an acyclic list. The keywords requires and ensures are used to specify method preconditions and postconditions, and the keywords reads and modifies are used to specify read effects [2, 3] and write effects. Read effects describe a set of locations that expressions depend on. Method Init is a map’s constructor; it can be used on the right-hand side of an assignment statement. The use of fresh in its postcondition allows one to add a newly allocated Node to fpt. The ghost variable fpt, declared as type region, is used to store abstract locations of a map, which is a map’s dynamic frame. fpt needs to be explicitly updated when the map changes, as in method Add. Such statements involving regions cause ghost state changes, which must not affect the programs’ control flow. For example, regions cannot appear in the tests of branching statements. Function\(^4\) Valid describes the structure of a map by logic formulas. In this example, Valid is considered as an invariant, which must be kept valid after a map is initialized, and before and after each method call. The detailed implementation of Valid is omitted. The method Add updates the value of the key, $k$, with a new value, $v$, if it is already in the map, otherwise, a new key-value pair, $(k, v)$, is prepended to the map.

The frame condition, modifies fpt, means that all the locations in the map may be modified by the method Add. It over-approximates Add’s real write effect, since if $k$ is held by a key-value pair, $p$, in the map, only the location region{p.val} will be changed; otherwise only the location region{this.fpt} will be changed. So one has to specify that the value of the extra locations are preserved. For example, the

---

\(^3\) FRL language is a blend of Dafny [10] and VERL [12].

\(^4\) Functions in FRL are boolean-valued methods that have no write effects.
specification in line 22 means all the values of the key-value pairs, whose key is not \( k \), are preserved. The expression \( \text{old}(\text{fpt}) \) denotes the values in the pre-state. That is the value evaluated before \text{Add} executes. We call such a postcondition a \textit{make-up frame condition}. The condition \( \text{o.key} = \text{old}(\text{o.key}) \) in line 21 is also a make-up frame condition. If we allow one to specify frame conditions more precisely, then this make-up frame conditions can be avoided, such as the following frame condition and postcondition:

\[
\text{modifies if } \text{Find(key)} = \text{null then region\{this.fpt\}} \\
\hspace{1em} \text{else region\{Find(key).val\}}; \\
\text{ensures Valid() && fresh(fpt - \text{old}(\text{fpt}))}; \\
\text{ensures } \exists \text{ region\{o.*\} } \subseteq \text{fpt.(o.key = key && o.val = val);} \\
\]

The frame condition uses the pure method \text{Find} that returns the reference that holds \( k \) if it is in the map, otherwise returns null reference. The technique of using pure methods in region logic is discussed in Banerjee and Naumann’s work [1]. The frame condition says if there is an object, \( p \), that holds the key, \( k \), then the location \text{region\{p.val\}} can be modified, otherwise the location \text{region\{this.fpt\}} can be modified. Because the frame condition is much more precise than the one in Fig. 1, the make-up frame conditions are avoided in the post-condition. The conditional effect is caused by conditional statements in the program where mutually exclusive conditions split the execution traces of programs. Our CONMASK1 and CONMASK2 rules discussed in section 5 can drop the condition from such conditional effects. Therefore conditional effects do not necessarily increase the complexity of the compositions of effects.

3 Programming Language

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

3.1 Syntax

Fig. 2 shows the syntax of the FRL language. A program consists of a statement \( S \) in the context of some class declarations. A class consists of fields and methods. A field is declared with type integer, boolean, a user-defined datatype, or \textit{region}. A method is declared in a class. In a method implementation, there are local variable declarations, update statements, condition statements, and loop statements.

\( E \) are pure expressions, \( HE \) are expressions that may depend on heaps. The empty region is written \textit{region\{\}}. For each expression \( HE \) that evaluates to an non-null object, a region expression of the form \textit{region\{HE.f\}} denotes a singleton set that stores the location of field \( f \) in the object that is the value of \( HE \). The form \textit{region\{HE.*\}} denotes a set containing the abstract locations represented by the reference \( HE \) and all its fields\(^5\). The conditional region expression, \textit{if HE then RE_1 else RE_2}, is stateful. It denotes that if \( HE \neq 0 \), then the region is \( RE_1 \), otherwise the region is \( RE_2 \). A

\(^5\) Since FRL does not have subclassing or subtyping, the fields in \textit{region\{HE.*\}} are based on the static type of the reference denoted by \( HE \), which will also be its dynamic type.
class Node<Key, Value> {
    var key: Key; var val: Value;
    var next: Node<Key, Value>;
}

class Map {
    var head: Node<Key, Value>;
    ghost var fpt: region; // footprint of this object

    function Valid(): bool {
        ...
    }

    constructor Init() modifies region{this.*} {
        fpt := region{this.*}; head := null;
    }

    method Add(k: Key, v: Value) requires Valid(); modifies fpt; ensures Valid() && fresh(fpt - old(fpt));
    ensures \exists region{o.*} \subseteq fpt.
    (old(o.key) = k => o.val = v && o.key = old(o.key));
    ensures \forall region{o.*} \subseteq old(fpt).
    (old(o.key) != k => o.key = old(o.key) && o.val = old(o.val));
    {
        var p = Find(k);
        if (p = null) {
            var h := new Node<Key, Value>;
            h.key := k; h.val := v; h.next := head;
            fpt := fpt + region{h.*};
        } else {
            p.val := val;
        }
    }

    pure method Find(k: Key) returns (p: Node<Key, Value>) requires Valid(); reads fpt; ensures p = null => \forall region{o.*} \subseteq fpt.(o.key!=k);
    ensures p != null => region{p.*} \subseteq fpt && p.key = k;
    {
        ...
    }
}

Fig. 1: The toy example of Map
region expression of the form \( \text{filter}\{RE, C, f\} \) denotes one of the following: if \( f \)'s type is region, it is the union of all the regions \( o.f \), where \((o, f)\) is in \( RE \) and \( o \) has the type \( C \); otherwise, it is the set of locations of form \((o, f)\), where each object reference, \( o \), has the type \( C \). A region expression of the form \( \text{filter}\{RE, C\} \) denotes the subset of \( RE \) with references of type \( C \). For example, let \( RE = \{o_1.f_1, o_1.f_2, o_2.f\} \), where only \( o_1 \) has type \( C \), then \( \text{filter}\{RE, C\} = \{o_1.f_1, o_1.f_2\} \). The operators \( +, - \), and \( * \) denote union, difference and intersection respectively. The statement for garbage collection or deallocation are excluded in our statements. We use \( \Gamma \) for type environments,

\[
\text{Class} ::= \text{class } C \{ \text{Member} \} \\
\text{Member} ::= \text{Field} | \text{Method} \\
\text{Field} ::= \text{var } f : T \\
\text{Method} ::= \text{method } m(x : T) \text{ returns } (x' : T) \{ S \} \\
\text{T} ::= \text{int} | \text{bool} | C | \text{region} \\
\text{E} ::= n | x | \text{null} \\
\text{HE} ::= E | HE.f | HE_1 \odot HE_2 \\
\text{RE} ::= x | \text{region}{ } | \text{region}(HE.*) | \text{region}(HE,f) \\
| \text{if } HE \text{ then } RE_1 \text{ else } RE_2 | \text{filter}(RE, C,f) | \text{filter}(RE, C) \\
| RE_1 \odot RE_2 \\
\text{F} ::= HE | RE \\
\text{S} ::= \text{var } x : T_1 ; | x := F_1 ; | x_1 := x_2 . f_1 ; | x.f := F_1 ; | x := \text{new } C ; \\
| \text{if } E \text{ then } \{ S_1 \} \text{ else } \{ S_2 \} ; | \text{while } E \{ S \} ; | S_1 ; S_2 \\
\od ::= = | + | - | * | \leq \ldots \\
\od ::= + | - | * \\
\]

Fig. 2: The syntax of FRL language.

which map variables to types: \( \Gamma \in \text{TypeEnv} = var \rightarrow T \). The typing rules for expressions is defined in Fig. 3, for region expressions are defined in Fig. 4, and for statements are defined in Fig. 5.

\[
\begin{align*}
\Gamma \vdash x : T \text{ where } (\Gamma x) = T & \quad \Gamma \vdash \text{null} : C \text{ where } \text{isClass}(C) & \quad \Gamma \vdash n : \text{int} \\
\Gamma \vdash HE : C \text{ where } \text{isClass}(C) & \quad (f : T) \in \text{fields}(C) \\
\Gamma \vdash HE.f : T \\
\Gamma \vdash HE_1 : T_1 & \quad \Gamma \vdash HE_2 : T_2 & \quad \Gamma \vdash \od : T_1 \rightarrow T_2 \rightarrow T \\
\Gamma \vdash HE_1 \od HE_2 : T \\
\end{align*}
\]

Fig. 3: Typing rules for expressions. The predicate \( \text{isClass} \) returns true just when \( C \) is a declared class name in the program.
3.2 Semantics

Fig. 6 shows the semantics of expressions and region expressions. The set \( \text{Loc} \) 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 \( \sigma \), 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: \( \text{Value} = \text{Boolean} + \text{Object} + \text{Int} + \text{PowerSet}(\text{Loc}) \). Pure expressions evaluate to Values. Heap dependent expressions may fault with value, \( \text{err} \). Region expressions evaluate to regions, i.e., sets of locations.

A \( \Gamma \)-state contains a store and a heap: \( \Gamma \text{-State}=\text{Store} \times \text{Heap} \), where \( \text{dom}(\sigma) = \text{dom}(\Gamma) \), 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. Also fieldNames is a function that takes a class table and a type and returns a list of the names of the declared fields of the type. The semantics of statements is standard. The disjointness of two regions can be represented by \( \text{RE}_1 \cap \text{RE}_2 = \text{region} \). We use \( \text{RE}_1 \cup \text{RE}_2 \) as a syntactic sugar for this boolean expression.

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. The semantics of statements are defined in Fig. 7.
4 Assertion Language

In this section, FRL’s assertion language is formalized.

4.1 Syntax and semantics of assertions

The syntax of assertions is shown in Fig. 8. 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. 9. The semantics of assertions is shown in Fig. 10. RE₁ ⊆ RE₂ means RE₁ is subregion of RE₂. The semantics of assertions identifies errors (err) with false, and is two-valued. For example, RE₁ ⊆ RE₂ is false if either RE₁ or RE₂ is err.

4.2 EFFECTS

Effects (ε, δ) are used in frame conditions. The keyword modifies specifies write effects and reads specifies read effects. 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 ε₁ else ε₂; it denotes that if E ≠ 0, the effect is ε₁, otherwise the effect is ε₂.

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
\[ E : E \rightarrow Type \rightarrow State \rightarrow Value + \{err\} \]
\[ \mathcal{E}_\cdot \mid \Gamma \vdash \cdot : T \models ((\sigma, H)) = \sigma (x) \]
\[ \mathcal{E}_\cdot \mid \Gamma \vdash \cdot \mid null : C \models ((\sigma, H)) = null \]
\[ \mathcal{E}_\cdot \mid \Gamma \vdash \cdot \mid int : \text{int} \models ((\sigma, H)) = N \models \text{int} \]
\[ \mathcal{E}_\cdot \mid HE, f \models ((\sigma, H)) = \]
\[ \text{let } o = E_\cdot [HE]((\sigma, H)) \text{ in} \]
\[ \text{if } o \neq \text{err} \text{ and } o \neq \text{null} \text{ then } H(o, f) \text{ else err} \]
\[ \mathcal{E}_\cdot \mid HE_1 \odot HE_2 \models ((\sigma, H)) = \]
\[ \text{let } v_1 = E_\cdot [HE_1]((\sigma, H)) \text{ in} \]
\[ \text{let } v_2 = E_\cdot [HE_2]((\sigma, H)) \text{ in} \]
\[ \text{if } v_1 \neq \text{err} \text{ and } v_2 \neq \text{err} \text{ then } v_1 \text{ MO } \odot \text{ v2 else err} \]

\[ \mathcal{R} : RE \rightarrow Type \rightarrow State \rightarrow Value + \{err\} \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid region() : region((\sigma, H)) = \emptyset \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid region(HE, f) \mid \sigma, H = \]
\[ \text{let } o = E_\cdot [HE]((\sigma, H)) \text{ in} \]
\[ \text{if } o \neq \text{err} \text{ and } o \neq \text{null} \text{ then} \]
\[ \{(o, f)\} \text{ else err} \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid region(HE, C, f) : region((\sigma, H)) = \]
\[ \text{let } v = E_\cdot [HE]((\sigma, H)) \text{ in} \]
\[ \text{cases } v \text{ of} \]
\[ \text{true } \rightarrow \mathcal{R}_\cdot \mid RE_1 \mid \sigma, H \]
\[ \text{false } \rightarrow \mathcal{R}_\cdot \mid RE_2 \mid \sigma, H \]
\[ \text{else err} \]
\[ \text{end} \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid filter(RE, C, f) : region((\sigma, H)) = \]
\[ \text{let } v = \mathcal{R}_\cdot \mid \Gamma \vdash RE : region((\sigma, H)) \text{ in} \]
\[ \text{if } v \neq \text{err} \text{ then} \]
\[ \{(o, f)\} \text{ if } v \land Type(o, \sigma) = C \}
\[ \text{else err} \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid filter(RE, C, f) : region((\sigma, H)) = \]
\[ \text{let } v = \mathcal{R}_\cdot \mid \Gamma \vdash RE : region((\sigma, H)) \text{ in} \]
\[ \text{if } v \neq \text{err} \text{ then} \]
\[ \text{if } f : \text{region} \in \text{fieldNames}(CT, C) \text{ then} \]
\[ \bigcup \{H(o, f)\} \text{ if } v \land Type(o, \sigma) = C \}
\[ \text{else} \{(o, f')\} \text{ if } v \land f' = f \land Type(o, \sigma) = C \}
\[ \text{else err} \]
\[ \mathcal{R}_\cdot \mid \Gamma \vdash \cdot \mid RE_1 \odot RE_2 : region((\sigma, H)) = \]
\[ \text{let } v_1 = \mathcal{R}_\cdot \mid \Gamma \vdash RE : region((\sigma, H)) \text{ in} \]
\[ \text{let } v_2 = \mathcal{R}_\cdot \mid \Gamma \vdash RE : region((\sigma, H)) \text{ in} \]
\[ \text{if } v_1 \neq \text{err} \land v_2 \neq \text{err} \text{ then } v_1 \text{ MO } \odot \text{ v2 else err} \]

Fig. 6: Semantics of properly typed expressions. \( N \) is the standard meaning function for numeric literals. The function MO gives the semantics of operators.
\[MS : \text{TypingJudgment} \rightarrow \text{ClassTable} \rightarrow \text{State} \rightarrow \text{State}\_\bot\]

\[MS[\Gamma \vdash (\text{var} \: x : T;:) : ok(\Gamma)](CT)(\sigma, H) =\]
\[\langle \sigma[x \mapsto \text{default}(T)], H \rangle\]

\[MS[\Gamma \vdash (x := F;) : ok(\Gamma)](CT)(\sigma, H) =\]
\[let \: T = \Gamma(x) \text{ in}\]
\[let \: val = E_r[\Gamma \vdash F : T][(\sigma, H)] \text{ in}\]
\[if \: val \neq \text{err} \text{ then } \langle \sigma[x \mapsto \text{val}], H \rangle \text{ else err}\]

\[MS[\Gamma \vdash (x.f := F;) : ok(\Gamma)](CT)(\sigma, H) =\]
\[let \: v_1 = E_r[\Gamma \vdash x.f : T][(\sigma, H)] \text{ in}\]
\[let \: v_2 = E_r[\Gamma \vdash F : T][(\sigma, H)] \text{ in}\]
\[if \: v_1 \neq \text{err} \wedge v_2 \neq \text{err} \text{ then } \langle \sigma, H[v_1 \mapsto v_2] \rangle \text{ else err}\]

\[MS[\Gamma \vdash (x := \text{new} \: C;) : ok(\Gamma)](CT)(\sigma, H) =\]
\[let \: (l, H') = \text{allocate}(C, H) \text{ in}\]
\[let \: (f_1, \ldots, f_n) = \text{fieldNames}(C) \text{ in}\]
\[let \: \sigma' = \sigma[x \mapsto l] \text{ in}\]
\[\langle \sigma', H'[(\sigma'(x), f_1) \mapsto \text{default}(T_1), \ldots, (\sigma'(x), f_n) \mapsto \text{default}(T_n)] \rangle\]

\[MS[\Gamma \vdash (\text{if} \: E \: \text{then} \{S_1\} \text{else} \{S_2\};) : ok(\Gamma)](CT)(\sigma, H) =\]
\[let \: v = E_r[\Gamma \vdash E : T][(\sigma, H)] \text{ in}\]
\[\text{cases } v \text{ of}\]
\[true \rightarrow MS[\Gamma \vdash S_1 : ok(\Gamma)](CT)(\sigma, H)\]
\[false \rightarrow MS[\Gamma \vdash S_2 : ok(\Gamma)](CT)(\sigma, H)\]
\[\text{else err}\]
\[\text{end}\]

\[MS[\Gamma \vdash (\text{while} \: E \: \{S\};) : ok(\Gamma)](CT)(\sigma, H) =\]
\[\text{fix}(\lambda g. \: MS).\]
\[let \: v = E_r[\Gamma \vdash E : \text{bool}][(\sigma, H)] \text{ in}\]
\[\text{cases } v \text{ of}\]
\[true \rightarrow \text{let } \: s' = MS[\Gamma \vdash S : ok(\Gamma)](CT)(s) \text{ in } g \circ s'\]
\[false \rightarrow s\]
\[\text{else err}\]
\[\text{end}(\sigma, H)\]

\[MS[\Gamma \vdash (S_1, S_2) : ok(\Gamma)](CT)(\sigma, H) =\]
\[let \: s' = MS[\Gamma \vdash S_1 : ok(\Gamma)](CT)(\sigma, H) \text{ in}\]
\[if \: s' \neq \bot \text{ then } MS[\Gamma \vdash S_2 : ok(\Gamma)](CT)(s') \text{ else } \bot\]

Fig. 7: The semantics of statements. The allocate function takes the heap and the class name as parameters, and returns a location and a new heap.
\[ P ::= E_1 = E_2 \mid x.f = E \mid \text{RE}_1 \leq \text{RE}_2 \mid P_1 \& P_2 \mid P_1 \mid P_2 \mid \neg P \]
\[
\forall \ x : \text{int}.P \mid \forall \ x, f.(\text{region}[x.f] \leq \text{RE} \& \& P) \\
\exists x : \text{int}.P \mid \exists x, f.(\text{region}[x.f] \leq \text{RE} \& \& P)
\]

Fig. 8: The syntax of assertions

\[
\begin{align*}
\Gamma \vdash E_1 : T, & \quad \Gamma \vdash E_2 : T \\
\Gamma \vdash E_1 = E_2 : \text{bool} \\
\Gamma \vdash x.f : T, & \quad \Gamma \vdash E : T \\
\Gamma \vdash x.f = E : \text{bool} \\
\Gamma \vdash \text{RE}_1 : \text{region}, & \quad \Gamma \vdash \text{RE}_2 : \text{region} \\
\Gamma \vdash \text{RE}_1 \leq \text{RE}_2 : \text{bool} \\
\Gamma \vdash P_1 : \text{bool}, & \quad \Gamma \vdash P_2 : \text{bool} \\
\Gamma \vdash P_1 \& P_2 : \text{bool} \\
\Gamma \vdash P_1 \mid | P_2 : \text{bool} \\
\Gamma \vdash \neg P : \text{bool} \\
\Gamma \vdash \text{RE} : \text{region}, & \quad \Gamma, \text{region}[x.f] : \text{region} \vdash P : \text{bool} \\
\Gamma \vdash \forall x, f.(\text{region}[x.f] \leq \text{RE} \& \& P) : \text{bool} \\
\Gamma, x : \text{int} \vdash P : \text{bool} \\
\Gamma \vdash \exists x : \text{int}.P : \text{bool} \\
\Gamma \vdash \text{RE} : \text{region}, & \quad \Gamma, \text{region}[x.f] : \text{region} \vdash P : \text{bool} \\
\Gamma \vdash \exists x, f.(\text{region}[x.f] \leq \text{RE} \& \& P) : \text{bool}
\end{align*}
\]

Fig. 9: Typing rules for assertions. The predicate isClass returns true just when \( C \) is a declared class name in the program.

The abbreviation of \( \text{if} \ E \ \text{then} \ \epsilon \ \text{else} \ \emptyset \). A verified method must make sure that the actual write effect in the method is the sub-effect of the specified effect in the frame condition. Consider verifying Add’s frame condition in Fig. 1. The frame obligation for line 30 is:
\[
\text{region}[\text{this}.fpt] \leq (\text{if} \ \text{Find}(k) = \text{null} \ \text{then} \ \text{region}[\text{this}.fpt] \ \text{else} \ \text{region}[\text{Find}(k).val])
\]
with the assumption \( \text{Find}(k) = \text{null} \). We use a sub-effect rule defined in Fig. 12 to reason about such cases; it encodes the standard properties of sets.

Effects must be healthy; for example, in a given state, \( \text{reads} \ x \) is meaningless if \( x \) is not in the domain of the store.

**Definition 1.** (Healthy Effects) Let \( \delta \) be an effect and \( (\sigma, H) \) be a state. \( \delta \) is healthy in the state \( (\sigma, H) \) if
1. for all \( \text{reads} \ x \in \delta, x \in \text{dom}(\sigma) \)
2. for all \( \text{reads} \ \text{RE} \leq \delta, \mathcal{R}[\text{RE}]((\sigma, H)) \neq \text{err} \).
Fig. 10: Semantics of assertions. Note that assertions with errors are false, so this is two-valued logic.

Definition 2. (changes allowed by write and freshness effects) Let \( \epsilon \) be a healthy effect in \( \Gamma \), and \((\sigma, h)\) and \((\sigma', h')\) be \( \Gamma' \)-states for some \( \Gamma' \supseteq \Gamma \). \( \epsilon \) allows change from \((\sigma, h)\) to \((\sigma', h')\), written \( (\sigma, h) \vdash \epsilon \) if and only if:

1. for all \( x \in \text{dom}(\Gamma') \), either \( \sigma(x) = \sigma'(x) \) or modifies \( x \) is in \( \epsilon \);
2. for all \((o, f) \in \sigma(\text{alloc})\), either \( h[o, f] = h'[o, f] \) or there is \( \text{RE} \) such that modifies\( \text{RE} \) is in \( \epsilon \), such that \((o, f) \in \mathcal{R}[\Gamma] \vdash \text{RE} : \text{region}[\sigma, h] \);
3. for all \( \text{RE} \) such that fresh\( \text{RE} \) is in \( \epsilon \), 
   \[ \mathcal{R}[\Gamma] \vdash \text{RE} : \text{region}[\sigma', h'] \subseteq \sigma'(\text{alloc}) - \sigma(\text{alloc}) \]
Definition 3 says that if two states agree on a read effect, \( \delta \), then the values of the expressions that depend on \( \delta \) are identical.

**Definition 3.** (Agreement on read effects) Let \( \delta \) be an effect that is type-checked in \( \Gamma \). Let \( \Gamma' \supseteq \Gamma \) and \( \Gamma'' \supseteq \Gamma \). Let \( (\sigma', h') \) and \( (\sigma'', h'') \) be a \( \Gamma' \)-state and a \( \Gamma'' \)-state respectively. Then \( (\sigma', h') \) and \( (\sigma'', h'') \) agree on \( \delta \), \( (\sigma', h') \overset{\delta}{=} (\sigma'', h'') \), if and only if:
1. for all reads \( x \in \delta \), \( \sigma'(x) = \sigma''(x) \)
2. for all \( RE, o, f \notin \text{reads} RE \) \( \leq \delta \), \( R[\sigma] {\mathcal{R}}\sigma, H = v, (o, f) \in v \), we have \( H'[o, f] = H''[o, f] \).

Definition 4 says that if two states agree on a read effect, \( \delta \), then the validity of an assertion, \( Q \), that depends on that read effect is preserved under condition, \( P \).

**Definition 4.** (Frame validity) \( P \vdash_\Gamma \delta \text{ from } Q \) is valid, written \( P \models_\Gamma \delta \text{ from } Q \), if and only if for all \( \Gamma \)-states \( (\sigma, h), (\sigma', h') \), if \( (\sigma, h) \overset{\delta}{=} (\sigma', h') \) and \( \sigma, h \models_\Gamma P \land_Q Q \), then \( \sigma', h' \models_\Gamma Q \).
Fine-Grained Region Logic

\[ \vdash \epsilon \leq \epsilon \]

\[ \epsilon, \epsilon' \leq \epsilon, \epsilon \]

\[ \vdash \text{fresh } RE, \epsilon \leq \epsilon \]

\[ \vdash \epsilon \leq \epsilon' \]

\[ \begin{align*}
P \vdash \epsilon_1 \leq \epsilon_2 & \quad P \vdash \epsilon_2 \leq \epsilon_3 \\
\vdash \text{modifies } RE_1, RE_2 & \leq \text{modifies } RE_1 + RE_2
\end{align*} \]

\[ \begin{align*}
P \vdash \epsilon_1 \leq \epsilon_2 \\
\vdash \text{modifies filter } \{ RE, C, f \} & \leq \text{modifies } RE
\end{align*} \]

\[ \begin{align*}
\vdash \text{modifies filter } \{ RE, C \} & \leq \text{modifies } RE \\
\vdash \text{modifies } RE_1 \leq \text{modifies } RE_2
\end{align*} \]

\[ \begin{align*}
RE_1 \leq RE_2 & \vdash \text{reads } RE_1 \leq \text{reads } RE_2 \\
\vdash \text{if } E \text{ then } \epsilon_1 \text{ else } \epsilon_2 & \leq \text{if } E_2 \text{ then } \epsilon_3 \text{ else } \epsilon_4
\end{align*} \]

Fig. 12: Subeffect rules. The sub-region rule is defined in Fig. 11.

Lemma 1. (frame soundness of expressions) Let \((\sigma, h)\) and \((\sigma', h')\) be arbitrary \(\Gamma\)-states. Let \(HE\) be an expression. If \((\sigma, h) \xrightarrow{\text{fps}(HE)} (\sigma', h')\), then \(E' \vdash HE : T \| (\sigma, h) = E' \| \Gamma \vdash HE : T \| ((\sigma', h'))\).

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

Lemma 2. (frame soundness of region expressions) Let \((\sigma, h)\) and \((\sigma', h')\) be arbitrary \(\Gamma\)-states. Let \(RE\) be a region expression. If \((\sigma, h) \xrightarrow{\text{fps}(RE)} (\sigma', h')\), then \(\mathcal{R} \vdash \Gamma \vdash \text{region} \| (\sigma, h) = \mathcal{R} \vdash \Gamma \vdash \text{region} \| (\sigma', h')\).

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

Lemma 3. (frame soundness of assertions) Every derivable framing judgment is valid.

Proof. By induction on a derivation of a framing judgment \(P \vdash \delta \text{ from } Q\).

We use \(\to\) to define the disjointness on effects in Fig. 16. \(\delta \to \epsilon\) means that the read effects in \(\delta\) are disjoint with the write effects in \(\epsilon\). We treat \(\text{reads } \delta\), where \(\delta\) is not a conditional effect, as \(\text{reads if } true \text{ then } \delta \text{ else } \emptyset\). For example, let \(RE\) be \(\text{if } x.f=0 \text{ then region } \{ y.f \} \text{ else region}\). Suppose \(x \neq y\) and \(x.f \neq 0\). The
fpt(x) = reads x
fpt(n) = ∅
fpt(null) = ∅
fpt(HE.f) = reads fpt(HE), region{HE.f}
fpt(HE1 ⊕ HE2) = fpt(HE1), fpt(HE2)
fpt(region{}) = ∅
fpt(region{HE.f}) = fpt(HE)
fpt(region{HE,*}) = fpt(HE)

fpt(if HE then RE1 else RE2) = \begin{cases} fpt(HE), fpt(RE1) & \text{if } HE \neq 0 \\ fpt(HE), fpt(RE2) & \text{otherwise} \end{cases}
fpt(filter{RE, f}) = fpt(RE)
fpt(filter{RE, C, f}) = fpt(RE)
fpt(RE1 ⊕ RE2) = fpt(RE1), fpt(RE2)
fpt(E1 = E2) = fpt(E1), fpt(E2)
fpt(x.f = E) = reads x, region{x.f}, fpt(E)
fpt(RE1 ⊆ RE2) = fpt(RE1), fpt(RE2)

Fig. 13: Footprint of expressions, region expressions and atomic assertions.

The separation of reads region{y.f} and modifies RE can be derived to reads region{y.f} /\ modifies region[] by the rule CONMASK introduced in the next section.

**Lemma 4.** Let RE1 and RE2 be two regions. Let σ be a state. If σ \models^\Gamma RE1 \parallel RE2, then reads RE1 /\ modifies RE2 and reads RE2 /\ modifies RE1.

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

**Lemma 5.** (separator agreement) Let ϵ and δ be effects that are type-checked in Γ. Let Γ' ⊇ Γ. Let (σ, h) and (σ', h') be Γ' states, such that (σ', h') = MSΓ' ⊢ S : ok(Γ') \[(CT)(σ, h).

Let ϵ be the write effect of executing S, and (σ, h) \models δ /\ ϵ, then (σ, h) \models δ /\ (σ', h').

**Proof.** According to Definition 3, there are two cases.
1. Let reads x \in δ be arbitrary. Since (σ, H) \models δ /\ ϵ, modifies x \notin ϵ. So we have σ(x) = σ'(x).
2. Let reads RE \in δ be arbitrary. Since (σ, H) \models δ /\ ϵ, for all modifies RE' \in ϵ, we have RE \parallel RE'. So we have for all (o, f) \in E, [Γ \models RE : region][(σ, H)], we have H[o, f] = H'[o, f]

To prevent the effects of two sequential statements, immune of two effects under certain condition is introduced. Consider the example, x := y; x.f := 5. The write effect of the first statement is modifies x, and that of the second statement is region{x, f}. The effect of their composition is not necessarily modifies x, region{x, f}. 


Fine-Grained Region Logic

\[
\begin{align*}
\text{FRMFTPT} & \quad P \text{ is atomic} \\
& \quad \text{true} \vdash \operatorname{ftpt}(P) \text{ from } P \\
\text{FRMSUB} & \quad R \vdash \delta_1 \text{ from } Q \\
& \quad Q \vdash \delta_1 \leq \delta_2 \quad P \Rightarrow R \\
& \quad P \vdash \delta_2 \text{ from } Q \\
\text{FRMCONJ} & \quad P \vdash \delta \text{ from } Q_1, Q_2 \\
& \quad P \vdash \delta \text{ from } Q_1 \& Q_2 \\
\text{FRMDISJ} & \quad P \vdash \delta \text{ from } Q_1, Q_2 \\
& \quad P \vdash \delta \text{ from } Q_1 \mid Q_2 \\
\text{FRMFTPTNEG} & \quad P \text{ is atomic} \\
& \quad \text{true} \vdash \operatorname{ftpt}(P) \text{ from } \lnot P \\
\text{FRM\ensuremath{\forall}_1} & \quad P \vdash \operatorname{reads} \operatorname{ftpt}(RE) \leq \delta \\
& \quad P \land \operatorname{reads} \text{region}[x.f] \leq \delta \vdash \text{reads } x \text{. region}[x.f] \text{ from } Q \\
& \quad P \vdash \delta \text{ from } \forall x : \text{int } Q \\
\text{FRM\ensuremath{\exists}_1} & \quad P \vdash \text{reads } x \leq \delta \text{. reads } x \text{ from } Q \\
& \quad P \vdash \delta \text{ from } \exists x : \text{int } Q \\
\text{FRM\ensuremath{\exists}_2} & \quad P \vdash \operatorname{reads} \operatorname{ftpt}(RE) \leq \delta \\
& \quad P \land \operatorname{reads} \text{region}[x.f] \leq \delta \vdash \text{reads } x \text{. region}[x.f] \text{ from } Q \\
& \quad P \vdash \delta \text{ from } \exists \text{region}[x.f] \leq RE.Q \\
\end{align*}
\]

Fig. 14: Rules for the framing judgment. \(\Gamma\) is omitted when it is the same in the judgment. \(\Sigma = (\sigma, H)\)

**Definition 5.** (immune) Let \(RE\) be a region expression, \(P\) be an assertion, and \(\epsilon\) and \(\delta\) be two effects. \(RE\) is **immune from** \(\epsilon\) under \(P\), written \(\text{immune}(P, RE, \epsilon)\), if and only if \(P \Rightarrow \text{ftpt}(RE) / \epsilon\). \(\delta\) is immune from \(\epsilon\) under \(P\), written \(\text{immune}(P, \epsilon, \epsilon)\), if and only if for all **modifies** \(RE\) in \(\delta\), \(\text{immune}(P, RE, \epsilon)\).

**Lemma 6.** Let \(\epsilon\) be an effect, \(RE\) be a region expression, and \(P\) be an assertion, such that \(\text{immune}(P, RE, \epsilon)\). Then \(\Sigma(RE) = \Sigma'(RE)\) for any \(\Sigma, \Sigma'\) such that \(\Sigma \Rightarrow \Sigma' \models \epsilon\) and \(\Sigma \models P\).

The following lemma is used in proving Theorem 1.

**Lemma 7.** (effect transfer) Let \(\Sigma_0, \Sigma_1, \Sigma_2\) be states. Let \(\epsilon_1\) and \(\epsilon_2\) 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. \(\Sigma_0 \rightarrow \Sigma_1 \models \epsilon_1\) and \(\Sigma_1 \rightarrow \Sigma_2 \models \epsilon_2\);
3. \(\text{immune}(P, \epsilon_2, \epsilon_1)\);
4. for all fresh \(RE \in \epsilon_1, \text{immune}(P_1, RE, \epsilon_2)\)
\( \vdash \text{region}\{\} \implies \text{RE} \quad \vdash \text{RE}_1 \implies \text{RE}_2 \implies \text{RE}_3 \quad \vdash f_1 \neq f_2 \quad \vdash \text{region}\{\text{HE}_1,f_1\} \implies \text{region}\{\text{HE}_2,f_2\} \)

\( \vdash \text{HE}_1 \neq \text{HE}_2 \)

\( \vdash \text{region}\{\text{HE}_1,f\} \implies \text{region}\{\text{HE}_2,f\} \)

\( P \vdash \text{RE}_1 \implies \text{RE}_2 \quad P \vdash \text{RE}_2 \implies \text{RE}_3 \)

\( P \vdash \text{RE}_1 \implies \text{RE}_2 \quad \vdash \text{RE}_1 \implies (\text{RE}_2 + \text{RE}_3) \)

\( P' \vdash \text{RE}_1 \implies \text{RE}_2 \quad P \vdash \text{RE}_1 \implies \text{RE}_2 \)

\( P' \vdash \text{RE}_1 \implies \text{RE}_2 \)

\( \vdash \text{if \ HE \ then \ RE}_1 \text{\ else \ RE}_2 \implies \text{RE} \quad \vdash \text{if \ HE \ then \ RE}_1 \text{\ else \ RE}_2 \implies \text{RE} \)

\( \vdash \text{filter}\{\text{RE}_1,C_1,f_1\} \implies \text{filter}\{\text{RE}_2,C_2,f_2\} \)

\( \vdash \text{filter}\{\text{RE}_1,C_1\} \implies \text{filter}\{\text{RE}_2,C_2\} \)

Fig. 15: Disjoint region rules

\( \text{reads} \text{RE}_1 \mid \text{modifies} \text{RE}_2 = \text{RE}_1 \implies \text{RE}_2 \text{ reads} \ y \mid \text{modifies} \ x \neq y \neq x \)

\( \delta \mid \epsilon = \text{true} \) for all other pairs of atomic effects \( \delta \mid \epsilon = \text{true} \) in case \( \delta \) or \( \epsilon \) is empty

\( \delta \mid (\epsilon \mid \epsilon') = (\delta \mid \epsilon) \land (\delta \mid \epsilon') \quad (\delta, \delta') \mid \epsilon = (\delta \mid \epsilon) \lor (\delta' \mid \epsilon) \)

\( \text{if E then } \delta_1 \text{ else } \delta_2 \)

\( = \begin{cases} \delta_1 \mid \epsilon_1 \text{ if } P \& \& P' \\ \delta_1 \mid \epsilon_2 \text{ if } P \& \& \lnot P' \\ \delta_2 \mid \epsilon_1 \text{ if } \lnot P \& \& P' \\ \delta_2 \mid \epsilon_2 \text{ if } \lnot P \& \& \lnot P' \end{cases} \)

(Let \( P = E \neq 0, \text{ and } P' = E' \neq 0 \))

Fig. 16: Separator. \( \delta \) is read effect and \( \epsilon \) is write effect.

Then \( \Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2 \).

Proof: We need to prove that \( \Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2 \) satisfies the conditions defined in Definition 2.

For condition (1) in Definition 2, 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, \( \text{modifies} \ x \) is either in \( \epsilon_1 \) or in \( \epsilon_2 \) or both.

For condition (2) in Definition 2, let \( (o,f) \in \Sigma_0(\text{alloc}) \), 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 \( \text{RE} \), such that \( \text{modifies} \ \text{RE} \in \epsilon_1 \) and \( \Sigma_0(\text{RE}) = \{(o,f)\} \). By assumption 3 and Lemma 6, \( \Sigma_1(\text{RE}) = \Sigma_2(\text{RE}) \). So \( \Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2 \)
2. $\Sigma_1(o,f) \neq \Sigma_2(o,f)$: By assumption 2, there is some RE, such that modifies RE $\in \epsilon_2$ and $\Sigma_1(RE) = \{(o,f)\}$. So $\Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2$

For condition (3) in Definition 2, there are two cases:

1. Suppose fresh(RE) $\in \epsilon_1$. By assumption 2, $\Sigma_1(RE) \subseteq \Sigma_1($alloc$) - \Sigma_0($alloc$)$, and $\Sigma_1($alloc$) \subseteq \Sigma_2($alloc$)$. By assumption 3, $\Sigma_1(RE) = \Sigma_2(RE)$. So $\Sigma_2(RE) \subseteq \Sigma_2(RE) - \Sigma_0($alloc$)$. So $\Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2$.

2. Suppose fresh(RE) $\in \epsilon_2$. By assumption 2, $\Sigma_2(RE) \subseteq \Sigma_2($alloc$) - \Sigma_1($alloc$)$, and $\Sigma_1($alloc$) \subseteq \Sigma_0($alloc$)$. So $\Sigma_2(RE) \subseteq \Sigma_2(RE) - \Sigma_0($alloc$)$. So $\Sigma_0 \rightarrow \Sigma_2 \models \epsilon_1, \epsilon_2$.

$\square$

5 Program Correctness

The validity of a Hoare-formula $\{P\} S \{Q\} [e]$ means that if a program, $S$, executes from an initial state satisfying $P$, $S$ does not cause an error, and $S$ terminates, then the final state satisfies $Q$, and any change happens in $e$. For brevity, we omit modifies when expressing the write effects. The formalization is shown as follows.

**Definition 6 (FRL Valid of Hoare-Formula).** Let $S$ be a method body, let $P_1$ and $P_2$ be assertions, let $e$ be effects, and let $(\sigma, H)$ be a state. Then $\{P_1\} S \{P_2\}[e]$ is valid in $(\sigma, H)$, written $\sigma, H \models_{\{P_1\} S \{P_2\}[e]}$, if and only if whenever $\sigma, H \models_{P_1}$ and $(\sigma', H') = MS[S][(\sigma, H)]$, then $\sigma', H' \models_{P_2}$ and for all $(o, f) \in \text{dom}(H)$,

$$H'[o, f] \neq H[o, f] \Rightarrow (o, f) \in R[e][\sigma, H].$$

A Hoare-formula $\{P_1\} S \{P_2\}[e]$ is valid, written $\models_{\{P_1\} S \{P_2\}[e]}$, if and only if for all states $(\sigma, H)$, $\sigma, H \models_{\{P_1\} S \{P_2\}[e]}$.

**Theorem 1.** Every derivable correctness judgment is valid.

**Proof.** We prove this by induction on the structure of the proof of $\{P\} S \{Q\} [e]$. Let $(\sigma, h)$ be an arbitrary state, and without loss of generality, let $(\sigma', h') = MS[\Gamma] \models (S : \text{ok}(\Gamma))[(\sigma, h)]$.

We assume $\Gamma \models \{P\} S \{Q\} [e]$, and $\sigma, h \models_{\Gamma} P$. Then we must prove $\sigma', h' \models_{\Gamma} Q$, and that all the changed locations are in $e$.

1. (ALLOC) In this case, $S$ is $x := \text{new} K$, $P$ is true, $Q$ is $x.\bar{T} = \text{default}(T)$ and $e = x.\text{alloc.fresh.region}(x, \ast)$. We derive $Q$ as below:

By the semantics, $(\sigma', h') = (\text{let}(\sigma'', h'' = \text{alloc}(h) \text{ in } (\sigma'', h'')[x][\bar{T} \mapsto \text{default}(T)])$, which entails $Q$.

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

2. (ASSGN) In this case, $S$ is $x := F$, $P$ is $x = x'$, $Q$ is $\{x = F'(x \mapsto x')\}$ and $e = x$. By the semantics, $(\sigma', H') = (\sigma[x \mapsto E, \Gamma \models F : \Gamma'(x)](\sigma, H))$, which entails $Q$.

For the frame condition, this statement only updates variable $x$. Therefore $e = x$ is a correct frame.
3. \((\text{FIELDUPD})\) In this case, \(S\) is \(x.f := F\), \(P\) is \(x \neq \text{null} \land x' = F\), \(Q\) is \(x.f = x'\) and \(\epsilon = \text{region}\{x.f\}\). By the semantics, \((\sigma', H') = (\sigma, H[(\mathcal{E}, \mathcal{F}((\sigma, H), f)) \mapsto \mathcal{E}, \mathcal{F}]]((\sigma, H)))\), which entails \(Q\).

For the frame condition, this statement changes the singleton heap location \((x, f)\).

Therefore \(\epsilon = \text{region}\{x.f\}\) is a correct frame.

4. \((\text{FIELDACC})\) In this case, \(S\) is \(x := x_2.f\), \(P\) is \(x_2 \neq \text{null} \land x_1 = x_2\), \(Q\) is \(x = x_1.f\), and \(\epsilon = x\). We derive \(Q\) as below:

By the semantics, \((\sigma', H') = (\sigma[x \mapsto H[(\mathcal{E}, \mathcal{F}((\sigma, H), f))]], H)\), which entails \(Q\).

For the frame condition, this statement only updates variable \(x\). Therefore \(\epsilon = x\) is a correct frame.

5. \((\text{SEQ})\) In this case, \(S\) is \(S_1; S_2\). By the inductive hypothesis for \(S_1\) and \(S_2\), \((\sigma'', H'') = \mathcal{M}\mathcal{S}[\Gamma \vdash S_1 : \text{ok}(\Gamma)](\sigma, H)\), and \((\sigma'', H'') \vdash^T P_1\). By the second premise and the semantics, \((\sigma', H') = \mathcal{M}\mathcal{S}[\Gamma \vdash S_2 : \text{ok}(\Gamma)](\sigma'', H'').\) Hence \((\sigma', H') \vdash^T P_2\).

For the frame condition, we must show \((\sigma, H) \rightarrow (\sigma', H') = \epsilon_1, \epsilon_2, \text{fresh}(RE)\).

It can be proved by the fourth, fifth premises and Lemma 7.

6. \((\text{IF})\) In this case, \(S\) is \(\text{if} E \text{ then } S_1 \text{ else } S_2\).

There are two cases:

Case 1: \(E \neq 0\). By the inductive hypothesis, \((\sigma', H') = \mathcal{M}\mathcal{S}[\Gamma \vdash S_1 : \text{ok}(\Gamma)](\sigma, H)\), which entails \(Q\).

Case 2: \(E = 0\). By the inductive hypothesis, \((\sigma', H') = \mathcal{M}\mathcal{S}[\Gamma \vdash S_2 : \text{ok}(\Gamma)](\sigma, H)\), which entails \(Q\).

For the frame condition, the induction hypothesis, \(\epsilon\) is a correct frame.

7. \((\text{WHILE})\) In this case, \(S\) is \(\text{while} E \text{ do } S\). \(P = I, Q = I \land E \neq 0\) and the frame conditions is \(\epsilon\). The premise is \(\vdash \{I \land E \neq 0\} S \{I\}[\epsilon]\).

By the semantics of this statement, let \(g\) be a recursive point function, such that \(g = \lambda s. \text{if } E \neq 0 \text{ then } (\mathcal{E}, \mathcal{F}((\sigma, H))) \text{ then let } s' = \mathcal{M}\mathcal{S}[\Gamma \vdash S : \text{ok}(\Gamma)](\sigma, H)\text{in } g s' \text{ else } s'\).

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

Base Case: \(\bot \vdash^T I\) 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 \((\sigma'', H'') \vdash^T I\) hold for an arbitrary iteration of \(g\), and \(\epsilon\) is the frame condition. Then we prove that \(\text{fix}(g)(\sigma'', H'') \vdash^T I\) holds, and the changed locations on the heap are fixed.

There are two cases:

Case 1: \(E \neq 0\). By the semantics, \(\text{fix}(g)(\sigma'', H'') = g(\mathcal{M}\mathcal{S}[\Gamma \vdash S : \text{ok}(\Gamma)](\sigma'', H''))\).

By the inductive hypothesis, \(g(\mathcal{M}\mathcal{S}[\Gamma \vdash S : \text{ok}(\Gamma)](\sigma'', H'')) \vdash^T I\) holds. Hence \(\text{fix}(g)(\sigma'', H'') \vdash^T I\) holds. For the frame condition, since the fixed point function always returns the same function \(g\), which is framed by \(\epsilon\) by the induction hypothesis, therefore \(\epsilon\) is the frame condition for an arbitrary iteration.

Case 2: \(E = 0\). By the semantics, \(\text{fix}(g)(\sigma'', H'') = (\sigma'', H'')\). Therefore by the inductive hypothesis,
$\text{fix}(q)(\sigma^H, H^H) \vdash_\Gamma I$ holds. For the frame condition, since the state does not change, the frame is $\text{region}[]$, which is the subset of $\epsilon$.

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

8. (SUBEFF) By the inductive hypothesis, $\vdash_\Gamma \{P\} S(Q)[\epsilon]$. Hence when applying the frame condition $\epsilon' \supseteq \epsilon$, the locations that may be changed are also contained in $\epsilon'$. Therefore $\epsilon'$ is a correct frame.

9. (CONSEQ) In this case, by the inductive hypothesis, $\vdash_\Gamma \{P'\} S(Q')[\epsilon]$. By the premise, $P \Rightarrow P'$ and $Q' \Rightarrow Q$. Hence $\vdash_\Gamma \{P\} S(Q)[\epsilon]$ is valid.

\[
\begin{align*}
\text{ALLOC} & \quad \text{Fields}(C) = \overline{f} : T \\
\Rightarrow & \quad \{\text{true}\} x := \text{new } C[x.\text{default}(T)]x, alloc, \text{fresh}[\text{region}(x*])
\end{align*}
\]

\[
\begin{align*}
\text{ASSIGN} & \quad x' \not\equiv x \\
\Rightarrow & \quad \{x = x'\} x := F\{x = (F[x \mapsto x'])\}[x]
\end{align*}
\]

\[
\begin{align*}
\text{FIELDACC} & \quad x_1 \not\equiv x \\
\Rightarrow & \quad \{x_2 \neq \text{null} \land x_1 = x_2\} x := x_2.f\{x = x_1.f\}[x]
\end{align*}
\]

\[
\begin{align*}
\text{FIELDUPD} & \quad \{x \neq \text{null} \land x' = F\} x.f := F\{x.f = x'\}[\text{region}(x.f)]
\end{align*}
\]

\[
\begin{align*}
\text{SEQ} & \quad \{P\} S_1 \{P_1\}[\epsilon_1, \text{fresh}(RE)] \\
& \quad \{P_1\} S_2 \{P'\}[\epsilon_2] \\
& \quad \epsilon_1 \text{ is fresh-free} \quad \text{immune}(P_1, \epsilon_2, \epsilon_1) \\
& \quad \text{immune}(P, RE, \epsilon_2) \\
\Rightarrow & \quad \{P\} S_1 S_2 \{P'\}[\epsilon_1, \epsilon_2, \text{fresh}(RE)]
\end{align*}
\]

\[
\begin{align*}
\text{IF} & \quad \vdash \{P \land E \neq 0\} S_1 \{P'\}[\epsilon] \quad \vdash \{P \land E = 0\} S_2 \{P'\}[\epsilon] \\
\Rightarrow & \quad \vdash \{P\} \text{if } \epsilon \text{ then } S_1 \text{ else } S_2 \{P'\}[\epsilon]
\end{align*}
\]

\[
\begin{align*}
\text{WHILE} & \quad \vdash \{I \land \epsilon \neq 0\} S(I)[\epsilon] \\
\Rightarrow & \quad \vdash \{I\} \text{ while } \epsilon \text{ do } S(I \land E = 0)[\epsilon]
\end{align*}
\]

Fig. 17: Correctness axioms and rules for statements. alloc is a special variable containing the set of locations. The keyword modifies is omitted when there is no confusion.
Fig. 18: Structural rules. alloc is a special variable containing the set of locations.
6 Examples

6.1 Examples with conditional effects

Consider the example:

\[
S \triangleq \text{if } t \text{ then } x.f = 5 \text{ else } y.f = 5
\]

\[
P \triangleq x \neq y \land x.f = 4 \land y.f = 4
\]

\[
P' \triangleq (t \neq 0 \Rightarrow x.f = 5) \land (t = 0 \Rightarrow y.f = 5)
\]

\[
\epsilon \triangleq \text{if } t \text{ then } \text{region}(x.f) \text{ else } \text{region}(y.f)
\]

We do forward reasoning:

1. \{x \neq y \land x.f = 4 \land y.f = 4\}
2. \text{if } t
3. \quad \text{then } \{
4. \quad \{t \neq 0 \land x \neq y \land x.f = 4 \land y.f = 4\}
5. \quad x.f := 5
6. \quad \{t \neq 0 \land x \neq y \land x.f = 5 \land y.f = 4\} \text{ [region}(x.f)\text{]}
7. \quad \text{else } \{
8. \quad \{t = 0 \land x \neq y \land x.f = 4 \land y.f = 4\}
9. \quad y.f := 5
10. \quad \{t = 0 \land x \neq y \land x.f = 4 \land y.f = 5\} \text{ [region}(y.f)\text{]}
11. \}

Formula in line 6 implies

\[
(t \neq 0 \land x \neq y \land x.f = 5 \land y.f = 4) \lor
(t = 0 \land x \neq y \land x.f = 4 \land y.f = 5)
\]

(1)

Formula in line 10 also implies (1) that is equivalent to \(P'\). Using the rule CONSEQ, we derive:

\[
\vdash \{t \neq 0 \land x \neq y \land x.f = 4 \land y.f = 4\} x.f := 5 \{P'\}
\]

\[
\text{ [region}(x.f)\text{]}
\]

(2)

\[
\vdash \{t = 0 \land x \neq y \land x.f = 4 \land y.f = 4\} y.f := 5 \{P'\}
\]

\[
\text{ [region}(y.f)\text{]}
\]

(3)

Consider the formula (2). Since

\[
\vdash \{t = 0 \land t \neq 0 \land x \neq y \land x.f = 4 \land y.f = 4\} y.f := 5 \{P'\}
\]

\[
\text{ [region}(x.f)\text{]}
\]

(4)

is vacuously true, we can use rule IF and derive to

\[
\vdash \{t \neq 0 \land x \neq y \land x.f = 4 \land y.f = 4\} S\{P'\}
\]

\[
\text{ [region}(x.f)\text{]}
\]

(5)
Similarly, we can derive (3) to
\[ \{ t = 0 \land x \neq y \land x.f = 4 \land y.f = 4 \} S[P'] \]
\[ \text{region}\{y.f\} \] (6)

Using the rule CONEFF, we can prove \( \{ P \} S[P'][\varepsilon] \).

We do backward reasoning. Approach 1 with the rule IF. We need to prove:
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t \neq 0 \} x.f := 5 \{ P' \} [\varepsilon] \] (7)

and
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t = 0 \} y.f = 5 \{ P' \} [\varepsilon] \] (8)

Consider the formula (7), since \( x \neq y \land x.f = 4 \land y.f = 4 \land t \neq 0 \Rightarrow t \neq 0 \), using
the rule CONMASK1, the proof obligation can be discharged to:
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t = 0 \} x.f := 5 \{ P' \} \]
\[ \text{region}\{x.f\} \] (9)

Let \( Q \) be \( x \neq y \land f = 4 \land t \neq 0 \) that is the frame. Let \( \delta \) be reads \( x, y, f, t \). Since
\[ x.f = 4 \vdash \delta \text{ frm } Q \] (10)

and
\[ x \neq y \land x.f = 4 \land y.f = 4 \land t = 0 \Rightarrow \]
\[ \delta/\text{modifies region}\{x.f\} \] (11)

Using the rule FRAME, the proof discharged to
\[ \{ x.f = 4 \} x.f := 5 \{ x.f = 5 \} \text{region}\{x.f\} \] (12)

Using the rule FIELDUPD, one can prove (12). The formula (8) can be proved in a
similar way.

Approach 2 with the rule CONEFF. We need to prove:
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t \neq 0 \} S[P'] \text{region}\{x.f\} \] (13)

and
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t = 0 \} S[P'] \text{region}\{y.f\} \] (14)

Consider the proof of the formula (13). Using the rule IF, one need to prove
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t \neq 0 \land t \neq 0 \} x.f := 5 \{ P' \} \]
\[ \text{region}\{x.f\} \] (15)

and
\[ \{ x \neq y \land x.f = 4 \land y.f = 4 \land t = 0 \land t \neq 0 \} y.f := 5 \{ P' \} \]
\[ \text{region}\{y.f\} \] (16)

The formula (16) is vacuously true. The other formula can be proved by the rules
FRAME and FIELDUPD.
6.2 Loop

Consider the example:

\[
\begin{align*}
\Gamma & \overset{\text{def}}{=} \text{alloc : region}, r : \text{region}, n : \text{Node} \\
S & \overset{\text{def}}{=} r := \text{region}[]; \text{while } \text{true } \text{do } S_w \\
S_w & \overset{\text{def}}{=} n := \text{new Node}; r := r + \text{region}\{n.*\}; \\
P & \overset{\text{def}}{=} \text{true} \\
Q & \overset{\text{def}}{=} \text{true} \\
\epsilon & \overset{\text{def}}{=} r, n, \text{alloc, fresh}(r)
\end{align*}
\]

This example is to show the derivation of write effects in a loop. We want to prove:

\[\models^\Gamma \{P\} S \{Q\}[\epsilon]\]

Consider the client code in the introduction. Before \textit{o.sync()} is called, we have the state that satisfies \(Q = \{o.data = 5 \land o.valid = \text{false}\}\). The postcondition of \textit{o.sync()}, \(P'\), is \(o.valid \implies o.data = 6\), and the frame condition, \(\epsilon\), is \(o.valid \implies \text{region}\{o.data\}\). Apply the rule CONMASK2, we obtain the new frame, \(\epsilon'\), that is \text{region}[]. \(Q\)'s read effects, \(\delta\), is \text{reads}\{o.o.data, o.valid\}. So \(\delta/\epsilon\). Therefore, we can apply the FRAME rule, and conclude that \(Q\) is valid after \textit{o.sync()} is called. Therefore the assertion \textit{assert o.data = 5} is valid.

7 Acknowledgments

Thanks to David Naumann for discussions about region logic.

References


