Conditional Effects in Fine-grained Region Logic

Yuyan Bao, Gary T. Leavens, and Gidon Ernst

CS-TR-15-01
March 2015

Keywords: Frame axiom, modifies clause, separation logic, dynamic frames, region logic, formal methods, Dafny language, DafnyR language.
2013 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, pre- and post-conditions, specification techniques;

Submitted for publication.

Computer Science
4000 Central Florida Blvd.
University of Central Florida
Orlando, Florida 32816, USA

Conditional Effects in Fine-grained Region Logic

Yuyan Bao[*]
University of Central Florida
Orlando, FL 32816 USA
ybao@eecs.ucf.edu

Gary T. Leavens
University of Central Florida
Orlando, FL 32816 USA
leavens@ucf.edu

Gidon Ernst
Universität Augsburg,
D-86135 Augsburg, Germany
ernst@isse.de

ABSTRACT

Specification languages have long featured ways to describe what does not change when an imperative procedure is executed: the so-called frame problem. Solutions to the frame problem are needed for formal verification in imperative programming, as otherwise a verification would not be able to accumulate information from one statement to the next. Region logic is one of the approaches to solving the frame problem. We present a modified version of region logic with fine-granularity and introduce conditional effects that allows one to specify more precise frame conditions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Program Verification—*Class invariants, correctness proofs, formal methods, programming by contract, object orientation*; D.3.1 [**Programming Languages**]: Definitions and Theory—*Classes and objects, modules, packages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques*

General Terms

Verification, Language

Keywords

Region logic, framing, formal specification, formal verification, Hoare logic

1. INTRODUCTION

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 = `**`old`**`(a)`, which says a's value is the same as the value before the method call. Dynamic frames [7, 8] 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 FRL. It is an adaption of the Dafny's solution [10] to the problem ♯4 in the VSI benchmark [**?**]. The purpose of this example is to overview 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[1] programing language is a sequential programming language. A FRL program can consist 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 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[2] `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, **`modifes`** `fpt`, means 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.

[*] The work of the first two authors is supported in part by US NSF under grant CCF-0916715.

[1] FRL language is a blend of Dafny [10] and VERL [11].

[2] Functions in FRL are boolean-valued methods that have no write effects.

So one has to specify that the value of the extra locations are preserved. For example, the specification in line 22 means all the values of the key-value pairs, whose key is not k, are preserved. The expression **old**(fpt) denotes the values in the pre-state. That is the value evaluated before Add executes. We call such a postcondition a *make-up frame condition*. The condition o.key = **old**(o.key) in line 21 is also a make-up frame condition. If we allow one to specify frame conditions more precisely, then the make-up frame conditions can be avoided, such as the following frame condition and postcondition:

```
modifies if Find(key) = null then region{this.fpt}
         else region{Find(key).val};
ensures Valid() && fresh(fpt - old(fpt));
ensures ∃ region{o.*} ⊆ fpt.(o.key = key && o.val = val);
```

The frame condition uses the pure method Find that returns the reference that holds k if it is in the map, otherwise returns null reference. The technique of pure methods in region logic is discussed in Banerjee and Naumann's work [1], and is out of the scope of this paper. The frame condition says if there is an object, p, that holds the key, k, then the location **region**{p.val} can be modified, otherwise the location **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 branches the execution traces of programs. Our CONMASK1 and CONMASK2 rules discussed in section 5 can drop the condition. Therefore conditional effects do not necessarily increase the complexity of the compositions of effects.

Contribution: in this paper, we propose a fine-grained region logic. It defines a region as a set of locations, that is as ghost state in a program that can be mentioned in a modifies clauses. We introduce a novel if-then-else effects, **if** E **then** ϵ_1 **else** ϵ_2, which allows one to specify frame properties more precisely. Such conditional effects are our major contribution.

2. RELATED WORK

Our work is an adaption of region logic [2, 3]. To ease comparison, we try to use the same symbols and definitions as the work [3], such as *ftpt* function and the separator $\cdot/\cdot$. Our work has two major differences from region logic. (1) In region logic, regions, G, are sets of references, possibly containing **null** [3]. For example, $\{o\}$ is a region containing a singleton object o. Image expressions $o\text{‘}f$ denotes the location which corresponds to our **region**$\{o.f\}$. Our work made an extension to region logic to uniformly use 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. 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. The pair $(\texttt{null}, f)$ is not allowed in the regions of our languages' semantics. Since we re-defined region expressions, the *ftpt* function and $\cdot/\cdot$ are re-defined. (2) The work on region logic [2, 3] does not consider conditional effects. Recent work [1] considers a logical analysis of framing for specifications with pure method calls. Our work can also be used in method calls with side effects. We re-defined the frame judgments and the new $\cdot/\cdot$ definitions based on our new syntax. Due to the introduction of conditional effects, we revised the definition of the POSTTOFR, FRTOPOST, VARMASK and FIELDMASK rules, and also introduce the CONEFF, CONMASK1 and CONMASK2 rules in proving program correctness. These are our contributions.

```
1  class Node<Key,Value> {
2   var key: Key; var val: Value;
3   var next: Node<Key,Value>;
4  }
5  class Map {
6   var head : Node<Key, Value>;
7   ghost var fpt: region;
8
9   function Valid() : bool{ ... };
10  constructor Init()
11    modifies region{this.*};
12    ensures Valid() && fresh(fpt - region{this.*});
13  {
14    fpt := region{this.*}; head := null;
15  }
16  method Add(k: Key, v: Value)
17   requires Valid();
18   modifies fpt;
19   ensures Valid() && fresh(fpt - old(fpt));
20   ensures ∃ region{o.*} ⊆ fpt.
21    (old(o.key)=k => o.val=v && o.key=old(o.key));
22   ensures ∀ region{o.*} ⊆ old(fpt).
23    (old(o.key) != k => o.key = old(o.key) &&
24     o.val = old(o.val));
25   {
26     var p = Find(k);
27     if (p = null){
28        var h := new Node<Key, Value>;
29        h.key := k; h.val := v; h. next := head;
30        fpt := fpt + region{h.*};
31     } else {  p.val := val; }
32   }
33
34   pure method Find(k: Key)returns(p: Node<Key,Value>)
35    requires Valid();
36    reads fpt;
37    ensures p = null => ∀ region{o.*} ⊆ fpt.(o.key!=k);
38    ensures p != null => region{p.*} ⊆ fpt && p.key = k;
39    { ... }
40 }
```

Figure 1: The toy example of Map

The work [9, 12, 13] allows one to write specifications for different cases. For example, the keyword **also** is used in [6]. We consider it as a syntactic sugar for our conditional effects.

3. AN OVERVIEW OF FRL

The FRL [3] programing language is a sequential programming language. A FRL program can consist of a set of classes with fields and methods. 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. The **decreases** clause specifies termination conditions for recursive methods.

The code in Fig. 2 shows a linked-list example written in FRL. To simplify the presentation, the specification only describes the shape of a linked-list. Method Init is a list'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 newly allocated Node to fpt. The **ghost** variable fpt, declared as type **region**, is used to store abstract locations of a linked list, which is a list's dynamic frame. fpt needs to be explicitly updated when the list changes, as in method Append. Such statements involving regions cause **ghost** state changes, which must not affect the programs' control flow. For example, regions cannot appear in the tests

[3] FRL language is a blend of Dafny [10] and VERL [11].

```
class Node {
  var val: int;  var next: Node; ghost var fpt: region;

  function Valid(): bool
    reads this, fpt; {
    region{this.*} <= fpt &&
    (this.next == null ==> region{this.*} == fpt) &&
    (this.next != null ==> region{next.*} < fpt &&
     next.fpt <= fpt && !(region{this.*} <= next.fpt) &&
     fpt == region{this.*} + next.fpt && next.Valid()) }

  constructor Init(d: int)
    modifies region{this.*};
    ensures Valid() && val == d &&  next == null;
    ensures fresh(fpt - region{this.*});
  {  val := d;  next := null;  fpt := region{this.*}; }

  method Append(v: int, n: Node) returns (ret : Node)
    requires n == null || n.Valid();
    modifies n != null => n.fpt;
    ensures ret != null && ret.Valid();
    ensures n == null ==> fresh(ret.fpt);
    ensures n != null ==> n == ret &&
            fresh(n.fpt - old(n.fpt));
    decreases if n == null then region{} else n.fpt;
  {  if(n == null){ ret := new Node.Init(v);
     }else {  n.next := Append(v, n.next);
        n.fpt := n.fpt + n.next.fpt;  ret := n;}}
}
```

Figure 2: A linked-list example

of branching statements. Function[4] `Valid` describes the structure of a linked-list by logic formulas. In this example, `Valid` is considered as an invariant, which must be kept valid after a list is initialized, and before and after each method call. The frame of `Append` is specified by a conditional effect. One could specify it by **`modifies`** `n.fpt`. But the problem is when `n = null`, `n.fpt` dereferences null reference, which makes no sense. The condition `n != null` is necessary to have meaningful frame conditions in the common programming concept, although the work of region logic [2, 3] defines null.fpt as an empty set.

4. PROGRAMMING LANGUAGE

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

4.1 Syntax

Fig. 3 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 **`region`**. A method is declared in a class. In a method implementation, there are local variable declarations, update statements, condition statements, and loop statements. Unlike the type **`rgn`** in region logic [2, 3], which ranges over sets of references, the type **`region`** stores locations of fields. Each field's location is represented by a pair of an allocated reference and a field name. The region expression **`region`**$\{\}$ denotes the empty region. For each x that evaluates to an non-null object, a 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 form **`region`**$\{x.*\}$ denotes a set containing the abstract locations represented by the reference x and all its fields [5]. A region expression of the form **`filter`**$\{RE, C, f\}$ denotes one of the two meanings: 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 type C; otherwise, it is the location of form (o, f) , where each object reference, o, has the type C. A region expression of the form **`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 **`filter`**$\{RE, C\} = \{o_1.f_1, o_1.f_2\}$. The operators $+$, $-$, and $*$ denote union, difference and intersection respectively.

$$
\begin{array}{l}
\mathit{Class} ::= \texttt{class}\ C\ \{\ \overline{\mathit{Member}}\ \}\\
\mathit{Member} ::= \mathit{Field} \mid \mathit{Method}\\
\mathit{Field} ::= \texttt{var}\ f\texttt{:}T\\
\mathit{Method} ::= \texttt{void}\ m\,(\overline{x : T})\,\texttt{returns}\,(x' \texttt{:} T)\ \{\ \ \overline{S}\ \ \}\\
T ::= \texttt{int} \mid \texttt{bool} \mid \texttt{C} \mid \texttt{region}\\
E ::= n \mid x \mid \texttt{null} \mid E \oplus E\\
\mathit{RE} ::= x \mid \texttt{region}\{\} \mid \texttt{region}\{x.f\} \mid \texttt{region}\{x.*\}\\
\qquad \mid \texttt{filter}\{\mathit{RE}, C, f\} \mid \texttt{filter}\{\mathit{RE}, C\}\\
\qquad \mid \mathit{RE}_1 \otimes \mathit{RE}_2\\
F ::= E \mid \mathit{RE}\\
S ::= \texttt{var}\ x\texttt{:}T\texttt{;} \mid x\texttt{:=}F\texttt{;} \mid x_1\texttt{:=}x_2\texttt{.}f\texttt{;} \mid x\texttt{.}f\texttt{:=}F\texttt{;}\\
\qquad \mid x\texttt{:=new}\ C\texttt{;} \mid \texttt{if}\ E\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\}\texttt{;}\\
\qquad \mid \texttt{while}\ E\ \{S\}\texttt{;} \mid S_1 S_2\\
\oplus ::= \ = \mid + \mid - \mid * \mid \leqslant\ \ldots\\
\otimes ::= + \mid - \mid *
\end{array}
$$

Figure 3: The syntax of FRL language.

We use Γ for type environments, which map variables to types: $\Gamma \in \mathit{TypeEnv} = \mathit{var} \to T$. The typing rules for expressions is defined in Fig. 4, for region expressions are defined in Fig. 5, and for statements are defined in Fig. 6.

4.2 Semantics

The set *Loc* represents locations in a heap. 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:$\mathit{Value} = \mathit{Boolean} + \mathit{Object} + \mathit{Int} + \mathit{PowerSet}(\mathit{Loc})$. Region expressions evaluate to regions, i.e., sets of locations. A program Γ-state contains a store and a heap: $\Gamma\text{-}\mathit{State}{=}\mathit{Store}{\times}\mathit{Heap}$. *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. Fig. 7 shows the semantics of expressions and region expressions. The semantics of statements is standard. The disjointness of two regions can be represented by $RE_1 * RE_2 = \texttt{region}\{\}$. We use $RE_1 !! RE_2$ as a syntactic sugar for this boolean expression.

5. ASSERTION LANGUAGE

In this section, we formalize RSL's assertion language.

5.1 Syntax and semantics of assertions

The syntax of assertions is shown in Fig. 9. 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. 10. The semantics of assertions are shown in Fig. 11. $RE_1 \leqslant RE_2$ means RE_1 is subregion of RE_2. Note that the semantics identifies errors (*err*) with false.

[4] Functions in FRL are boolean-valued methods that have no write effects.

[5] Since we are not considering 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.

$$\Gamma \vdash x : T \textbf{ where } (\Gamma\ x) = T \qquad\qquad \Gamma \vdash \texttt{null} : C \textbf{ where } isClass(C) \qquad\qquad \Gamma \vdash n : \texttt{int}$$

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

Figure 4: Typing rules for expressions. The predicate *isClass* returns true just when C is a declared class name in the program.

$$\Gamma \vdash \texttt{region}\{\} : \texttt{region} \qquad\qquad \frac{\Gamma \vdash x : C}{\Gamma \vdash \texttt{region}\{x.f\} : \texttt{region}} \ \begin{array}{l}\textbf{where } \textit{isClass(C)} \\ \textbf{and } (f : T) \in fields(C)\end{array}$$

$$\frac{\Gamma \vdash x : C}{\Gamma \vdash \texttt{region}\{x.*\} : \texttt{region}} \ \textbf{where } isClass(C) \qquad\qquad \frac{\Gamma \vdash RE : \texttt{region}}{\Gamma \vdash \texttt{filter}\{RE, C, f\} : \texttt{region}}$$

$$\frac{\Gamma \vdash RE : \texttt{region}}{\Gamma \vdash \texttt{filter}\{RE, C\} : \texttt{region}} \ \textbf{where } isClass(C) \qquad\qquad \frac{\Gamma \vdash RE_1 : \texttt{region}, \qquad \Gamma \vdash RE_2 : \texttt{region}}{\Gamma \vdash RE_1 \otimes RE_2 : \texttt{region}}$$

Figure 5: Typing rules for region expressions. The predicate $isClass$ returns true just when C is a declared class name in the program. The auxiliary function *fields* takes a class name and returns a list of its declared field names and their types.

$$\Gamma \vdash \texttt{var}\, x : T; : ok(\Gamma, x : T) \qquad \frac{\Gamma \vdash x : T,\ \Gamma \vdash F : T}{\Gamma \vdash x \ := \ F; : ok(\Gamma)} \qquad \frac{\Gamma \vdash x_1 : T,\ \Gamma \vdash x_2.f : T}{\Gamma \vdash x_1 \ := \ x_2.f; : ok(\Gamma)} \qquad \frac{\Gamma \vdash x.f : T,\ \Gamma \vdash F : T}{\Gamma \vdash x.f \ := \ F; : ok(\Gamma)}$$

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

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

Figure 6: Typing rules for statements.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{E} : E \to \textit{TypingJudgment} \to \textit{State} \to \textit{Value}} \\
\quad \mathcal{E}\llbracket \Gamma \vdash x : T \rrbracket(\sigma, H) = \sigma(x) & \mathcal{E}\llbracket \Gamma \vdash E_1 \oplus E_2 : T \rrbracket(\sigma, H) = \\
\quad \mathcal{E}\llbracket \Gamma \vdash \texttt{null} : C \rrbracket(\sigma, H) = \textit{null} & \quad \textbf{let}\ v_1 \ = \ \mathcal{E}\llbracket \Gamma \vdash E_1 : T \rrbracket(\sigma, H) \textbf{in} \\
\quad \mathcal{E}\llbracket \Gamma \vdash n : \texttt{int} \rrbracket(\sigma, H) = \mathcal{N}\llbracket n \rrbracket & \quad\quad \textbf{let}\ v_2 \ = \ \mathcal{E}\llbracket \Gamma \vdash E_2 : T \rrbracket(\sigma, H)\ \textbf{in}\ v_1\ \mathcal{MO}\llbracket \oplus \rrbracket\ v_2
\end{array}$$

$$\begin{array}{ll}
\mathcal{R} : RE \to \textit{TypingJudgment} \to \textit{State} \to \textit{Value} + \{\textit{err}\} & \mathcal{R}\llbracket \Gamma \vdash \texttt{filter}\{RE, C, f\} : \texttt{region} \rrbracket(\sigma, H) = \\
 & \quad \textbf{let}\ v \ = \ \mathcal{R}\llbracket \Gamma \vdash RE : \texttt{region} \rrbracket(\sigma, H) \textbf{in} \\
\mathcal{R}\llbracket \Gamma \vdash \texttt{region\{\}} : \texttt{region} \rrbracket(\sigma, H) = \varnothing & \quad\quad \textbf{if}\ v \neq \textit{err}\ \textbf{then} \\
\mathcal{R}\llbracket \Gamma \vdash \texttt{region}\{x.f\} : \texttt{region} \rrbracket(\sigma, H) = & \quad\quad\quad \textbf{if}\ f : \texttt{region} \in fieldNames(CT, C)\ \textbf{then} \\
\quad \textbf{if}\ \sigma(x) \neq \textit{null}\ \textbf{then}\ \{(\sigma(x), f)\}\ \textbf{else}\ \textit{err} & \quad\quad\quad\quad \bigcup\{H[o, f] | (o, f) \in v \wedge Type(o) = C\} \\
\mathcal{R}\llbracket \Gamma \vdash \texttt{region}\{x.*\} : \texttt{region} \rrbracket(\sigma, H) = & \quad\quad\quad \textbf{else}\ \{(o, f') | (o, f') \in \ v \wedge f' = f \wedge Type(o) = C\} \\
\quad \textbf{if}\ \sigma(x) \neq \textit{null}\ \textbf{then} & \quad\quad \textbf{else}\ \textit{err} \\
\quad\quad \{(o, f) \mid (f : T) \in fieldNames(CT, \Gamma(x))\}\ \textbf{else}\ \textit{err} & \mathcal{R}\llbracket \Gamma \vdash RE_1 \otimes RE_2 : \texttt{region} \rrbracket(\sigma, H) = \\
\mathcal{R}\llbracket \Gamma \vdash \texttt{filter}\{RE, C\} : \texttt{region} \rrbracket(\sigma, H) = & \quad \textbf{let}\ v_1 \ = \ \mathcal{R}\llbracket \Gamma \vdash RE : \texttt{region} \rrbracket(\sigma, H)\ \textbf{in} \\
\quad \textbf{let}\ v \ = \ \mathcal{R}\llbracket \Gamma \vdash RE : \texttt{region} \rrbracket(\sigma, H) \textbf{in} & \quad\quad \textbf{let}\ v_2 \ = \ \mathcal{R}\llbracket \Gamma \vdash RE : \texttt{region} \rrbracket(\sigma, H)\ \textbf{in} \\
\quad\quad \textbf{if}\ v \neq \textit{err}\ \textbf{then}\ \{(o, f) | (o, f) \in v \wedge Type(o) = C\} \textbf{else}\ \ \textit{err} & \quad\quad\quad \textbf{if}\ v_1 \neq \textit{err} \wedge v_2 \neq \textit{err}\ \textbf{then}\ \ v_1\ \mathcal{MO}\llbracket \otimes \rrbracket\ v_2\ \textbf{else}\ \textit{err}
\end{array}$$

Figure 7: Semantics of Expressions. $\mathcal{N}$ is the standard meaning function for numeric literals. The function MO gives the semantics of operators [4].

$$\frac{\Gamma \vdash E_1 : T, \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 = E_2 : \texttt{\textbf{bool}}} \qquad \frac{\Gamma \vdash x.f : T, \qquad \Gamma \vdash E : T}{\Gamma \vdash x.f = E : \texttt{\textbf{bool}}} \qquad \frac{\Gamma \vdash RE_1 : \texttt{\textbf{region}}, \qquad \Gamma \vdash RE_2 : \texttt{\textbf{region}}}{\Gamma \vdash RE_1 \leqslant RE_2 : \texttt{\textbf{bool}}}$$

$$\frac{\Gamma \vdash P_1 : \texttt{\textbf{bool}}, \qquad \Gamma \vdash P_2 : \texttt{\textbf{bool}}}{\Gamma \vdash P_1 \;\&\&\; P_2 : \texttt{\textbf{bool}}} \qquad \frac{\Gamma \vdash P_1 : \texttt{\textbf{bool}}, \qquad \Gamma \vdash P_2 : \texttt{\textbf{bool}}}{\Gamma \vdash P_1 \,||\, P_2 : \texttt{\textbf{bool}}} \qquad \frac{\Gamma \vdash P : \texttt{\textbf{bool}}}{\Gamma \vdash \neg P : \texttt{\textbf{bool}}} \qquad \frac{\Gamma, x : \texttt{\textbf{int}} \vdash P : \texttt{\textbf{bool}}}{\Gamma \vdash \forall x : \texttt{\textbf{int}}.P : \texttt{\textbf{bool}}}$$

$$\frac{\Gamma \vdash RE : \texttt{\textbf{region}}, \qquad \Gamma, x : C \vdash P : \texttt{\textbf{bool}}}{\Gamma \vdash \forall (x, f) \in RE.P : \texttt{\textbf{bool}}} \textbf{ where } isClass(C) \qquad \frac{\Gamma, x : int \vdash P : \texttt{\textbf{bool}}}{\Gamma \vdash \exists x : \texttt{\textbf{int}}.P : \texttt{\textbf{bool}}}$$

$$\frac{\Gamma \vdash RE : \texttt{\textbf{region}}, \qquad \Gamma, x : C \vdash P : \texttt{\textbf{bool}}}{\Gamma \vdash \exists (x, f) \in RE.P : \texttt{\textbf{bool}}} \textbf{ where } isClass(C)$$

Figure 10: Typing rules for assertions. The predicate $isClass$ returns true just when C is a declared class name in the program.

$$\begin{array}{l}
\sigma, H \models^{\Gamma} E_1 = E_2 \iff \mathcal{E}[\![\Gamma \vdash E_1 : T]\!](\sigma, H) = \mathcal{E}[\![\Gamma \vdash E_2 : T]\!](\sigma, H) \\
\sigma, H \models^{\Gamma} x.f = E \iff \begin{cases} H[\sigma(x), f] = \mathcal{E}[\![\Gamma \vdash E : T]\!](\sigma, H) & if\ \sigma(x) \neq null\ and (\sigma(x), f) \in dom(H) \\ false & otherwise \end{cases} \\
\sigma, H \models^{\Gamma} RE_1 \leqslant RE_2 \iff \begin{cases} \mathcal{R}[\![\Gamma \vdash RE_1 : \texttt{\textbf{region}}]\!](\sigma, H) \subseteq \mathcal{R}[\![\Gamma \vdash RE_2 : \texttt{\textbf{region}}]\!](\sigma, H) \\ \qquad\qquad \text{if } \mathcal{R}[\![\Gamma \vdash RE_i : \texttt{\textbf{region}}]\!](\sigma, H) \neq err, \text{and } i = 1, 2 \\ false \qquad otherwise \end{cases} \\
\sigma, H \models^{\Gamma} P_1 \;\&\&\; P_2 \iff \sigma, H \models^{\Gamma} P_1\ and\ \sigma, H \models^{\Gamma} P_2 \\
\sigma, H \models^{\Gamma} P_1 \;||\; P_2 \iff \sigma, H \models^{\Gamma} P_1\ or\ \sigma, H \models^{\Gamma} P_2 \\
\sigma, H \models^{\Gamma} \neg P \iff \quad \sigma, H \not\models^{\Gamma} P \\
\sigma, H \models^{\Gamma} \forall x : \texttt{\textbf{int}}.\, P \iff forall\ v.(\sigma[x \mapsto v], H \models^{\Gamma, x:\texttt{\textbf{int}}} P) \\
\sigma, H \models^{\Gamma} \forall\, \texttt{\textbf{region}}\{x.f\} \subseteq RE.\, P \iff \begin{cases} forall\ o, C.((o, f) \in \mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma, H)\ and\ type(o) = C\ and \\ \qquad \sigma[x \mapsto o], H \models^{\Gamma, x:C} P) \quad \text{if } \mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma, H) \neq err \\ false \qquad\qquad otherwise \end{cases} \\
\sigma, H \models^{\Gamma} \exists x : \texttt{\textbf{int}}.P \iff exists\ v.\ \sigma[x \mapsto v], H \models^{\Gamma, x:\texttt{\textbf{int}}} P \\
\sigma, H \models^{\Gamma} \exists\, \texttt{\textbf{region}}\{x.f\} \subseteq RE.P \iff \begin{cases} exists\ o, C.((o, f) \in \mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma, H)\ and\ type(o) = C\ and \\ \qquad \sigma[x \mapsto o], H \models^{\Gamma, x:C} P) \quad \text{if } \mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma, H) \neq err \\ false \qquad\qquad otherwise \end{cases}
\end{array}$$

Figure 11: Semantics of Assertions

5.2 EFFECTS

Effects (ϵ) are used in frame conditions. The keyword **`modifies`** specifies write effects and **`reads`** specifies read effects. **`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.

ϵ ::= `(empty)` | ϵ`,`ϵ | **`if`** E **`then`** ϵ_1 **`else`** ϵ_2
| **`reads`** RE | **`reads`** x | **`modifies`** RE
| **`modifies`** x | **`fresh`**`(`RE`)`

The latter five forms are called atomic effects. We omit **`modifies`** and **`reads`** when the context is obvious. For example, in a modifies clause we write **`if`** E **`then`** RE **`else`** x instead of **`if`** E **`then modifies`** RE **`else modifies`** x. And **`if`** E **`then`** ϵ is an abbreviation of **`if`** E **`then`** ϵ **`else`** (). 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 in line 30 is: **`region`**$\{this.fpt\} \leqslant$ (**`if`** `Find(k)` = null **`then`** **`region`**$\{this.fpt\}$ **`else`** **`region`**$\{Find(k).val\}$) with the assumption `Find(k)` = null. We use the sub-effect rule [4] to reason about such cases. It encodes the standard properties of sets.

5.3 Framing

Let R be the region that the frame condition of a method, m, specifies in a given state. R contains the locations that may be modified in m. The locations that are preserved are the compliment of R, written $\bar{R}$. Let R' be locations that may be used in evaluating an assertion, P, written **`reads`**$R' frm P$. If $R' \subseteq \bar{R}$, i.e., $R'!!R$, then P's validity is preserved after m is called. We use *ftpt* in Fig. 14 to define R' for expressions, atomic region expressions, and atomic assertions. 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 state that satisfies P. Note that we use δ to denote reads effects and ϵ to denote write effects, and Γ is omitted when the type context is the same in the judgment. Fig. 15 shows the judgment for non-atomic region expressions and assertions.

Definition 1 says that if two states agree on a read effect, δ, then the values of the expressions that depend on δ are identical.

Definition 1. (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 and σ' be Γ'-state and Γ''-state respectively. σ and σ' *agree* on δ, $\sigma \overset{\delta}{\equiv} \sigma'$, if and only if:

1. for all **`reads`** $x \in \delta$, $\sigma(x) = \sigma'(x)$
2. for all **`reads region`**$\{x.f\} \leqslant \delta$, $\mathcal{E}[\![\Gamma \vdash x.f : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash x.f : T]\!](\sigma')$.

$$\vdash \texttt{\textbf{region}}\{\} \leqslant RE \qquad \vdash RE \leqslant RE \qquad \frac{P \vdash RE_1 \leqslant RE_2 \qquad P \vdash RE_2 \leqslant RE_3}{P \vdash RE_1 \leqslant RE_3} \qquad \frac{P' \Rightarrow P \qquad P \vdash RE_1 \leqslant RE_2}{P' \vdash RE_1 \leqslant RE_2}$$

$$\vdash RE_1 \leqslant RE_1 \cup RE_2 \qquad \vdash RE_1 - RE_2 \leqslant RE_1 \qquad \vdash RE_1 \cap RE_2 \leqslant RE_1 \qquad \vdash RE_1 \cap RE_2 \leqslant RE_2$$

$$\vdash \texttt{\textbf{region}}\{x.f\} \leqslant \texttt{\textbf{region}}\{x.*\} \qquad \frac{\vdash \Gamma(x) = C \qquad \vdash \texttt{\textbf{region}}\{x.f\} \leqslant RE}{\vdash \texttt{\textbf{region}}\{x.f\} \leqslant \texttt{\textbf{filter}}\{RE, C, f\}} \qquad \frac{\vdash \texttt{\textbf{region}}\{x.f\} \leqslant \texttt{\textbf{filter}}\{RE, C, f\}}{\vdash \texttt{\textbf{filter}}\{RE, C, f\} \leqslant \texttt{\textbf{region}}\{x.f\}}$$

$$\vdash \texttt{\textbf{filter}}\{RE, C, f\} \leqslant \texttt{\textbf{filter}}\{RE, C\} \qquad \frac{P \vdash RE_1 \leqslant RE_2}{P \vdash \texttt{\textbf{filter}}\{RE_1, C, f\} \leqslant \texttt{\textbf{filter}}\{RE_2, C, f\}} \qquad \vdash \texttt{\textbf{filter}}\{RE, C\} \leqslant RE$$

$$\frac{P \vdash RE_1 \leqslant RE_2}{P \vdash \texttt{\textbf{filter}}\{RE_1, C\} \leqslant \texttt{\textbf{filter}}\{RE_2, C\}}$$

Figure 12: Sub-region rules.

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

$$\frac{P \vdash \epsilon_1 \leqslant \epsilon_2 \qquad P \vdash \epsilon_2 \leqslant \epsilon_3}{P \vdash \epsilon_1 \leqslant \epsilon_3} \qquad \frac{P' \Rightarrow P \qquad P \vdash \epsilon_1 \leqslant \epsilon_2}{P' \vdash \epsilon_1 \leqslant \epsilon_2} \qquad \frac{P \vdash \epsilon_1 \leqslant \epsilon_2}{P \vdash \epsilon_1, \epsilon \leqslant \epsilon_2, \epsilon}$$

$$\vdash \texttt{\textbf{modifies}}\, RE_1, RE_2 \;\lesseqgtr\; \texttt{\textbf{modifies}}\, RE_1 + RE_2 \qquad \vdash \texttt{\textbf{reads}}\, RE_1, RE_2 \;\lesseqgtr\; \texttt{\textbf{reads}}\, RE_1 + RE_2$$

$$\vdash \texttt{\textbf{modifies}}\, \texttt{\textbf{filter}}\{RE, C, f\} \leqslant \texttt{\textbf{modifies}}\, RE \qquad \vdash \texttt{\textbf{modifies}}\, \texttt{\textbf{filter}}\{RE, C\} \leqslant \texttt{\textbf{modifies}}\, RE$$

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

$$\vdash \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2 \leqslant \epsilon_1, \epsilon_2 \qquad \frac{\begin{array}{ll} P \wedge E_1 \neq 0 \wedge E_2 \neq 0 \vdash \epsilon_1 \leqslant \epsilon_3 & P \wedge E_1 = 0 \wedge E_2 \neq 0 \vdash \epsilon_2 \leqslant \epsilon_3 \\ P \wedge E_1 \neq 0 \wedge E_2 = 0 \vdash \epsilon_1 \leqslant \epsilon_4 & P \wedge E_1 = 0 \wedge E_2 = 0 \vdash \epsilon_2 \leqslant \epsilon_4 \end{array}}{P \vdash \texttt{\textbf{if}}\, E_1\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2 \leqslant \texttt{\textbf{if}}\, E_2\, \texttt{\textbf{then}}\, \epsilon_3\, \texttt{\textbf{else}}\, \epsilon_4}$$

Figure 13: Subeffect rules. The sub-region rule is defined in Fig. 12.

Definition 2 says that if two states agree on a read effect, δ, then the validity of assertions, Q, that depend on that read effect are preserved under condition, P.

Definition 2. (Frame validity) $P \vdash^{\Gamma} \delta\ frm\ Q$ is valid, written $P \models^{\Gamma} \delta\ frm\ Q$, if and only if for all states σ, σ', if $\sigma \overset{\delta}{\equiv} \sigma'$ and $\sigma \models^{\Gamma} P \wedge Q$, then $\sigma' \models^{\Gamma} Q$.

Lemma 1. (frame soundness of expressions) Let σ and σ' be arbitrary states. Let E be an expression. If $\sigma \overset{ftpt(\Gamma, \mathrm{E})}{\equiv} \sigma'$, then $\mathcal{E}[\![\Gamma \vdash E : T]\!](\sigma) = \mathcal{E}[\![\Gamma \vdash E : T]\!](\sigma')$.

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

Lemma 2. (frame soundness of atomic region expressions) Let σ and σ' be arbitrary states. Let RE be an atomic region expression. If $\sigma \overset{ftpt(\Gamma, \mathrm{RE})}{\equiv} \sigma'$, then $\mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma) = \mathcal{R}[\![\Gamma \vdash RE : \texttt{\textbf{region}}]\!](\sigma')$

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 $P \vdash \delta frm Q$. The proof is similar to [3]. We leave it as future work. □

We use $\cdot/\cdot$ to define the disjointness on effects in Fig. 17. The region disjoint rules are defined in Fig. 16. We treat **reads** δ, where δ is not a conditional effect, as **reads if** $true$ **then** δ **else** . 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 is reduced to **reads region**$\{y.f\}$ $\cdot/\cdot$ **modifies region**$\{\}$.

Lemma 4. Let RE_1 and RE_2 be two regions. Let σ be a state. If $\sigma \models^{\Gamma} RE_1 \mathbin{!!} RE_2$, 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 5. (separator agreement) Let $\sigma' = \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma)]\!](CT)(\sigma)$. Let ϵ be the write effect of exe-

$$
\begin{array}{lcl}
\mathit{ftpt}(x) & = & \texttt{\textbf{reads}}\ x \\
\mathit{ftpt}(n) & = & \emptyset \\
\mathit{ftpt}(\mathit{null}) & = & \emptyset \\
\mathit{ftpt}(E_1 \oplus E_2) & = & \mathit{ftpt}(\Gamma, E_1), \mathit{ftpt}(\Gamma, E_2) \\
\mathit{ftpt}(\texttt{\textbf{region}}\{\}) & = & \emptyset \\
\mathit{ftpt}(\texttt{\textbf{region}}\{x.f\}) & = & \texttt{\textbf{reads}}\ x
\end{array}
\qquad
\begin{array}{lcl}
\mathit{ftpt}(\texttt{\textbf{region}}\{x.*\}) & = & \texttt{\textbf{reads}}\ x \\
\mathit{ftpt}(\texttt{\textbf{filter}}\{RE, f\}) & = & \mathit{ftpt}(RE) \\
\mathit{ftpt}(\texttt{\textbf{filter}}\{RE, C, f\}) & = & \mathit{ftpt}(RE) \\
\mathit{ftpt}(RE_1 \otimes RE_2) & = & \mathit{ftpt}(RE_1), \mathit{ftpt}(RE_2) \\
\mathit{ftpt}(E_1 = E_2) & = & \mathit{ftpt}(E_1), \mathit{ftpt}(E_2) \\
\mathit{ftpt}(x.f = E) & = & \texttt{\textbf{reads}}\ x, \texttt{\textbf{region}}\{x.f\}, \mathit{ftpt}(E) \\
\mathit{ftpt}(RE_1 \leqslant RE_2) & = & \mathit{ftpt}(RE_1), \mathit{ftpt}(RE_2)
\end{array}
$$

Figure 14: Footprint of expressions, atomic region expressions and atomic assertions

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

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

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

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

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

FRM$\forall_1$
$$\frac{P \vdash^{\Gamma, x:\mathit{int}} \delta, \texttt{\textbf{reads}}\ x\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ \forall\, x : \texttt{\textbf{int}}\ Q}$$

FRM$\forall_2$
$$\frac{P \vdash \texttt{\textbf{reads}}\, \mathit{ftpt}(\Gamma, RE) \leqslant \delta \qquad P \wedge \texttt{\textbf{reads}}\ \texttt{\textbf{region}}\{x.f\} \leqslant \delta \vdash^{\Gamma, x:C} \epsilon, \texttt{\textbf{reads}}\ x, \texttt{\textbf{region}}\{x.f\}\ \mathit{frm}\ Q}{P \vdash \delta\ \mathit{frm}\ \forall\, \texttt{\textbf{region}}\{x.f\} \subseteq RE.Q}$$

FRM$\exists_1$
$$\frac{P \vdash^{\Gamma, x:\mathit{int}} \delta, \texttt{\textbf{reads}}\ x\ \mathit{frm}\ Q}{P \vdash^{\Gamma} \delta\ \mathit{frm}\ \exists\, x : \texttt{\textbf{int}}\ Q}$$

FRM$\exists_2$
$$\frac{P \vdash \texttt{\textbf{reads}}\, \mathit{ftpt}(\Gamma, RE) \leqslant \delta \qquad P \wedge \texttt{\textbf{reads}}\ \texttt{\textbf{region}}\{x.f\} \leqslant \delta \vdash^{\Gamma, x:C} \delta, \texttt{\textbf{reads}}\ x, \texttt{\textbf{region}}\{x.f\}\ \mathit{frm}\ Q}{P \vdash \delta\ \mathit{frm}\ \exists\, \texttt{\textbf{region}}\{x.f\} \subseteq RE.Q}$$

Figure 15: Rules for the framing judgment.Γ is omitted when it is the same in the judgment.

$$\vdash \texttt{\textbf{region}}\{\}\ !!\ RE \qquad \frac{\vdash RE_1\ !!\ RE_2}{\vdash RE_2\ !!\ RE_1} \qquad \frac{\vdash f_x = f_y}{\vdash \texttt{\textbf{region}}\{x.f_x\}\ !!\ \texttt{\textbf{region}}\{y.f_y\}} \qquad \frac{P \vdash RE_1 \leqslant RE_2 \qquad P \vdash RE_2\ !!RE_3}{P \vdash RE_1\ !!\ RE_3}$$

$$\frac{P' \Rightarrow P \qquad P \vdash RE_1\ !!\ RE_2}{P' \vdash RE_1\ !!\ RE_2} \qquad \frac{\vdash RE_1\ !!\ RE_2}{\vdash \texttt{\textbf{filter}}\{RE_1, C_1, f_1\}\ !!\ \texttt{\textbf{filter}}\{RE_2, C_2, f_2\}}$$

$$\frac{\vdash C_1\ \neq\ C_2}{\vdash \texttt{\textbf{filter}}\{RE_1, C_1, f_1\}\ !!\ \texttt{\textbf{filter}}\{RE_2, C_2, f_2\}} \qquad \frac{\vdash f_1\ \neq\ f_2}{\vdash \texttt{\textbf{filter}}\{RE_1, C_1, f_1\}\ !!\ \texttt{\textbf{filter}}\{RE_2, C_2, f_2\}}$$

$$\frac{\vdash RE_1\ !!\ RE_2}{\vdash \texttt{\textbf{filter}}\{RE_1, C_1\}\ !!\ \texttt{\textbf{filter}}\{RE_2, C_2\}} \qquad \frac{\vdash C_1\ \neq\ C_2}{\vdash \texttt{\textbf{filter}}\{RE_1, C_1\}\ !!\ \texttt{\textbf{filter}}\{RE_2, C_2\}}$$

Figure 16: Disjoint region rules

$$
\begin{array}{ll}
\texttt{\textbf{reads}}\, RE_1 \mathbin{\cdot\!/\!\cdot} \texttt{\textbf{modifies}}\ RE_2 = RE_1\ !!\ RE_2 & \texttt{\textbf{reads}}\ y \mathbin{\cdot\!/\!\cdot} \texttt{\textbf{modifies}}\ x = y \not\equiv x \\
\delta \mathbin{\cdot\!/\!\cdot} \epsilon = \mathit{true} \text{ for all other pairs of atomic effects} & \delta \mathbin{\cdot\!/\!\cdot} \epsilon = \mathit{true} \text{ in case } \delta \text{ or } \epsilon \text{ is empty} \\
\delta \mathbin{\cdot\!/\!\cdot} (\epsilon, \epsilon') = (\delta \mathbin{\cdot\!/\!\cdot} \epsilon) \wedge (\delta \mathbin{\cdot\!/\!\cdot} \epsilon') & (\delta, \delta') \mathbin{\cdot\!/\!\cdot} \epsilon = (\delta \mathbin{\cdot\!/\!\cdot} \epsilon) \wedge (\delta' \mathbin{\cdot\!/\!\cdot} \epsilon)
\end{array}
$$

$$
\begin{array}{c}
\texttt{\textbf{if}}\ E\ \texttt{\textbf{then}}\ \delta_1\ \texttt{\textbf{else}}\ \delta_2 \\
\cdot\!/\!\cdot \\
\texttt{\textbf{if}}\ E'\ \texttt{\textbf{then}}\ \epsilon_1\ \texttt{\textbf{else}}\ \epsilon_2 \\
(\text{Let } P = E \neq 0, \text{and } P' = E' \neq 0)
\end{array}
=
\begin{cases}
\delta_1 \mathbin{\cdot\!/\!\cdot} \epsilon_1 & \text{if} \quad P \,\&\,\& \ P' \\
\delta_1 \mathbin{\cdot\!/\!\cdot} \epsilon_2 & \text{if} \quad P \,\&\,\& \ \neg P' \\
\delta_2 \mathbin{\cdot\!/\!\cdot} \epsilon_1 & \text{if} \quad \neg P \,\&\,\& \ P' \\
\delta_2 \mathbin{\cdot\!/\!\cdot} \epsilon_2 & \text{if} \quad \neg P \,\&\,\& \ \neg P'
\end{cases}
$$

Figure 17: Separator. δ is read effect and ϵ is write effect. The region disjoint rules are defined in the technique report [4]

```
$\mathcal{MS}$ : $\mathit{TypingJudgment} \rightarrow \mathit{ClassTable} \rightarrow \mathit{State}_\perp \rightarrow \mathit{State}_\perp$
$\mathcal{MS}[\![\Gamma \vdash (\mathtt{var}\, x : T;) : ok(\Gamma)]\!](CT)(\rho, \sigma, H) =$
  cases $T$ of
     $C \ \rightarrow \ (\rho[x \mapsto C], \sigma[x \mapsto initVal(C)], H)$
     else $(\rho, \sigma[x \mapsto initVal(C)], H)$
  end
$\mathcal{MS}[\![\Gamma \vdash (x := F;) : ok(\Gamma)]\!](\rho, \sigma, H) =$
  let $T \ = \ \mathit{typeOf}(\Gamma, x)$ in
    let $val \ = \ \mathcal{E}[\![\ \Gamma \ \vdash F : T]\!](\rho, H, \sigma)$ in
      if $val \neq \mathit{err}$ then $(\rho, \sigma[x \mapsto val], H)$ else $\perp$
$\mathcal{MS}[\![\Gamma \vdash (x.f := F;) : ok(\Gamma)]\!](\rho, \sigma, H) =$
  let $v_1 \ = \ \mathcal{E}[\![\Gamma \vdash x.f : T]\!]\rho, \sigma, H$ in
    let $v_2 \ = \ \mathcal{E}[\![\Gamma \vdash F : T]\!]\rho, \sigma, H$ in
      if $v_1 \neq \mathit{err} \wedge v_2 \neq \mathit{err}$ then $(\rho, \sigma, H[v_1 \mapsto v_2])$ else $\perp$
$\mathcal{MS}[\![\Gamma \vdash (x_1 := x_2.f;) : ok(\Gamma)]\!](\rho, \sigma, H) =$
  let $v_1 \ = \ \mathcal{E}[\![\Gamma \vdash x_1 : T]\!]\rho, \sigma, H$ in
    let $v_2 \ = \ \mathcal{E}[\![\Gamma \vdash x_2.f : T]\!]\rho, \sigma, H$ in
      if $v_1 \neq \mathit{err} \wedge v_2 \neq \mathit{err}$ then $(\rho, \sigma, H[v_1 \mapsto v_2])$ else $\perp$
$\mathcal{MS}[\![(x := \mathtt{new}\, C;)]\!]\rho, \sigma, H \ =$
  let $(l, H') \ = \ allocate(C, H)$ in
    let $(f_1, \ldots, f_n) \ = \ \mathit{fieldNames}(C)$ in
      let $\sigma' \ = \ \sigma[x \mapsto l]$ in
        $(\rho, \sigma', H'[(\sigma'(x), f_1) \mapsto 0, \ldots, (\sigma'(x), f_n) \mapsto 0])$
$\mathcal{MS}[\![\Gamma \vdash (\mathtt{if}\, E\, \mathtt{then}\{S_1\}\mathtt{else}\{S_2\};) : ok(\Gamma)]\!](\rho, \sigma, H) =$
  let $v \ = \ \mathcal{E}[\![\Gamma \vdash E : T]\!]\rho, \sigma, H$ in
    cases $v$ of
      $true \ \rightarrow \ \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma)]\!](\rho, \sigma, H)$
      $false \ \rightarrow \ \mathcal{MS}[\![\Gamma \vdash S_2 : ok(\Gamma)]\!](\rho, \sigma, H)$
      else $\perp$
    end
$\mathcal{MS}[\![\Gamma \vdash (\mathtt{while}\, E\, \{S\};: ok(\Gamma)]\!](\rho, \sigma, H) =$
  $\mathit{fix}(\lambda g \,.\, \underline{\lambda} s \,.$
    let $v \ = \ \mathcal{E}[\![\Gamma \vdash E : \mathtt{bool}]\!](\rho, \sigma, H)$ in
      cases $v$ of
        $true \ \rightarrow$ let $s' \ = \ \mathcal{MS}[\![\Gamma \vdash S : ok(\Gamma)]\!](\rho, \sigma, H)$ in $gs'$
        $false \ \rightarrow \ s$
        else $\perp$
      end$)(\rho, \sigma, H)$
$\mathcal{MS}[\![\Gamma \vdash (S_1 S_2) : ok(\Gamma)]\!](\rho, \sigma, H) =$
  let $s' \ = \ \mathcal{MS}[\![\Gamma \vdash S_1 : ok(\Gamma)]\!](\rho, \sigma, H)$ in
    if $s' \neq \perp$ then $\mathcal{MS}[\![\Gamma \vdash S_2 : ok(\Gamma)]\!](s')$ else $\perp$
```

Figure 8: The semantics of statements. The *allocate* function takes the heap and the class name as parameters, and returns a location and a new heap. We define a function *initVal* that takes the variable's type as a parameter and returns an arbitrary value of that type.

$$
\begin{array}{ll}
P \ ::= & E_1{=}E_2 \ \mid \ x.f{=}E \ \mid \ RE_1{\leqslant}RE_2 \mid \ P_1\,\&\,\&P_2 \ \mid \ P_1\,|\,|\,P_2 \\
& \mid \ \neg \ P \ \mid \ \forall \ x{:}\mathtt{int}.P \ \mid \ \forall \ \mathtt{region}\{x.f\} \subseteq \ RE.P \\
& \mid \ \exists x{:}\mathtt{int}.P \ \mid \ \exists \ \mathtt{region}\{x.f\} {\subseteq} RE.P
\end{array}
$$

Figure 9: The syntax of assertions

cuting S. Let δ be a read effects, such that $\sigma' \models \delta \cdot\!/\!\cdot \epsilon$, then $\sigma \overset{\delta}{\equiv} \sigma'$.

Proof. We leave it as future work. □

6. PROGRAM CORRECTNESS

The validity of a Hoare-formula $\{P\}S\{Q\}[\epsilon]$ 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 ϵ. For simplicity, we omit **`modifies`** when expressing the write effects. It means that if one can prove that a statement, S, can have different effects under the complementarity conditions, $E \neq 0$ and $E = 0$, then one can conclude a conditional effect for S. Examples of using CONEFF rules are in our technical report [4]. We derive the IF2 rule with the IF rule in the work of region logic [3]:

$$
\text{IF} \quad \frac{\vdash \{P \wedge E \neq 0\}\ S_1\ \{P'\}[\epsilon] \qquad \vdash \{P \wedge E = 0\}\ S_2\ \{P'\}[\epsilon]}{\vdash \{P\}\mathtt{if}\, E\, \mathtt{then}\, S_1\, \mathtt{else}\, S_2\{P'\}[\epsilon]}
$$

The IF rule can over-approximates write effects in the sense that ϵ is the union of the write effects of S_1 and S_2. The derived IF2 rule shows that if one can prove two branches with different effects, then one can prove the IF statement with conditional effects, which gives a more precise frame condition, because the write effects of IF statement is either those of S_1 or those of S_2 depending on the branch a program actually takes. To derive the rule IF2, we begin with $\vdash \ \{P \wedge E \neq 0\}\ S_1\ \{P'\}[\epsilon_1]$, and $\vdash \{P \wedge E = 0\}\ S_2\ \{P'\}[\epsilon_2]$. Using the sub-effect rule, we have $P \wedge E \neq 0 \vdash \epsilon_1 \leqslant \mathtt{if}\, E\, \mathtt{then}\, \epsilon_1\, \mathtt{else}\, \epsilon_2$ and $P \wedge E = 0 \vdash \epsilon_2 \leqslant \mathtt{if}\, E\, \mathtt{then}\, \epsilon_1\, \mathtt{else}\, \epsilon_2$. Then we can derive the IF2 rule by using the IF rule. Fig. 18 and Fig. 19 lists the axioms and structural rules.

Consider the example:

$$
\begin{array}{lcl}
S & \doteq & \mathtt{if}\, t\, \mathtt{then}\, x.f = 5\, \mathtt{else}\, y.f = 5 \\
P & \doteq & x \neq y \wedge x.f = 4 \wedge y.f = 4 \\
P' & \doteq & (t \neq 0 \Rightarrow x.f = 5) \wedge (t = 0 \Rightarrow y.f = 5) \\
\epsilon & \doteq & \mathtt{if}\, t\, \mathtt{then}\, \mathtt{region}\{x.f\}\, \mathtt{else}\, \mathtt{region}\{y.f\}
\end{array}
$$

We do forward reasoning:

```
1   {x ≠ y ∧ x.f = 4 ∧ y.f = 4 }
2       if  t
3       then  {
4   {t ≠ 0 ∧ x ≠ y ∧ x.f = 4 ∧ y.f = 4}
5         x.f := 5
6   {t ≠ 0 ∧ x ≠ y ∧ x.f = 5 ∧ y.f = 4} [region{x.f}]
7       } else {
8   {t = 0 ∧ x ≠ y ∧ x.f = 4 ∧ y.f = 4}
9         y.f := 5
10  {t = 0 ∧ x ≠ y ∧ x.f = 4 ∧ y.f = 5} [region{y.f}]
11      }
```

Formula in line 6 implies

$$
\begin{array}{c}
(t \neq 0 \wedge x \neq y \wedge x.f = 5 \wedge y.f = 4) \vee \\
(t = 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 5)
\end{array}
\tag{1}
$$

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

$$\vdash \{t \neq 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 4\}x.f := 5\{P'\} [\texttt{\textbf{region}}\{x.f\}] \quad (2)$$

$$\vdash \{t = 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 4\}y.f := 5\{P'\} [\texttt{\textbf{region}}\{y.f\}] \quad (3)$$

Consider the formula (2). Since

$$\vdash \{t = 0 \wedge t \neq 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 4\}y.f := 5\{P'\} [\texttt{\textbf{region}}\{x.f\}] \quad (4)$$

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

$$\vdash \{t \neq 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 4\}S\{P'\} [\texttt{\textbf{region}}\{x.f\}] \quad (5)$$

Similarly, we can derive (3) to

$$\vdash \{t = 0 \wedge x \neq y \wedge x.f = 4 \wedge y.f = 4\}S\{P'\} [\texttt{\textbf{region}}\{y.f\}] \quad (6)$$

Using the rule CONEFF, we can prove $\{P\}S\{P'\}[\epsilon]$.

We do backward reasoning:

Approach 1 with the rule IF:

Using the rule IF, we need to prove

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0\}x.f := 5\{P'\}[\epsilon] \quad (7)$$

and

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t = 0\}y.f = 5\{P'\}[\epsilon] \quad (8)$$

Consider the formula (7), since $x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0 \Rightarrow t \neq 0$, using the rule CONMASK1, the proof obligation can be discharged to:

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0\}x.f := 5\{P'\} [\texttt{\textbf{region}}\{x.f\}] \quad (9)$$

Let Q be $x \neq y \wedge y.f = 4 \wedge t \neq 0$ that is the frame. Let δ be **reads** $x, y, y.f, t$. Since

$$x.f = 4 \vdash \delta\, frm\, Q \quad (10)$$

and

$$x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0 \Rightarrow \delta \cdot\!/\!\cdot \texttt{\textbf{modifies region}}\{x.f\} \quad (11)$$

Using the rule FRAME, the proof discharged to

$$\vdash \{x.f = 4\}x.f := 5\{x.f = 5\}[\texttt{\textbf{region}}\{x.f\}] \quad (12)$$

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

Approach 2 with the rule CONEFF:

Usint the rule CONEFF, we need to prove

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0\}S\{P'\}[\texttt{\textbf{region}}\{x.f\}] \quad (13)$$

and

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t = 0\}S\{P'\}[\texttt{\textbf{region}}\{y.f\}] \quad (14)$$

Consider the proof of the formula (13). Using the rule IF, one need to prove

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t \neq 0 \wedge t \neq 0\}x.f := 5\{P'\} [\texttt{\textbf{region}}\{x.f\}] \quad (15)$$

and

$$\vdash \{x \neq y \wedge x.f = 4 \wedge y.f = 4 \wedge t = 0 \wedge t \neq 0\}y.f := 5\{P'\} [\texttt{\textbf{region}}\{y.f\}] \quad (16)$$

The formula (16) is vacuously true. The other formula can be proved by the rules FRAME and FIELDUPD. Consider the example in Fig. 1; the execution of S_1 causes the write effect, **region**`{this.fpt}`, and the execution of S_2 causes the write effect, **region**`{Find(key).val}`. So we can prove that the frame condition is **if** `Find(key) = null` **then region{this.fpt} else region**`{Find(key).val}`. Moreover, a more precise frame condition not only means fewer locations that may be modified in some states, but also more locations are not changed and the validity of more assertions can be preserved. Consider the FRAME [6] rule in Fig. 18. Let Q be $o.data = 5 \wedge o.valid = false$, and ϵ be **modifies if** $o.valid$ **then region**$\{o.data\}$ **else region**$\{\}$, which is reduced to **modifies region**$\{\}$ by the similar analysis in section 4.1. δ = **reads** $o, o.data, o.valid$, which frames Q. So $\delta \cdot\!/\!\cdot$ **modifies region**$\{\}$ by the definition of separator in Fig. 17. Therefore we can prove Q is the frame. If ϵ approximates to just **region**$\{o.data\}$ without conditions, we cannot prove Q is the frame.

We introduce two rules CONMASK1 and CONMASK2 that drop conditional frames. The rule CONMASK1 says if **if** E **then** RE_1 **else** RE_2 is in the frame condition, and $E \neq 0$ is true in the precondition, then it is sound to simplify the conditional frames to RE_1. The rule CONMASK2 is similar. One can always drop conditional frames by appying the CONMASK1 and CONMASK2 rules before applying the FRAME rule. Therefore, we can keep the FRAME rule unchanged.

We also revised the rules POSTTOFR and FRTOPOST . They manipulate conditional freshness effects. Note that the un-conditional freshness effects, **fresh**(RE), can be considered as **if** $true$ **then fresh**(RE) **else region**$\{\}$. The special variable, $alloc$, has type **region**, and contains all allocated locations in the heap. The keyword **old** denotes values in the pre-state. If one can prove that the postcondition of a statement, S, implies that all fresh objects may be allocated in region RE_1 or RE_2 according to a complementarity conditions, $E \neq 0$ and $E = 0$. Then it is sound to add the effect **if** E **then fresh**(RE_1) **else fresh** (RE_2) to S's effect. The rule FRTOPOST is the reverse of the rule POSTTOFR. If **if** E **then fresh**(RE_1) **else fresh**(RE_2) is the fresh effect, and **reads** $old(alloc)$ separates ϵ in the post-condition P', then it is sound to conjoin $old(E) \neq 0 \Rightarrow RE_1 \,!!\, old(alloc)$ and $old(E) = 0 \Rightarrow RE_2 \,!!\, old(alloc)$ to the post-condition P'. The rules VARMASK1, VARMASK2, FIELDMASK1 and FIELDMASK2 drop write effects. Consider of the example of proving $\{true\}$ **if**

[6] The frame rule is the same as the work [3]

$true$ **`then`** $x{:=}\ x\{true\}$. Using the rule ASSIGN [4], CONSEQ [4] and the rule IF, one can derive:

$$\{true\}\ \mathtt{\textbf{if}}\ true\ \mathtt{\textbf{then}}\ x := x\ \{true\}[\mathtt{\textbf{if}}\ true\ \mathtt{\textbf{then}}\ x]. \quad (17)$$

The variable y is not written by the statement. And in the pre-state and post-state, x is equal to the value of y that is not changed. So, $P \vee P' \Rightarrow x = y$ is valid. And Therefore, using the rule VARMASK1, one can derive $\{true\}$ **`if`** $true$ **`then`** $x{:=}\ x\{true\}$ from (17). Similarly, FIELDMASK1 and FIELDMASK2 are used to prove, e.g., $\{x.f = 0\}$ **`if`** $true$ **`then`** $x.f{:=}0\{true\}[]$.

Definition 3 defines the validity of correctness judgment.

Definition 3. (Validity) The judgment $\Gamma \vdash \{P\}\ S\ \{Q\}[\epsilon]$ is valid, written $\sigma \models^{\Gamma} \{P\}\ S\ \{Q\}[\epsilon]$, if and only if: if $\sigma \models^{\Gamma} P$, $\sigma' = \mathcal{MS}[\![\Gamma \vdash (S) : ok(\Gamma)]\!](\sigma)$, and $\sigma' \neq \bot$, then $\sigma' \models^{\Gamma} Q$ and the effect from σ to σ' is covered by ϵ.

Theorem 1. Every derivable correctness judgment is valid.

7. EXAMPLE REVISITED

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

8. ACKNOWLEDGMENTS

Thanks to David Naumann for discussions about region logic. Thanks to Jośe Sánchez's comments for the draft.

9. REFERENCES

[1] A. Banerjee and D. Naumann. A logical analysis of framing for specifications with pure method calls. In D. Giannakopoulou and D. Kroening, editors, *Verified Software: Theories, Tools and Experiments*, Lecture Notes in Computer Science, pages 3–20. Springer International Publishing, 2014.

[2] 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.

[3] A. Banerjee, D. A. Naumann, and S. Rosenberg. Local reasoning for global invariants, part i: Region logic. *J. ACM*, 60(3):18:1–18:56, June 2013.

[4] Y. Bao, G. T. Leavens, and G. Ernst. Conditional framing in fine-grained region logic. Technical Report CS-TR-15-01, Computer Science, University of Central Florida, Orlando, Florida, Mar. 2015. `http://www.eecs.ucf.edu/~leavens/tech-reports/UCF/CS-TR-15-01/TR.pdf`.

[5] 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.

[6] 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.

[7] 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.

[8] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.

[9] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[10] 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.

[11] S. Rosenberg. Verifier for region logic. Web page at http://www.cs.stevens.edu/ naumann/pub/VERL/., 2011.

[12] A. Wills. Specification in Fresco. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[13] J. M. Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

ALLOC

$$\frac{Fields(C) = \overline{f} : \overline{T}}{\vdash \{true\}\; x := \texttt{\textbf{new}}\; C\{x.\overline{f} = default(\overline{T})\}[x, alloc, \texttt{\textbf{fresh}}(\texttt{\textbf{region}}\{x.*\})]}$$

ASSIGN

$$\frac{x' \not\equiv x}{\vdash \{x = x'\}\; x := E\; \{x = (E/x \mapsto x')\}[x]}$$

FIELDACC

$$\frac{x_1 \not\equiv x}{\{x_2 \neq null \wedge x_1 = x_2\}\; x := x_2.f\; \{x = x_1.f\}[x]}$$

FIELDUPD

$$\{x \neq null \wedge x' = E\}\; x.f := E\; \{x.f = x'\}[\texttt{\textbf{region}}\{x.f\}]$$

SEQ

$$\frac{\begin{array}{c}\{P\}\; S_1\; \{P_1\}[\epsilon_1, \texttt{\textbf{fresh}}(RE)] \\ \{P_1\}\; S_2\; \{P'\}[\epsilon_2, RE'] \qquad RE' <= RE \qquad \epsilon_1 \text{ is fresh-free} \qquad P \vdash \epsilon_2 \cdot\!/\!\cdot \epsilon_1 \qquad P_1 \vdash RE \cdot\!/\!\cdot (\epsilon_2, RE')\end{array}}{\vdash \{P\}\; S_1 S_2\; \{P'\}[\, RE_1, RE_2, \texttt{\textbf{fresh}}(RE)]}$$

IF

$$\frac{\vdash \{P \wedge E \neq 0\}\; S_1\; \{P'\}[\epsilon_1] \qquad \vdash \{P \wedge E = 0\}\; S_2\; \{P'\}[\epsilon_2]}{\vdash \{P\}\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, S_1\, \texttt{\textbf{else}}\, S_2\{P'\}[\texttt{\textbf{if}}\, E \neq 0\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2]}$$

IF2

$$\frac{\vdash \{P \wedge E \neq 0\}\; S_1\; \{P'\}[\epsilon_1] \qquad \vdash \{P \wedge E = 0\}\; S_2\; \{P'\}[\epsilon_2]}{\vdash \{P\}\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, S_1\, \texttt{\textbf{else}}\, S_2\{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2]}$$

WHILE

$$\frac{\vdash \{I \wedge E \neq 0\}\; S\; \{I\}[\epsilon, RE] \qquad \epsilon \text{ is fresh-free} \qquad I \Rightarrow RE\; !!\; old(alloc) \qquad P \vdash \epsilon \cdot\!/\!\cdot \epsilon \qquad \texttt{\textbf{modifies}}\; alloc \notin \epsilon}{\vdash \{I\}\; while\, E\, do\, S\{I \wedge E = 0\}[\epsilon]}$$

Figure 18: axioms. $alloc$ is a special variable containing the set of locations. The keyword **`modifies`** is omitted when there is no confusion.

FRAME

$$\frac{\vdash \{P\}\, S\, \{P'\}[\epsilon] \qquad P \vdash \delta\, \mathit{frm}\, Q \qquad P \land Q \Rightarrow \delta \cdot\!/\!\cdot \epsilon}{\vdash \{P \land Q\}\, S\, \{P' \land Q\}[\epsilon]}$$

SUBEFF

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

CONSEQ

$$\frac{\vdash \{P_1\}\, S\, \{P_1'\}[\epsilon] \qquad P_2 \Rightarrow P_1 \qquad P_1' \Rightarrow P_2'}{\vdash \{P_2\}\, S\{P_2'\}[\epsilon]}$$

CONEFF

$$\frac{\vdash \{P \land E \neq 0\}\, S\, \{P'\}[\epsilon_1] \qquad \vdash \{P \land E = 0\}\, S\, \{P'\}[\epsilon_2]}{\vdash \{P\}\, S\, \{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2]}$$

CONMASK1

$$\frac{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2] \qquad P \Rightarrow E \neq 0}{\vdash \{P\}\, S\, \{P'\}[\epsilon, \epsilon_1]}$$

CONMASK2

$$\frac{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2] \qquad P \Rightarrow E = 0}{\vdash \{P\}\, S\, \{P'\}[\epsilon, \epsilon_2]}$$

POSTTOFR

$$\frac{\vdash \{P\}\, S\, \{P'\}[\epsilon] \qquad P \Rightarrow (E \neq 0 \land \mathit{RE}_1 \,!!\, \mathit{old}(\mathit{alloc})) \qquad P \Rightarrow (E = 0 \land \mathit{RE}_2 \,!!\, \mathit{old}(\mathit{alloc}))}{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \texttt{\textbf{fresh}}(RE_1)\, \texttt{\textbf{else}}\, \texttt{\textbf{fresh}}(RE_2)]}$$

FRTOPOST

$$\frac{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \texttt{\textbf{fresh}}(RE_1)\, \texttt{\textbf{else}}\, \texttt{\textbf{fresh}}(RE_2)]}{\begin{array}{c}\vdash \{P\}\, S\, \{P' \land (\mathit{old}(E) \neq 0 \Rightarrow RE_1!!\mathit{old}(\mathit{alloc})) \land \qquad (\mathit{old}(E) = 0 \Rightarrow RE_2!!\mathit{old}(\mathit{alloc}))\} \\ [\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \texttt{\textbf{fresh}}(RE_1)\, \texttt{\textbf{else}}\, \texttt{\textbf{fresh}}(RE_2)]\end{array}}$$

VARMASK1

$$\frac{\vdash \{P\}\, S\, \{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, x, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2 \epsilon] \qquad P \Rightarrow E \neq 0 \qquad P \lor P' \Rightarrow x = y \qquad P \land \mathit{old}(E) \neq 0 \vdash \texttt{\textbf{reads}}\, y \cdot\!/\!\cdot (x, \epsilon)}{\vdash \{P\}\, S\, \{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2, \epsilon]}$$

VARMASK2

$$\frac{\vdash \{P\}\, S\, \{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, x, \epsilon_2, \epsilon] \qquad P \Rightarrow E = 0 \qquad P \lor P' \Rightarrow x = y \qquad P \land \mathit{old}(E) = 0 \vdash \texttt{\textbf{reads}}\, y \cdot\!/\!\cdot (x, \epsilon)}{\vdash \{P\}\, S\, \{P'\}[\texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2, \epsilon]}$$

FIELDMASK1

$$\frac{\begin{array}{c}\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \texttt{\textbf{region}}\{x.f\}, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2] \qquad P \Rightarrow E \neq 0 \\ P \lor P' \Rightarrow x.f = y \qquad P' \land \mathit{old}(E) \neq 0 \vdash \texttt{\textbf{reads}}\, x \cdot\!/\!\cdot\, \texttt{\textbf{modifies}}\, \epsilon \qquad P' \land \mathit{old}(E) \neq 0 \vdash \texttt{\textbf{reads}}\, y \cdot\!/\!\cdot \texttt{\textbf{modifies}}\, \epsilon\end{array}}{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2]}$$

FIELDMASK2

$$\frac{\begin{array}{c}\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \texttt{\textbf{region}}\{x.f\}, \epsilon_2] \qquad P \Rightarrow E = 0 \\ P \lor P' \Rightarrow x.f = y \qquad P' \land \mathit{old}(E) = 0 \vdash \texttt{\textbf{reads}}\, x \cdot\!/\!\cdot\, \texttt{\textbf{modifies}}\, \epsilon \qquad P' \land \mathit{old}(E) = 0 \vdash \texttt{\textbf{reads}}\, y \cdot\!/\!\cdot \texttt{\textbf{modifies}}\, \epsilon\end{array}}{\vdash \{P\}\, S\, \{P'\}[\epsilon, \texttt{\textbf{if}}\, E\, \texttt{\textbf{then}}\, \epsilon_1\, \texttt{\textbf{else}}\, \epsilon_2]}$$

Figure 19: Structural rules. $alloc$ is a special variable containing the set of locations.