A Formal Soundness Proof of Region-based Memory Management for Object-Oriented Paradigm

Florin Craciun[1], Shengchao Qin[1], and Wei-Ngan Chin[2]

[1] Department of Computer Science, Durham University, UK
{florin.craciun,shengchao.qin}@durham.ac.uk
[2] Department of Computer Science, National University of Singapore, Singapore
chinwn@comp.nus.edu.sg

Abstract. Region-based memory management has been proposed as a viable alternative to garbage collection for real-time applications and embedded software. In our previous work we have developed a region type inference algorithm that provides an automatic compile-time region-based memory management for object-oriented paradigm. In this work we present a formal soundness proof of the region type system that is the target of our region inference. More precisely, we prove that the object-oriented programs accepted by our region type system achieve region-based memory management in a safe way. That means, the regions follow a stack-of-regions discipline and regions deallocation never create dangling references in the store and on the program stack. Our contribution is to provide a simple syntactic proof that is based on induction and follows the standard steps of a type safety proof. In contrast the previous safety proofs provided for other region type systems employ quite elaborate techniques.

1 Introduction

Modern object-oriented programming languages provide a run-time system that automatically reclaims memory using tracing garbage collection [23]. A correct garbage collector can guarantee that the memory is not collecting too early, and also that all memory is eventually reclaimed if the program terminates. However the space and time requirements of garbage-collected programs are very difficult to estimate in practice. Therefore many different solutions have been proposed for real-time applications and embedded software running on resource-limited platforms. These solutions either completely omit the use of garbage collectors (e.g. JavaCard platform), or use real-time garbage collectors [1], or use region-based memory management (e.g. Real-Time Specification for Java (RTSJ) [3]).

Region-based memory management systems allocate each new object into a program-specified *region*, with the entire set of objects in each region deallocated simultaneously when the region is deleted. Various studies have shown that region-based memory management can provide memory management with good real-time performance. Individual object deallocation is accurate but time unpredictable, while region deletion presents a better temporal behavior, at the cost of some space overhead. Data locality may also improve when related objects are placed together in the same region. Classifying objects into regions based on their lifetimes may deliver better memory utilization if regions are deleted in a timely manner.

The first safe region-based memory system was introduced by Tofte and Talpin [21, 22] for a functional language. Using a region type inference system, they have provided an automatic static region-based memory management for Standard ML. More precisely, their compiler can group heap allocations into regions and it can statically determine the program points where it is safe to deallocate the regions. Later, several projects have investigated the use of region-based memory management for C-like languages (e.g. Cyclone [12]) and object-oriented languages [9, 5]. These projects provide region type checkers and require programmers to annotate their programs with region declarations. The type checkers then use these declarations to verify that well-typed programs safely use the region-based memory.

In our previous work [8], we have developed the first automatic region type inference system for object-oriented paradigm. Our compiler automatically augments unannotated object-oriented programs with regions type declarations and inserts region allocation/deallocation instructions that achieve a safe memory management. In this paper we provide the safety proof of our region type system that is the target of our previous region inference algorithm.

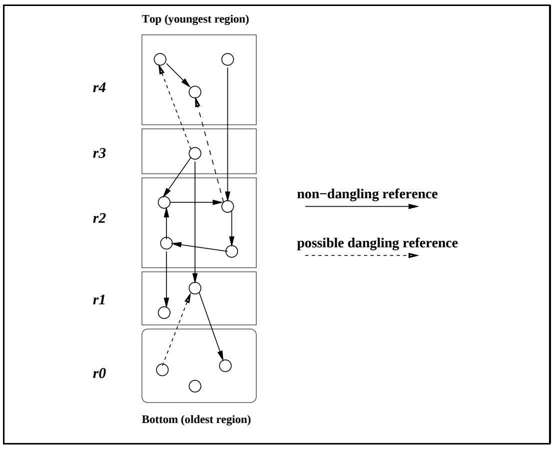

Fig. 1. Lexically-Scoped Regions

In our work, we use *lexically scoped region* such that the memory is organised as a *stack of regions*, as illustrated in Fig. 1. Regions are memory blocks that are allocated and deallocated by the construct `letreg r in e`, where the region `r` can only be used to allocate objects in the program `e`. The older regions (with longer lifetime) are allocated at the bottom of the stack while the younger regions (with shorter lifetime) are at the top. The region lifetime relations are expressed using a transitive *outlive relation*, denoted by $\succeq$. Thus, we can define the lifetime constraints $\texttt{r0}\succeq\texttt{r1}\wedge\texttt{r1}\succeq\texttt{r2}\wedge\texttt{r2}\succeq\texttt{r3}\wedge\texttt{r3}\succeq\texttt{r4}$ on the regions of Fig. 1. Region lifetime constraints (as shown in Fig. 2) are of two main forms $r_1\succeq r_2$ and $r_1{=}r_2$. The constraint $r_1\succeq r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2, while the constraint $r_1{=}r_2$ denotes that r_1 and r_2 must be the same region. The equality can be expressed as an outlive relation such that $r_1{=}r_2$ iff $r_1\succeq r_2$ and $r_2\succeq r_1$.

Dangling references are a safety issue for region-based memory management. Fig. 1 shows two kinds of references: non-dangling references and possible dangling references. Non-dangling references originate from objects placed in a younger region and point to objects placed either in an older region or inside the same region. Possible dangling references occur when objects placed in an older region point to objects placed in a younger region. They turn into dangling references when the younger region is deallocated. Using a dangling reference to access memory is unsafe because the accessed memory may have been recycled to store other objects. There are two approaches to eliminating this problem. The first approach allows the program to create dangling references, but uses an effect-based region type system to ensure that the program never accesses memory through a dangling reference [21, 22, 9, 12]. The second approach uses a region type system to prevent the program from creating dangling references at all [5]. Our work has adopted the second approach.

Contributions. The main contribution of this paper is the soundness proof of our region type system for object-oriented paradigm. We prove that our region type system guarantees that well-typed programs use lexically-scoped regions and never create dangling references in the store and on the program stack. We provide a simple syntactic proof based on induction, that follows the standard steps of a type safety proof [24]. Our small-step dynamic semantics decomposes high-level expression `letreg r in e` into three intermediate operations: allocation of region `r` on the stack, evaluation of program `e`, and deallocation of region `r`. The difficulty is to prove that after deallocation of region `r`, there is not any reference to `r` (and to the objects stored in `r`) in the store, on the program stack and in the remaining code. To prove that region deallocation is safe, we use the region constraints of our type system and a syntactic condition that we imposed to restrict the valid intermediate code. However our syntactic restriction does not restrict high-level source code, it only defines the correct intermediate code to which high-level code can be evaluated.

Related Work. In original effect-based region type system, Tofte and Talpin [22, 20, 2] and later Christiansen and Velschow [9], in their region calculus for object-oriented languages make use of co-induction to prove the soundness. Their proof requires co-induction partly because they prove two properties at the same time: type soundness and translation soundness. The latter property guarantees that there exists a semantic relation between source program and its region-annotated counterpart. Our safety theorems are only focused on the problem of type soundness, thus are simpler to prove. A co-inductive definition is required in their proof also because they use a big-step semantics where certain information is lost when deleting a region from the store, as discussed in [14, 7]. Our system uses a small-step operational semantics instrumented with regions which makes the consistency definition and the proof easier. Calcagno [6] uses a stratified operational semantics to avoid co-induction in the proof of safety properties of a simple version of Tofte and Talpin's region calculus, while Helsen et al. [14, 13] introduces a special constant for defunct regions in their big-step semantics which makes the soundness proof simpler. A similar proof with ours is the safety proof of Niss [18], that in addition to a simple functional language handles an imperative calculus, and like our proof avoids explicit co-induction by using store typing. Cyclone [12] also has an effect system used for a soundness proof and does not use co-induction. Elsman [11] refines Tofte and Talpin's region type system in order to forbid the dangling references and proves by induction the safety for a small functional language. There are many differences between his proof and ours. His proof is based on a small-step contextual semantics [16], while in our proof we explicitly model the heap as a stack of regions

$$
\begin{array}{lll}
t & ::= \; cn\langle r^{+}\rangle \;\mid\; prim\langle\rangle \;\mid\; \bot & (\textbf{region types}) \\
prim & ::= \textbf{int} \mid \textbf{boolean} \mid \textbf{void} & \\
\varphi & ::= r_1 \succeq r_2 \mid r_1 = r_2 \mid \mathit{true} \mid \varphi_1 \wedge \varphi_2 & (\textbf{region constraints}) \\
P & ::= \mathit{def}^{*} & (\textbf{region annotated program}) \\
\mathit{def} & ::= \textbf{class}\; cn_1\langle r^{+}\rangle \;\textbf{extends}\; cn_2\langle r^{+}\rangle \;\textbf{where}\; \varphi & \\
 & \qquad\qquad \{(t\,f)^{*}\; meth^{*}\} & (\textbf{region annotated class declaration}) \\
meth & ::= t\; mn\langle r^{*}\rangle\textbf{(}(t\,v)^{*}\textbf{)}\;\textbf{where}\;\varphi\;\{e\} & (\textbf{region annotated method}) \\
e & ::= \textbf{null} \mid k \mid lhs \mid lhs = e \mid e_1 \;\textbf{;}\; e_2 & (\textbf{region annotated expression}) \\
 & \mid \{(t\,v)\; e\} \mid \textbf{new}\; cn\langle r^{+}\rangle\textbf{(}v^{*}\textbf{)} \mid v.mn\langle r^{*}\rangle\textbf{(}v^{*}\textbf{)} & \\
 & \mid \textbf{if}\; v\; \textbf{then}\; e_1\; \textbf{else}\; e_2 \mid \textbf{while}\; v\; e & \\
 & \mid \textbf{letreg}\; r\; \textbf{in}\; e & (\textbf{region declaration}) \\
 & cn \in \mathit{class\ names} & r \in \mathit{region\ variable\ names} \\
 & mn \in \mathit{method\ names} & k \in \mathit{integer\ or\ boolean\ constants} \\
 & f \in \mathit{field\ names} & v \in \mathit{variable\ names}
\end{array}
$$

Fig. 2. The Syntax of Region-annotated Core-Java

and we use a consistency relation between the static and dynamic semantics. In addition Elsman uses a syntax-directed containment relation to express the regions of the program values and also to force the stack discipline for regions'allocation and deallocation. In our case the region requirements and the order among regions are expressed by the region constraints of the type system. However we also impose a syntactic condition to restrict the valid intermediate (non-source) programs. Boudol [4] refines Tofte and Talpin's region calculus to a flow-sensitive effect-based region type system, that explicitly records the deallocations effects. He provides a simple proof for a functional language by means of a subject reduction property up to simulation. Although his simulation is half-bisimulation, his proof does not employ co-induction. In contrast our region type system is a flow-insensitive calculus. However our syntactic restriction on intermediate code has a similar role as the flow-sensitive deallocation effect. Our type system is similar to SafeJava' type system of Boyapati et al. [5], but in addition we support region subtyping principle [12]. However SafeJava does not provide a formal proof for its region type system.

Outline. The paper is organized as follows. In Section 2 we introduce the syntax of our region calculus. Section 3 presents our region type system, while Section 4 defines the dynamic semantics of our region calculus. In Section 5, we derive the soundness proof. A brief conclusion is given. For lack of space, most of the technical details of our proofs are in Appendix A and Appendix B.

2 Region Calculus

Our region calculus is designed by annotating with regions a Java-like object-oriented language, named Core-Java [10]. The full syntax of the region annotated Core-Java language is given in Fig. 2. Core-Java is designed in the same minimalist spirit as the pure functional calculus Featherweight Java [15]. Despite its expression-oriented syntax, Core-Java supports imperative features.

Each class definition is parameterized with one or more regions to form a *region type*. For instance, a region type *cn*$\langle r_1, ..., r_n\rangle$ is a class name cn annotated with region parameters $r_1...r_n$. Parameterization allows us to obtain a region-polymorphic type for each class whose fields can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which the instance object of this class is allocated. The fields of the objects, if any, are allocated in the other regions $r_2...r_n$ which should *outlive* the region of the object. This is expressed by the constraint $\bigwedge_{i=2}^{n}(r_i \succeq r_1)$, which captures the property that the regions of the fields (in $r_2...r_n$) should have lifetimes no shorter than the lifetime of the region (namely r_1) of the object that refers to them. This condition, called *no-dangling requirement*, prevents dangling references completely, as it guarantees that each object never references another object in a younger region. In general the class invariant, φ, of a class consists of the no-dangling requirement for the region type of the current class, the no-dangling requirements for the fields' region types, and the class invariant of the parent class We do not require region parameters for primitive types, since primitive values can be copied and stored directly on the stack or they are part of an object. In order to keep the same notation, we use *prim*$\langle\rangle$ to denote a region annotated primitive type. Although null values are of object type, they are regarded as primitive values. The type of a null value is denoted by $\perp$.

$$\frac{\begin{array}{l}\textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..m}\rangle .. \in P' \\ n{\geq}m{\geq}p \quad \vdash \mathit{cn}'\langle x_{1..m}\rangle{<:}\mathit{cn}''\langle x'_{1..p}\rangle,\ \varphi\end{array}}{\vdash \mathit{cn}\langle x_{1..n}\rangle{<:}\mathit{cn}''\langle x'_{1..p}\rangle,\ \varphi}\,[\underline{\textbf{SubClass}}] \qquad \frac{}{\vdash\perp{<:}\mathit{cn}\langle x_{1..n}\rangle,\ \mathit{true}}\,[\underline{\textbf{Null}}] \qquad \frac{\varphi{=}(x_1{\succeq}\hat{x}_1)\wedge\bigwedge_{i=2}^{n}(x_i{=}\hat{x}_i)}{\vdash \mathit{cn}\langle x_{1..n}\rangle{<:}\mathit{cn}\langle\hat{x}_{1..n}\rangle,\ \varphi}\,[\underline{\textbf{RegSub}}]$$

Fig. 3. Region Subtyping Rules

The *region subtyping principle* allows an object from a region with longer lifetime to be assigned to a location where a region with a shorter lifetime is expected. This principle is illustrated by the subtyping rule [**RegSub**] of Fig. 3. This rule relies on the fact that once an object is allocated in a particular region, it stays within the same region and never migrates to another region. This property allows us to apply covariant subtyping to the region of the current object. However, the object fields are mutable (in general) and must therefore use invariant subtyping to ensure the soundness of subsumption. The other two rules, [**SubClass**] and [**Null**] from Fig. 3 denote the class subtyping and the fact that a null value can be assigned to any object, respectively.

Every method is decorated with zero or more region parameters; these parameters capture the regions used by each method's parameters (including `this`) and result. For simplicity, no other externally defined regions are made available for a method. Thus, all regions used in a method either are mapped to these region parameters or are localised by `letreg` in the method body. Each method also has a method precondition, φ expressed as a region lifetime constraint that is consistent with the operations performed in the method body. The method precondition also contains the class invariants of its parameters including the receiver and its result. The instance methods of a subclass can override the instance methods of the superclass.

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1 {
   Object⟨r2⟩ fst;
   Object⟨r3⟩ snd;

void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o) where r4⪰r3∧r2⪰r1∧r3⪰r1
     {snd=o;}
void swap⟨r1,r2,r3⟩() where r2=r3∧r2⪰r1
    { Object⟨r2⟩ tmp=fst; fst=snd; snd=tmp; }
Pair⟨r5,r6,r7⟩ exalloc⟨r1,r2,r3,r5,r6,r7⟩()
    where r7⪰r5∧r6⪰r5∧r2⪰r1∧r3⪰r1
  {letreg r in {
    Pair⟨r7,r7,r7⟩ p4;
    Pair⟨r,r,r⟩ p3;
    Pair⟨r5,r6,r7⟩ p2;
    Pair⟨r,r,r⟩ p1;
    p4 = new Pair⟨r7,r7,r7⟩(null,null);
    p3 = new Pair⟨r,r,r⟩(p4,null);
    p2 = new Pair⟨r5,r6,r7⟩(null,p4);
    p1 = new Pair⟨r,r,r⟩(p2,null);
    p1.setSnd⟨r,r,r,r⟩(p3); p2} }
}
```

Fig. 4. Region Annotated Core-Java Program

Consider the `Pair` class in Fig. 4. As there are two fields in this class, a distinct region is introduced for each of them, `r2` for `fst` field and `r3` for `snd` field. The `Pair` object is placed in the region `r1`. To ensure that every `Pair` instance satisfies the no-dangling requirement, the region lifetime constraint `r2⪰r1∧r3⪰r1` is added to the class invariant.

Consider the `getFst`, `swap`, and `exalloc` methods of the `Pair` class. A set of distinct region parameters are introduced for the methods' parameters, and the results, as shown in Fig. 4. The receiver regions are taken from the class definition. Moreover, the methods' region lifetime constraints are based on the possible operations of the respective methods. For example, due to an assignment operation and region subtyping, we have `r4⪰r3` for `setSnd`, while `r2=r3` is present due to the swapping operation on the receiver object in the `swap` method. Though the `swap` method's region constraint is exclusively on the regions of the current object, we associate the constraint with the method. In this way, only those objects that might call the method are required to satisfy this constraint. The class invariants of methods' parameters (including the receiver and their result) are also added to the methods' region constraints. The `exalloc` method's body introduces a local region `r` using `letreg`. Since the `p1` and `p3` objects do not escape from the `exalloc` method's body, they are stored in the local region `r`. The `p2` and `p4` objects escape through the method result, therefore they are stored in the method result's regions `r5` and `r7`, respectively.

$$\frac{\begin{array}{c}\textit{WFClasses}(P)\\ P = \textit{def}_{i:1..n}\\ \textit{FieldsOnce}(\textit{def})_{i:1..n}\\ \textit{MethodsOnce}(\textit{def})_{i:1..n}\\ P \vdash \textit{InheritanceOK}(\textit{def})_{i:1..n}\\ P \vdash_{\textit{def}} \textit{def}_{i:1..n}\end{array}}{\vdash P}\;[\mathbf{RC{-}PROG}]$$

$$\frac{\begin{array}{c} def = \textbf{class } \textit{cn}\langle r_{1..n}\rangle \textbf{extends } c\langle r_{1..m}\rangle\\ \textbf{where } \varphi\ \{\textit{field}_{1..p}\ \textit{meth}_{1..q}\}\\ r_1 \notin \bigcup_{i=1}^{p}\ \textit{reg}(\textit{field}_i)\\ \varphi \Rightarrow r_i \succeq r_1 \quad i = 2..n \quad R = \{r_1, \ldots, r_n\}\\ P; \{\textit{this} : \textit{cn}\langle r_{1..n}\rangle\}; R; \varphi \vdash_{\textit{meth}} \textit{meth}_i \quad i = 1..q\\ P; R; \varphi \vdash_{\textit{field}} \textit{field}_i \quad i = 1..p\end{array}}{P \vdash_{\textit{def}} def}\;[\mathbf{RC{-}CLASS}]$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma + (v_j : t_j)_{j:1..p} \quad R' = R \cup \{r_1, \ldots, r_m\}\\ \varphi' = \varphi \wedge \varphi_0 \qquad P; R'; \varphi' \vdash_{\textit{type}} t_j,\ j = 0..p\\ P; \Gamma'; R'; \varphi' \vdash e : t_0' \qquad P; R'; \varphi' \vdash t_0' <: t_0\end{array}}{P; \Gamma; R; \varphi \vdash_{\textit{meth}} t_0\ \textit{mn}\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p})\textbf{where } \varphi_0\ \{e\}}\;[\mathbf{RC{-}METH}]$$

$$\frac{\begin{array}{c} P; R; \varphi \vdash_{\textit{type}} t'\\ \Gamma' = \Gamma + (v : t')\\ P; \Gamma'; R; \varphi \vdash e : t\end{array}}{P; \Gamma; R; \varphi \vdash \{(t'\ v)\ e\} : t}\;[\mathbf{RC{-}EB}]$$

$$\frac{(v : t) \in \Gamma}{P; \Gamma; R; \varphi \vdash v : t}\;[\mathbf{RC{-}VAR}]$$

$$\frac{\begin{array}{c} P; R; \varphi \vdash_{\textit{type}} \textit{cn}\langle r_{1..n}\rangle \quad \textit{fieldlist}(\textit{cn}\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p}\\ (v_i : t_i') \in \Gamma \quad P; R; \varphi \vdash t_i' <: t_i \quad i = 1..p\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{new } \textit{cn}\langle r_{1..n}\rangle(v_1, .., v_p) : \textit{cn}\langle r_{1..n}\rangle}\;[\mathbf{RC{-}NEW}]$$

$$\frac{\begin{array}{c}(v_0 : \textit{cn}\langle a^+\rangle) \in \Gamma \quad P; R; \varphi \vdash_{\textit{type}} \textit{cn}\langle a^+\rangle\\ P \vdash (t\ \textit{mn}\langle a^+ r'^+\rangle((t_i\ v_i)_{i:1..n})\textbf{where } \varphi_0\ \{e\}) \in \textit{cn}\langle a^+\rangle\\ (v_i' : t_i')_{i:1..n} \in \Gamma \quad a'^+ \in R \quad \rho = [r'^+ \mapsto a'^+]\\ \varphi \Rightarrow \rho\, \varphi_0 \quad P; R; \varphi \vdash t_i' <: \rho\, t_i \quad i = 1..n\end{array}}{P; \Gamma; R; \varphi \vdash v_0.\textit{mn}\langle a^+ a'^+\rangle(v_1'..v_n') : \rho\, t}\;[\mathbf{RC{-}INVOKE}]$$

$$\frac{\begin{array}{c} a = \textit{fresh}()\\ \varphi' = \varphi \wedge \bigwedge_{r' \in R}(r' \succeq a)\\ P; \Gamma; R \cup \{a\}; \varphi' \vdash [r \mapsto a]e : t\\ \textit{reg}(t) \subseteq R\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{letreg } r \textbf{ in } e : t}\;[\mathbf{RC{-}LETR}]$$

$\rho t,\ \rho\varphi,\ \rho e$ region substitution on a type, a constraint, and an expression
fresh() returns one or more new/unused region names

Fig. 5. Region Type Checking Rules

3 Region Type System: Static Semantics

Our region type system guarantees that region-annotated Core-Java programs never create dangling references. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase. A part of region type checking rules are depicted in Fig. 5, with some auxiliary rules in Fig. 6 (a complete description of region type system is given in Appendix B). Judgments of the following forms are employed:

- $\vdash P$ denoting that a program P is well-typed.
- $P \vdash_{\textit{def}} \textit{def}$ denoting that a class declaration *def* is well-formed.
- $P; \Gamma; R; \varphi \vdash_{\textit{meth}} \textit{meth}$ denoting that a method *meth* is well-defined with respect to the program P, the type environment Γ, the set of live regions R, and the region constraint φ.
- $P; \Gamma; R; \varphi \vdash e : t$ denoting that an expression e is well-typed with respect to the program P, the type environment Γ, the set of live regions R, and the region constraint φ.

$$\mathit{reg}(\{\}) =_{def} \{\} \quad \mathit{reg}(\{v{:}\tau\langle r^*\rangle\} \cup \Gamma) =_{def} \{r^*\} \cup \mathit{reg}(\Gamma) \quad \mathit{reg}(\tau\langle r^*\rangle) =_{def} \{r^*\}$$

$$\mathit{reg}((\tau\langle r^*\rangle\ f)) =_{def} \{r^*\} \quad \mathit{reg}(\mathit{true}) =_{def} \{\} \quad \mathit{reg}(r_1 = r_2) =_{def} \{r_1, r_2\}$$

$$\mathit{reg}(r_1 \succeq r_2) =_{def} \{r_1, r_2\} \quad \mathit{reg}(q\langle r_1..r_n\rangle\} =_{def} \{r_1..r_n\} \quad \mathit{reg}(\varphi_1 \wedge \varphi_2) =_{def} \mathit{reg}(\varphi_1) \cup \mathit{reg}(\varphi_2)$$

$$\frac{}{\mathit{fieldlist}(\mathbf{Object}\langle r\rangle) =_{def} [\,]} \qquad \frac{\begin{array}{c}\mathbf{class}\ \mathit{cn}_1\langle r_{1..n}\rangle\ \mathbf{extends}\ \mathit{cn}_2\langle r_{1..m}\rangle..\{(t_i\ f_i)_{i:1..p}..\} \in P' \\ \ell = \mathit{fieldlist}(\rho\ \mathit{cn}_2\langle r_{1..m}\rangle) \quad \rho = [r_i \mapsto x_i]_{i=1}^{n}\end{array}}{\mathit{fieldlist}(\mathit{cn}_1\langle x_{1..n}\rangle) =_{def} \ell \mathbin{+\!\!+} [(\rho\ t_i)\ f_i]_{i=1}^{p}}$$

Fig. 6. Auxiliary Region Checking Rules

- $P; R; \varphi \vdash_{\mathit{type}} t$ denoting that a type t is well-formed, namely, the regions of the type t are from the set of the live regions R, and the invariant of the type t is satisfied by the constraint context φ.
- $P; R \vdash_{\mathit{constr}} t,\ \varphi$ denoting that the regions of the type t are from the set of the live regions R, while φ is the invariant of the type t.
- $P; R; \varphi \vdash_{\mathit{field}} \mathit{field}$ denoting that the type of a field *field* is well-formed with respect to $\vdash_{\mathit{type}}$ judgment.
- $P; R; \varphi \vdash t <: t'$ denoting that the type t is a subtype of the type t', namely both types are well-formed and the region constraint of the subtyping relation (defined in Fig. 3) is satisfied by the constraint context φ.

The rule [RC−PROG] denotes that a region-annotated program is well-typed if all declared classes are well-typed. The predicates in the premise are used to capture the standard well-formedness conditions for the object-oriented programs such as no duplicate definitions of classes and no cycle in the class hierarchy; no duplicate definitions of fields; no duplicate definitions of methods; and soundness of class subtyping and method overriding.

The rule [RC−CLASS] indicates that a class is well-formed if all its fields and methods are well-formed, and the class invariant ensures the necessary lifetime relations among class region parameters. In addition, the rule does not allow the first region of the class to be used by the region types of the fields. Using the first region on a field would break the object (region) subtyping (rule [RegSub] of Fig. 3). Function $\mathit{reg}(\mathit{field}_i)$ returns the region variables of a field type (see Fig. 6).

The rule [RC−METH] checks the well-formedness of a method declaration. Each region type is checked to be well-formed, that means its regions are in the current set of live regions and its invariant is satisfied by the current constraint context. The method body is checked using the type relation for expressions such that the gathered type has to be a subtype of the declared type.

Our type relation for expressions is defined in a syntax-directed fashion. Take note that region constraints of the variables are not checked at their uses ([RC−VAR]), but at their declaration sites ([RC−EB]). The region invariant of an object is also checked when that object is created ([RC−NEW]). In the rule for object creation ([RC−NEW]), the function $\mathit{fieldlist}(\mathit{cn}\langle x_{1..n}\rangle)$ returns a list comprising all declared and inherited fields of the class $\mathit{cn}\langle x_{1..n}\rangle$ and their region types according to the regions $x_1..x_n$ of the class *cn* (see Fig. 6). They are organized in an order determined by the constructor function.

The rule [RC−INVOKE] is used to check a method call. It ensures that the method region parameters are live regions and the method precondition is satisfied by the current constraint context as $\varphi \Rightarrow \rho\varphi_0$. A substitution ρ is computed for the method's formal region parameters. The current arguments are also checked to be subtypes of the method's formal parameters.

The rule [RC−LETR] is used to check a local region declaration. The local expression is checked with an extra live region a (that is a fresh region), and an extra constraint $\bigwedge_{r' \in R}(r' \succeq a)$ that ensures that newly introduced region is on the top of the region stack. The rule uses a region substitution on the expressions. Note that the region substitutions on expressions, constraints and types are defined as expected. The gathered region type of the local expression is checked to contain only live regions (from R excepting a). This guarantees that the localized region a does not escape. Function $reg(t)$ returns all region variables of t (see Fig. 6).

4 Dynamic Semantics

We define the dynamic semantics of region-annotated Core-Java as a small-step rewriting relation from machine states to machine states. A machine state is of the form $\langle \varpi, \Pi \rangle[e]$, where ϖ is the heap organized as a stack of regions, Π is the variable environment, and e is the current program. Our dynamic semantics was inspired by the previous work on abstract models of memory management [17] and region-based memory management [9, 12]. The following notations are used:

Region Variables :	r, a	$\in RegVar$
Offset :	o	$\in Offset$
Locations :	ℓ *or* (r, o)	$\in Location = RegVar \times Offset$
Primitive Values :	k | `null`	$\in Prim$
Values :	δ	$\in Value = Prim \uplus Location$
Variable Environment :	Π	$\in VEnv = Var \rightharpoonup_{\text{fin}} Value$
Field Environment :	V	$\in FEnv = FieldName \rightharpoonup_{\text{fin}} Value$
Object Values :	$cn\langle r^* \rangle(V)$	$\in ObjVal = ClassName \times (RegVar)^n \times FEnv$
Store :	ϖ	$\in Store = [\,] \vert [r \mapsto Rgn]Store$
Runtime Regions :	Rgn	$\in Region = Offset \rightharpoonup_{\text{fin}} ObjVal$

Regions are identified by region variables. We assume a denumerably infinite set of region variables, $RegVar$. The store ϖ is organized as a stack, that defines an ordered map from region variables, r to runtime regions Rgn. The notation $[r \mapsto Rgn]\varpi$ denotes a stack with the region r on the top, while $[\,]$ denotes an empty store. The store can only be extended with new region variables. A runtime region Rgn is an unordered finite map from offsets to object values. We assume a denumerably infinite set of offsets, $Offset$ for each runtime region Rgn.

The set of values that can be assigned to variables and fields is denoted by $Value$. Such a value is either a primitive value (a constant or a null value) or it is a location in the store. A location consists of a pair of a region variable and an offset.

An object value consists of a region type $cn\langle r^*\rangle$, and a field environment V mapping field names to values. V is not really an environment since it can only be updated, never extended. An update of field f with value δ is written as $V{+}\{f{\mapsto}\delta\}$.

The variable environment Π is a mapping $\mathit{Var} \rightharpoonup_{\text{fin}} \mathit{Value}$, while the type environment Γ that corresponds to the runtime variable environment is also a mapping $\mathit{Var} \rightharpoonup_{\text{fin}} \mathit{Type}$. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase.

Notation $f : A \rightharpoonup_{\text{fin}} B$ denotes a partial function from A to B with a finite domain, written $A = \mathit{dom}(f)$. We write $f{+}\{a \mapsto b\}$ for the function like f but mapping a to b (if $a{\in}\mathit{dom}(f)$ and $f(a){=}c$ then $(f{+}\{a \mapsto b\})(a){=}b$).

The notation $\{\}$ (or $\emptyset$) stands for an undefined function. Given a function $f : A \rightharpoonup_{\text{fin}} B$, the notation $f{-}C$ denotes the function $f_1 : (A{-}C) \rightharpoonup_{\text{fin}} B$ such that $\forall x{\in}(A{-}C){\cdot}f_1(x){=}f(x)$.

We require some intermediate expressions for the small-step dynamic semantics to follow through. The intermediate expressions help our proof to use simpler induction techniques rather than a more elaborate co-induction machinery. The syntax of intermediate expressions is thus extended from the original expression syntax, as follows:

$$e ::= \ldots \mid (r,o) \mid \mathtt{ret}(v,e) \mid \mathtt{retr}(r,e)$$

The expression $\mathtt{ret}(v,e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The variable associated with `ret` denotes either a block local variable or a method receiver or a method parameter. This variable is popped from the variable environment at the end of the block's evaluation. In the case of a method invocation there are multiple nested `ret`s which pop off the receiver and the method parameters from the variable environment at the end of the method's evaluation. The expression $\mathtt{retr}(r,e)$ is used to pop off the top region, r of the store stack at the end of expression e evaluation.

Dynamic semantics rules of region annotated Core-Java are shown in Fig. 7 and Fig. 8. The evaluation judgment is of the form:

$$\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']$$

where ϖ (ϖ') denotes the store before (after) evaluation, while Π (Π') denotes the variable environment before (after) evaluation. The store ϖ organized as a stack establishes the outlive relations among regions at runtime. The function $\mathit{ord}(\varpi)$ returns the outlive relations for a given store. The function $\mathit{dom}(\varpi)$ returns the set of the store regions, while the function $\mathit{location\_dom}(\varpi)$ returns the set of all locations from the store. They are defined as follows:

$$\mathit{ord}([r_1{\mapsto}\mathit{Rgn}_1][r_2{\mapsto}\mathit{Rgn}_2]\varpi){=}_{def}(r_2{\succeq}r_1){\wedge}\mathit{ord}([r_2{\mapsto}\mathit{Rgn}_2]\varpi)$$
$$\mathit{ord}([r{\mapsto}\mathit{Rgn}]) ={}_{def}\ \mathit{true} \quad \mathit{ord}([\,]) ={}_{def}\ \mathit{true}$$
$$\mathit{dom}([r{\mapsto}\mathit{Rgn}]\varpi){=}_{def}\{r\}{\cup}\mathit{dom}(\varpi) \quad \mathit{dom}([r{\mapsto}\emptyset]\varpi){=}_{def}\{r\}{\cup}\mathit{dom}(\varpi) \quad \mathit{dom}([\,]){=}_{def}\emptyset$$
$$\mathit{location\_dom}(\varpi){=}_{def}\{(r,o) \mid \varpi{=}\varpi_1[r{\mapsto}\mathit{Rgn}]\varpi_2 \wedge \mathit{Rgn}{\neq}\emptyset \wedge o{\in}\mathit{dom}(\mathit{Rgn})\}$$

Notation $\varpi(r)(o)$ denotes an access into the region r at the offset o, as follows:

$$\varpi(r)(o){=}_{def}\mathit{Rgn}(o)\ \ where\ \ \varpi{=}\varpi_1[r{\mapsto}\mathit{Rgn}]\varpi_2$$

We define the meaning of *no-dangling references* property at runtime. The property refers to two kinds of references: (1) references from variable environment to store locations, and (2) references from store locations to other store locations. Note that the notion of *no-dangling references* was introduced in Fig. 1, and a reference is formalized as a location (r,o).

$$\begin{array}{c}[\underline{\text{D-VAR}}]\\ \dfrac{v \in dom(\mathit{\Pi})}{\langle\varpi, \mathit{\Pi}\rangle[v]\hookrightarrow\langle\varpi, \mathit{\Pi}\rangle[\mathit{\Pi}(v)]}\end{array}
\qquad
\begin{array}{c}[\underline{\text{D-FD}}]\\ \dfrac{\mathit{\Pi}(v)=(r,o) \quad \varpi=\varpi_1[r\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V)}{\langle\varpi, \mathit{\Pi}\rangle[v.f]\hookrightarrow\langle\varpi, \mathit{\Pi}\rangle[V(f)]}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-ASSGN1}}]\\ \dfrac{\langle\varpi, \mathit{\Pi}\rangle[e]\hookrightarrow\langle\varpi', \mathit{\Pi}'\rangle[e']}{\langle\varpi, \mathit{\Pi}\rangle[lhs = e]\hookrightarrow\langle\varpi', \mathit{\Pi}'\rangle[lhs = e']}\end{array}
\quad
\begin{array}{c}[\underline{\text{D-ASSGN2}}]\\ \dfrac{\begin{array}{c}v{\in}dom(\mathit{\Pi}) \quad \mathit{\Pi}'{=}\mathit{\Pi}{+}\{v{\mapsto}\delta\}\\ \delta{=}(r_1,o_1) \ \wedge \ r_1{\in}dom(\varpi)\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[v = \delta]\hookrightarrow\langle\varpi, \mathit{\Pi}'\rangle[\mathbf{()}]}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-ASSGN2-DANGLERR}}]\\ \dfrac{\begin{array}{c}v \in dom(\mathit{\Pi})\\ \delta{=}(r_1,o_1) \ \wedge \ r_1{\notin}dom(\varpi)\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[v{=}\delta]\hookrightarrow\mathtt{danglingerr}}\end{array}
\quad
\begin{array}{c}[\underline{\text{D-ASSGN3}}]\\ \dfrac{\begin{array}{c}\mathit{\Pi}(v){=}(a,o) \quad \varpi{=}\varpi_1[a{\mapsto}Rgn]\varpi_2 \quad Rgn(o){=}cn\langle a^+\rangle(V)\\ Rgn'{=}Rgn{+}\{o{\mapsto}cn\langle a^+\rangle(V{+}\{f{\mapsto}\delta\})\}\ \varpi'{=}\varpi_1[a{\mapsto}Rgn']\varpi_2\\ \delta{=}(r_1,o_1) \ \wedge \ \mathbf{ord}(\varpi){\Rightarrow}(r_1{\succeq}\mathbf{fieldregion}(cn\langle a^+\rangle, f))\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[v.f = \delta]\hookrightarrow\langle\varpi', \mathit{\Pi}\rangle[\mathbf{()}]}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-ASSGN3-DANGLERR}}]\\ \dfrac{\begin{array}{c}\mathit{\Pi}(v){=}(a,o) \quad \varpi{=}\varpi_1[a{\mapsto}Rgn]\varpi_2 \quad Rgn(o){=}cn\langle a^+\rangle(V)\\ \delta{=}(r_1,o_1) \ \wedge \ \neg\ (\mathbf{ord}(\varpi){\Rightarrow}(r_1{\succeq}\mathbf{fieldregion}(cn\langle a^+\rangle, f)))\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[v.f = \delta]\hookrightarrow\mathtt{danglingerr}}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-NEW}}]\\ \dfrac{\begin{array}{c}\mathbf{class}\ cn\langle r_{1..n}\rangle\ \mathbf{extends}\ c\langle...\rangle\ \mathbf{where}\ \varphi_{inv}\ \{...\} \in P \qquad \mathbf{ord}(\varpi){\Rightarrow}\varphi_{inv}\\ \varpi{=}\varpi_1[r_1{\mapsto}Rgn]\varpi_2 \quad V{=}\{f_1{\mapsto}\mathit{\Pi}(v_1), ..., f_p{\mapsto}\mathit{\Pi}(v_p)\} \quad \mathbf{fieldlist}(cn\langle r_{1..n}\rangle){=}(t_i\ f_i)_{i:1..p}\\ \mathit{if}\ \mathit{\Pi}(v_i){=}(r'_i, o'_i)\ \mathit{then}\ \mathbf{ord}(\varpi){\Rightarrow}(r'_i{\succeq}\mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)) \quad i{=}1..p\\ o{\notin}dom(Rgn) \quad Rgn'{=}Rgn{+}\{o{\mapsto}cn\langle r_{1..n}\rangle(V)\} \quad \varpi'{=}\varpi_1[r_1{\mapsto}Rgn']\varpi_2\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[\mathbf{new}\, cn\langle r_{1..n}\rangle(v_{1..p})]\hookrightarrow\langle\varpi', \mathit{\Pi}\rangle[(r_1, o)]}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-NEW-DANGLERR}}]\\ \dfrac{\begin{array}{c}\mathbf{class}\ cn\langle r_{1..n}\rangle\ \mathbf{extends}\ c\langle...\rangle\ \mathbf{where}\ \varphi_{inv}\ \{...\}\ \in\ P\\ V{=}\{f_1{\mapsto}\mathit{\Pi}(v_1), ..., f_p{\mapsto}\mathit{\Pi}(v_p)\} \quad \mathbf{fieldlist}(cn\langle r_{1..n}\rangle){=}(t_i\ f_i)_{i:1..p}\\ \neg(\mathbf{ord}(\varpi){\Rightarrow}\varphi_{inv}) \vee\ (\exists i{\in}\{1..p\}\cdot \mathit{\Pi}(v_i){=}(r'_i, o'_i)\ \wedge\\ \qquad\qquad \neg(\mathbf{ord}(\varpi){\Rightarrow}(r'_i{\succeq}\mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)))\end{array}}{\langle\varpi, \mathit{\Pi}\rangle[\mathbf{new}\, cn\langle r_{1..n}\rangle(v_{1..p})]\hookrightarrow\mathtt{danglingerr}}\end{array}$$

$$\begin{array}{c}[\underline{\text{D-INVOKE}}]\\ \dfrac{\begin{array}{c}\{a^+, a'^+\}{\subset}\mathbf{dom}(\varpi)\\ \mathit{\Pi}(v'_0) = (a_1, o) \quad \varpi(a_1)(o) = cn\langle a^+\rangle(V)\\ (t_0\ mn\langle a^+ r'^+\rangle((t\,v)_{1..p})\mathbf{where}\ \varphi\ \{e\}) \in cn\langle a^+\rangle\\ n_i{=}fresh()\ \ i = 0..p \quad \rho{=}[r'^+{\mapsto}a'^+]\\ \mathit{\Pi}'{=}\mathit{\Pi}{+}\{n_i{\mapsto}\mathit{\Pi}(v'_i)_{i:0..p}\}\\ e'{=}\mathtt{ret}(n_0, ..\mathtt{ret}(n_p, [this{\mapsto}n_0][v_i{\mapsto}n_i]_{i:1}^{p}\rho e))\end{array}}{\langle\varpi,\mathit{\Pi}\rangle[v'_0.mn\langle a^+ a'^+\rangle(v'_{1..p})]\hookrightarrow\langle\varpi,\mathit{\Pi}'\rangle[e']}\end{array}
\quad
\begin{array}{c}[\underline{\text{D-INVOKE-DANGLERR}}]\\ \dfrac{\neg(r^+{\in}\mathbf{dom}(\varpi))}{\langle\varpi,\mathit{\Pi}\rangle[v.mn\langle r^+\rangle(v^*)]\hookrightarrow\mathtt{danglingerr}}\end{array}$$

Fig. 7. Dynamic Semantics for Region-Annotated Core-Java: Part I

[D−EB]

$$\frac{n{=}\mathit{fresh}()\quad \mathit{\Pi}'{=}\mathit{\Pi}{+}\{(n{\mapsto}\mathbf{init}(t))\}\quad e'{=}\texttt{ret}(n,e)}{\langle\varpi,\mathit{\Pi}\rangle[\{(t\,v)\ e\}]{\hookrightarrow}\langle\varpi,\mathit{\Pi}'\rangle[e']}$$

[D−RET1]

$$\frac{\langle\varpi,\mathit{\Pi}\rangle[e]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[e']}{\langle\varpi,\mathit{\Pi}\rangle[\texttt{ret}(v,e)]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[\texttt{ret}(v,e')]}$$

[D−RET2]

$$\frac{}{\langle\varpi,\mathit{\Pi}\rangle[\texttt{ret}(v,\delta)]{\hookrightarrow}\langle\varpi,\mathit{\Pi}{-}\{v\}\rangle[\delta]}$$

[D−LETR]

$$\frac{a{=}\mathit{fresh}()}{\langle\varpi,\mathit{\Pi}\rangle[\mathbf{letreg}\ r\ \mathbf{in}\ e]{\hookrightarrow}\langle[a{\mapsto}\emptyset]\varpi,\mathit{\Pi}\rangle[\texttt{retr}(a,[r{\mapsto}a]e)]}$$

[D−RETR1]

$$\frac{\langle\varpi,\mathit{\Pi}\rangle[e]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[e']}{\langle\varpi,\mathit{\Pi}\rangle[\texttt{retr}(a,e)]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[\texttt{retr}(a,e')]}$$

[D−RETR2]

$$\frac{\begin{array}{c}(\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall v{\in}\mathit{\Pi}\cdot(\mathit{\Pi}(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi)\wedge\\ \forall f\in \mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\mathit{\Pi}\rangle[\texttt{retr}(a,\delta)]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[\delta]}$$

[D−RETR2−DANGLERR]

$$\frac{\begin{array}{c}\neg(a{=}a_1)\vee\\ \neg((\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi)))\vee\neg((\forall v{\in}\mathit{\Pi}\cdot(\mathit{\Pi}(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))))\\ \vee\neg(\forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi)\wedge\\ \forall f\in \mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi)))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\mathit{\Pi}\rangle[\texttt{retr}(a_1,\delta)]{\hookrightarrow}\texttt{danglingerr}}$$

[D−IF1]

$$\frac{\mathit{\Pi}(v){=}\mathit{true}}{\langle\varpi,\mathit{\Pi}\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[e_1]}$$

[D−IF2]

$$\frac{\mathit{\Pi}(v){=}\mathit{false}}{\langle\varpi,\mathit{\Pi}\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[e_2]}$$

[D−LOOP1]

$$\frac{\mathit{\Pi}(v){=}\mathit{true}}{\langle\varpi,\mathit{\Pi}\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[e\,\mathbf{;}\,\mathbf{while}\ v\ e]}$$

[D−LOOP2]

$$\frac{\mathit{\Pi}(v){=}\mathit{false}}{\langle\varpi,\mathit{\Pi}\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[\mathbf{()}]}$$

[D−SEQ1]

$$\frac{\langle\varpi,\mathit{\Pi}\rangle[e_1]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[e_1']}{\langle\varpi,\mathit{\Pi}\rangle[e_1\,\mathbf{;}\,e_2]{\hookrightarrow}\langle\varpi',\mathit{\Pi}'\rangle[e_1'\,\mathbf{;}\,e_2]}$$

[D−SEQ2]

$$\frac{}{\langle\varpi,\mathit{\Pi}\rangle[\delta_1\,\mathbf{;}\,e_2]{\hookrightarrow}\langle\varpi,\mathit{\Pi}\rangle[e_2]}$$

[D−NULLERR1]

$$\frac{\mathit{\Pi}(v){=}\texttt{null}}{\langle\varpi,\mathit{\Pi}\rangle[v.f]{\hookrightarrow}\texttt{nullerr}}$$

[D−NULLERR2]

$$\frac{\mathit{\Pi}(v){=}\texttt{null}}{\langle\varpi,\mathit{\Pi}\rangle[v.f=\delta]{\hookrightarrow}\texttt{nullerr}}$$

[D−NULLERR3]

$$\frac{\mathit{\Pi}(v){=}\texttt{null}}{\langle\varpi,\mathit{\Pi}\rangle[v.mn\langle a^*\rangle(u^*)]{\hookrightarrow}\texttt{nullerr}}$$

Fig. 8. Dynamic Semantics for Region-Annotated Core-Java: Part II

Definition 1. *(live location) A location* (r, o) *is* live *with respect to a store* ϖ*, if* $r \in dom(\varpi)$.

Definition 2. *(no-dangling)*

1. *A variable environment* Π *is* no-dangling *with respect to a store* ϖ *if for all* $v \in dom(\Pi)$,$\Pi(v)$ *is either a primitive value, or a live location* (r, o) *with respect to* ϖ.
2. *A runtime store* ϖ *is* no-dangling *if each region* $r_1 \in dom(\varpi)$ *contains only references to regions older than itself, that means that for each location* $(r_1, o) \in location\_dom(\varpi)$ *containing an object value* $\varpi(r_1)(o)=cn\langle r_{1..n}\rangle(V)$*, that object value satisfies the* non-dangling requirement for a class, *such that* $ord(\varpi)\Rightarrow \bigwedge_{i:2..n}(r_i \succeq r_1)$ *and the current values of the fields are either primitives or references to regions older than those expected by the region type* $cn\langle r_{1..n}\rangle$*, as follows:*
   $\forall f \in dom(V)\ .\ V(f)=(r_f, o_f) \quad ord(\varpi)\Rightarrow r_f \succeq fieldregion(cn\langle r_{1..n}\rangle, f)$
   Function $fieldregion(cn\langle r_{1..n}\rangle, f)$ *computes the region type of the class field f and then returns its first region where the field is expected to be stored.*

The dynamic semantics evaluation rules may yield two possible runtime errors, namely:

$$Error ::= \texttt{nullerr} \mid \texttt{danglingerr}$$

The first error `nullerr` is due to null pointers (by accessing fields or methods of null objects). The second error `danglingerr` is reported when a store updating operation or a variable environment updating operation creates a dangling reference. Our dynamic semantics rules use runtime checks to guarantee that a `danglingerr` error is reported (and the execution is aborted) whenever the program evaluation tries to create a dangling reference. There are five situations that require no-dangling reference checks at runtime:

- *Creation of a new object value.* Rule [D−NEW] checks whether the class invariant holds, $ord(\varpi)\Rightarrow\varphi_{inv}$ (mainly whether the fields regions $r_{i:2..n}$ outlive the region r_1 of the object). The initial value of a field is also checked to be stored in a region that outlives the expected region of that field $r'_i \succeq fieldregion(cn\langle r_{1..n}\rangle, f_i)$. The function $fieldlist(cn\langle r_{1..n}\rangle)$ is defined in Fig. 6.
- *Updating of an object's field.* Rule [D−ASSGN3] checks whether the region r_1 of the new location $\delta=(r_1, o_1)$ outlives the expected region for the object field f, $r_1 \succeq fieldregion(cn\langle a^+\rangle, f)$.
- *Updating a variable from the variable environment.* Rule [D−ASSGN2] checks whether the new location $\delta=(r_1, o_1)$ assigned to a variable is live, namely its region is in the current store, $r_1 \in dom(\varpi)$.
- *Deallocation of a region.* Rule [D−RETR2] checks whether the region a is on the top of the store stack. Then it checks whether a reference to a does not escape neither through the value result δ, nor through the program variable environment Π, nor through the object values of the store ϖ. Note that when a new region is allocated, in rule [D−LETR], a fresh region name is used in order to avoid region name duplication in the store.
- *Calling a method.* Rule [D−INVOKE] checks whether the method's region arguments are in the current store and then prepares the variable environment for the method's body execution.

The corresponding rules [D-NEW-DANGLERR], [D-ASSGN3-DANGLERR], [D-ASSGN2-DANGLERR], [D-RETR2-DANGLERR], and [D-INVOKE-DANGLERR] generate a `danglingerr` error due to the failure of their runtime checks. In the rules [D-ASSGN2], [D-ASSGN3], and [D-LOOP2] the result **()** denotes the singleton value of type **void**. Note that the type **void** is assumed to be isomorphic to type **unit**. In rule [D-EB], the locally declared variable is assigned, with the help of the function **init**, an initial value according to its type as follows:

$$\begin{array}{lll}
\textbf{init}(t) \ =_{def} \ \textit{case } t \textit{ of} & & \\
\quad \textit{boolean} & \rightarrow & \textit{false} \\
\quad \textit{int} & \rightarrow & \textit{0} \\
\quad cn\langle r_{1..n}\rangle & \rightarrow & \texttt{null}
\end{array}$$

5 Soundness Proof

5.1 Extended Static Semantics

The static semantics of the language is also extended to include the new intermediate expressions. The process requires introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics [19]. In our case, it is denoted by: $\Sigma \in \textit{StoreType} = \textit{RegVar} \rightharpoonup_{\mathsf{fin}} \textit{Offset} \rightharpoonup_{\mathsf{fin}} \textit{Type}$. The judgments of static semantics are extended with store typing, as follows:

$$P; \Gamma; R; \varphi; \Sigma \vdash e : t$$

For a store typing $\Sigma : R \rightharpoonup_{\mathsf{fin}} O \rightharpoonup_{\mathsf{fin}} \textit{Type}$, a region r, a location (r, o), and a type t we introduce the following notations:

$$\begin{array}{l}
\textit{dom}(\Sigma){=}R \qquad \Sigma(r)(o){=}f(o),\ \textit{where}\ f{=}\Sigma(r) \\
\textit{location\_dom}(\Sigma){=}_{def}\{(r,o) \mid r{\in}\textit{dom}(\Sigma) \wedge f{=}\Sigma(r) \wedge f{\neq}\emptyset \wedge o{\in}\textit{dom}(f)\} \\
\Sigma{-}r{=}_{def}\Sigma_1\ \textit{such that}\ \Sigma_1 : (R{-}\{r\}){\rightharpoonup_{\mathsf{fin}}}O{\rightharpoonup_{\mathsf{fin}}}\textit{Type} \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_1(r'){=}\Sigma(r') \\
\Sigma{+}r{=}_{def}\Sigma_2\ \textit{such that}\ \Sigma_2 : (R{\cup}\{r\}){\rightharpoonup_{\mathsf{fin}}}O{\rightharpoonup_{\mathsf{fin}}}\textit{Type} \wedge \Sigma_2(r){=}\emptyset \wedge \forall r'{\in}R \cdot \Sigma_2(r'){=}\Sigma(r') \\
\Sigma{-}(r,o){=}_{def}\Sigma_3\ \textit{such that}\ \Sigma_3 : R{\rightharpoonup_{\mathsf{fin}}}O{\rightharpoonup_{\mathsf{fin}}}\textit{Type} \\
\qquad\qquad {\wedge} r{\in}R \wedge \Sigma_3(r){=}\Sigma(r){-}\{o\} \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_3(r'){=}\Sigma(r') \\
\Sigma{+}((r,o) : t){=}_{def}\Sigma_4\ \textit{such that}\ \Sigma_4 : R{\rightharpoonup_{\mathsf{fin}}}O{\rightharpoonup_{\mathsf{fin}}}\textit{Type} \\
\qquad\qquad {\wedge} r{\in}R \wedge \Sigma_4(r){=}\Sigma(r){+}\{o{\mapsto}t\} \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_4(r'){=}\Sigma(r')
\end{array}$$

Definition 3. *The function* vars(e) *computes the set of all program variables which occur in the expression* e*, excepting those variables introduced by e's block subexpressions, as follows:*

$$\begin{array}{lll}
\textit{vars}(e) \ =_{def} \ \textit{case } e \textit{ of} & & \\
\quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup \ \textit{vars}(e) \\
\quad \{(t\, v)\ \ e\} & \rightarrow & \textit{vars}(e) \setminus \{v\} \\
\quad \texttt{retr}(r, e) \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \textit{vars}(e) \\
\quad v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow & \{v\} \cup \textit{vars}(e) \\
\quad v.f \mid v & \rightarrow & \{v\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \{v\} \cup \textit{vars}(e_1) \cup \textit{vars}(e_2) \\
\quad e_1\ \textbf{;}\ e_2 & \rightarrow & \textit{vars}(e_1) \cup \textit{vars}(e_2) \\
\quad \textbf{new}\ cn\langle r^{+}\rangle\textbf{(}v^{*}\textbf{)} & \rightarrow & \{v^{*}\} \\
\quad v.mn\langle v^{*}\rangle\textbf{(}v^{*}\textbf{)} & \rightarrow & \{v\} \cup \{v^{*}\} \\
\quad \textit{otherwise} & \rightarrow & \emptyset
\end{array}$$

Definition 4. *The function* $\mathit{retvars}(e)$ *computes the set of all program variables which occur in the* ret *subexpressions of the expression* e*, as follows:*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{retvars}(e)\ =_{def}\ \ \mathit{case}\ e\ \mathit{of}} \\
\quad \texttt{ret}(v,e) & \rightarrow\ \{v\} \cup \mathit{retvars}(e) \\
\quad \texttt{retr}(r,e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow\ \mathit{retvars}(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow\ \mathit{retvars}(e) \\
\quad e_1\ \textbf{;}\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow\ \mathit{retvars}(e_1) \cup \mathit{retvars}(e_2) \\
\quad \mathit{otherwise} & \rightarrow\ \emptyset
\end{array}$$

Definition 5. *The function* $\mathit{regs}(e)$ *computes the set of all region variables which occur in the expression* e*, excepting those regions introduced by e's letreg subexpressions, as follows:*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{regs}(e)\ =_{def}\ \ \mathit{case}\ e\ \mathit{of}} \\
\quad \{(t\ v)\ \ e\} & \rightarrow\ \mathit{reg}(t) \cup \mathit{regs}(e) \\
\quad \texttt{retr}(r,e) & \rightarrow\ \{r\} \cup \mathit{regs}(e) \\
\quad \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow\ \mathit{regs}(e) \setminus \{r\} \\
\quad \texttt{ret}(v,e) \mid v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow\ \mathit{regs}(e) \\
\quad (r,o) & \rightarrow\ \{r\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid e_1\ \textbf{;}\ e_2 & \rightarrow\ \mathit{regs}(e_1) \cup \mathit{regs}(e_2) \\
\quad \textbf{new}\ cn\langle r^+\rangle\textbf{(}v^*\textbf{)} \mid v.mn\langle r^+\rangle\textbf{(}v^*\textbf{)} & \rightarrow\ \{r^+\} \\
\quad \mathit{otherwise} & \rightarrow\ \emptyset
\end{array}$$

where reg(t) is defined in the Figure 6.

Definition 6. *The function* $\mathit{retregs}(e)$ *computes the set of all region variables which occur in the* retr *subexpressions of the expression* e*, as follows:*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{retregs}(e)\ =_{def}\ \ \mathit{case}\ e\ \mathit{of}} \\
\quad \texttt{retr}(r,e) & \rightarrow\ \{r\} \cup\ \mathit{retregs}(e) \\
\quad \texttt{ret}(v,e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow\ \mathit{retregs}(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow\ \mathit{retregs}(e) \\
\quad e_1\ \textbf{;}\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow\ \mathit{retregs}(e_1) \cup \mathit{retregs}(e_2) \\
\quad \mathit{otherwise} & \rightarrow\ \emptyset
\end{array}$$

Definition 7. *(valid program)*

1. *An expression* e *is a* valid expression *if the predicate* $\mathit{valid}(e)$ *holds, where* $\mathit{valid}(e)$ *is defined as follows:*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{valid}(e)\ =_{def}\ \ \mathit{case}\ e\ \mathit{of}} \\
\quad \{(t\ v)\ e\} & \rightarrow\ \mathit{retvars}(e){=}\emptyset \wedge \mathit{retregs}(e){=}\emptyset \\
\quad \mathit{lhs} = e & \rightarrow\ \mathit{retvars}(e){\cap}\mathit{vars}(\mathit{lhs}){=}\emptyset \wedge \mathit{valid}(e) \\
\quad e_1\ \textbf{;}\ e_2 & \rightarrow\ \mathit{retregs}(e_2){=}\emptyset \wedge \mathit{retvars}(e_2){=}\emptyset \wedge \mathit{valid}(e_1) \\
 & \quad\ \wedge \mathit{retvars}(e_1){\cap}\mathit{vars}(e_2){=}\emptyset \wedge \mathit{retregs}(e_1){\cap}\mathit{regs}(e_2){=}\emptyset \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow\ \mathit{retregs}(e_1){=}\emptyset \wedge \mathit{retvars}(e_1){=}\emptyset \\
 & \quad\ \wedge \mathit{retregs}(e_2){=}\emptyset \wedge \mathit{retvars}(e_2){=}\emptyset \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow\ \mathit{retregs}(e){=}\emptyset \wedge \mathit{retvars}(e){=}\emptyset \\
\quad \texttt{ret}(v,e) & \rightarrow\ v \notin \mathit{retvars}(e) \wedge \mathit{valid}(e) \\
\quad \texttt{retr}(r,e) & \rightarrow\ r \notin \mathit{retregs}(e) \wedge \mathit{valid}(e) \\
\quad \mathit{otherwise} & \rightarrow\ \mathit{true}
\end{array}$$

2. *A method is a* valid method *if the method's body* e*, is a valid block expression such that* $\mathit{retvars}(e){=}\emptyset$ *and* $\mathit{retregs}(e){=}\emptyset$*.*

$$\frac{r{\in}R \quad \Sigma(r)(o) = t}{P;\Gamma;R;\varphi;\Sigma \vdash (r,o) : t}\,[\textbf{RC-LOCATION}] \qquad \frac{\begin{array}{c}P;R;\varphi \vdash_{type} cn\langle r_{1..n}\rangle \quad fieldlist(cn\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p}\\ P;\Gamma;R;\varphi;\Sigma \vdash V(f_i) : t'_i \quad P;R;\varphi \vdash t'_i <: t_i \quad i{=}1..p\end{array}}{P;\Gamma;R;\varphi;\Sigma \vdash cn\langle r_{1..n}\rangle(V) : cn\langle r_{1..n}\rangle}\,[\textbf{RC-ObjVal}]$$

$$\frac{v{\in}\Gamma \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t}{P;\Gamma;R;\varphi;\Sigma \vdash \texttt{ret}(v,e) : t}\,[\textbf{RC-RET}] \qquad \frac{P;\Gamma;R;\varphi;\Sigma \vdash e : t' \quad P;R;\varphi \vdash t' <: t}{P;\Gamma;R;\varphi;\Sigma \vdash e : t}\,[\textbf{SUBSUMPTION}]$$

$$\frac{\begin{array}{c}a{\in}R \quad R_t{=}R{-}lreg(e){-}\{a\} \quad \varphi{\Rightarrow} \bigwedge_{r\in R_t}(r{\succeq}a)\\ reg(t){\subseteq}R_t \quad reg(\Gamma{-}lvar(e)) \subseteq R_t \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t\end{array}}{P;\Gamma;R;\varphi;\Sigma \vdash \texttt{retr}(a,e) : t}\,[\textbf{RC-RETR}]$$

Fig. 9. Region Type Checking Rules for Valid Intermediate Expressions

3. *A class is a* valid class *if all the class's methods are valid methods.*
4. *A program is a* valid program *if all the program's classes are valid classes.*

Note that a source language Core-Java program is by default a valid program since it does not contain any intermediate expression.

Definition 8. *Using the evaluation rules from Fig. 7 and Fig. 8, the function* $lvar(e)$ *estimates the set of variables which may be popped off from the variable environment* Π *during the evaluation of the valid expression* e *(note that only* $\texttt{ret}(v,e)$ *may affect* Π*), as follows:*

$$\begin{array}{lll} lvar(e) \ =_{def}\ \textit{case e of} & & \\ \quad \texttt{ret}(v,e) & \rightarrow & \{v\} \cup lvar(e) \\ \quad \texttt{retr}(r,e) \mid lhs = e \mid e\,\texttt{;}\,e_1 & \rightarrow & lvar(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

Definition 9. *Using the evaluation rules from Fig. 7 and Fig. 8, the function* $lreg(e)$ *estimates the set of regions which may be popped off from the store* ϖ *during the evaluation of the valid expression* e *(note that only* $\texttt{retr}(r,e)$ *may affect* ϖ*), as follows:*

$$\begin{array}{lll} lreg(e) \ =_{def}\ \textit{case e of} & & \\ \quad \texttt{retr}(r,e) & \rightarrow & \{r\} \cup lreg(e) \\ \quad \texttt{ret}(v,e) \mid lhs = e \mid e\,\texttt{;}\,e_1 & \rightarrow & lreg(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

Definition 10. *Using the evaluation rules from Fig. 7 and Fig. 8, the function* $lloc(e)$ *estimates the new location which may be created into an existing region during one evaluation step of the valid expression* e *(note that only* **new** *may create a new location), as follows:*

$$\begin{array}{lll} lloc(e) \ =_{def}\ \textit{case e of} & & \\ \quad \textbf{new}\ cn\langle r_1,..,r_n\rangle\textbf{(}v^*\textbf{)} & \rightarrow & \{(r_1,o)\} \\ \quad \texttt{ret}(v,e) \mid \texttt{retr}(r,e) \mid lhs = e \mid e\,\texttt{;}\,e_1 & \rightarrow & lloc(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

where the offset o *of the region* r *is the offset where the next allocation in* r *is done.*

The judgments of the new intermediate expressions are presented in Fig. 9. They assume that the expressions are valid with respect to the Definition 7. The first two rules [RC−LOCATION] and [RC−ObjVal] are used to type the store, either a location or an object value (i.e. a location's content). Rule [RC−ObjVal] preserves the same invariants as those of the rule [RC−NEW]. Rule [RC−RET] ensures that the variable to be popped off, v is in the current environment $\mathit{\Gamma}$. The subsumption rule [SUBSUMPTION] simplifies the next theorems and their proofs.

Rule [RC−RETR] is similar to rule [RC−LETR], but it takes into account the evaluation of the expression $\mathtt{retr}(r, e)$. The first check ensures that the region to be deallocated, a is in R. The R_t denotes the regions from R which are different than a and are not younger than a. Note that $\mathit{lreg}(e)$ denotes the regions which are younger than a. The second check ensures that our type system uses only lexically scoped regions such that the region to be deallocated, a is always on the top of the regions stack. The third and the fourth check ensure that the region a and the regions younger than a do not escape either through the result or through the live variables of the type environment. Note that $\mathit{lvar}(e)$ denotes the local variables of the expression e which are deallocated from the variable environment during the evaluation of e.

5.2 Soundness Theorem

By using the techniques from [24] we prove that a valid program well-typed by the type system we have presented, never creates dangling references. In what follows, we formulate the type preservation theorem and the progress theorem. The soundness of our static semantics relies on the following consistency relationship between the static and dynamic semantics.

Definition 11. *(consistency) A run-time environment $(\varpi, \mathit{\Pi})$ is* consistent *with a static environment $(\mathit{\Gamma}, R, \varphi, \mathit{\Sigma})$, written $\mathit{\Gamma}, R, \varphi, \mathit{\Sigma} \models \langle \varpi, \mathit{\Pi} \rangle$, if the following judgment holds:*

$$\frac{\begin{array}{c} \mathit{dom}(\mathit{\Gamma}){=}\mathit{dom}(\mathit{\Pi}) \quad \forall v \in \mathit{dom}(\mathit{\Pi}) \cdot P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash \mathit{\Pi}(v) : \mathit{\Gamma}(v) \quad \mathit{reg}(\mathit{\Gamma}){\subseteq}R \\ \mathit{location\_dom}(\mathit{\Sigma}){=}\mathit{location\_dom}(\varpi) \quad \mathit{dom}(\mathit{\Sigma}){=}\mathit{dom}(\varpi) \quad R{=}\mathit{dom}(\varpi) \quad \mathit{ord}(\varpi){\Rightarrow}\varphi \\ \forall (r, o){\in}\mathit{location\_dom}(\varpi) \cdot P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash \varpi(r)(o) : \mathit{\Sigma}(r)(o) \end{array}}{\mathit{\Gamma}, R, \varphi, \mathit{\Sigma} \models \langle \varpi, \mathit{\Pi} \rangle}$$

Note that $\varpi(r)(o)$ returns an object value $\mathit{cn}\langle r^* \rangle(V)$ whose type is $\mathit{cn}\langle r^* \rangle$. In our instrumented operational semantics an object value and its type are stored together.

The subject reduction theorem ensures that the region type is preserved during the evaluation of a valid program, as follows:

Theorem 1. *(Subject Reduction): If*

$$\begin{array}{c} \mathit{valid}(e) \quad P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash e : t \\ \mathit{\Gamma}, R, \varphi, \mathit{\Sigma} \models \langle \varpi, \mathit{\Pi} \rangle \\ \langle \varpi, \mathit{\Pi} \rangle [e] \hookrightarrow \langle \varpi', \mathit{\Pi}' \rangle [e'] \end{array}$$

then there exist $\mathit{\Sigma}'$, $\mathit{\Gamma}'$, R', and φ', such that

$$\begin{array}{c} (\mathit{\Sigma}' - (\mathit{lreg}(e') - \mathit{lreg}(e))) - (\mathit{lloc}(e) - \mathit{lloc}(e')) = \mathit{\Sigma} - (\mathit{lreg}(e) - \mathit{lreg}(e')) \\ \mathit{\Gamma}' - (\mathit{lvar}(e') - \mathit{lvar}(e)) = \mathit{\Gamma} - (\mathit{lvar}(e) - \mathit{lvar}(e')) \\ R' - (\mathit{lreg}(e') - \mathit{lreg}(e)) = R - (\mathit{lreg}(e) - \mathit{lreg}(e')) \\ \varphi' - (\mathit{lreg}(e') - \mathit{lreg}(e)) \Rightarrow \varphi - (\mathit{lreg}(e) - \mathit{lreg}(e')) \\ \mathit{\Gamma}', R', \varphi', \mathit{\Sigma}' \models \langle \varpi', \mathit{\Pi}' \rangle \\ \mathit{valid}(e') \quad P; \mathit{\Gamma}'; R'; \varphi'; \mathit{\Sigma}' \vdash e' : t. \end{array}$$

Proof: By structural induction on *e*. The detailed proof is in Appendix A.2.

Although the hypothesis of the above theorem contains an evaluation relation, the proof does not use the runtime checks associated with the evaluation rules to prove that the result of the evaluation (result and dynamic environment) is well-typed, valid and consistent.

The following theorem guarantees that the evaluation of a valid program cannot generate `danglingerr` errors, by proving that those runtime checks are redundant for a well-typed valid program (the runtime checks are proved by the static semantics).

Theorem 2. *(Progress) If*

$$valid(e) \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \models \langle \varpi, \Pi \rangle$$

then either

- *e is a value, or*
- $\langle \varpi, \Pi \rangle [e] \hookrightarrow \texttt{nullerr}$ *or*
- *there exist* ϖ', Π', e' *such that* $\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e']$.

Proof: By induction over the depth of the type derivation for expression *e*. The detailed proof is in Appendix A.2.

We conclude with the following soundness theorem for region annotated Core-Java. The theorem states that if a valid program is well-typed and is evaluated in a runtime environment consistent with the static environment, the result is (1) either an error different from a dangling error, (2) or a value, (3) or that the program diverges. The evaluation never reports dangling errors, namely the program never creates dangling references.

Theorem 3. *(Soundness) Given a well-typed valid Core-Java program* $P{=}def^*$ *and the main function* $(void\ \mathbf{main}(void)\{e_0\}){\in}P$*, where* e_0 *is a well-typed valid closed term (without free regions and free variables), such that* $retvars(e_0){=}\emptyset \wedge retregs(e_0){=}\emptyset$ *and* $P;\Gamma_0;R_0;\varphi_0;\Sigma_0 \vdash e_0 : void$*, where* $\Gamma_0{=}\emptyset$*,* $R_0{=}\emptyset$*,* $\varphi_0{=}true$*, and* $\Sigma_0{=}\emptyset$ *. Starting from the initial runtime environment* $\langle \varpi_0, \Pi_0 \rangle$*, where* $\varpi_0{=}[\,]$*,* $\Pi_0{=}\emptyset$*, such that* $\Gamma_0, R_0, \varphi_0, \Sigma_0 \models \langle \varpi_0, \Pi_0 \rangle$*. Then either*

(1) $$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \texttt{nullerr}$$

or there exist a store ϖ*, a variable environment* Π*, a value* δ*, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

(2) $$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle [\delta] \quad \Gamma, R, \varphi, \Sigma \models \langle \varpi, \Pi \rangle \quad P;\Gamma;R;\varphi;\Sigma \vdash \delta : void$$

or for a store ϖ*, a variable environment* Π*, a valid expression e, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

$$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle [e] \quad \Gamma, R, \varphi, \Sigma \models \langle \varpi, \Pi \rangle \quad P;\Gamma;R;\varphi;\Sigma \vdash e : void \quad valid(e)$$

there exist a store ϖ'*, a variable environment* Π'*, an expression* e'*, a type environment* Γ'*, a set of regions* R'*, a region constraint* φ'*, a store typing* Σ' *such that*

(3) $$\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e'] \quad \Gamma',R',\varphi',\Sigma' \models \langle \varpi', \Pi' \rangle \quad P;\Gamma';R';\varphi';\Sigma' \vdash e' : void \quad valid(e')$$

Proof: The proof is an induction on the number of the reduction steps. We can repeatedly use the progress theorem (Theorem 2) to prove that there is a reduction step and then the preservation theorem (Theorem 1) to prove that the runtime environment after evaluation is still well-typed and the evaluation result is valid.

6 Conclusion

We have considered a region calculus consisting of an object-oriented core language annotated with regions. We have defined the dynamic semantics for our region calculus based on a simpler small-step rewriting relation. Some of the region calculus constructions (e.g. `letreg`) are firstly evaluated to intermediate constructions. Therefore the static semantics must also be extended to include these new intermediate constructions. We have used a novel syntactic condition (*valid(e)*) to restrict the places where the intermediate constructions may occur in a program. This condition does not restrict source-level region calculus, since intermediate constructions are generated during the program evaluation. Our dynamic semantics is instrumented with runtime checks to guarantee that a special `danglingerr` error is reported whenever the program evaluation tries to create a dangling reference. We have defined an important consistency relationship between the static and dynamic semantics. A store typing technique is used to ensure that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. We have proven the soundness of the region calculus by using a syntactic proof method [24], based on subject reduction and progress. The subject reduction theorem ensures that the region type of a valid program is preserved during the evaluation. The progress theorem guarantees that the evaluation of a valid program cannot generate `danglingerr` errors (namely those runtime checks are redundant for a well-typed valid program). We have proven both theorems in a modular fashion using just a simple induction. This simpler soundness proof adds confidence to our region-based memory inference and execution systems.

References

1. David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–298, 2003.
2. L. Birkedal and M. Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258(1–2):299–392, 2001.
3. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
4. Gerard Boudol. Typing safe deallocation. In *European Symposium on Programming (ESOP)*, pages 116–130, 2008.
5. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, 2003.
6. C. Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 155–165, 2001.

7. C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.
8. Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin C. Rinard. Region inference for an object-oriented language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 243–254, 2004.
9. M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.
10. Florin Craciun, Hong Yaw Goh, and Wei-Ngan Chin. A framework for object-oriented program analyses via Core-Java. In *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 197–205, Cluj-Napoca, Romania, 2006.
11. Martin Elsman. Garbage collection safety for region-based memory management. In *ACM Workshop on Types in Language Design and Implementation (TLDI)*, pages 123–134, 2003.
12. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.
13. S. Helsen. *Region-Based Program Specialization*. PhD thesis, Universität Freiburg, 2002.
14. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. *Electronic Notes in Theoretical Computer Science*, 41(3), 2000.
15. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 132–146, 1999.
16. Gregory Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.
17. J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 66–77, 1995.
18. H. Niss. *Regions are imperative. Unscoped regions and control-sensitive memory management*. PhD thesis, University of Copenhagen, 2002.
19. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
20. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, 1998.
21. M. Tofte and J. Talpin. Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 188–201, 1994.
22. M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
23. Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management (IWMM)*, pages 1–42, 1992.
24. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information Computation*, 115(1):38–94, 1994.

A Proof Details

A.1 Auxiliary Definitions and Lemmas

Lemma 1. *Suppose* $P; \Gamma; R; \varphi; \Sigma \vdash e : t$*. If* $\rho = [(r_i \mapsto a_i)_{1..p}]$*, and for all* $i{=}1..p$*, either* $a_i \notin R$ *or* $\varphi {\Rightarrow} (r_i{=}a_i)$*, then* $P; \rho\Gamma; \rho R; \rho\varphi; \rho\Sigma \vdash \rho e : \rho t$*.*

Proof: By structural induction on e.

Lemma 2. *Suppose* $\varphi_1 \Rightarrow \varphi_2$.
If $a \notin \mathit{reg}(\varphi_1 \wedge \varphi_2)$*, then*

1. $(r \succeq a \wedge \varphi_1) \Rightarrow (r \succeq a \wedge \varphi_2)$.
2. $(a \succeq r \wedge \varphi_1) \Rightarrow (a \succeq r \wedge \varphi_2)$.

Proof: By induction on the form of a region constraint.

Definition 12. *Given a region constraint* $\varphi = \bigwedge_i (r_i \succeq r'_i) \wedge \bigwedge_j (r_j = r'_j)$ *and a set of regions* R*, we define the following notations:*

$r \succeq r' \in \varphi$ *iff* $\exists i$ *such that* $r = r_i \wedge r' = r'_i$
$r = r' \in \varphi$ *iff* $\exists j$ *such that* $r = r_j \wedge r' = r'_j$
$\mathit{TransClosure}(\varphi) =_{\mathit{def}} \bigwedge_{i'} (r_{i'} \succeq r'_{i'}) \wedge \bigwedge_{j'} (r_{j'} = r'_{j'})$ *such that*
$i' \geq i \wedge j' \geq j$
$\forall r_1, r_2, r_3 \cdot r_1 \succeq r_2 \in \mathit{TransClosure}(\varphi) \wedge r_3 \succeq r_1 \in \mathit{TransClosure}(\varphi) \Rightarrow r_3 \succeq r_2 \in \mathit{TransClosure}(\varphi)$
$\forall r_1, r_2, r_3 \cdot r_1 = r_2 \in \mathit{TransClosure}(\varphi) \wedge r_3 = r_1 \in \mathit{TransClosure}(\varphi) \Rightarrow r_3 = r_2 \in \mathit{TransClosure}(\varphi)$
$\varphi - R$ *or* $\varphi \backslash R =_{\mathit{def}} \varphi'$ *such that*
$\forall r \succeq r' \in \mathit{TransClosure}(\varphi) \wedge r \notin R \wedge r' \notin R \Rightarrow r \succeq r' \in \varphi'$
$\forall r = r' \in \mathit{TransClosure}(\varphi) \wedge r \notin R \wedge r' \notin R \Rightarrow r = r' \in \varphi'$

Lemma 3. *Suppose* $\varphi_1 \Rightarrow \varphi_2$ *and a region* r.

1. $\varphi_1 - \{r\} \Rightarrow \varphi_2 - \{r\}$.
2. *If* $r \notin \mathit{reg}(\varphi_2)$*, then* $\varphi_1 - \{r\} \Rightarrow \varphi_2$.

Proof: By case analysis on φ_1 and φ_2.

Lemma 4. *Suppose* $P; R; \varphi \vdash_{\mathit{type}} t$.

1. *If* $r \notin R$*, then* $P; R \cup \{r\}; \varphi \vdash_{\mathit{type}} t$.
2. *If* $\varphi' \Rightarrow \varphi$*, then* $P; R; \varphi' \vdash_{\mathit{type}} t$.

Proof: By structural induction on the $\vdash_{\mathit{type}}$ derivation.

Lemma 5. *Suppose* $P; R; \varphi \vdash t_1 <: t_2$.

1. *If* $r \notin R$*, then* $P; R \cup \{r\}; \varphi \vdash t_1 <: t_2$.
2. *If* $\varphi' \Rightarrow \varphi$*, then* $P; R; \varphi' \vdash t_1 <: t_2$.
3. *If* $r \in R$*,* $r \notin \mathit{reg}(t_1)$*, and* $r \notin \mathit{reg}(t_2)$*, then* $P; R - \{r\}; \varphi - \{r\} \vdash t_1 <: t_2$.

Proof: By structural induction on the subtyping derivation using the Lemma 4.

Lemma 6. *Suppose* $P; \Gamma; R; \varphi; \Sigma \vdash e : t$.

1. *If* $v \notin \mathit{dom}(\Gamma)$*, then* $P; \Gamma + (v : t_1); R; \varphi; \Sigma \vdash e : t$.
2. *If* $\varphi' \Rightarrow \varphi$*, then* $P; \Gamma; R; \varphi'; \Sigma \vdash e : t$.
3. *If* $r \notin R$*, then* $P; \Gamma; R \cup \{r\}; \varphi; \Sigma + r \vdash e : t$.
4. *If* $(r, o) \notin \Sigma$ *and* $r \in R$*, then* $P; \Gamma; R; \varphi; \Sigma + ((r, o) : t_1) \vdash e : t$.

Proof: By structural induction on e.

Lemma 7. *Suppose $P;\Gamma;R;\varphi;\Sigma\vdash e:t$.*

1. *If $v\in dom(\Gamma)$ and $v\notin vars(e)$, then $P;\Gamma{-}\{v\};R;\varphi;\Sigma\vdash e:t$. where $vars(e)$ is defined by Definition 3.*

Proof: By structural induction on e.

Lemma 8. *Suppose $P;\Gamma;R{\cup}\{a\};\varphi;\Sigma\vdash e:t$.*

1. *If $a\notin reg(\Gamma)$, and $a\notin regs(e)$, then $P;\Gamma;R;\varphi{-}a;\Sigma{-}a\vdash e:t$. where $regs(e)$ is defined by Definition 5.*

Proof: By structural induction on e.

Lemma 9. *Suppose an expression e.*

1. *If $retvars(e){=}\emptyset$ and $retregs(e){=}\emptyset$ then $valid(e)$ holds.*
2. *If $retvars(e){=}\emptyset$ then $lvar(e){=}\emptyset$.*
3. *If $retregs(e){=}\emptyset$ then $lreg(e){=}\emptyset$.*

Proof: By structural induction on e.

Lemma 10. *(Canonical Forms)*
Suppose $P;\Gamma;R;\varphi;\Sigma\vdash\delta:t$ and $\Gamma,R,\varphi,\Sigma\models\langle\varpi,\Pi\rangle$. Then:

1. *if $t=void$ then $\delta=()$.*
2. *if $t=boolean$ then either $\delta=true$ or $\delta=false$.*
3. *if $t=int$ then $\delta=i$ for some integer i.*
4. *if $t=\bot$ then $\delta=$ `null`.*
5. *if $t=cn\langle r_{1..n}\rangle$ then*
   - *either the value is a location, $\delta=(r_1,o)$. The content of that location is an object value $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$ that is well-typed such that $P;\Gamma;R;\varphi;\Sigma{\vdash}cn\langle r_{1..n}\rangle(V){:}cn\langle r_{1..n}\rangle$ (it contains the fields and the methods of the class cn according to the program P).*
   - *or the value is $\delta=$ `null`*

Proof: By the definition of values and inspection of type checking rules.

Lemma 11. *Given any source language Core-Java program, P.*
Suppose $\vdash P\Rrightarrow P'$.

1. *If $\tau\neq Object$ and $\vdash\tau{\Rrightarrow}t,\varphi$, then $reg(t)\subseteq reg(\varphi)$.*
2. *Given any $t\in P'$ and $t'\in P'$.*
   If $\vdash t{<:}t'{\Rrightarrow}\varphi$, then $(reg(t){\cup}reg(t'))\subseteq reg(\varphi)$.

3. *Given any source language Core-Java expression, e.*
   If $\Gamma \vdash e \Rrightarrow e'{:}t,\ \varphi$, then $reg(t) \subseteq (regs(e') \cup reg(\Gamma) \cup reg(\varphi))$.

   Proof:

1. Since φ is the region class invariant of a class type. By induction on the inference rules [$\textsc{ri-class-1}$] and [$\textsc{ri-class-2}$] we can prove that the region class invariant always contain all the regions of a region type. An exception is the region type $\text{Object}\langle r\rangle$ since its invariant is just *true*.
2. Using the case (1) we can prove the conclusion for all t and t' such that $t \neq \text{Object}\langle r\rangle$ and $t' \neq \text{Object}\langle r\rangle$. However the first region of a region type is always used by the region constraint φ_0 of a region subtyping relation $\vdash t{<:}t',\ \varphi_0$ (see rules of Table 3). Since the exceptional case $\text{Object}\langle r\rangle$ contains only one region, the conclusion is proved.
3. By structural induction on e. The proof is straightforward by inspection of the type inference rules.

A.2 Proof of Theorem 1 (Subject Reduction)

By structural induction on e.

Case: v

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the consistency relation of the hypothesis, as $\Pi(v)$ and v have the same type $\Gamma(v)$. The validity is straightforward proved.

Case: $v.f$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. From the operational semantics $\Pi(v) = (r, o)$. By the consistency relation of the hypothesis (r, o), v, and $\varpi(r)(o)$ have the same type $\Sigma(r)(o){=}cn\langle a^+\rangle$. Note that the type of a location is the type of its content (by the rule [$\textsc{rc-location}$]). Using the hypothesis of type rule [$\textsc{rc-obj}$] for $\varpi(r)(o)$, we prove that the type of $V(f)$ is a subtype of the type of $v.f$. By subsumption the type judgment is proved. The validity is straightforward proved.

Case: $v = \delta$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. The type judgment is trivial as the type remains **void** as before the evaluation. We only have to prove the consistency relation for the updated variable environment Π'. By the type rule of the hypothesis the type of $\delta{=}\Pi'(v)$ is a subtype of the type of v. Using subsumption, we prove that v and $\Pi'(v)$ have the same type. The validity is straightforward proved.

Case: $v.f = \delta$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$.
The update on ϖ preserves the consistency relation except the object value $\varpi(r)(o)$. By the type rule of the hypothesis the type of value $\delta = (r_1, o_1)$ is a subtype of the type of the field $v.f$. Thus the type of $V(f)$ after updating is a subtype of the type of $v.f$. By the consistency relation of the hypothesis for the object value $\varpi(r)(o)$ before

updating combined with the previous subtyping relation for the updated field *V(f)* we can prove that object value after updating is still well typed. The type judgment is trivial as the type remains **void** as before the evaluation. The validity is straightforward proved.

Case: $\delta\,;e_2$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$.
By the validity from the hypothesis, $\mathit{valid}(\delta\,;e_2)$ we get that $\mathit{retvars}(e_2){=}\emptyset$ and $\mathit{retregs}(e_2){=}\emptyset$. Applying case (1), (2) and (3) of Lemma 9 we prove that $\mathit{valid}(e_2)$, $\mathit{lvar}(e_2){=}\emptyset$, and $\mathit{lreg}(e_2){=}\emptyset$. Note that $\mathit{lloc}(\delta) = \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis.

Case: $\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. By the validity from the hypothesis, $\mathit{valid}(\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)$ we get that $\mathit{retvars}(e_1){=}\emptyset$, $\mathit{retregs}(e_1){=}\emptyset$, $\mathit{retvars}(e_2){=}\emptyset$, and $\mathit{retregs}(e_2){=}\emptyset$. Applying case (1), (2) and (3) of Lemma 9 we prove that $\mathit{valid}(e_1)$, $\mathit{lvar}(e_1){=}\emptyset$, $\mathit{lreg}(e_1){=}\emptyset$, $\mathit{valid}(e_2)$, $\mathit{lvar}(e_2){=}\emptyset$, and $\mathit{lreg}(e_2){=}\emptyset$.
Note that $\mathit{lloc}(\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) = \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis and the subsumption.

Case: $\mathbf{while}\ v\ e$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. By the validity from the hypothesis, $\mathit{valid}(\mathbf{while}\ v\ e)$ we get that $\mathit{retvars}(e){=}\emptyset$, $\mathit{retregs}(e){=}\emptyset$. Applying case (1), (2) and (3) of Lemma 9 we prove that $\mathit{valid}(e\,;\mathbf{while}\ v\ e)$, $\mathit{lvar}(e\,;\mathbf{while}\ v\ e){=}\emptyset$, and $\mathit{lreg}(e\,;\mathbf{while}\ v\ e){=}\emptyset$. Note that $\mathit{lloc}(\mathbf{while}\ v\ e) = \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis. The second case of the loop evaluation rule (when the condition is false) is straightforward proved.

Case: $\mathbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_{1..p})$

We let $\Sigma' = \Sigma + \{(r_1, o) : \mathit{cn}\langle r_{1..n}\rangle\},\ \Gamma' = \Gamma,\ \varphi' = \varphi,\ R' = R$.
Note that $\mathit{lloc}(\mathbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_{1..p})){=}(r_1, o)$, while the functions *lvar* and *lreg* return $\emptyset$ for both $\mathbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_{1..p})$ and (r_1, o). The conclusion's type judgment is straightforward proved as the type of new location is given by the Σ'.
The store is extended with one more location (r_1, o) and $\mathit{location\_dom}(\varpi'){=}$ $\mathit{location\_dom}(\varpi)\cup\{(r_1, o)\}{=}\mathit{location\_dom}(\Sigma)\cup\{(r_1, o)\}{=}\mathit{location\_dom}(\Sigma')$. In order to prove the conclusion's consistency relation we use the case (4) of Lemma 6 to extend the typing relations of the hypothesis and the hypothesis's consistency relation. The object value from the new location is proved to be well typed by reconstructing the hypotheses of the type rule $[\textsc{objVal}]$ as follows: $P; R; \varphi \vdash_{\mathit{type}} \mathit{cn}\langle r_{1..n}\rangle$ is proved by the type rule $[\textsc{new}]$ from hypothesis; by the evaluation rule $V(f_i) = \Pi(v_i)$, while from the hypothesis consistency relation $P; \Gamma; R; \varphi; \Sigma \vdash \Pi(v_i) : \Gamma(v_i)$; using the hy-

pothesis type judgment [NEW] we get $t'_i = \Gamma(v_i)$ and $P; R; \varphi \vdash t'_i <: t_i$. The validity is straightforward proved,

Case: $\{(t\,v)\ \ e\}$

We let $\Sigma' = \Sigma$, $\Gamma' = \Gamma + (v : t)$, $\varphi' = \varphi$, and $R' = R$. By the validity from the hypothesis, $\mathit{valid}(\{(t\,v)\ \ e\})$ we get that $\mathit{retvars}(e){=}\emptyset$ and $\mathit{retregs}(e){=}\emptyset$. Applying the cases (1), (2) and (3) of Lemma 9 we prove that $\mathit{valid}(\texttt{ret}(v, e))$, $\mathit{lvar}(\texttt{ret}(v, e)){=}\{v\}$, and $\mathit{lreg}(\texttt{ret}(v, e)){=}\emptyset$. Note that $\mathit{lloc}(\{(t\,v)\ \ e\}) = \emptyset$. By the hypothesis's type judgment [EB], the conclusion's type judgment [RET] is proved.

We prove that $P; \Gamma; R; \varphi; \Sigma \vdash \Pi(v) : \Gamma(v)$ as follows: $\Pi(v) = \mathbf{init}(t)$ and the type of $\mathbf{init}(t)$ is a subtype of t, while $\Gamma(v) = t$. By the hypothesis's consistency relation we get that $\mathit{reg}(\Gamma){\subseteq}R$. By the hypothesis's type judgment [EB], we get that $P; R; \varphi \vdash_{\mathit{type}} t'$, that ensures $\mathit{reg}(t){\subseteq}R$. Thus $\mathit{reg}(\Gamma + (v : t)){\subseteq}R$. Then, by the consistency relation of the hypothesis and by the case (1) of Lemma 6 the rest of the conclusion's consistency relation is proved.

Case: $\texttt{ret}(v, \delta)$

We let $\Sigma' = \Sigma$, $\varphi' = \varphi$, $R' = R$, and $\Gamma' = \Gamma - \{v\}$. Note that $\mathit{lvar}(\texttt{ret}(v, \delta)){=}\{v\}$, $\mathit{lreg}(\texttt{ret}(v, \delta)){=}\emptyset$, $\mathit{lloc}(\texttt{ret}(v, \delta)) = \emptyset$, $\mathit{lvar}(\delta){=}\emptyset$, and $\mathit{lreg}(\delta){=}\emptyset$. The valid relation $\mathit{valid}(\delta)$ holds. By the hypothesis's type judgment and consistency and the case (1) of Lemma 7 (the variable v is not used neither in δ nor by object values) the type judgment and the consistency of the conclusion are straightforward proved.

Case: $\mathbf{letreg}\ r\ \mathbf{in}\ e$

We use Lemma 1 for region substitutions. For simplicity, we consider that the region substitution is already done both for static and dynamic environment.

We let $R' = R \cup \{a\}$, $\Sigma' = \Sigma{+}a$, $\Gamma' = \Gamma$, and $\varphi' = (\varphi \wedge \bigwedge_{r \in R}(r \succeq a))$. By the hypothesis's valid relation $\mathit{valid}(\mathbf{letreg}\ r\ \mathbf{in}\ e)$ we get that $\mathit{retvars}(e){=}\emptyset$, $\mathit{retregs}(e){=}\emptyset$. Applying the cases (1), (2) and (3) of Lemma 9 we prove that $\mathit{valid}(\texttt{retr}(a, e))$, $\mathit{lvar}(\texttt{retr}(a, e)){=}\emptyset$, and $\mathit{lreg}(\texttt{retr}(a, e)){=}\{a\}$. Note that $\mathit{lloc}(\mathbf{letreg}\ r\ \mathbf{in}\ e) = \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. We prove the conclusion's type judgment $P;\Gamma';R';\varphi';\Sigma'{\vdash}\texttt{retr}(a, e){:}t$ as follows: $R_t = R$; by the hypothesis's consistency relation we get that $\mathit{reg}(\Gamma){\subseteq}R$; by the hypothesis's type judgment [LETR] we prove that $\mathit{reg}(t) \subseteq R$ and $P; \Gamma; R'; \varphi' \vdash e : t$; the entailment $\varphi'{\Rightarrow}\bigwedge_{r \in R}(r{\succeq}a)$ is straightforward. By the hypothesis's consistency relation, $\mathit{ord}(\varpi){\Rightarrow}\varphi$ and Lemma 2 we prove that $\mathit{ord}([a{\mapsto}\emptyset]\varpi){\Rightarrow}\varphi'$. The remaining part of the conclusion's consistency follows directly from the hypothesis's consistency using the cases (3) and (4) of Lemma 6.

Case: $\texttt{retr}(a, \delta)$

We let $R' = R{-}\{a\}$, $\Gamma' = \Gamma$, $\varphi' = \varphi{-}a$, $\Sigma' = \Sigma{-}\{a\}$. The validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. By the type judgment of the hypothesis and the case (1) of Lemma 8 the type judgment of the conclusion is proved.

The consistency is proved as follows:

By the hypothesis's consistency and $\mathit{reg}(\Gamma){\subseteq}R{-}\{a\}$ (from the hypothesis type judgment [RETR]) the type of each $v \in \Pi'$ does not contain the region a since that type is given by $\Gamma(v)$. But $\Pi(v)$ is either a location or a constant. Note that the type of a location always contains the region's location. Hence $\mathit{vars}(\Pi'(v)){\subseteq}R{-}\{a\}$. Thus we can apply the case (1) of Lemma 8 to prove that $\mathit{\Pi}$ is well typed when the region a is deallocated. Note that this means that there are not any references from the program variable environment to the deallocated region.

By the consistency and type judgment of the hypothesis, each object value of the store ϖ is well typed. By the type rule [$\textsc{ObjVal}$], an object value can have references only to regions older that the region of the current location. We can use the case (1) of Lemma 8 to type the store ϖ. By the consistency of the hypothesis and Lemma 3 we can prove that $\mathit{ord}(\varpi){\Rightarrow}\varphi'$. As we mentioned before, the hypothesis's type judgment [$\textsc{retr}$] ensures that $\mathit{reg}(\mathit{\Gamma}){\subseteq}R{-}\{a\}$.

Case: $v'_0.\mathit{mn}\langle a^+\rangle(v'_{1..p})$

According to the evaluation rule ([$\textsc{d-invoke}$]), the actual type of the object stored at the location given by $\mathit{\Pi}(v'_0)$ is $\mathit{cn}\langle a_{1..n}\rangle$, which is a subtype of the expected type of v'_0, say $\mathit{cn}'\langle a_{1..m}\rangle$ at compile time.

We let $R' = R$, $\mathit{\Gamma}' = \mathit{\Gamma}{+}\{(n_i : \mathit{\Gamma}(v'_i))_{i:0..p}\}$, $\varphi' = \varphi{\wedge}\mathit{inv.cn}\langle a_{1..n}\rangle$, and $\mathit{\Sigma}' = \mathit{\Sigma}$.

Since P is a valid program, the method's body e is valid. Hence we get that $\mathit{valid}(\mathit{ret}(n_{0..p}, e))$, $\mathit{lreg}(\mathit{ret}(n_{0..p}, e)) = \emptyset$ and $\mathit{lvar}(\mathit{ret}(n_{0..p}, e)) = \{n_{0..p}\}$.

Note that $\mathit{lloc}(\mathit{ret}(n_{0..p}, e)) = \emptyset$. The conclusion's consistency relation is proved as follows:

Since the hypothesis $reg(\mathit{\Gamma}){\subseteq}R$, we get that also $reg(\mathit{\Gamma}{+}\{(n_i : \mathit{\Gamma}(v'_i))_{i:0..p}\})$ holds. By the consistency of the hypothesis and the cases (1) and (2) of Lemma 6, we can prove that $\mathit{\Pi}'$ and ϖ are well typed (type environment is extended according to the program environment extension). In order to prove that $\mathit{ord}(\varpi) \Rightarrow \varphi'$, we have the following two sub cases to prove: (a) $\mathit{ord}(\varpi) \Rightarrow \varphi$ that is true from the hypothesis's consistency and (b) $\mathit{ord}(\varpi) \Rightarrow \mathit{inv.cn}\langle a_{1..n}\rangle$ that is true because each object value of the store is well typed: class invariant is checked at object creation, in the type rule [$\textsc{rc-ObjVal}$] by the judgment $P; R; \varphi \vdash_{\mathit{type}} \mathit{cn}\langle r_{1..n}\rangle$ and from the hypothesis's consistency $R{=}\mathit{dom}(\varpi)$ and $\mathit{ord}(\varpi) \Rightarrow \varphi$. In order to prove the type judgment of the conclusion, $P; \mathit{\Gamma}'; R'; \varphi'; \mathit{\Sigma}' \vdash \mathit{ret}(n_{0..p}, e) : t$ we have to prove the assumptions of the rule [$\textsc{rc-meth}$]. By its definition $\mathit{\Gamma}'$ is well formed. By the hypothesis's type rule [$\textsc{rc-invoke}$] we get that all regions that annotate the method are in R; $\varphi{\Rightarrow}inv.cn'$ (the invariant of the superclass), and the fact the types of the method arguments are well formed $P; R; \varphi \vdash_{\mathit{type}} t_j,\ j = 0..p$. By type judgment of the hypothesis φ implies the precondition of superclass $\mathit{cn}'\langle a_{1..m}\rangle$ method, $\varphi{\Rightarrow}\mathit{pre.cn}'.\mathit{mn}$. But adding the subclass invariant to both sides of the entailment we have the following entailment: $\varphi'{\Rightarrow}\mathit{pre.cn}'.\mathit{mn}{\wedge}\mathit{inv.cn}\langle a_{1..n}\rangle$ By the soundness of the method overriding: $\mathit{pre.cn}'.\mathit{mn}{\wedge}\mathit{inv.cn}\langle a_{1..n}\rangle{\Rightarrow}\mathit{pre.cn.mn}$. Thus, the region constraint required by rule [$\textsc{rc-meth}$] is proved. Using $\mathit{\Gamma}'$, R' and φ' we can typecheck the method body. Type of the method body is a subtype of the expected type, thus we use the subsumption. We also used Lemma 1 for region substitutions. Hence both the type judgment and the consistency relation hold.

Case: $e_1\,;e_2$

By induction hypothesis for $\langle\varpi, \mathit{\Pi}\rangle[e_1]{\hookrightarrow}\langle\varpi', \mathit{\Pi}'\rangle[e'_1]$ there exist $\hat{\mathit{\Sigma}}$, $\hat{\mathit{\Gamma}}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e'_1)$, $\hat{\mathit{\Gamma}}, \hat{R}, \hat{\varphi}, \hat{\mathit{\Sigma}} \vDash \langle\varpi', \mathit{\Pi}'\rangle$, $reg(\hat{\mathit{\Gamma}}) \subseteq \hat{R}$,
$(\hat{\mathit{\Sigma}}{-}(\mathit{lreg}(e'_1){-}\mathit{lreg}(e_1))){-}(\mathit{lloc}(e_1){-}\mathit{lloc}(e'_1)) = \mathit{\Sigma}{-}(\mathit{lreg}(e_1){-}\mathit{lreg}(e'_1))$,
$\hat{\mathit{\Gamma}}{-}(\mathit{lvar}(e'_1){-}\mathit{lvar}(e_1)) = \mathit{\Gamma}{-}(\mathit{lvar}(e_1){-}\mathit{lvar}(e'_1))$,

$\hat{R}-(lreg(e_1')-lreg(e_1)) = R-(lreg(e_1)-lreg(e_1'))$, and
$\hat{\varphi}-(lreg(e_1')-lreg(e_1)) \Rightarrow \varphi-(lreg(e_1)-lreg(e_1'))$.
We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(e_1; e_2)$ we get that $retregs(e_2) = \emptyset$, $retvars(e_2) = \emptyset$, $retregs(e_1) \cap regs(e_2) = \emptyset$, $retvars(e_1) \cap vars(e_2) = \emptyset$, and $valid(e_1)$. Then, by the Lemma 9 we get that $valid(e_2)$, $lvar(e_2) = \emptyset$, and $lreg(e_2) = \emptyset$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. In order to prove that $P; \Gamma'; R'; \varphi'; \Sigma' \vdash e_1'; e_2 : t_2$, we have to prove that $P; \Gamma'; R'; \varphi'; \Sigma' \vdash e_2 : t_2$. Note that the hypothesis contains the type judgment $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ We also have to prove that $valid(e_1'; e_2)$ holds. We use a case based analysis on the expression e_1. We discuss only the main sub cases that change either ϖ or Π (the other cases are straightforward):

- $e_1 = \mathtt{retr}(a, \delta)$
  $\hat{R} = R-\{a\}$, $\hat{\Gamma} = \Gamma$, $\hat{\varphi} = \varphi-a$, $\hat{\Sigma} = \Sigma-\{a\}$. From hypothesis $reg(\Gamma) \subseteq R-\{a\}$ and $a \notin regs(e_2)$. Applying Lemma 8 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward.
- $e_1 =$ **letreg** r **in** e
  $\hat{R} = R \cup \{a\}$, $\hat{\Sigma} = \Sigma + a$, $\hat{\Gamma} = \Gamma$, and $\hat{\varphi} = (\varphi \wedge \bigwedge_{r \in R}(r \succeq a))$. Note that a is a fresh region. Applying the cases (2) and (3) of Lemma 6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward since a is a fresh region.
- $e_1 = \mathtt{ret}(v, \delta)$
  $\hat{\Sigma} = \Sigma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$, and $\hat{\Gamma} = \Gamma-\{v\}$.
  By the hypothesis's valid relation we get that $retvars(\mathtt{ret}(v, \delta)) \cap vars(e_2) = \emptyset$, hence $v \notin vars(e_2)$. Applying the case (1) of Lemma 7 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward.
- $e_1 = \{(t\, v)\ \ e\}$
  $\hat{\Sigma} = \Sigma$, $\hat{\Gamma} = \Gamma + (v : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v is a fresh variable. Applying the case (1) of Lemma 6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward since v is a fresh variable.
- $e_1 =$ **new** $cn\langle r_{1..n}\rangle(v_{1..p})$
  $\hat{\Sigma} = \Sigma + \{(r_1, o) : cn\langle r_{1..n}\rangle\}$, $\hat{\Gamma} = \Gamma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$.
  Applying the case (4) of Lemma 6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward.
- $e_1 = v_0'.mn\langle a^+\rangle(v_{1..p}')$
  $\hat{R} = R$, $\hat{\Gamma} = \Gamma + \{(n_i : \Gamma(v_i'))_{i:0..p}\}$, $\hat{\varphi} = \varphi \wedge inv.cn\langle a_{1..n}\rangle$, and $\hat{\Sigma} = \Sigma$. Note that all n_i variables are fresh variables. Applying the cases (1) and (2) of Lemma 6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward since n_i are fresh variables and the method body is a valid block expression.

Case: $lhs = e$
By induction hypothesis for $\langle\varpi, \Pi\rangle[e] \hookrightarrow \langle\varpi', \Pi'\rangle[e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma}, \hat{R}, \hat{\varphi}, \hat{\Sigma} \vDash \langle\varpi', \Pi'\rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$,
$(\hat{\Sigma}-(lreg(e')-lreg(e)))-(lloc(e)-lloc(e')) = \Sigma-(lreg(e)-lreg(e'))$,
$\hat{\Gamma}-(lvar(e')-lvar(e)) = \Gamma-(lvar(e)-lvar(e'))$,

$\hat{R}-(lreg(e')-lreg(e)) = R-(lreg(e)-lreg(e'))$, and
$\hat{\varphi}-(lreg(e')-lreg(e)) \Rightarrow \varphi-(lreg(e)-lreg(e'))$.
We let $R' = \hat{R}$, $\mathit{\Gamma}' = \hat{\mathit{\Gamma}}$, $\varphi' = \hat{\varphi}$, and $\mathit{\Sigma}' = \hat{\mathit{\Sigma}}$. From the hypothesis's valid relation $valid(lhs = e)$ we get that $retvars(e) \cap vars(lhs) = \emptyset$. Hence, the conclusion's relations between $\mathit{\Gamma}$ and $\mathit{\Gamma}'$, $\mathit{\Sigma}$ and $\mathit{\Sigma}'$, R and R', and φ and φ' are straightforward proved. In order to prove that $P; \mathit{\Gamma}'; R'; \varphi'; \mathit{\Sigma}' \vdash lhs{=}e' : \textbf{void}$, we have to prove that $P; \mathit{\Gamma}'; R'; \varphi'; \mathit{\Sigma}' \vdash lhs : t$ and $P; R'; \varphi' \vdash t' <: t$, while $P; \mathit{\Gamma}'; R'; \varphi'; \mathit{\Sigma}' \vdash e' : t'$,
$P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$, and $P; R; \varphi \vdash t' <: t$ are given by the induction hypothesis. We also have to prove that $valid(lhs = e')$ holds. We use a case based analysis on the expression e. We discuss only the main sub cases that change either ϖ or $\mathit{\Pi}$ (the other cases are straightforward):

- $e = \mathtt{retr}(a, \delta)$
  $\hat{R} = R-\{a\}$, $\hat{\mathit{\Gamma}} = \mathit{\Gamma}$, $\hat{\varphi} = \varphi-a$, $\hat{\mathit{\Sigma}} = \mathit{\Sigma}-\{a\}$. From hypothesis $reg(\mathit{\Gamma}) \subseteq R-\{a\}$. Since $lhs = v \mid v.f$ we get that $regs(lhs) = \emptyset$. Applying Lemma 8 on
  $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. Since $a \notin reg(t)$ and $a \notin reg(t')$, applying the case (3) of Lemma 5 on $P; R; \varphi \vdash t' <: t$ we prove the subtype judgment. The valid relation is straightforward.
- $e = \textbf{letreg}\, r\, \textbf{in}\, e_1$
  $\hat{R} = R \cup \{a\}$, $\hat{\mathit{\Sigma}} = \mathit{\Sigma}{+}a$, $\hat{\mathit{\Gamma}} = \mathit{\Gamma}$, and $\hat{\varphi} = (\varphi \wedge \bigwedge_{r \in R}(r \succeq a))$. Note that a is a fresh region. Applying the cases (2) and (3) of Lemma 6 on $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. Applying the cases (1) and (2) of Lemma 5 on $P; R; \varphi \vdash t' <: t$ we prove the subtype judgment. From the hypothesis valid relation $valid(\textbf{letreg}\, r\, \textbf{in}\, e_1)$ we get that $retvars(e_1) = \emptyset$. By the induction hypothesis we get that $valid(\mathtt{retr}(a, e_1))$. Hence $valid(lhs = \mathtt{retr}(a, e_1))$ holds.
- $e = \mathtt{ret}(v, \delta)$
  $\hat{\mathit{\Sigma}} = \mathit{\Sigma}$, $\hat{\varphi} = \varphi$, $\hat{R} = R$, and $\hat{\mathit{\Gamma}} = \mathit{\Gamma}-\{v\}$. By the hypothesis's valid relation $retvars(\mathtt{ret}(v, \delta)) \cap vars(lhs) = \emptyset$, hence $v \notin vars(lhs)$. Applying the case (1) of Lemma 7 on $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. The subtype judgment is the same as that of the hypothesis. The valid relation is straightforward.
- $e = \{(t\, v)\ \ e_1\}$
  $\hat{\mathit{\Sigma}} = \mathit{\Sigma}$, $\hat{\mathit{\Gamma}} = \mathit{\Gamma}{+}(v : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v is a fresh variable. By the hypothesis valid relation we get $valid((t\, v)\ \ e_1\})$ and then $retvars(e_1) = \emptyset$. Since v is a fresh variable, $valid(lhs = \mathtt{ret}(v, e_1))$ holds. Applying the case (1) of Lemma 6 on $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. The subtype judgment is the same as that of the hypothesis.
- $e = \textbf{new}\; cn\langle r_{1..n}\rangle(v_{1..p})$
  Applying the case (4) of Lemma 6 on $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. The valid relation and the subtype judgment are straightforward.
  $\hat{\mathit{\Sigma}} = \mathit{\Sigma} + \{(r_1, o) : cn\langle r_{1..n}\rangle\}$, $\hat{\mathit{\Gamma}} = \mathit{\Gamma}$, $\hat{\varphi} = \varphi$, $\hat{R} = R$.
- $e = v'_0.mn\langle a^+\rangle(v'_{1..p})$
  $\hat{R} = R$, $\hat{\mathit{\Gamma}} = \mathit{\Gamma}{+}\{(n_i : \mathit{\Gamma}(v'_i))_{i:0..p}\}$, $\hat{\varphi} = \varphi \wedge inv.cn\langle a_{1..n}\rangle$, and $\hat{\mathit{\Sigma}} = \mathit{\Sigma}$. Note that all n_i variables are fresh variables. Applying the cases (1) and (2) of Lemma 6 on $P; \mathit{\Gamma}; R; \varphi; \mathit{\Sigma} \vdash lhs : t$ we prove the type judgment. Applying the case (2) of Lemma 5 on $P; R; \varphi \vdash t' <: t$ we prove the subtype judgment. The valid relation is straightforward since n_i are fresh variables and the method body is a valid block expression.

Case: $\mathtt{ret}(v, e)$

By induction hypothesis for $\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma}, \hat{R}, \hat{\varphi}, \hat{\Sigma} \vDash \langle \varpi', \Pi' \rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$, $P; \hat{\Gamma}; \hat{R}; \hat{\varphi}; \hat{\Sigma} \vdash e' : t$
$(\hat{\Sigma} - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$,
$\hat{\Gamma} - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$,
$\hat{R} - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$, and
$\hat{\varphi} - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$.
We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(\mathtt{ret}(v, e))$ we get that $v \notin retvars(e)$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. By the hypothesis's type judgment $P; \Gamma; R; \varphi; \Sigma \vdash \mathtt{ret}(v, e) : t$ we get that $v \in \Gamma$. In order to prove the type judgment $P; \Gamma'; R'; \varphi'; \Sigma' \vdash \mathtt{ret}(v, e') : t$ we only have to prove that $v \in \Gamma'$, since $P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : t$ is given by the induction hypothesis. In order to prove the valid relation $valid(\mathtt{ret}(v, e'))$ we have to prove that $v \notin retvars(e')$ since $valid(e')$ is given by the induction hypothesis. We use a case based analysis on the expression e. We discuss only the main sub cases that change Π (the other cases are straightforward):

- $e = \mathtt{ret}(v', \delta)$
  $\hat{\Sigma} = \Sigma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$, and $\hat{\Gamma} = \Gamma - \{v'\}$. From hypothesis $v \notin retvars(e)$ holds, hence $v \neq v'$. Therefore $v \in \hat{\Gamma}$ holds since from hypothesis we have that $v \in \Gamma$. The relation $v \notin retvars(\delta)$ is straightforward.
- $e = \{(t\,v')\ \ e_1\}$
  $\hat{\Sigma} = \Sigma$, $\hat{\Gamma} = \Gamma + (v' : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v' is a fresh variable. The relation $v \in \hat{\Gamma}$ holds since from hypothesis $v \in \Gamma$. From hypothesis $valid(\{(t\,v')\ \ e_1\})$ holds, therefore $retvars(e_1) = \emptyset$. Since v' is a fresh variable $v \neq v'$ holds. Hence the relation $v \notin retvars(ret(v', e_1))$ holds.
- $e = v'_0.mn\langle a^+ \rangle (v'_{1..p})$
  $\hat{R} = R$, $\hat{\Gamma} = \Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\}$, $\hat{\varphi} = \varphi \wedge inv.cn\langle a_{1..n} \rangle$, and $\hat{\Sigma} = \Sigma$. Note that all n_i variables are fresh variables. The relation $v \in \hat{\Gamma}$ holds since from hypothesis $v \in \Gamma$. From hypothesis the method body is a valid block expression, therefore there is not any ret in the method's body. Since n_i are fresh variables we get that $v \neq n_i\ i = 0..p$. Hence the relation $v \notin retvars(e')$ holds.

Case: $\mathtt{retr}(a, e)$

By induction hypothesis for $\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma}, \hat{R}, \hat{\varphi}, \hat{\Sigma} \vDash \langle \varpi', \Pi' \rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$, $P; \hat{\Gamma}; \hat{R}; \hat{\varphi}; \hat{\Sigma} \vdash e' : t$
$(\hat{\Sigma} - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$,
$\hat{\Gamma} - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$,
$\hat{R} - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$, and
$\hat{\varphi} - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$.
We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(\mathtt{retr}(a, e))$ we get that $a \notin retregs(e)$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. In order to prove the valid relation $valid(\mathtt{retr}(a, e'))$ we have to prove that $a \notin retregs(e')$ since $valid(e')$ is given by the induction hypothesis.
In order to prove the type judgment $P; \Gamma'; R'; \varphi'; \Sigma' \vdash \mathtt{retr}(a, e') : t$ we have to prove that $a \in R'$, $reg(t) \subseteq R' - lreg(e') - \{a\}$, $reg(\Gamma - lvar(e')) \subseteq R' - lreg(e') - \{a\}$,

and $\varphi' \Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r \succeq a)$, while $P;\Gamma';R';\varphi';\Sigma' \vdash e' : t$ is given by the induction hypothesis. We use a case based analysis on the expression e. We discuss only the main sub cases that change $lreg$, $lvar$ and $retregs$ (the other cases are straightforward):

- $e = \texttt{retr}(r,\delta)$ then $e' = \delta$.
  $R' = R - r$, $\varphi' = \varphi - r$, and $\Gamma' = \Gamma$. From hypothesis $a \notin retregs(e)$ holds, therefore $r \neq a$. The relation $a \notin retregs(e')$ is straightforward proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$. Applying the case (2) of Lemma 3 we get that $\varphi' \Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r \succeq a)$.
- $e = \textbf{letreg}\, r\, \textbf{in}\ e_1$ then $e' = \texttt{retr}(r', e_1)$ and r' is a fresh region.
  $R' = R \cup \{r'\}$, $\varphi' = (\varphi \wedge \bigwedge_{r\in R}(r \succeq r'))$, and $\Gamma' = \Gamma$. The region r' is a fresh region, therefore $r' \neq a$. From the hypothesis relation $valid(e)$, we get that $retregs(e_1) = \emptyset$. Hence, the relation $a \notin retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$.
  Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$. Since $\varphi'\Rightarrow\varphi$ we get that $\varphi'\Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r\succeq a)$.
- $e = \texttt{ret}(v',\delta)$ then $e' = \delta$.
  $R' = R$, $\varphi' = \varphi$, and $\Gamma' = \Gamma-\{v'\}$. The relation $a \notin retregs(e')$ is straightforward proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $\varphi\Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r\succeq a)$, therefore $\varphi'\Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r\succeq a)$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$.
- $e = \{(t\,v')\ \ e_1\}$ then $e' = \texttt{ret}(v_1, e_1)$ and v_1 is a fresh variable.
  $\Gamma' = \Gamma+(v_1 : t)$, $\varphi' = \varphi$, and $R' = R$. From the hypothesis's valid relation $valid(e)$, we get that $retregs(e_1) = \emptyset$ and $retvars(e_1) = \emptyset$. By the cases (2) and (3) of Lemma 9 we get that $lreg(e_1) = \emptyset$ and $lvar(e_1) = \emptyset$. Hence, the relation $a \notin retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$, therefore $\varphi' \Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r\succeq a)$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$, therefore $reg(\Gamma'-lvar(e'))\subseteq R'-lreg(e')-\{a\}$.

- $e = v'_0.mn\langle a^+\rangle(v'_{1..p})$ then $e' = \texttt{ret}(n_{1..p}, e_1)$, $n_{1..p}$ are fresh variables, and e_1 is a valid block expression such that $retvars(e_1) = \emptyset$ and $retregs(e_1) = \emptyset$.
  $R' = R$, $\Gamma' = \Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\}$, and $\varphi' = \varphi \wedge inv.cn\langle a_{1..n}\rangle$. By the cases (2) and (3) of Lemma 9 we get that $lreg(e_1)=\emptyset$ and $lvar(e_1)=\emptyset$. Hence, the relation $a \notin retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R' - lreg(e') - \{a\} = R - lreg(e) - \{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R - lreg(e) - \{a\}$, therefore $reg(t) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r \in (R - lreg(e) - \{a\})}(r \succeq a)$.
  Since $\varphi' \Rightarrow \varphi$, we get $\varphi' \Rightarrow \bigwedge_{r \in (R' - lreg(e') - \{a\})}(r \succeq a)$. From the hypothesis's type judgment we get that $reg(\Gamma - lvar(e)) \subseteq R - lreg(e) - \{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$.

□

A.3 Proof of Theorem 2 (Progress)

By structural induction over the depth of the type derivation for expression *e* and using the Lemma 10.

Cases: [RC−LOCATION, RC−ObjVal, RC−CONS1, RC−CONS2]
e is a value.

Case: [RC−VAR]
We let $\varpi' = \varpi$, $\Pi' = \Pi$, and $e' = \Pi(v)$. By hypothesis we get that $(v : t) \in \Gamma$ and $dom(\Gamma) = dom(\Pi)$, thus the check of the evaluation rule [D−VAR] does not fail.

Case: [RC−FD]
By the type judgment and the consistency of the hypothesis we get that $(v : cn\langle r_{1..n}\rangle) \in \Gamma$ and $\Pi(v) : \Gamma(v)$. According to the Lemma 10, there are two cases for $\Pi(v)$:

1. $\Pi(v) = \texttt{null}$ then the rule [D−NULLERR1] generates an error $\texttt{nullerr}$.
2. $\Pi(v) = (r_1, o)$, $\varpi(r_1)(o) = cn\langle r_{1..n}\rangle(V)$, and $P; \Gamma; R; \varphi; \Sigma \vdash cn\langle r_{1..n}\rangle(V) : cn\langle r_{1..n}\rangle$.
   We let $\varpi' = \varpi$, $\Pi' = \Pi$, and $e' = V(f)$. Then rule [D−FD] is used.

Case: [RC−ASSGN]
We deal with expression $lhs = e$. From type judgment of the hypothesis we have $P; \Gamma; R; \varphi; \Sigma \vdash e : t'$. By the induction hypothesis, we have the following cases:

1. $\langle \varpi, \Pi\rangle[e] \hookrightarrow \texttt{nullerr}$,
   then the error is propagated as $\langle \varpi, \Pi\rangle[lhs{=}e] \hookrightarrow \texttt{nullerr}$
2. $\langle \varpi, \Pi\rangle[e] \hookrightarrow \langle \hat{\varpi}, \hat{\Pi}\rangle[e']$.
   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $lhs = e'$. Then the evaluation rule [D−ASSGN1] is used.
3. *e* is a value $e = \delta$.
   There are the following two sub cases based on the form of $lhs = v \mid v.f$:

SubCase: $v = \delta$
By hypothesis's type judgment we get that $(v{:}t) \in \Gamma$ and $dom(\Gamma) = dom(\Pi)$, thus $v \in dom(\Pi)$. By the type judgment of the hypothesis, the type of δ is well-formed and is a subtype of type of *v*. If $\delta = (r_1, o_1)$ the type rule

[RC−LOCATION] ensures that $r_1 \in R$, but from the hypothesis' consistency $dom(\varpi) = R$. Thus $r_1 \in dom(\varpi)$. We let $\varpi' = \varpi$ and $\Pi' = \Pi + \{v \mapsto \delta\}$.
The evaluation rule [D−ASSGN2] can be applied, since we proved that its runtime checks hold. Note that the rule [D−ASSGN1−DANGLERR] is never used for a well typed expression.

SubCase: $v.f = \delta$

By the type judgment and the consistency of the hypothesis we get that $(v : cn\langle a_{1..n}\rangle) \in \Gamma$ and $\Pi(v) : \Gamma(v)$. According to the Lemma 10, there are two cases for $\Pi(v)$:

(a) $\Pi(v) = \texttt{null}$ then the rule [D−NULLERR2] generates an error `nullerr`.

(b) $\Pi(v) = (a_1, o)$ and $\varpi(a_1)(o) = cn\langle a_{1..n}\rangle(V)$.
By the hypothesis type rule [RC−ASSGN] we get that the type of δ is well-formed and is a subtype of type of $v.f$. If $\delta = (r_1, o_1)$, the subtyping rule of the type rule [RC−ASSGN] and the subtyping judgment [ObjRegSub] ensures that the first region of type of δ, r_1 outlives the first region of type of $v.f$ according to the region constraint φ. But from the consistency of hypothesis $ord(\varpi) \Rightarrow \varphi$ holds. Thus the rule [D−ASSGN3] can be applied, since we proved that its runtime checks hold.
Note that the rule [D−ASSGN3−DANGLERR] is never used for a well typed expression.

Case: [RC−NEW]

By the consistency of the hypothesis we get that $ord(\varpi) \Rightarrow \varphi$ and $R = dom(\varpi)$. By the type judgment $P; R; \varphi \vdash_{type} cn\langle r_{1..n}\rangle$ of the hypothesis we get that $\varphi \Rightarrow \varphi_{inv}$. Thus the first runtime check $ord(\varpi) \Rightarrow \varphi_{inv}$ holds. By the hypothesis's type judgment the type of each variable v_i is a subtype of the corresponding class field f_i type. By the subtyping judgment [ObjRegSub], the first region of the region type of v_i outlives the first region of the region type of f_i according to the region constraint φ. Thus the runtime check $\mathbf{ord}(\varpi) \Rightarrow (r'_i \succeq \mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i))$ holds for each object field f_i. Thus the rule [D−NEW] can be applied, since we proved that its runtime checks hold.
Note that the rule [D−NEW−DANGLERR] is never used for a well typed expression.

Case: [RC−EB]

We let $\varpi' = \varpi$, $\Pi' = \Pi + \{n \mapsto \mathbf{init}(t)\}$, and $e' = \texttt{ret}(n, e)$ where n is a fresh variable. Then the evaluation rule [D−EB] can be applied.

Case: [RC−RET]

We deal with $\texttt{ret}(v, e)$. Based on the expression e, there are two cases:

1. e is a value and then the rule [D−RET2] can be applied.
2. e is not a value. By the induction hypothesis, there are two sub cases:
   (a) $\langle \varpi, \Pi\rangle[e] \hookrightarrow \langle \hat{\varpi}, \hat{\Pi}\rangle[e']$.
   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $\texttt{ret}(v, e')$. The evaluation rule [D−RET1] can be applied.
   (b) $\langle \varpi, \Pi\rangle[e] \hookrightarrow \texttt{nullerr}$
   The error is propagated as $\langle \varpi, \Pi\rangle[\texttt{ret}(v, e)] \hookrightarrow \texttt{nullerr}$.

Case: [RC−LETR]

We let $\varpi' = [a \mapsto \emptyset]\varpi$ and $\Pi' = \Pi$. The evaluation rule [D−LETR] can be applied.

Case: [RC-RETR]

We deal with $\texttt{retr}(a, e)$. Based on the expression e, there are two sub cases:

1. e is a value such that $e{=}\delta$.

   In order to apply the rule [D-RETR2], we have to prove that its runtime checks are redundant. Note that $lreg(\delta) = \emptyset$ and $lvar(\delta) = \emptyset$. By the type judgment of the hypothesis we get that $a \in R$, but from the hypothesis's consistency we get that $R = dom(\varpi)$. Thus $a \in dom(\varpi)$. In addition, by the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r \in R_t} (r \succeq a)$ where $R_t{=}R{-}\{a\}$. By the hypothesis consistency we get that $ord(\varpi) \Rightarrow \varphi$. Thus, we proved that a is the region on the top of the stack ϖ such that $\varpi{=}[a{\mapsto}Rgn]\varpi'$. If $\delta = (r, o)$ then its type t contains the region r, but from the hypothesis's type judgment we get that $reg(t) {\subseteq} R{-}a$. Thus we proved that $r \in dom(\varpi')$. By the hypothesis's type judgment we get that $reg(\Gamma){\subseteq}R{-}a$, but from the consistency relation $dom(\Gamma) = dom(\Pi)$ holds and for each $v \in \Pi$ the type of $\Pi(v)$ is $\Gamma(v)$. Since the type of a location contains the location's region, we proved that $\forall v{\in}\Pi \cdot (\Pi(v){=}(r, o)) {\Rightarrow} (r{\in}dom(\varpi'))$. By the hypothesis's consistency relation and the type judgment for an object value [RC-ObjVal] we get the following relations for each location $(r_1, o){\in}dom(\varpi')$ with $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$: $r_i \succeq r_1, i = 2..n$ and for each field $f \in dom(V)$ its type is a subtype of the expected type given by $fieldlist(cn\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p}$, (that means the regions of its type are older than $r_1, .., r_n$). Since $r_1 \succeq a$ holds, we proved the last check (about ϖ') of the rule [D-RETR2]. Thus, we can apply the rule [D-RETR2], while the rule [D-RETR2-DANGLERR] is never used for a well typed program.
2. e is not a value. By the induction hypothesis, there are two sub cases:
   (a) $\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \hat{\varpi}, \hat{\Pi} \rangle [e']$.

   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $\texttt{retr}(a, e')$. The evaluation rule [D-RETR1] can be applied.

   (b) $\langle \varpi, \Pi \rangle [e] \hookrightarrow \texttt{nullerr}$

   The error is propagated as $\langle \varpi, \Pi \rangle [\texttt{retr}(a, e)] \hookrightarrow \texttt{nullerr}$.

Case: [RC-IF]

By the hypothesis's type judgment the type of v is $boolean$. According to the Lemma 10, there are two cases: either v is *true* and the rule [D-IF1] is applied, or v is *false* and the rule [D-IF2] is applied.

Case: [RC-LOOP]

By the hypothesis's type judgment the type of v is $boolean$. According to the Lemma 10, there are two cases: either v is *true* and the rule [D-LOOP1] is applied, or v is *false* and the rule [D-LOOP2] is applied.

Case: [RC-SEQ]

We deal with $e_1; e_2$. Based on the expression e_1, there are two cases:

1. e_1 is a value and then the rule [D-SEQ2] can be applied.
2. e_1 is not a value. By the induction hypothesis, there are two sub cases:
   (a) $\langle \varpi, \Pi \rangle [e_1] \hookrightarrow \langle \hat{\varpi}, \hat{\Pi} \rangle [e_1']$.

   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $e_1'; e_2$. The evaluation rule [D-SEQ1] can be applied.

   (b) $\langle \varpi, \Pi \rangle [e_1] \hookrightarrow \texttt{nullerr}$

   The error is propagated as $\langle \varpi, \Pi \rangle [e_1; e_2] \hookrightarrow \texttt{nullerr}$.

Case: [$\textsc{rc-invoke}$]

We deal with $v'_0.mn\langle a^+ a'^+\rangle(v'_{1..p})$. By the hypothesis's type judgment [$\textsc{rc-invoke}$] we get that the regions $\{a^+ a'^+\} \subset R$. By the hypothesis's consistency relation we get that $dom(\varpi) = R$. Thus the runtime check of [$\textsc{rc-invoke}$] is proved and the rule

[$\textsc{d-invoke-danglerr}$] is never used by a well typed program. By the hypothesis's type judgment the type of v'_0 is $cn\langle a^+\rangle$. By the hypothesis's consistency $dom(\Gamma) = dom(\Pi)$, thus $v'_0 \in dom(\Pi)$. According to the Lemma 10, there are two cases:

1. $\Pi(v'_0) = \texttt{null}$. Then the rule [$\textsc{d-nullerr3}$] generates an error `nullerr`.
2. $\Pi(v'_0) = (a_1, o)$ that is well-typed. Thus the rule [$\textsc{d-invoke}$] can be applied.

□

B Region Type System Rules

$$\frac{\begin{array}{c}[\textbf{RC-PROG}]\\ \mathit{WFClasses}(P)\\ P = \mathit{def}_{i:1..n}\\ \mathit{FieldsOnce}(\mathit{def})_{i:1..n}\\ \mathit{MethodsOnce}(\mathit{def})_{i:1..n}\\ P \vdash \mathit{InheritanceOK}(\mathit{def})_{i:1..n}\\ P \vdash_{\mathit{def}} \mathit{def}_{i:1..n}\end{array}}{\vdash P}$$

$$\frac{\begin{array}{c}[\textbf{RC-CLASS}]\\ def = \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle \textbf{extends}\ c\langle r_{1..m}\rangle\\ \textbf{where}\ \varphi\ \{\mathit{field}_{1..p}\ \mathit{meth}_{1..q}\}\\ r_1 \notin \bigcup_{i=1}^{p} \mathit{reg}(\mathit{field}_i)\\ \varphi \Rightarrow r_i \succeq r_1\ \ i = 2..n \quad R = \{r_1, \ldots, r_n\}\\ P; \{\mathit{this} : \mathit{cn}\langle r_{1..n}\rangle\}; R; \varphi \vdash_{\mathit{meth}} \mathit{meth}_i \quad i = 1..q\\ P; R; \varphi \vdash_{\mathit{field}} \mathit{field}_i \quad i = 1..p\end{array}}{P \vdash_{\mathit{def}} def}$$

$$\frac{\begin{array}{c}[\textbf{RC-METH}]\\ \Gamma' = \Gamma + (v_j : t_j)_{j:1..p} \quad R' = R \cup \{r_1, \ldots, r_m\}\\ \varphi' = \varphi \wedge \varphi_0 \qquad P; R'; \varphi' \vdash_{\mathit{type}} t_j,\ j = 0..p\\ P; \Gamma'; R'; \varphi' \vdash e : t'_0 \qquad P; R'; \varphi' \vdash t'_0 <: t_0\end{array}}{P; \Gamma; R; \varphi \vdash_{\mathit{meth}} t_0\ \mathit{mn}\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p})\textbf{where}\ \varphi_0\ \{e\}}$$

$$\frac{\begin{array}{c}[\textbf{RC-EB}]\\ P; R; \varphi \vdash_{\mathit{type}} t'\\ \Gamma' = \Gamma + (v : t')\\ P; \Gamma'; R; \varphi \vdash e : t\end{array}}{P; \Gamma; R; \varphi \vdash \{(t'\ v)\ e\} : t}$$

$$\frac{[\textbf{RC-CONS1}]}{P; \Gamma; R; \varphi \vdash \textbf{null} : \bot} \qquad \frac{[\textbf{RC-CONS2}]}{P; \Gamma; R; \varphi \vdash k : \mathit{prim}\langle\rangle} \qquad \frac{\begin{array}{c}[\textbf{RC-VAR}]\\ (v : t) \in \Gamma\end{array}}{P; \Gamma; R; \varphi \vdash v : t}$$

$$\frac{\begin{array}{c}[\textbf{RC-FD}]\\ (v : \mathit{cn}\langle a_{1..n}\rangle) \in \Gamma\\ (t\,f) \in \mathit{fieldlist}(\mathit{cn}\langle r_{1..n}\rangle)\end{array}}{P; \Gamma; R; \varphi \vdash v.f : [r_1 \mapsto a_1 ... r_n \mapsto a_n]t}$$

$$\frac{\begin{array}{c}[\textbf{RC-NEW}]\\ P; R; \varphi \vdash_{\mathit{type}} \mathit{cn}\langle r_{1..n}\rangle\\ \mathit{fieldlist}(\mathit{cn}\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p}\\ (v_i : t'_i) \in \Gamma \quad P; R; \varphi \vdash t'_i <: t_i \quad i = 1..p\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_1, .., v_p) : \mathit{cn}\langle r_{1..n}\rangle}$$

$$\frac{\begin{array}{c}[\textbf{RC-IF}]\\ v : \textbf{boolean}\langle\rangle \in \Gamma\\ P; \Gamma; R; \varphi \vdash e_1 : t_1\ \ P; R; \varphi \vdash t_1 <: t\\ P; \Gamma; R; \varphi \vdash e_2 : t_2\ \ P; R; \varphi \vdash t_2 <: t\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : t}$$

$$\frac{\begin{array}{c}[\textbf{RC-ASSGN}]\\ P; \Gamma; R; \varphi \vdash \mathit{lhs} : t\\ P; \Gamma; R; \varphi \vdash e : t'\\ P; R; \varphi \vdash t' <: t\end{array}}{P; \Gamma; R; \varphi \vdash \mathit{lhs} = e : \textbf{void}}$$

$$\frac{\begin{array}{c}[\textbf{RC-SEQ}]\\ P; \Gamma; R; \varphi \vdash e_1 : t_1\\ P; \Gamma; R; \varphi \vdash e_2 : t_2\end{array}}{P; \Gamma; R; \varphi \vdash e_1\textbf{;}e_2 : t_2} \qquad \frac{\begin{array}{c}[\textbf{RC-LOOP}]\\ v : \textbf{boolean}\langle\rangle \in \Gamma\\ P; \Gamma; R; \varphi \vdash e : \textbf{void}\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{while}\ v\ e : \textbf{void}}$$

$$\frac{\begin{array}{c}[\textbf{RC-INVOKE}]\\ (v_0 : \mathit{cn}\langle a^+\rangle) \in \Gamma \quad P; R; \varphi \vdash_{\mathit{type}} \mathit{cn}\langle a^+\rangle\\ P \vdash (t\ \mathit{mn}\langle a^+ r'^+\rangle((t_i\ v_i)_{i:1..n})\textbf{where}\ \varphi_0\ \{e\}) \in \mathit{cn}\langle a^+\rangle\\ (v'_i : t'_i)_{i:1..n} \in \Gamma \quad a'^+ \in R \quad \rho = [r'^+ \mapsto a'^+]\\ \varphi \Rightarrow \rho\,\varphi_0 \quad P; R; \varphi \vdash t'_i <: \rho\, t_i \quad i = 1..n\end{array}}{P; \Gamma; R; \varphi \vdash v_0.\mathit{mn}\langle a^+ a'^+\rangle(v'_1..v'_n) : \rho\, t}$$

$$\frac{\begin{array}{c}[\textbf{RC-LETR}]\\ a = \mathit{fresh}()\\ \varphi' = \varphi \wedge \bigwedge_{r' \in R}(r' \succeq a)\\ P; \Gamma; R \cup \{a\}; \varphi' \vdash [r \mapsto a]e : t\\ \mathit{reg}(t) \subseteq R\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{letreg}\ r\ \textbf{in}\ e : t}$$

$\rho t,\ \rho\varphi,\ \rho e$ region substitution on a type, a constraint, and an expression
$\mathit{fresh}()$ returns one or more new/unused region names

Fig. 10. Region Type Checking Rules

$$\mathit{reg}(\{\})=_{def}\{\} \quad \mathit{reg}(\{v{:}\tau\langle r^*\rangle\}\cup\Gamma)=_{def}\{r^*\}\cup\mathit{reg}(\Gamma) \quad \mathit{reg}(\tau\langle r^*\rangle)=_{def}\{r^*\}$$

$$\mathit{reg}((\tau\langle r^*\rangle\ f))=_{def}\{r^*\} \quad \mathit{reg}(\mathit{true})=_{def}\{\} \quad \mathit{reg}(r_1{=}r_2)=_{def}\{r_1,r_2\}$$

$$\mathit{reg}(r_1{\succeq}r_2)=_{def}\{r_1,r_2\} \quad \mathit{reg}(q\langle r_1..r_n\rangle\}=_{def}\{r_1..r_n\} \quad \mathit{reg}(\varphi_1\wedge\varphi_2)=_{def}\mathit{reg}(\varphi_1)\cup\mathit{reg}(\varphi_2)$$

$$\frac{}{\mathit{fieldlist}(\textbf{Object}\langle r\rangle)=_{def}[\,]} \qquad \frac{\begin{array}{c}\textbf{class}\ \mathit{cn}_1\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}_2\langle r_{1..m}\rangle..\{(t_i\ f_i)_{i:1..p}..\}\in P'\\ \ell=\mathit{fieldlist}(\rho\ \mathit{cn}_2\langle r_{1..m}\rangle) \quad \rho=[r_i\mapsto x_i]_{i=1}^{n}\end{array}}{\mathit{fieldlist}(\mathit{cn}_1\langle x_{1..n}\rangle)=_{def}\ell +\!\!+ [(\rho\ t_i)\ f_i]_{i=1}^{p}}$$

$$\frac{P=\ldots\mathit{def}\ldots}{\mathit{def}\in P} \qquad \frac{P\vdash \mathit{mbr}\in_D \mathit{cn}\langle r_{1..n}\rangle}{P\vdash \mathit{mbr}\in \mathit{cn}\langle r_{1..n}\rangle} \qquad \frac{\mathit{mbr}=\mathit{field}|\mathit{meth}\ \ \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ldots\{\ldots\mathit{mbr}\ldots\}\in P}{P\vdash \mathit{mbr}\in_D \mathit{cn}\langle r_{1..n}\rangle}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..m}\rangle\ldots\in P\\ P\vdash \mathit{mbr}\in \mathit{cn}'\langle r_{1..m}\rangle\ \ \neg(P\vdash \mathit{mbr}{\in_D}\mathit{cn}\langle r_{1..n}\rangle)\end{array}}{P\vdash \mathit{mbr}\in \mathit{cn}\langle r_{1..n}\rangle} \qquad \frac{\begin{array}{c}\vdash t<:t',\varphi' \quad \varphi\Rightarrow\varphi'\\ P;R;\varphi\vdash_{\mathit{type}} t \quad P;R;\varphi\vdash_{\mathit{type}} t'\end{array}}{P;R;\varphi\vdash t{<:}t'}$$

$$\frac{P;R\vdash_{\mathit{constr}} t,\varphi' \quad \varphi\Rightarrow\varphi'}{P;R;\varphi\vdash_{\mathit{type}} t} \qquad \frac{}{P;R\vdash_{\mathit{constr}} \mathit{prim}\langle\rangle,\ \mathit{true}} \qquad \frac{r\in R}{P;R\vdash_{\mathit{constr}} \mathit{Object}\langle r\rangle,\ \mathit{true}}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ c\langle\ldots\rangle\ \textbf{where}\ \varphi\ \{\ldots\}\ \in\ P\\ R\supseteq\{x_1,\ldots,x_n\}\end{array}}{P;R\vdash_{\mathit{constr}} \mathit{cn}\langle x_{1..n}\rangle,[r_1\mapsto x_1..r_n\mapsto x_n]\varphi} \qquad \frac{P;R;\varphi\vdash_{\mathit{type}} t}{P;R;\varphi\vdash_{\mathit{field}} t\ v}$$

$$\frac{\begin{array}{c}P=\mathit{def}_{1..n} \qquad \mathit{def}_i=\textbf{class}\ \mathit{cn}_i\langle\ldots\rangle\ \textbf{extends}\ \mathit{cn}_{i'}\langle\ldots\rangle\ldots\\ \mathit{IR}=\{(\mathit{cn}_i,\mathit{cn}_{i'})\mid 1\leq i\leq n\} \quad \mathit{ID}=\{(\mathit{cn}_i,\mathit{cn}_i)\mid 1\leq i\leq n\}\\ \mathit{TransClosure}(\mathit{IR})\cap\mathit{ID}=\emptyset \quad \forall i,j{:}i\neq j\cdot \mathit{cn}_i\neq\mathit{cn}_j\end{array}}{\mathit{WFClasses}(P)}$$

$$\frac{\begin{array}{c}\mathit{def}=\textbf{class}\ \mathit{cn}\langle\ldots\rangle\ldots\{(\mathit{fd}_j)_{j:1..p}\ldots\}\\ \forall j,l{:}j\neq l\cdot \mathit{name}(\mathit{fd}_j)\neq\mathit{name}(\mathit{fd}_l)\end{array}}{\mathit{FieldsOnce}(\mathit{def})} \qquad \frac{\begin{array}{c}\mathit{def}=\textbf{class}\ \mathit{cn}\langle\ldots\rangle\ldots\{\ldots(m_j)_{j:1..q}\}\\ \forall j,l{:}j\neq l\cdot \mathit{name}(m_j)\neq\mathit{name}(m_l)\end{array}}{\mathit{MethodsOnce}(\mathit{def})}$$

$$\frac{\begin{array}{c}\mathit{def}=\textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..m}\rangle\ \textbf{where}\ \varphi\ \{\mathit{fd}_{1..p}\ \mathit{meth}_{1..q}\}\\ n\geq m \quad P;\{r_1,..,r_m\}\vdash_{\mathit{constr}} \mathit{cn}'\langle r_{1..m}\rangle,\varphi' \quad \varphi\Rightarrow\varphi'\\ \forall j{\in}1..q\cdot\exists \mathit{meth}'\cdot P{\vdash}\mathit{meth}'{\in}\mathit{cn}'\langle r_{1..m}\rangle\wedge\mathit{name}(\mathit{meth}')=\mathit{name}(\mathit{meth}_j)\\ \Rightarrow(P;\varphi{\vdash}\mathit{OverridesOK}(\mathit{meth}_j,\mathit{meth}'))\end{array}}{P\vdash \mathit{InheritanceOK}(\mathit{def})}$$

$$\frac{\begin{array}{c}\mathit{meth}=t_0\ \mathit{mn}\langle x_{1..p(p+1)..q},r_{1..n}\rangle((t\,v)_{i:1..m})\ \textbf{where}\ \varphi\ \ldots\\ \mathit{meth}'=t_0\ \mathit{mn}\langle x_{1..p},r_{1..n}\rangle((t\,v)_{i:1..m})\ \textbf{where}\ \varphi'\ldots \quad \varphi_0\wedge\varphi'\Rightarrow\varphi\end{array}}{P;\varphi_0\vdash \mathit{OverridesOK}(\mathit{meth},\mathit{meth}')}$$

Fig. 11. Auxiliary Region Checking Rules