Immutable Objects for a Java-like Language

C. Haack[1][⋆], E. Poll[1], J. Schäfer[2][⋆⋆], and A. Schubert[1,3][⋆⋆⋆]

[1] Radboud Universiteit Nijmegen, The Netherlands
[2] Technische Universität Kaiserlautern, Germany
[3] Warsaw University, Poland

Abstract. We extend a Java-like language with immutability specifications and a static type system for verifying immutability. A class modifier `immutable` specifies that all class instances are immutable objects. Ownership types specify the depth of object states and enforce encapsulation of representation objects. The type system guarantees that the state of immutable objects does not visibly mutate during a program run. Provided immutability-annotated classes and methods are `final`, this is true even if immutable classes are composed with untrusted classes that follow Java's type system, but not our immutability type system.

1 Introduction

An object is immutable if it does not permit observable mutations of its object state. A class is immutable if all its instances are immutable objects. In this article, we present an extension of a Java-like language with immutability specifications and a static type system for verifying them.

For many reasons, favoring immutability greatly simplifies object-oriented programming [Blo01]. It is, for instance, impossible to break invariants of immutable objects, as these are established once and for all by the object constructor. This is especially pleasing in the presence of aliasing, because maintaining invariants of possibly aliased objects is difficult and causes headaches for program verification and extended static checking tools. Sharing immutable objects, on the other hand, causes no problems whatsoever. Object immutability is particularly useful in multi-threaded programs, as immutable objects are thread-safe. Race conditions on the state of immutable objects are impossible, because immutable objects do not permit writes to their object state. Even untrusted components cannot mutate immutable objects. This is why immutable objects are important in scenarios where some components (e.g. applets downloaded from the web) cannot be trusted. If a security-sensitive component checks data that it has received from an untrusted component, it typically relies on the fact that the data does not mutate after the check. A prominent example of an immutable class whose immutability is crucial for many security-sensitive applications is Java's immutable `String` class.

Unfortunately, statically enforcing object immutability for Java is not easy. The main reason for this is that an object's local state often includes more than just the object's fields. If local object states never extended beyond the object's fields, Java's

[⋆] Supported by the EU under the IST-2005-015905 MOBIUS project.
[⋆⋆] Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation).
[⋆⋆⋆] Supported by an EU Marie Curie Intra-European Fellowship.

`final` field modifier would be enough to enforce object immutability. However, `String` objects, for instance, refer to an internal character array that is considered part of the `String`'s local state. It is crucial that this character array is encapsulated and any aliasing from outside is prevented. Java does not provide any support for specifying deep object states and enforcing encapsulation. Fortunately, ownership type systems come to rescue. Ownership type systems have been proposed to better support encapsulation in object-oriented languages, e.g., [CPN98,CD02,BLS03,MPH01,DM05]. In order to permit immutable objects with deep states, we employ a variant of ownership types. The core of our ownership type system is contained (in various disguises) in all of the ownership type systems listed above. In addition, our type system distinguishes between read-only and read-write objects. The difference between read-only objects and immutable objects is that the latter have no public mutator methods at all, whereas the former have public mutator methods that are prohibited to be called. We need read-only objects in order to support sharing mutable (but read-only) representation objects among immutable objects. Unlike read-only *references* [MPH01,BE04,TE05], our read restrictions for immutable and read-only objects are per object, not per reference.

Our type system guarantees immutability in an *open world* [PBKM00] where immutable objects are immutable even when interacting with *unchecked components* that do not follow the rules of our immutability type system. The immutability type system guarantees that unchecked components cannot break from outside the immutability of checked immutable objects. All we assume about unchecked components is that they follow the standard Java typing rules. Unchecked components could, for instance, represent legacy code or untrusted code. Our decision to support an open world has several important impacts on the design of our type system. For instance, we have to ensure that the types of public methods of immutable objects do not constrain callers beyond the restrictions imposed by Java's standard type system. Technically, this is easily achieved by restricting the ownership types of methods. Furthermore, we cannot assume that clients of immutable objects follow a read-only policy that is not already enforced by Java's standard type system. For this reason, we define read-only types in context `world` to be equivalent to read-write types.

A difficulty in enforcing object immutability is that even immutable objects mutate for some time, namely during their construction phase. This is problematic for several reasons. Firstly, Java does not restrict constructor bodies in any way. In particular, Java allows passing self-references from constructors to outside methods. This is undesirable for immutable objects as it would allow observing immutable objects while they are still mutating. Moreover, the rules that control aliasing for constructors should be different from the rules that control aliasing for methods. Constructors should be allowed to pass dynamic aliases to their internals to outside methods as long as these methods do not store any static aliases to the internals. Methods, on the other hand, must be disallowed to leak dynamic aliases to internals, if our goal is immutability in an open world.

2 A Java-like Language with Immutability

In this section, we present Core Jimuva, a core language for an immutability extension of Java. We use the same syntax conventions as Featherweight Java (FJ) [IPW01]. In particular, we indicate sequences of X's by an overbar: $\bar{X}$. We assume that field declarations $\bar{F}$, constructor declarations $\bar{K}$, method declarations $\bar{M}$ and parameter declarations $\bar{ty}\,\bar{x}$ do not contain duplicate declarations. We also use some regular expression syntax: X**?** for an optional X, X***** for a possibly empty list of X's, and $X \mid Y$ for an X or a Y. For any entity X (e.g., X an expression or a type), we write $\mathsf{oids}(X)$ for the set of object identifiers occurring in X and $\mathsf{vars}(X)$ for the set of variables occurring in X (including the special access variable `myaccess`). For a given class table $\bar{c}$, we write $C\,\texttt{ext}_{\bar{c}}\,D$ whenever $\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\{..\} \in \bar{c}$. The *subclassing relation* $<:_{\bar{c}}$ is the reflexive, transitive closure of $\texttt{ext}_{\bar{c}}$. We omit the subscript $\bar{c}$ if it is clear from the context. Like in FJ, we assume the following sanity conditions on class tables $\bar{c}$: (1) subclassing $<:_{\bar{c}}$ is antisymmetric, (2) if C (except `Object`) occurs anywhere in $\bar{c}$ then C is declared in $\bar{c}$ and (3) $\bar{c}$ does not contain duplicate declarations or a declaration of `Object`.

Core Jimuva — a Java-like Core Language with Immutability Annotations:

$C, D, E \in \mathsf{ClassId}$	class identifiers (including `Object`)
$f, g \in \mathsf{FieldId}$	field identifiers
$m, n \in \mathsf{MethId}$	method identifiers
$k, l \in \mathsf{ConsId}$	constructor identifiers
$o, p, q, r \in \mathsf{ObjId}$	object identifiers (including `world`)
$x, y, z \in \mathsf{Var}$	variables (including `this`, `myowner`)
$\mathit{ca} ::= \texttt{immutable}\textbf{?}$	class attributes
$\mathit{ea} ::= \texttt{anon}\textbf{?}\ \texttt{rdonly}\textbf{?}\ \texttt{wrlocal}\textbf{?}$	expression attributes
$\mathit{ar} ::= \texttt{rd} \mid \texttt{rdwr} \mid \texttt{myaccess}$	access rights for objects
$\mathit{fm} ::= \texttt{final}\textbf{?}$	final modifier
$u, v, w \in \mathsf{Val} ::= \texttt{null} \mid o \mid x$	values
$\mathit{ty} \in \mathsf{ValTy} ::= C\texttt{<}\mathit{ar}, v\texttt{>} \mid \texttt{void}$	value types
$T \in \mathsf{ExpTy} ::= \mathit{ea}\,\mathit{ty}$	expression types
$c, d ::= \mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\{\bar{F}\ \bar{K}\ \bar{M}\}$	class declaration (where $C \neq$ `Object`)
$F ::= C\texttt{<}\mathit{ar}, v\texttt{>}\,f\,\texttt{;}$	field
$K ::= \mathit{ea}\,C.k(\bar{ty}\,\bar{x})\{e\}$	constructor (scope of $\bar{x}$ is e)
$M ::= \mathit{fm}\texttt{<}\bar{y}\texttt{>}\,T\,m(\bar{ty}\,\bar{x})\{e\}$	method (scope of $\bar{y}$ is $(T, \bar{ty}, e)$, of $\bar{x}$ is e)
$e \in \mathsf{Exp} ::=$	expressions and statements
$v \mid v.f \mid v.f\texttt{=}e \mid v.m\texttt{<}\bar{v}\texttt{>}(\bar{e}) \mid \texttt{new}\,C\texttt{<}\mathit{ar},v\texttt{>}.k(\bar{e}) \mid \texttt{let}\,x\texttt{=}e\,\texttt{in}\,e \mid (C)e \mid C.k(\bar{e})$	

Derived Forms:

$$\text{If } e \notin \mathsf{Val}, x \notin \mathsf{vars}(e, e', \bar{v}, \bar{e})\text{:}\quad e.f \triangleq \texttt{let}\,x\texttt{=}e\,\texttt{in}\,x.f \quad e.f\texttt{=}e' \triangleq \texttt{let}\,x\texttt{=}e\,\texttt{in}\,x.f\texttt{=}e'$$

$$e.m\texttt{<}\bar{v}\texttt{>}(\bar{e}) \triangleq \texttt{let}\,x\texttt{=}e\,\texttt{in}\,x.m\texttt{<}\bar{v}\texttt{>}(\bar{e}) \qquad \text{If } x \notin \mathsf{vars}(e')\text{:}\quad e\,\texttt{;}\,e' \triangleq \texttt{let}\,x\texttt{=}e\,\texttt{in}\,e'$$

$$\texttt{skip} \triangleq \texttt{null} \qquad e\,\texttt{;} \triangleq e\,\texttt{;}\,\texttt{skip} \qquad \texttt{let}\,x,\bar{x}\texttt{=}e,\bar{e}\,\texttt{in}\,e' \triangleq \texttt{let}\,x\texttt{=}e\,\texttt{in}\,\texttt{let}\,\bar{x}\texttt{=}\bar{e}\,\texttt{in}\,e'$$

$$e.m(\bar{e}) \triangleq e.m\texttt{<>}(\bar{e}) \qquad \mathit{fm}\,T\,m(\bar{ty}\,\bar{x})\{e\} \triangleq \mathit{fm}\texttt{<>}\,T\,m(\bar{ty}\,\bar{x})\{e\}$$

$$C\texttt{<}\mathit{ar}\texttt{>} \triangleq C\texttt{<}\mathit{ar},\texttt{world}\texttt{>} \qquad C\texttt{<}v\texttt{>} \triangleq C\texttt{<}\texttt{rdwr}, v\texttt{>} \qquad C \triangleq C\texttt{<}\texttt{world}\texttt{>}$$

Core Jimuva extends a Java core language by *immutability specifications*: the class attribute `immutable` specifies that all instances of a class are immutable objects, i.e., their object state does not visibly mutate.

The other Java extensions are auxiliary and specify constraints on objects and methods that `immutable` objects depend on: *Ownership types* are used to ensure encapsulation of representation [CPN98,CD02,BLS03]. The `rdonly`-attribute (*read-only*) is used to disallow methods of immutable objects to write to their own object state. The `wrlocal`-attribute (*write-local*) is used to constrain constructors of immutable objects not to write to the state of other immutable objects of the same class. Vitek and Bokowski's `anon` (*anonymous*) attribute [VB01] is used to constrain constructors of immutable objects not to leak references to `this`. For a given class table with `immutable`-specifications, these additional expression attributes can be automatically inferred, but we prefer to make them syntactically explicit in this paper.

Object types are of the form C<ar, v>, where ar specifies the access rights for the object and v specifies the object owner. Omitted access rights default to `rdwr`, omitted owners default to `world`. The expression `new`C<ar,v>.$k(\bar{e})$ creates a new object of type C<ar, v> and then executes the body of constructor $C.k()$ to initialize the new object. Access rights and ownership information have no effect on the dynamic behaviour of programs.

Access rights specify access constraints for *objects* (in contrast to Java's access modifiers `protected` and `private`, which specify access constraints for *classes*). The access rights are `rdwr` (*read-write, i.e., no constraints*) and `rd` (*read-only*). Read-only access to o forbids writes to o's state and calls to o's non-`rdonly` methods. Objects are implicitly parameterized by the *access variable* `myaccess`, which refers to the access rights for `this`. Consider, for instance, the following class:

```
class C ext Object {
  C<myaccess,myowner> x;
  wrlocal C.k(C<myaccess,myowner> x){ this.set(x); }
  rdonly C<myaccess,myowner> get(){ x }
  wrlocal void set(C<myaccess,myowner> x){ this.x = x; } }
```

If, for instance, o is an object of type `C<rd,`p`>`, then access to o is read-restricted. Furthermore, access to all objects in the transitive reach of o is read-restricted, too: o.`get()`, o.`get().get()`, etc., all have type C<`rd`, p> and therefore permit only `rd`-access. The following example shows how `C` can be used:

```
class D ext Object {
  C<rd,this> x;  C<myaccess,myowner> y;  C<rdwr,this> z;
  ...
  void m() {
    x = new C<rd,this>(new C<rd,this>(null)); // legal
    y = new C<myaccess,myowner>(new C<myaccess,myowner>(null)); // legal
    z = new C<rdwr,this>(new C<rdwr,this>(null)); // legal
    new C<rd,this>(new C<myaccess,myowner>(null)); // illegal
    x.get(); y.get(); z.get(); y.set(null); z.set(null); // legal
    x.set(null); // illegal call of non-rdonly method on rd-object }
  rdonly void n() {
    y.set(null); // illegal call of non-rdonly method } }
```

It may perhaps be slightly surprising that the call `y.set(null)` in `m()` is legal, although the access variable `myaccess` may possibly get instantiated to `rd`. This call is safe, because it is illegal to call the non-`rdonly` method `m()` on a `rd`-object and, hence, the call `y.set(null)` inside `m()` is never executed when `myaccess` instantiates to `rd`.

Ownership types. Objects of type C<ar, o> are considered *representation objects owned by o*, that is, they are not visible to the outside and can only be accessed via o's interface. Objects without owners have types of the form C<ar,world>. The special variable myowner refers to the owner of this. Our type system restricts myowner and world to only occur inside angle brackets <·>. The myowner variable corresponds to the first class parameter in parametric ownership type systems [CD02,BLS03] and to the owner ghost field in JML's encoding of the Universe type system [DM05]. Furthermore, the Universe type system's rep and peer types [MPH01] relate to our types as follows: rep$C$ corresponds to C<rdwr,this>, and peer$C$ to C<rdwr,myowner>.

Jimuva has *owner-polymorphic methods*: In a method declaration <$\bar{y}$>$T\,m(\bar{ty}\,\bar{x})\{e\}$, the scope of owner parameters $\bar{y}$ includes the types $T, \bar{ty}$ and the method body e. The type system restricts occurrences of owner parameters to inside angle brackets <·>. Owner parameters get instantiated by the values $\bar{v}$ in method call expressions $u.m$<$\bar{v}$>$(\bar{e})$.

Owner-polymorphic methods permit *dynamic aliasing of representation objects*. Consider, for instance, a method of the following type:

```
<x,y> void copy(C<x> from, C<y> to)
```

A client may invoke copy with one or both of x and y instantiated to this, for instance, copy<world,this>(o,mine), where mine refers to an internal representation object owned by the client. Dynamic aliasing of representation objects is often dangerous, but can sometimes be useful. For immutability, dynamic aliasing is useful during the object construction phase, but dangerous thereafter. For instance, the constructor String(char[] a) of Java's immutable String class passes an alias to the string's internal character array to a global arraycopy() method, which does the job of defensively copying a's elements to the string's representation array. Our type system uses owner-polymorphic methods to permit dynamic aliasing during the construction phase of immutable objects, but prohibit it thereafter. The latter is achieved by prohibiting rdonly-expressions to instantiate a method's owner parameters by anything but world.

For String to be immutable, it is important that the arraycopy() method does not create a static alias to the representation array that is handed to it from the constructor String(char[] a). Fortunately, owner-polymorphic methods prohibit the creation of dangerous static aliases! This is enforced merely by the type signature. Consider again the copy() method: From the owner-polymorphic type we can infer that an implementation of copy does not introduce an alias to the to-object from inside the transitive reach of the from-object. This is so, because all fields in from's reach have types of the form D<ar,x> or D<ar,from> or D<ar,world> or D<ar,o> where o is in from's reach. None of these are supertypes of C<y>, even if D is a supertype of C. Therefore, copy's polymorphic type forbids assigning the to-object to fields inside from's reach.

Let-bindings. Unlike FJ [IPW01] but like other languages that support ownership through dependent types [CD02,BLS03], we restrict some syntactic slots to values instead of expressions, for instance, $v.f$ instead of $e.f$. This is needed for our typing rules to meaningfully instantiate occurrences of this in types. We obtain an expression language similar to FJ through derived forms, see above. An automatic typechecker for full Jimuva will work on an intermediate language with let-bindings.

Constructors. Our language models object constructors. This is important, as object construction is a critical stage in the lifetime of immutable objects: during construction even immutable objects still mutate! For simplicity, Core Jimuva's constructors are *named*. Moreover, we have simplified explicit constructor calls: instead of calling constructors using `super()` and `this()`, constructors are called by concatenating class name C and constructor name k, i.e., $C.k()$. Constructors $C.k()$ are only visible in C's subclasses. We allow direct constructor calls $C.k()$ from constructors, and even from methods, of arbitrary subclasses of C. That is more liberal than real Java, but unproblematic for the properties we care about.

Protected fields. Jimuva's type system ensures that fields are visible in subclasses only. This is similar to Java's `protected` fields.[4] Our reason for using `protected` instead of `private` fields is proof-technical: a language with `private` fields does not satisfy the type preservation (aka subject reduction) property. On the other hand, soundness of a type system with `private` fields obviously follows from soundness of our less restrictive type system with `protected` fields.

3 Operational Semantics

Our operational semantics is small-step and similar to the semantics from Zhao et al [ZPV06]. However, in contrast to [ZPV06], we also model a mutable heap. The operational semantics is given by a state reduction relation $h :: s \rightarrow_{\bar{c}} h' :: s'$, where h is a *heap*, s a *stack* and $\bar{c}$ the underlying set of classes. We omit the subscript $\bar{c}$ if it is clear from the context. *Stack frames* are of the form $(e \,\mathsf{in}\, o)$, where e is a (partially executed) method body and o is the `this`-binding. Keeping track of the `this`-binding will be needed for defining the semantics of immutability. The `world` identifier is used as a dummy for the `this`-binding of the top-level main program. *Evaluation contexts* are expressions with a single "hole" [ ], which acts as a placeholder for the expression that is up for evaluation in left-to-right evaluation order. If $\mathscr{E}$ is an evaluation context and e an expression, then $\mathscr{E}[e]$ denotes the expression that results from replacing $\mathscr{E}$'s hole by e. Evaluation contexts are a standard data structure for operational semantics [WF94].

Runtime Structures:

$state ::= h :: s \in \mathsf{State} = \mathsf{Heap} \times \mathsf{Stack}$	states
$h ::= \overline{obj} \in \mathsf{Heap} = \mathsf{ObjId} \rightarrow (\mathsf{FieldId} \rightarrow \mathsf{Val})$	heaps
$obj ::= o\{\bar{f} = \bar{v}\} \in \mathsf{Obj} = \mathsf{ObjId} \times (\mathsf{FieldId} \rightarrow \mathsf{Val})$	objects
$s ::= \bar{fr} \in \mathsf{Stack} = \mathsf{Frame}^*$	stacks
$fr ::= e \,\mathsf{in}\, o \in \mathsf{Frame} = \mathsf{Exp} \times \mathsf{ObjId}$	stack frames
$\mathscr{E} ::= [\,] \mid v.f\texttt{=}\mathscr{E} \mid v.m\texttt{<}\bar{v}\texttt{>}(\bar{v},\mathscr{E},\bar{e}) \mid \texttt{new}\, C\texttt{<}ar,v\texttt{>}.k(\bar{v},\mathscr{E},\bar{e}) \mid$ $\texttt{let}\, x \texttt{=} \mathscr{E}\, \texttt{in}\, e \mid (C)\mathscr{E} \mid C.k(\bar{v},\mathscr{E},\bar{e})$	evaluation contexts

We assume that every object identifier $o \neq \texttt{world}$ is associated with a unique type $\mathsf{ty}(o)$ of the form $C\texttt{<}ar, p\texttt{>}$ such that $p = \texttt{world}$ implies $ar = \texttt{rdwr}$ and C is `immutable` implies $p = \texttt{world}$. We define $\mathsf{rawty}(o) \stackrel{\Delta}{=} C$, if $\mathsf{ty}(o) = C\texttt{<}ar, p\texttt{>}$.

We use *substitution* to model parameter passing: Substitutions are finite functions from variables, including `myaccess`, to values and access rights. We let meta-variable σ range over substitutions and write $(\bar{x} \leftarrow \bar{v})$ for the substitution that maps each x_i in $\bar{x}$ to the corresponding v_i in $\bar{v}$. We write id for the identity. We write $e[\sigma]$ for the expression that results from e by substituting variables x by $\sigma(x)$. Similarly for types, $T[\sigma]$. The following abbreviations are convenient:

[4] Java's `protected` fields are slightly more permissive and package-visible, too.

$$\mathsf{self}(u,ar,v) \overset{\Delta}{=} (\mathtt{this},\mathtt{myaccess},\mathtt{myowner}{\leftarrow}u,ar,v)$$
$$\sigma,\bar{y}{\leftarrow}\bar{v} \overset{\Delta}{=} (\bar{x},\bar{y}{\leftarrow}\bar{u},\bar{v}),\ \text{if } \sigma = (\bar{x}{\leftarrow}\bar{u}) \text{ and } \bar{x}\cap\bar{y} = \emptyset$$

We use several auxiliary functions that are exactly as in FJ [IPW01]: The function $\mathsf{mbody}_{\bar{c}}(C,m)$ looks up the method for m on C-objects in class table $\bar{c}$. Similarly, $\mathsf{cbody}_{\bar{c}}(C.k)$ for constructors. The function $\mathsf{fd}_{\bar{c}}(C)$ computes the field set for C-objects based on class table $\bar{c}$. These functions are defined in Appendix B. We omit the subscript $\bar{c}$ if it is clear from the context.

State Reductions, $state \rightarrow_{\bar{c}} state'$**:**

(Red Get) $\quad h = h', o\{..f = v..\}$
$\quad h :: s, \mathscr{E}[o.f] \,\mathsf{in}\, p \rightarrow h :: s, \mathscr{E}[v] \,\mathsf{in}\, p$

(Red Set)
$\quad h, o\{f = u, \bar{g} = \bar{w}\} :: s, \mathscr{E}[o.f\texttt{=}v] \,\mathsf{in}\, p \rightarrow h, o\{f = v, \bar{g} = \bar{w}\} :: s, \mathscr{E}[v] \,\mathsf{in}\, p$

(Red Call) $\quad s = s', \mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})] \,\mathsf{in}\, p \quad \mathsf{ty}(o) = C\texttt{<}ar,w\texttt{>} \quad \mathsf{mbody}(C,m) = \texttt{<}\bar{y}\texttt{>}(\bar{x})(e)$
$\quad h :: s \rightarrow h :: s, e[\mathsf{self}(o,ar,w), \bar{y}{\leftarrow}\bar{u}, \bar{x}{\leftarrow}\bar{v}] \,\mathsf{in}\, o$

(Red New) $\quad s = s', \mathscr{E}[\mathtt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v})] \,\mathsf{in}\, p \quad o \notin \mathsf{dom}(h) \quad \mathsf{ty}(o) = C\texttt{<}ar,w\texttt{>} \quad \mathsf{fd}(C) = \bar{ty}\,\bar{f}$
$\quad h :: s \rightarrow h, o\{\bar{f} = \mathtt{null}\} :: s, C.k(\bar{v})\,;o \,\mathsf{in}\, o$

(Red Cons) $\quad s = s', \mathscr{E}[C.k(\bar{v})] \,\mathsf{in}\, p \quad \mathsf{cbody}(C.k) = (\bar{x})(e) \quad \mathsf{ty}(p) = D\texttt{<}ar,w\texttt{>}$
$\quad h :: s \rightarrow h :: s, e[\mathsf{self}(p,ar,w), \bar{x}{\leftarrow}\bar{v}] \,\mathsf{in}\, p$

(Red Rtr) $\quad e = q.m\texttt{<}\bar{u}\texttt{>}(\bar{v})$ or $e = \mathtt{new}\,C\texttt{<}ar,u\texttt{>}.k(\bar{v})$ or $e = C.k(\bar{v})$
$\quad h :: s, (\mathscr{E}[e] \,\mathsf{in}\, o), (v \,\mathsf{in}\, p) \rightarrow h :: s, \mathscr{E}[v] \,\mathsf{in}\, o$

(Red Let)
$\quad h :: s, \mathscr{E}[\mathtt{let}\,x\texttt{=}v\,\mathtt{in}\,e] \,\mathsf{in}\, p \rightarrow h :: s, \mathscr{E}[e[x{\leftarrow}v]] \,\mathsf{in}\, p$

(Red Cast) $\quad v = \mathtt{null}$ or $\mathsf{rawty}(v) <: C$
$\quad h :: s, \mathscr{E}[(C)v] \,\mathsf{in}\, p \rightarrow h :: s, \mathscr{E}[v] \,\mathsf{in}\, p$

4 Semantic Immutability

Intuitively, an object o is immutable in a given program P, if during execution of P no other object p can see two distinct states of o. A class is immutable if all its instances are immutable in all programs.

In order to formalize this definition, we have to describe the meaning of the phrase "p sees o's state". The object p can read o's fields directly or it can call o's methods and observe possible state changes that way. Thus, if o's object state is always the same on external field reads and in the prestate of external method calls on o, we can be sure that no object p ever sees mutations of o's state.

Definition 1 (Visible States). A *visible state for o* is a state of the form $(h :: s, \mathscr{E}[o.f] \,\mathsf{in}\, p)$ or $(h :: s, \mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})] \,\mathsf{in}\, p)$ where $p \neq o$.

We also have to formalize what o's object state is. Just including the fields of an object is often not enough, because this only allows shallow object states. We interpret the ownership type annotations on fields as specifications of the depth of object states: if a field f's type annotation has the form $C\texttt{<}ar,\mathtt{this}\texttt{>}$ then the state of the object that f refers to is included in `this`'s state; if f's type annotation has the form $C\texttt{<}ar,\mathtt{myowner}\texttt{>}$ then the state of the object that f refers to is included in `myowner`'s state. This is formalized by the following inductive definition:

Definition 2 (Object State). For any heap h, the binary relation $\_ \in \mathsf{state}(h)(\_)$ over $\mathsf{Obj} \times \mathsf{ObjId}$ is defined inductively by the following rules:

(a) If $o\{\bar{f} = \bar{v}\} \in h$, then $o\{\bar{f} = \bar{v}\} \in \mathsf{state}(h)(o)$.
(b) If $o\{..f = q..\} \in h$ and $C\texttt{<}ar,\texttt{this>}\,f \in \mathsf{fd}(\mathsf{rawty}(o))$ and $obj \in \mathsf{state}(h)(q)$, then $obj \in \mathsf{state}(h)(o)$.
(c) If $p \neq o$ and $p\{..f = q..\} \in \mathsf{state}(h)(o)$ and $C\texttt{<}ar,\texttt{myowner>}\,f \in \mathsf{fd}(\mathsf{rawty}(p))$ and $obj \in \mathsf{state}(h)(q)$, then $obj \in \mathsf{state}(h)(o)$.

Let $\mathsf{state}(h)(o) \overset{\Delta}{=} \{obj \,|\, obj \in \mathsf{state}(h)(o)\}$.

Example 1 (Object State).

```
class C ext Object { D<..,this> x; D<..,world> y; constructors methods }
class D ext Object { E<..,myowner> x; E<..,this> y; constructors methods }
class E ext Object { Object<..,myowner> x; constructors methods }
```

Let $c\{\mathtt{x} = d_1, \mathtt{y} = d_2\}$, $d_1\{\mathtt{x} = e_1, \mathtt{y} = e_2\}$, $e_1\{\mathtt{x} = o_1\}$, $e_2\{\mathtt{x} = o_2\}$ be instances of C, D, E in heap h. Then $\mathsf{state}(h)(e_1)$ consists of (the object whose identifier is) e_1; $\mathsf{state}(h)(e_2)$ consists of e_2; $\mathsf{state}(h)(d_1)$ consists of d_1, e_2, o_2; and $\mathsf{state}(h)(c)$ consists of $c, d_1, e_1, o_1, e_2, o_2$. □

Definition 3 (Immutability in a Fixed Program). Suppose $P = (\bar{c}; e_0)$ is a Jimuva-program and C is declared in $\bar{c}$. We say that C *is immutable in* P whenever the following statement holds:

If $\emptyset :: e_0\,\mathsf{in}\,\mathtt{world} \rightarrow^*_{\bar{c}} h_1 :: s_1 \rightarrow^*_{\bar{c}} h_2 :: s_2$,
and $h_1 :: s_1$ and $h_2 :: s_2$ are visible states for o,
and $\mathsf{rawty}(o) <: C$, then $\mathsf{state}(h_1)(o) = \mathsf{state}(h_2)(o)$.

This immutability definition disallows some immutable classes that intuitively could be allowed, because the last line requires $\mathsf{state}(h_1)(o)$ and $\mathsf{state}(h_2)(o)$ to be *exactly identical*. A more liberal definition would allow object state mutations that are unobservable to the outside. For instance, immutable objects with an invisible internal mutable cache for storing results of expensive and commonly called methods could be allowed. However, standard type-based verification techniques would probably disallow unobservable object mutations. Because our primary goal is the design of a sound static type system, we do not attempt to formalize a more permissive definition of immutability up to a notion of observational equivalence of object states, but instead work with our strict definition that is based on exact equality of object states.

We are interested in immutability in an open world, where object immutability cannot be broken by unchecked components. To formally capture the open world model, we define a type erasure mapping $|\cdot|$ from Jimuva to Core Java, see Appendix G for details. This mapping erases ownership information, access rights, expression attributes and class attributes. The operational semantics, $\rightarrow_{\mathsf{java}}$, and typing judgment, $\vdash_{\mathsf{java}}$, for Core Java are defined in Appendix G. The Jimuva typing judgment, $\vdash$, will be defined in Section 6. A Java-*program* is a pair $(\bar{c}; e)$ such that $(\vdash_{\mathsf{java}} \bar{c} : \mathsf{ok})$ and $(\vdash_{\mathsf{java},\bar{c}} e : ty)$ for some Java-type ty. The semantics of Jimuva and Core Java are related as follows:

- If $(\vdash \bar{c} : \mathsf{ok})$, then $(state \rightarrow_{\bar{c}} state')$ iff $(|state| \rightarrow_{\mathsf{java},|\bar{c}|} |state'|)$.

– If $(\vdash \bar{c} : \mathsf{ok})$, then $(\vdash_{\mathsf{java}} |\bar{c}| : \mathsf{ok})$.

There is also an embedding e that maps a Jimuva class table $\bar{c}$ and a Java class table $\bar{d}$ (which refers to $|\bar{c}|$) to a Jimuva class table $\mathsf{e}_{\bar{c}}(\bar{d})$ such that $|\mathsf{e}_{\bar{c}}(\bar{d})| = \bar{d}$, see Appendix G for details. This embedding inserts the annotations `rdwr` and `world` wherever access or ownership parameters are required. One can think of a Java-class as a Jimuva-class without any Jimuva-specific annotations. The embedding e inserts Jimuva-defaults where Jimuva-annotations are syntactically required.

Our type system is sound in an *open world with legal subclassing*. That is, we assume that unchecked classes do not extend Jimuva-annotated classes or override Jimuva-annotated methods. We could easily modify our system to guarantee immutability in an open world without this subclassing restriction, by requiring Jimuva-annotated classes and methods to be `final`. We choose not to, because we find that a bit too restrictive. Note, in this context, that Java's Extension Mechanism already supports *sealed optional packages*, which cannot be extended from outside the JAR file that contains them.[5] It would be nice if there was also a way to mark public classes as sealed, in order to prohibit subclassing from outside their containing JAR file.

Jimuva-annotated classes and methods: A *field declaration* $C\texttt{<}ar,v\texttt{>}\,f$ is *Jimuva-annotated* if $ar \neq \texttt{rdwr}$ or $v \neq \texttt{world}$. A *method* $fm\texttt{<}\bar{y}\texttt{>}\,ea\,ty'\,m(\bar{ty}\,\bar{x})\{e\}$ is Jimuva-*annotated* if $\bar{y}$, ea or $\mathsf{vars}(ty', \bar{ty})$ is non-empty. A *class* $fm\,ca\,\texttt{class}\,C\,\texttt{ext}\,D\{..\}$ is Jimuva-*annotated*, if it contains Jimuva-annotated field declarations or ca is non-empty.

Legal subclassing: A Java class table $\bar{d}$ *legally subclasses* a Jimuva class table $\bar{c}$, if no class declared in $\bar{d}$ extends a Jimuva-annotated class and no method declared in $\bar{d}$ overrides a Jimuva-annotated method.

Definition 4 (Immutability in an Open World). Suppose C is declared in Jimuva-class-table $\bar{c}$ and $(\vdash \bar{c} : \mathsf{ok})$. We say that *$C$ is immutable in $\bar{c}$* whenever C is immutable in $(\bar{c}, \mathsf{e}_{\bar{c}}(\bar{d}); \mathsf{e}_{\bar{c}}(e))$ for all Java-programs $(|\bar{c}|, \bar{d}; e)$ where $\bar{d}$ legally subclasses $\bar{c}$.

Let us say that a class table $\bar{c}$ is *correct for immutability* whenever every class that is declared `immutable` in $\bar{c}$ is in fact immutable in $\bar{c}$. Jimuva's type system is sound in the following sense:

Theorem 1 (Soundness). *If* $(\vdash \bar{c} : \mathsf{ok})$*, then $\bar{c}$ is correct for immutability.*

5 The Immutability Type System – Informally

The simplest example of an `immutable` class is:[6]

```
immutable class ImmutableInt ext Object {
  int value;
  anon wrlocal ImmutableInt.k(int i) { this.value=i; }
  rdonly int get() { this.value } }
```

Here the state of an `ImmutableInt` object just consists of its instance field `value`. For more complicated immutable objects, ownership annotations are needed to specify if objects referenced by instance fields are part of the (immutable) state:

[5] An attempt to extend a sealed optional package results in a `SecurityException` at runtime.
[6] For readability, keywords that could be left implicit are written in italics.

```
  class Mutable ext Object {
    int value;
    anon Mutable.k(int i) { this.value=i; }
    rdonly int get() { this.value }
    void set(int i) { this.value=i; } }

 immutable class EncapsulatedMutable ext Object {
    Mutable<this> m;
    anon wrlocal EncapsulatedMutable.k(Mutable m) {
        this.m = new Mutable<this>.k(m.get()); }
    rdonly int get(){ this.m.get() } }
```

Here the annotation `<this>` on the type of field `m` declares that the state of the object referenced by `m` is considered part of the state of an `EncapsulatedMutable` object. The type system enforces that constructor `EncapsulatedMutable.k(m)` makes a defensive copy of `m` to prevent representation exposure. Technically, this is achieved because `m`'s type `Mutable`, which is short for `Mutable<world>`, is not a subtype of `Mutable<this>` and, thus, a direct assignment to the field `this.m` is disallowed.

Restrictions on methods with `rdonly`. Obviously, methods of an immutable object should not modify their object state. One could try to ensure this by requiring that methods of immutable objects are side-effect free. However, ensuring side-effect freeness is not so simple, because even side-effect free methods must be allowed to call constructors that write to the heap. Limiting constructor writes for side effect freeness in a practical and safe way requires alias control [SR05]. Therefore, instead of requiring side-effect freeness, Jimuva uses a weaker restriction that is simpler to enforce on top of the ownership infrastructure.

> `rdonly`: An expression is *read-only*, if it (1) contains no field assignments, (2) all its method calls have the form $v.m\texttt{<}\bar{u}\texttt{>}(\bar{e})$ where either (a) m is `rdonly` or (b) $\bar{u} = \texttt{world}$ and v has a type $C\texttt{<}ar, \texttt{world}\texttt{>}$, and (3) all its `new`-calls have the form $\texttt{new}\,C\texttt{<}ar,\texttt{world}\texttt{>}.k(\bar{e})$.

`rdonly`-methods are guaranteed to not write to the state of immutable receivers. The `rdonly`-restriction allows important side-effecting methods. For instance, the method `getChars(int srcBegin,int srcEnd,char[] dst,int dstBegin)` from Java's immutable `String` class writes to the array `dst` (owned by `world`). It is an example of a `rdonly` method that is not side-effect free.

Restrictions on constructors with `wrlocal` *and* `anon`. A constructor of an immutable object typically will have side-effects to initialize the object state. We have to restrict constructors of immutable objects for two reasons: (i) we have to prevent them from modifying other objects of the same class, (ii) we have to prevent them from leaking the partially constructed `this` [Goe02].

Issue (i) stems from the fact that visibility modifiers in Java constrain per-class, not per-object, visibility. So it is possible for a constructor of an immutable object to see and modify other immutable objects of the same class. For example:

```
 immutable class Wrong {
     Mutable<this> m;
     rdonly int get(){ m.get() }
```

```
    anon wrlocal Wrong.k(Wrong o) {
          this.m = new Mutable<this>.k(o.get());
          o.m.set(23); /* unwanted side-effect on other object! */ } }
```

To prevent such immutability violations, we require constructors of immutable objects to be write-local in the following sense:

> `wrlocal`: An expression is *write-local*, if (1) all its field assignments have the form $v.f$=e where either $v = \texttt{this}$ or v has a type C<`rdwr`,`this`> and (2) all its method calls have the form $v.m$<$\bar{u}$>$(\bar{e})$ where either (a) m is `rdonly` or (b) m is `wrlocal` and $v = \texttt{this}$ or (c) v has a type C<`rdwr`,`this`> or (d) v is has a type C<ar,`world`>.

To prevent constructors of immutable objects from leaking `this`, we use Vitek et al's notion of anonymity of [VB01,ZPV06]:

> `anon`: An expression is *anonymous*, if it (1) is not `this`, (2) does not pass `this` to foreign methods, (3) does not assign `this` to fields, and (4) all its method calls have the form $v.m$<$\bar{u}$>$(\bar{e})$ where either v or m is `anon`.

Owner-polymorphic methods. The example below uses an owner-polymorphic method to permit dynamic aliasing of the representation object `this.m` during object construction. As explained in Section 2, the polymorphic type of `copy()` prevents this method from creating a static alias to its parameter `to`. This example is a small model of Java's `String` constructor `String(char[] a)`, which gives an alias to a representation object to a global `arraycopy()` method.

```
class Utilities ext Object {
  Utilities.k(){ skip }
  <x,y> void copy(Mutable<x> from, Mutable<y> to){ to.set(from.get()); } }

immutable class EncapsulatedMutable2 ext Object {
  Mutable<this> m;
  anon wrlocal EncapsulatedMutable2.k(Mutable m) {
     this.m = new Mutable<this>.k(null);
     new Utilities.k().copy<world,this>(m,this.m); }
  rdonly int get(){ m.get() } }
```

Now is a good point to present the subtyping relation: Subtyping is defined against a type environment Γ that assigns types to variables. The following function is used in its definition:

$$\mathsf{atts}(\texttt{Object}) \triangleq \emptyset \qquad \mathsf{atts}(C) \triangleq ca,\ \text{if}\, fm\, ca\, \texttt{class}\, C\, \texttt{ext}\, D\{..\} \qquad \mathsf{atts}(\texttt{void}) \triangleq \emptyset$$
$$\mathsf{atts}(C\texttt{<}ar,v\texttt{>}) \triangleq \mathsf{atts}(C) \cup \{ar\} \quad \mathsf{atts}(ea\, ty) \triangleq ea \cup \mathsf{atts}(ty) \quad \mathsf{atts}(o) \triangleq \mathsf{atts}(\mathsf{ty}(o))$$

We interpret expression attributes ea as subsets of $\{\texttt{anon},\texttt{rdonly},\texttt{wrlocal}\}$ ordered by set inclusion.

Subtyping, $\Gamma \vdash T \preceq U$:

(Sub Rep)
$$\frac{\Gamma \vdash ar, v, v' : \mathsf{ok} \qquad C <: C' \quad ea' \subseteq ea}{\Gamma \vdash ea\, C\texttt{<}ar,v\texttt{>} \preceq ea'\, C'\texttt{<}ar,v\texttt{>}}$$

(Sub World)
$$\frac{\Gamma \vdash ar, ar' : \mathsf{ok} \quad ea' \subseteq ea \quad C <: C'}{\Gamma \vdash ea\, C\texttt{<}ar,\texttt{world}\texttt{>} \preceq ea'\, C'\texttt{<}ar',\texttt{world}\texttt{>}}$$

(Sub Void)
$$\frac{ea' \subseteq ea}{\Gamma \vdash ea\,\texttt{void} \preceq ea'\,\texttt{void}}$$

(Sub Share)
$$\frac{ea' \subseteq ea \quad C <: C' \quad \Gamma \vdash v, v' : D, D' \text{ in } \texttt{world} \quad \texttt{immutable} \in \mathsf{atts}(D) \cap \mathsf{atts}(D')}{\Gamma \vdash ea\,C\texttt{<rd},v\texttt{>} \preceq ea'\,C'\texttt{<rd},v'\texttt{>}}$$

The interesting rules are (Sub Share) and (Sub World). The former allows flows of read-restricted objects with immutable owners into locations for read-restricted objects of other immutable owners. That is, our type system permits sharing representation objects among immutable objects as long as those are read-restricted. The rule (Sub World) expresses that ownerless objects do not have to follow access policies. It is needed to ensure that our type system is sound in an open world that includes clients that do not follow Jimuva-policies. Compared to type systems with read references, e.g., the Universe type system [MPH01], it is noteworthy that we do not allow upcasting read-write objects to read objects. Allowing this would lead to an unsoundness in our system. This means that read-restricted objects have to be created as read-restricted objects. Of course, we then must allow constructors of read-restricted objects to initialize their own state. This is safe, as long as constructors of read-restricted objects are `wrlocal`.

Sharing mutable representation objects. This example illustrates sharing of mutable representation objects. The subtyping rule (Sub Share) is used to upcast `o.m`'s type from `SharedRepObject<rd,o>` to `SharedRepObj<rd,this>` so that the assignment to `this.m` becomes possible.

```
immutable class SharedRepObject ext Object {
  Mutable<rd,this> m;
  rdonly int get(){ m.get() } }
  anon wrlocal SharedRepObject.k1(int i) {
      this.m = new Mutable<rd,this>.k(i); }
  anon wrlocal SharedRepObject.k2(SharedRepObject o) {
      this.m = o.m; } /* sharing of mutable representation object */ }
```

6 The Immutability Type System – Formally

A *type environment* $\Gamma = (\Gamma_{\mathsf{acc}}, \Gamma_{\mathsf{own}}, \Gamma_{\mathsf{val}})$ is a triple of partial functions $\Gamma_{\mathsf{acc}} \in \{\texttt{myaccess}\} \to \{\bullet\}$, $\Gamma_{\mathsf{own}} \in \mathsf{Var} \cup \mathsf{ObjId} \to \{\bullet\}$ and $\Gamma_{\mathsf{val}} \in \mathsf{Var} \cup \mathsf{ObjId} \to \mathsf{ExpTy}$. If $v \notin \mathsf{dom}(\Gamma_{\mathsf{val}}) \cup \{\texttt{null}\}$, we define $\Gamma_{\mathsf{val}}, v : T \triangleq \Gamma_{\mathsf{val}} \cup \{(x, T)\}$. Similarly, for Γ_{acc} and Γ_{own}. We define $\Gamma, v : T \triangleq (\Gamma_{\mathsf{acc}}, \Gamma_{\mathsf{own}}, (\Gamma_{\mathsf{val}}, v : T))$. Similarly, for Γ_{acc} and Γ_{own}. We often write $\Gamma(v) = T$ as an abbreviation for $\Gamma_{\mathsf{val}}(v) = T$. Similarly, for Γ_{acc} and Γ_{own}. We define $\mathsf{dom}(\Gamma) \triangleq \mathsf{dom}(\Gamma_{\mathsf{acc}}) \cup \mathsf{dom}(\Gamma_{\mathsf{own}}) \cup \mathsf{dom}(\Gamma_{\mathsf{val}})$.

Substitution Application for Environments, $\Gamma[\sigma]$:

$\Gamma[\sigma] \triangleq (\Gamma_{\mathsf{acc}}[\sigma], \Gamma_{\mathsf{own}}[\sigma], \Gamma_{\mathsf{val}}[\sigma]) \qquad \Gamma_{\mathsf{val}}[\sigma] \triangleq \{(v, T[\sigma]) \mid (v, T) \in \Gamma_{\mathsf{val}}\}$
$\Gamma_{\mathsf{acc}}[\sigma] \triangleq \{(ar[\sigma], \bullet) \mid ar \in \mathsf{dom}(\Gamma_{\mathsf{acc}})\} \cap \{(\texttt{myaccess}, \bullet)\}$
$\Gamma_{\mathsf{own}}[\sigma] \triangleq \{(v[\sigma], \bullet) \mid v \in \mathsf{dom}(\Gamma_{\mathsf{own}})\} \cap (\mathsf{Var} \cup \mathsf{ObjId}) \times \{\bullet\}$

In addition to subtyping, there are judgments of the following forms:

$\vdash c : \mathsf{ok}$	"c is a good class declaration"
$\Gamma \vdash e : T \text{ in } v, ar$	"if $\texttt{this} = v$ and v has access rights ar, then e has type T"

In useful judgments ($\Gamma \vdash e : T$ in v, ar), the `this`-binding v is either `this` itself or an object identifier. For type-checking class declarations, it is sufficient to consider judgments where $\mathsf{dom}(\Gamma) \subseteq \mathsf{Var} \cup \{\texttt{world}, \texttt{myaccess}\}$ and $v = \texttt{this}$. We allow arbitrary object identifiers in type environments and as `this`-binders, so that we can type runtime states, which is needed for proving type soundness.

The typing judgments are defined with respect to an underlying class table. This class table remains fixed in all typing rules and we leave it implicit. In contexts where we want to explicitly mention it, we subscript the turnstyle: ($\Gamma \vdash_{\bar{c}} e : T$ in v, ar). We use auxiliary functions $\mathsf{ctype}(C.k)$ and $\mathsf{mtype}(C, m)$ that compute the types of constructors and methods based on the underlying class table. These are essentially as in FJ [IPW01]. *Method subtyping* treats methods invariantly in the parameter types and covariantly in the result type. See Appendix B for more details.

Auxiliary Predicates and Judgments:

$ea\,C\texttt{<}ar, v\texttt{>}$ legal $\stackrel{\Delta}{=}$ $(v = \texttt{myowner} \Leftrightarrow ar = \texttt{myaccess})$ $\qquad$ $ea\,\texttt{void}$ legal $\stackrel{\Delta}{=}$ true

$C\texttt{<}ar, v\texttt{>}$ generative $\stackrel{\Delta}{=}$ $(\texttt{immutable} \in \mathsf{atts}(C) \Rightarrow v = \texttt{world}$ and $v = \texttt{world} \Rightarrow ar = \texttt{rdwr})$

(ea, u, ar_u, v_u) wrloc in v $\stackrel{\Delta}{=}$ $(u = v, \texttt{wrlocal} \in ea)$ or $(ar_u, v_u) = (\texttt{rdwr}, v)$

ar wrsafe in ar' $\stackrel{\Delta}{=}$ $(ar = \texttt{rdwr}$ or $ar' = \texttt{rd}$ or $ar = ar')$

$(\vdash \bar{c} : \mathsf{ok}) \stackrel{\Delta}{=} (\forall c \in \bar{c})(\vdash c : \mathsf{ok})$ $\qquad$ $(\Gamma \vdash e : T) \stackrel{\Delta}{=} (\Gamma \vdash e : T$ in $\texttt{myaccess})$

$(\Gamma \vdash e : T$ in $ar) \stackrel{\Delta}{=} (\Gamma \vdash e : T$ in $\texttt{this}, ar)$ $\qquad$ $(\Gamma \vdash e : T$ in $v) \stackrel{\Delta}{=} (\Gamma \vdash e : T$ in $v, \texttt{rdwr})$

$(\Gamma \vdash \bar{e}, e : \bar{T}, T$ in $v, ar) \stackrel{\Delta}{=} (\Gamma \vdash \bar{e} : \bar{T}$ in v, ar and $\Gamma \vdash e : T$ in $v, ar)$

$(\Gamma \vdash e : T \preceq U$ in $v, ar) \stackrel{\Delta}{=} (\Gamma \vdash e : T$ in v, ar and $\Gamma \vdash T \preceq U)$

$(\Gamma \vdash \diamond) \stackrel{\Delta}{=} (\texttt{world} \in \mathsf{dom}(\Gamma_{\mathsf{own}})$ and $(\forall v \in \mathsf{dom}(\Gamma_{\mathsf{val}}))(v \neq \texttt{world}$ and $\Gamma \vdash \Gamma_{\mathsf{val}}(v) : \mathsf{ok}))$

$(\Gamma \vdash v : \bullet) \stackrel{\Delta}{=} (\Gamma \vdash \diamond$ and $\Gamma(v) = \bullet)$ $\qquad$ $(\Gamma \vdash v : \mathsf{ok}) \stackrel{\Delta}{=} (\Gamma \vdash \diamond$ and $v \in \mathsf{dom}(\Gamma) \cup \{\texttt{null}\})$

$(\Gamma \vdash ar : \mathsf{ok}) \stackrel{\Delta}{=} (\Gamma \vdash \diamond$ and $ar \in \mathsf{dom}(\Gamma) \cup \{\texttt{rdwr}, \texttt{rd}\})$

$(\Gamma \vdash ea\,\texttt{void} : \mathsf{ok}) \stackrel{\Delta}{=} (\Gamma \vdash \diamond)$ $\qquad$ $(\Gamma \vdash ea\,C\texttt{<}ar, v\texttt{>} : \mathsf{ok}) \stackrel{\Delta}{=} (\Gamma \vdash ar : \mathsf{ok}$ and $\Gamma \vdash v : \mathsf{ok})$

$(\Gamma \vdash ty$ legal$) \stackrel{\Delta}{=} (\Gamma \vdash ty : \mathsf{ok}$ and ty legal$)$

Good Class Declarations, $\vdash c : \mathsf{ok}$**:**

(Cls Dcl)
$$\frac{\begin{array}{l} D \text{ is not } \texttt{final} \quad \Gamma = (\texttt{world}, \texttt{myowner}, \texttt{myaccess}, \texttt{this} : \bullet) \\ ca \neq \emptyset \Rightarrow (\mathsf{atts}(D) \neq \emptyset \text{ or } D = \texttt{Object}) \quad \mathsf{atts}(D) \neq \emptyset \Rightarrow ca \neq \emptyset \\ \Gamma, \texttt{this} : \texttt{rdonly wrlocal}\, C\texttt{<myaccess,myowner>} \vdash \bar{F}, \bar{K}, \bar{M} : \mathsf{ok} \text{ in } C \end{array}}{\vdash fm\, ca\, \texttt{class}\, C\, \texttt{ext}\, D\, \{\bar{F}\ \bar{K}\ \bar{M}\} : \mathsf{ok}}$$

(Fld Dcl)
$$\frac{C\,\texttt{ext}\,D \Rightarrow f \notin \mathsf{fd}(D) \quad E\texttt{<}ar, v\texttt{>} \text{ legal} \quad \Gamma \vdash ar : \mathsf{ok} \quad \Gamma \vdash v : \bullet}{\Gamma \vdash E\texttt{<}ar, v\texttt{>}\, f : \mathsf{ok} \text{ in } C}$$

(Cons Dcl)
$$\frac{\begin{array}{l} \bar{ty} \text{ legal} \quad \texttt{this} \notin \mathsf{vars}(\bar{ty}) \\ \mathsf{atts}(C) \neq \emptyset \Rightarrow \texttt{anon}, \texttt{wrlocal} \in ea \quad \Gamma, \bar{x} : \texttt{anon rdonly wrlocal}\, \bar{ty} \vdash e : ea\,\texttt{void} \end{array}}{\Gamma \vdash ea\, C.k(\bar{ty}\,\bar{x})\{e\} : \mathsf{ok} \text{ in } C}$$

(Mth Dcl)
$$\frac{\begin{array}{l} C\,\texttt{ext}\,D \Rightarrow \Gamma \vdash \mathsf{mtype}(m, C) \preceq \mathsf{mtype}(m, D) \quad \mathsf{atts}(C) \neq \emptyset \Rightarrow \texttt{rdonly} \in \mathsf{atts}(T) \\ ar = \texttt{myaccess} \text{ or } (\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea = \emptyset, ar = \texttt{rdwr}) \quad \sigma = (\texttt{myaccess} \leftarrow ar) \\ \Gamma[\sigma], \bar{y} : \bullet, \bar{x} : \texttt{anon rdonly wrlocal}\, \bar{ty}[\sigma] \vdash e[\sigma] : T[\sigma] \text{ in } ar \quad \bar{ty}, T \text{ legal} \quad \texttt{this} \notin \mathsf{vars}(\bar{ty}, T) \end{array}}{\Gamma \vdash fm\texttt{<}\bar{y}\texttt{>}\, T\, m(\bar{ty}\,\bar{x})\{e\} : \mathsf{ok} \text{ in } C}$$

Well-typed Expressions, $\Gamma \vdash e : T \ \mathsf{in}\ v, ar$**:**

$$(\mathrm{Var})\ \frac{\begin{array}{l}\Gamma(x) = T \\ \Gamma \vdash ar_v, v : \mathsf{ok}, \bullet\end{array}}{\Gamma \vdash x : T \ \mathsf{in}\ v, ar_v} \qquad (\mathrm{Obj})\ \frac{\begin{array}{l}\Gamma(o) = T \\ ea = \{\texttt{anon} \,|\, o \neq p\} \quad \Gamma \vdash ar_p, p : \mathsf{ok}, \bullet\end{array}}{\Gamma \vdash o : ea\, T \ \mathsf{in}\ p, ar_p} \qquad (\mathrm{Sub})\ \frac{\Gamma \vdash e : T \preceq U \ \mathsf{in}\ v, ar_v}{\Gamma \vdash e : U \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{Null})\ \frac{\Gamma \vdash T, ar_v, v : \mathsf{ok}, \mathsf{ok}, \bullet}{\Gamma \vdash \texttt{null} : T \ \mathsf{in}\ v, ar_v} \qquad (\mathrm{Let})\ \frac{\begin{array}{l}\Gamma \vdash e : ea_e\, ty_e \ \mathsf{in}\ v, ar_v \quad x \notin \mathsf{vars}(ty_{e'}) \\ \Gamma, x : ea_e\, ty_e \vdash e' : ea_{e'}\, ty_{e'} \ \mathsf{in}\ v, ar_v \quad ea = \bigcap(ea_e, ea_{e'})\end{array}}{\Gamma \vdash \texttt{let}\, x \texttt{=} e\, \texttt{in}\, e' : ea\, ty_{e'} \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{Cast})\ \frac{\begin{array}{l}C \text{ declared} \\ \Gamma \vdash e : ea_e\, C_e\texttt{<}ar_e, v_e\texttt{>} \ \mathsf{in}\ v, ar_v\end{array}}{\Gamma \vdash (C)e : ea_e\, C\texttt{<}ar_e, v_e\texttt{>} \ \mathsf{in}\ v, ar_v} \qquad (\mathrm{Get})\ \frac{\begin{array}{l}ty\, f \in \mathsf{fd}(C_u) \quad \sigma = \mathsf{self}(u, ar_u, v_u) \\ \Gamma \vdash u, v : ea_u\, C_u\texttt{<}ar_u, v_u\texttt{>}, C_u\texttt{<}ar_v, w_v\texttt{>} \ \mathsf{in}\ v, ar_v\end{array}}{\Gamma \vdash u.f : \texttt{anon}\ \texttt{rdonly}\ \texttt{wrlocal}\, ty[\sigma] \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{Set})\ \frac{\begin{array}{l}ty\, f \in \mathsf{fd}(C_u) \quad \Gamma \vdash v_u : \bullet \\ ea = \bigcap(\{x \ \mathsf{as}\ \texttt{wrlocal} \,|\, (\{x\}, u, ar_u, v_u) \ \mathsf{wrloc\ in}\ v\} \cup \{\texttt{anon}\}, ea_e) \quad ar_u \ \mathsf{wrsafe\ in}\ ar_v \\ \Gamma \vdash u, v, e : ea_u\, C_u\texttt{<}ar_u, v_u\texttt{>}, C_u\texttt{<}ar_v, w_v\texttt{>}, ea_e\, ty[\sigma] \ \mathsf{in}\ v, ar_v \quad \sigma = \mathsf{self}(u, ar_u, v_u)\end{array}}{\Gamma \vdash u.f \texttt{=} e : ea\, ty[\sigma] \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{Call})\ \frac{\begin{array}{l}\mathsf{mtype}(m, C_u) = fm\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow ea_m\, ty' \\ (\texttt{rdonly} \in ea_m) \text{ or } (ar_u \ \mathsf{wrsafe\ in}\ ar_v) \qquad \sigma = \mathsf{self}(u, ar_u, v_u), \bar{y} \leftarrow \bar{w} \\ ea = \bigcap \bar{ea}_{\bar{e}} \cap \bigcup(\ \{\texttt{anon}\} \cap (ea_m \cup ea_u),\ \{x \ \mathsf{as}\ \texttt{rdonly} \,|\, x \in ea_m \text{ or } v_u, \bar{w} = \texttt{world}\}, \\ \quad \{\texttt{wrlocal} \,|\, (ea_m, u, ar_u, v_u) \ \mathsf{wrloc\ in}\ v \text{ or } \texttt{rdonly} \in ea_m \text{ or } v_u = \texttt{world}\}\ ) \\ \Gamma \vdash u, \bar{e} : ea_u\, C_u\texttt{<}ar_u, v_u\texttt{>}, \bar{ea}_{\bar{e}}\, \bar{ty}[\sigma] \ \mathsf{in}\ v, ar_v \quad \Gamma \vdash \bar{w} : \bullet \quad (ar_u = \texttt{rd} \text{ or } \Gamma \vdash v_u : \bullet)\end{array}}{\Gamma \vdash u.m\texttt{<}\bar{w}\texttt{>}(\bar{e}) : ea\, ty'[\sigma] \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{New})\ \frac{\begin{array}{l}\mathsf{ctype}(C.k) = \bar{ty} \rightarrow ea_k\, \texttt{void} \quad (ar = \texttt{rdwr}) \text{ or } (\texttt{wrlocal}, \texttt{anon} \in ea_k) \\ ea = \bigcap(\{\texttt{rdonly} \,|\, w = \texttt{world}\} \cup \{\texttt{wrlocal}, \texttt{anon}\}, \bar{ea}_{\bar{e}}) \quad \Gamma \vdash ar, w : \mathsf{ok}, \bullet \\ \Gamma \vdash \bar{e} : \bar{ea}_{\bar{e}}\, \bar{ty}[\sigma] \ \mathsf{in}\ v, ar_v \quad \sigma = \mathsf{self}(\texttt{null}, ar, w) \quad C\texttt{<}ar, w\texttt{>} \ \mathsf{generative}\end{array}}{\Gamma \vdash \texttt{new}\, C\texttt{<}ar, w\texttt{>}.k(\bar{e}) : ea\, C\texttt{<}ar, w\texttt{>} \ \mathsf{in}\ v, ar_v}$$

$$(\mathrm{Cons})\ \frac{\begin{array}{l}\mathsf{ctype}(C.k) = \bar{ty} \rightarrow ea_k\, \texttt{void} \quad \sigma = \mathsf{self}(v, ar_v, w_v) \\ \Gamma \vdash \bar{e}, v : \bar{ea}_{\bar{e}}\, \bar{ty}[\sigma], C\texttt{<}ar_v, w_v\texttt{>} \ \mathsf{in}\ v, ar_v \quad ea = \bigcap(ea_k, \bar{ea}_{\bar{e}})\end{array}}{\Gamma \vdash C.k(\bar{e}) : ea\, \texttt{void} \ \mathsf{in}\ v, ar_v}$$

7 Conclusion

More on related work. We have already referenced and compared to some related work throughout the text and have no space to repeat all of that. Ernst et al's Javari language [BE04,TE05] statically checks reference immutability, i.e., read-only references. They report an impressive implementation. They do not support object immutability in an open world, like we do. In particular, their system does not fully prevent representation exposure. Pechtchanski et al [PS05] and Porat et al [PBKM00] present immutability analyses for Java. Their analyses are implementation driven and are not designed against a formal semantics like ours. Parts of our formal type system are inspired by similar informal static rules from Jan Schäfer's masters thesis [Sch04]. Clarke and Drossopolous [CD02] and Lu and Potter [LP06b,LP06a] combine ownership type systems with systems to control write- and/or read-effects. In spirit, this is similar to our system which contains a write-effect analysis (for `rdonly` and `wrlocal`) on top of an ownership type system. In contrast to the above mentioned systems, our sytem supports an open world and treats object constructors. Our system does not control read-effects. However, a read-effect analysis would be desirable, because for many applications of immutability, e.g., thread safety, it is important that immutable objects do not read from mutable state. We expect that we could combine our system with a variant of [CD02]'s read effect analysis to achieve this.

Summary. We have presented a core Java language with statically checkable immutability specifications in the form of a type system, which has been proved sound w.r.t. a formal semantic definition of object immutablity. The system is quite flexible and employs, for instance, owner-polymorphic methods to permit dynamic aliasing during object construction, and read-only objects to permit sharing of mutable representation objects among immutable objects of the same class. We view this paper as the careful design for a sound, type-based immutability analysis and plan to implement an immutability checker for Java based on this system.

Appendix

A More Examples

Building Immutable Objects from Immutable Objects. Jimuva supports building immutable objects from immutable objects. This is a good way of building immutable objects from scratch.

```
immutable class IntList ext Object { }

immutable class ConsList ext IntList {
  Int val;
  IntList tl;
  anon wrlocal ConsList.k(Int val, IntList tl) {
    this.val = val; this.tl = tl;
  }
}

immutable class NilList ext IntList {
  anon wrlocal NilList.k(){ skip }
}
```

Access Right Parametricity for Flexible Sharing. Our final example shows an immutable list constructed by encapsulating a mutable one. To avoid extensive copying, mutable list nodes can be shared. First we present a simple implementation of a mutable linked list class Node. The method deepcopy builds a read-restricted Node object. To make deepcopy more generic, it may be desirable to parameterize it on the access rights for the constructed Node. We believe that access parametricity for methods would be sound and useful, but decided not to support this in order to make the system not too complex.

```
class Node ext Object { // this class fakes a Java interface
  wrlocal void setVal(int val){ skip }
  wrlocal void setNext(Node<myaccess,myowner> next){ skip }
  rdonly int getVal(){ 0 }
  rdonly Node<myaccess,myowner> getNext(){ null }
  <x> Node<rd,x> deepcopy(){ new NilNode<rd,x>.k() }
}

class ValNode ext Node {
  int val;
  Node<myaccess,myowner> next;

  wrlocal ValNode.k(int val, Node<myaccess,myowner> next){
    this.setVal(val); this.setNext(next); }

  wrlocal void setVal(int val){ this.val = val; }
  wrlocal void setNext(Node<myaccess,myowner> next){ this.next = next; }
  rdonly int getVal(){ val }
  rdonly Node<myaccess,myowner> getNext(){ next }

  <x> Node<rd,x> deepcopy() {
    new ValNode<rd,x>.k(this.val, this.next.deepcopy<x>()) }
}

class NilNode ext Node {
  wrlocal NilNode.k(){ skip }
}
```

The class `IntList2` has three constructors. Typechecking the constructor `IntList2.tl` requires that the type `Node<rd,cons>` of `hd` is a subtype of the type `Node<rd,this>` of `this.hd`, which holds by (Sub Share). In `IntList2.cons`, we need to instantiate the `myaccess`-parameter of `ValNode`. Note that we would not be able to type `IntList2.cons` without access parametricity for classes.

```
immutable class IntList2 ext Object {
  Node<rd,this> hd;

  anon wrlocal IntList2.k(Node hd){
    this.hd = hd.deepcopy<this>();
  }
  anon wrlocal IntList2.tl(IntList2 cons) {
    let /*Node<rd,cons>*/ hd = cons.hd in this.hd = hd.getNext();
  }
  anon wrlocal IntList2.cons(int hd, IntList2 tl) {
    let /*Node<rd,tl>*/ tl = tl.hd
    in this.hd = new ValNode<rd,this>.k(hd,tl);
  }

  rdonly int hd(){ hd.getVal() }
  rdonly IntList2 tl(){ new IntList2.tl(this) }
  rdonly IntList2 cons(int hd){ new IntList2.cons(hd,this) }
}
```

B Auxiliary Definitions

In the following definitions, we omit the underlying class table $\bar{c}$. For instance, we write $\mathsf{cbody}(C.k)$ instead of $\mathsf{cbody}_{\bar{c}}(C.k)$ and $\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{\bar{F}\;\bar{K}\;\bar{M}\}$ instead of $\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{\bar{F}\;\bar{K}\;\bar{M}\} \in \bar{c}$.

Field Lookup, $\mathsf{fd}(C) = \bar{ty}\,\bar{f}$**:**

(Fields Base)

$$\frac{}{\mathsf{fd}(\texttt{Object}) = \emptyset}$$

(Fields Ind)

$$\frac{\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{\bar{ty}\,\bar{f}\;\bar{K}\;\bar{M}\} \quad \mathsf{fd}(D) = \bar{ty}_0\,\bar{f}_0}{\mathsf{fd}(C) = \bar{ty}_0\,\bar{f}_0\,\bar{ty}\,\bar{f}}$$

Constructor and Method Lookup:

$$\begin{array}{rll}
\mathsf{cbody}(C.k) & \triangleq (\bar{x})(e) & \text{if } \mathsf{lkup}(C.k) = \mathit{ea}\,C.k(\bar{ty}\,\bar{x})\{e\} \\
\mathsf{mbody}(m,C) & \triangleq \texttt{<}\bar{y}\texttt{>}(\bar{x})(e) & \text{if } \mathsf{lkup}(m,C) = \mathit{fm}\texttt{<}\bar{y}\texttt{>}\,T\,m(\bar{ty}\,\bar{x})\{e\} \\
\mathsf{ctype}(C.k) & \triangleq \bar{ty} \rightarrow \mathit{ea}\,\texttt{void} & \text{if } \mathsf{lkup}(C.k) = \mathit{ea}\,C.k(\bar{ty}\,\bar{x})\{e\} \\
\mathsf{mtype}(m,C) & \triangleq \mathit{fm}\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow T & \text{if } \mathsf{lkup}(m,C) = \mathit{fm}\texttt{<}\bar{y}\texttt{>}\,T\,m(\bar{ty}\,\bar{x})\{e\}
\end{array}$$

(Clkup) $C \neq \texttt{Object}$

$$\frac{\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{..\mathit{ea}\,C.k(\bar{ty}\,\bar{x})\{e\}..\}}{\mathsf{lkup}(C.k) = \mathit{ea}\,C.k(\bar{ty}\,\bar{x})\{e\}}$$

(Clkup Obj)

$$\frac{\mathit{ea} = \texttt{anon}\;\texttt{rdonly}\;\texttt{wrlocal}}{\mathsf{lkup}(\texttt{Object}.k) = \mathit{ea}\,\texttt{Object}.k(\,)\{\texttt{skip}\}}$$

(Mlkup Base)

$$\frac{\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{..\mathit{fm}'\,\texttt{<}\bar{y}\texttt{>}\,T\,m(\bar{ty}\,\bar{x})\{e\}..\}}{\mathsf{lkup}(m,C) = \mathit{fm}'\,\texttt{<}\bar{y}\texttt{>}\,T\,m(\bar{ty}\,\bar{x})\{e\}}$$

(Mlkup Ind)

$$\frac{\mathit{fm}\,\mathit{ca}\,\texttt{class}\,C\,\texttt{ext}\,D\,\{\bar{F}\;\bar{K}\;\bar{M}\} \quad m \text{ not defined in } \bar{M} \quad \mathsf{lkup}(m,D) = \mathit{mdef}}{\mathsf{lkup}(m,C) = \mathit{mdef}}$$

Method Subtyping, $\Gamma \vdash \mathit{fm}\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow T \preceq \mathit{fm}'\,\texttt{<}\bar{y}'\texttt{>}\bar{ty}' \rightarrow T'$**:**

(Meth Sub)

$$\frac{\texttt{final} \notin \mathit{fm}' \quad \Gamma, \bar{y} : \bullet \vdash T \preceq T'}{\Gamma \vdash \mathit{fm}\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow T \preceq \mathit{fm}'\,\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow T'}$$

Definitions for Environments:

Saturated Environments:

Γ is *saturated* $\triangleq \Gamma \vdash \diamond$ and $\mathsf{dom}(\Gamma_{\mathsf{own}}) \subseteq \mathsf{dom}(\Gamma_{\mathsf{val}}) \cup \{\texttt{world}\}$

Closed Environments:

Γ is *closed* $\triangleq \mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId}$ and Γ is saturated

Environment Containment:

$\Gamma \subseteq \Gamma' \triangleq (\Gamma_{\mathsf{acc}} \subseteq \Gamma'_{\mathsf{acc}}$ and $\Gamma_{\mathsf{own}} \subseteq \Gamma'_{\mathsf{own}}$ and $\Gamma_{\mathsf{val}} \subseteq \Gamma'_{\mathsf{val}})$

Domain Restriction:

$f|\bar{v} \triangleq \{(v, f(v)) \mid v \in \mathsf{dom}(f) \cap \bar{v}\}$

Environment Restriction:

$\Gamma|\bar{v} \triangleq (\Gamma_{\mathsf{acc}}, \Gamma_{\mathsf{own}}|\bar{v}, \Gamma_{\mathsf{val}}|\bar{v})$

Environment Modification:

$att\,\Gamma_{\mathsf{val}} \overset{\Delta}{=} \{(v,T) \mid (v, att\,T) \in \Gamma_{\mathsf{val}}\}$ $\qquad att\,\Gamma \overset{\Delta}{=} (\Gamma_{\mathsf{acc}}, \Gamma_{\mathsf{own}}, att\,\Gamma_{\mathsf{val}})$

Global type environment induced by the class map ty:

$\mathsf{env}(\mathsf{ty}) \overset{\Delta}{=} \mathtt{rdonly}\ \mathtt{wrlocal}\,(\emptyset, \{(\mathtt{world}, \bullet)\}, \mathsf{ty})$

C General Properties

Lemma 1 (Good Environments and Types).

(a) *If* $(\Gamma \vdash \diamond)$*, then* $(\Gamma \vdash \Gamma(x) : \mathsf{ok})$ *for all* x *in* $\mathsf{dom}(\Gamma_{\mathsf{val}})$.
(b) *If* $(\Gamma \vdash T : \mathsf{ok})$, $(\Gamma \vdash v : \mathsf{ok})$, $(\Gamma \vdash v : \bullet)$ *or* $(\Gamma \vdash ar : \mathsf{ok})$*, then* $(\Gamma \vdash \diamond)$.
(c) *If* $(\Gamma \vdash T \preceq U)$*, then* $(\Gamma \vdash T, U : \mathsf{ok})$.
(d) *If* $(\bar{c} : \mathsf{ok})$ *and* $(\Gamma \vdash_{\bar{c}} e : T \text{ in } v, ar)$*, then* $\mathsf{vars}(e) \subseteq \mathsf{dom}(\Gamma)$ *and* $(\Gamma \vdash T, ar, v : \mathsf{ok}, \mathsf{ok}, \bullet)$.

Lemma 2. *The subtyping relation* $\preceq$ *is a partial order:*

- *If* $(\Gamma \vdash T : \mathsf{ok})$*, then* $(\Gamma \vdash T \preceq T)$.
- *If* $(\Gamma \vdash T \preceq U)$ *and* $(\Gamma \vdash U \preceq V)$*, then* $(\Gamma \vdash T \preceq V)$.

Proof. Reflexivity is obvious. For transitivity, one inspects all possible reasons for $(\Gamma \vdash T \preceq U)$ and $(\Gamma \vdash U \preceq V)$. The following table summarizes the proof.

reason for $(\Gamma \vdash T \preceq U)$	reason for $(\Gamma \vdash U \preceq V)$	reason for $(\Gamma \vdash T \preceq V)$
(Sub Void)	(Sub Void)	(Sub Void)
(Sub Void)	other than (Sub Void)	premise impossible
other than (Sub Void)	(Sub Void)	premise impossible
(Sub Rep)	(Sub Rep)	(Sub Rep)
(Sub Rep)	(Sub Share)	(Sub Share)
(Sub Rep)	(Sub World)	(Sub World)
(Sub Share)	(Sub Rep)	(Sub Share)
(Sub Share)	(Sub Share)	(Sub Share)
(Sub Share)	(Sub World)	premise impossible
(Sub World)	(Sub Rep)	(Sub World)
(Sub World)	(Sub Share)	premise impossible
(Sub World)	(Sub World)	(Sub World)

□

Part (e) of the following Lemma 3 is important. We make extensive use of it in the type preservation proof. This is the technical reason why we require that field types, method types and constructor types are legal (see typing rules (Fld Dcl), (Cons Dcl) and (Mth Dcl)).

Lemma 3 (Subtyping Properties).

(a) $(\Gamma \vdash ea\,ty \preceq ea'\,ty')$ *if and only if* $ea' \subseteq ea$ *and* $(\Gamma \vdash ty \preceq ty')$.
(b) *If* $(\Gamma \vdash C\texttt{<}ar, v\texttt{>} \preceq C'\texttt{<}ar', v'\texttt{>})$*, then* $C <: C'$.
(c) *If* $(\Gamma \vdash C\texttt{<}ar, v\texttt{>} \preceq C'\texttt{<}ar', v'\texttt{>})$ *and* $D <: D'$,
then $(\Gamma \vdash D\texttt{<}ar, v\texttt{>} \preceq D'\texttt{<}ar', v'\texttt{>})$.
(d) *If* $(\Gamma \vdash C_1\texttt{<}ar_1, v_1\texttt{>} \preceq C_2\texttt{<}ar_2, v_2\texttt{>})$,
then $(\Gamma \vdash C_1\texttt{<}ar_2, v_2\texttt{>} \preceq C_2\texttt{<}ar_1, v_1\texttt{>})$

(e) *If* $(\Gamma \vdash C_1\texttt{<}ar_1, v_1\texttt{>} \preceq C_2\texttt{<}ar_2, v_2\texttt{>})$, $(\Gamma \vdash ty\ \mathsf{legal})$, $(\Gamma \vdash \sigma(x) : \mathsf{ok})$ *for all* x *in* $\mathsf{dom}(\sigma)$ *and* $\texttt{myaccess}, \texttt{myowner} \notin \mathsf{dom}(\sigma)$, *then* $(\Gamma \vdash ty[\sigma, \texttt{myaccess}, \texttt{myowner} \leftarrow ar_1, v_1] \preceq ty[\sigma, \texttt{myaccess}, \texttt{myowner} \leftarrow ar_2, v_2])$.

Proof. Parts (a) through (d) are straightforward to prove by inspection of the subtyping rules. We prove part (e):

(1) $\Gamma \vdash C_1\texttt{<}ar_1, v_1\texttt{>} \preceq C_2\texttt{<}ar_2, v_2\texttt{>}$	assumption
(2) $\Gamma \vdash ty\ \mathsf{legal}$	assumption
(3) $\texttt{myaccess}, \texttt{myowner} \notin \mathsf{dom}(\sigma)$	assumption
(4) $\Gamma \vdash \sigma(x) : \mathsf{ok}$ for all x in $\mathsf{dom}(\sigma)$	assumption
(5) $\sigma_1 \triangleq \sigma, \texttt{myaccess}, \texttt{myowner} \leftarrow ar_1, v_1$	abbreviation
(6) $\sigma_2 \triangleq \sigma, \texttt{myaccess}, \texttt{myowner} \leftarrow ar_2, v_2$	abbreviation

Case 1, $ty = \texttt{void}$: Then $ty[\sigma_1] = ty = ty[\sigma_2]$.

Case 2, $ty = C_3\texttt{<}ar_3, v_3\texttt{>}$: By part (c) of this lemma, we get:

(2.1) $\Gamma \vdash C_3\texttt{<}ar_1, v_1\texttt{>} \preceq C_3\texttt{<}ar_2, v_2\texttt{>}$

Case 2.1, $ar_3 = \texttt{myaccess}, v_3 = \texttt{myowner}$: Then $\Gamma \vdash ty[\sigma_1] = C_3\texttt{<}ar_1, v_1\texttt{>} \preceq C_3\texttt{<}ar_2, v_2\texttt{>} = ty[\sigma_2]$, by (2.1).

Case 2.2, $ar_3 \neq \texttt{myaccess}$ or $v_3 \neq \texttt{myowner}$: Then both $ar_3 \neq \texttt{myaccess}$ and $v_3 \neq \texttt{myowner}$, because ty is legal. Then $ty[\sigma_1] = ty[\sigma] = ty[\sigma_2]$. □

We use the meta-variable $\mathscr{J}$ to range over right-hand-sides of judgments, i.e., over the forms $\diamond$, $(ar : \mathsf{ok})$, $(v : \bullet)$, $(v : \mathsf{ok})$, $(T : \mathsf{ok})$, $(T \preceq U)$, and $(e : T\ \mathsf{in}\ v, ar)$.

Lemma 4 (Weakening). *If* $(\Gamma \vdash \mathscr{J})$, $(\Gamma \subseteq \Gamma')$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma' \vdash \mathscr{J})$.

Lemma 5 (Type Specialization). *If* $(\Gamma, v : U \vdash \mathscr{J})$ *and* $(\Gamma, v : T \vdash T \preceq U)$, *then* $(\Gamma, v : T \vdash \mathscr{J})$.

Recall that Γ is called saturated iff $(\Gamma \vdash \diamond)$ and $\mathsf{dom}(\Gamma_{\mathsf{own}}) \subseteq \mathsf{dom}(\Gamma_{\mathsf{val}}) \cup \{\texttt{world}\}$. The next lemma captures that for some judgments the owner component Γ_{own} of saturated environments Γ does not matter. In particular, this is is the case for subtyping judgments.

Lemma 6 (Independence of Γ_{own}). *Suppose* $\mathscr{J}$ *is* $\diamond$, $(ar : \mathsf{ok})$, $(v : \mathsf{ok})$, $(T : \mathsf{ok})$, $(T \preceq U)$ *or* $(v : T\ \mathsf{in}\ \texttt{world}, ar)$. *Then the following holds: If* $(\Gamma \vdash \mathscr{J})$, Γ, Γ' *are saturated,* $\Gamma_{\mathsf{acc}} = \Gamma'_{\mathsf{acc}}$ *and* $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}}$, *then* $(\Gamma' \vdash \mathscr{J})$.

For $att \in \{\texttt{anon}, \texttt{rdonly}, \texttt{wrlocal}\}$, we define:

$$att\,(ea\,ty) \triangleq (ea \cup \{att\})\,ty \qquad (ea\,ty) - att \triangleq (ea - \{att\})\,ty$$

Lemma 7 (Monotonicity of Union and Difference).

(a) *If* $(\Gamma \vdash T \preceq U)$, *then* $(\Gamma \vdash att\,T \preceq att\,U)$ *and* $(\Gamma \vdash T - att \preceq U - att)$.
(b) *If* $(\Gamma, u : T \vdash e : T\ \mathsf{in}\ v, ar)$, *then* $(\Gamma, u : T - att \vdash e : T - att\ \mathsf{in}\ v, ar)$.

Lemma 8 (Independence of `this`-Binding and Access for Values).

(a) *If* $(\Gamma \vdash v : T \text{ in } w, ar)$ *and* $(\Gamma \vdash ar' : \mathsf{ok})$*, then* $(\Gamma \vdash v : T \text{ in } w, ar')$.
(b) *If* $(\Gamma \vdash v : T \text{ in } w, ar)$ *and* $(\Gamma \vdash u : \mathsf{ok})$*, then* $(\Gamma \vdash v : (T - \texttt{anon}) \text{ in } u, ar)$.
(c) *If* $(\Gamma \vdash v : T \text{ in } w, ar)$*, then* $(\Gamma \vdash v : T \text{ in } \texttt{world})$.

Lemma 9 (Substitutivity). *If* $(\vdash \bar{c} : \mathsf{ok})$, $(\Gamma[x\leftarrow v] \vdash v : T[x\leftarrow v] \text{ in } u[x\leftarrow v], ar)$ *and* $(\Gamma, x : T \vdash \mathscr{J})$*, then the following statements hold:*

(a) *If* $\mathscr{J}$ *is* $\diamond$ *or* $(ar : \mathsf{ok})$ *or* $(v : \bullet)$ *or* $(v : \mathsf{ok})$ *or* $(U : \mathsf{ok})$ *or* $(U \preceq V)$*,*
then $(\Gamma[x\leftarrow v] \vdash \mathscr{J}[x\leftarrow v])$.
(b) *If* $\mathscr{J}$ *is* $(e : U \text{ in } u, ar_u)$*, then* $(\Gamma[x\leftarrow v] \vdash e[x\leftarrow v] : U[x\leftarrow v] \text{ in } u[x\leftarrow v], ar_u)$.

Proof. By induction on $(\Gamma, x : T \vdash \mathscr{J})$'s derivation:

(1)	$\vdash \bar{c} : \mathsf{ok}$	assumption
(2)	$\sigma = (x\leftarrow v)$	abbreviation
(3)	$\Gamma[\sigma] \vdash v : T[\sigma] \text{ in } u[\sigma], ar$	assumption
(4)	$\Gamma' = (\Gamma, x : T)$	abbreviation

Case 1, (Var), $y \in \mathsf{Var} - \{x\}$:

$$\frac{\Gamma'(y) = ea\,C\texttt{<}ar_y, v_y\texttt{>} \quad \Gamma' \vdash ar_u, u : \mathsf{ok}, \bullet}{\Gamma' \vdash y : ea\,C\texttt{<}ar_y, v_y\texttt{>} \text{ in } u, ar_u}$$

(1.1)	$\Gamma[\sigma] \vdash ar_u, u[\sigma] : \mathsf{ok}, \bullet$	by i.h.
(1.2)	$(\Gamma[\sigma])(y) = (\Gamma(y))[\sigma] = ea\,C\texttt{<}ar_y, v_y[\sigma]\texttt{>}$	
(1.3)	$\Gamma[\sigma] \vdash y : ea\,C\texttt{<}ar_y, v_y[\sigma]\texttt{>} \text{ in } u[\sigma], ar_u$	by (Var)

Case 2, (Var), $y = x$:

$$\frac{\Gamma'(x) = ea\,C\texttt{<}ar_x, v_x\texttt{>} \quad \Gamma' \vdash ar_u, u : \mathsf{ok}, \bullet}{\Gamma' \vdash x : ea\,C\texttt{<}ar_x, v_x\texttt{>} \text{ in } u, ar_u}$$

We have $T = ea\,C\texttt{<}ar_x, v_x\texttt{>} = \Gamma'(x)$ and $x[\sigma] = v$. Therefore, $(\Gamma[\sigma] \vdash x[\sigma] : ea\,C\texttt{<}ar_x, v_x[\sigma]\texttt{>} \text{ in } u[\sigma], ar)$, by assumption (3). By Lemma 8, we obtain $(\Gamma[\sigma] \vdash x[\sigma] : ea\,C\texttt{<}ar_x, v_x[\sigma]\texttt{>} \text{ in } u[\sigma], ar_u)$.

Case 3, (Sub Share):

$$\frac{\begin{array}{l} ea' \subseteq ea \quad C <: C' \\ \Gamma' \vdash u, u' : D, D' \text{ in } \texttt{world} \quad \texttt{immutable} \in \mathsf{atts}(D) \cap \mathsf{atts}(D') \end{array}}{\Gamma' \vdash ea\,C\texttt{<rd}, u\texttt{>} \preceq ea'\,C'\texttt{<rd}, u'\texttt{>}}$$

(3.1)	$\Gamma[\sigma] \vdash v : T[\sigma] \text{ in } \texttt{world} = \texttt{world}[\sigma]$	by assumption (3) and Lemma 8
(3.2)	$\Gamma[\sigma] \vdash u[\sigma], u'[\sigma] : D, D' \text{ in } \texttt{world}$	by i.h.
(3.3)	$\Gamma[\sigma] \vdash ea\,C\texttt{<rd}, u[\sigma]\texttt{>} \preceq ea'\,C'\texttt{<rd}, u'[\sigma]\texttt{>}$	by (Sub Share)

All other proof cases are straightforward. □

Lemma 10 (Owner Substitutivity). *If* $\vdash \bar{c} : \mathsf{ok}$*,* $(\Gamma[\bar{x}\leftarrow\bar{v}] \vdash \bar{v} : \bullet)$*,* $\bar{x} \cap \mathsf{dom}(\Gamma_{\mathsf{val}}) = \emptyset$*, and* $(\Gamma, \bar{x} : \bullet \vdash \mathscr{J})$ *then* $(\Gamma[\bar{x}\leftarrow\bar{v}] \vdash \mathscr{J}[\bar{x}\leftarrow\bar{v}])$.

Proof. By induction on $(\Gamma, \bar{x} : \bullet \vdash \mathscr{J})$'s derivation. □

Lemma 11 (Access Right Substitutivity). *If* $ar \in \{\texttt{rdwr}, \texttt{rd}\}$ *and* $(\Gamma, \texttt{myaccess} : \bullet \vdash \mathscr{J})$*, then* $(\Gamma[\texttt{myaccess}{\leftarrow}ar] \vdash \mathscr{J}[\texttt{myaccess}{\leftarrow}ar])$.

Proof. By induction on $(\Gamma, \texttt{myaccess} : \bullet \vdash \mathscr{J})$'s derivation. The only typing rules that inspect the access rights in an interesting way are the subtyping rules and (Set) and (Call). The proof cases for (Set) and (Call) "go through", because the predicates "ar $\mathsf{wrsafe\ in}$ ar'" and "(ea, u, ar_u, v_u) $\mathsf{wrloc\ in}$ v" are designed to be closed under access right substitutions. □

Lemma 9 is meant to deal with method calls in the type preservation proof. However, Lemma 9 alone is not quite good enough for this purpose, because there is a mismatch between types of actual and formal method parameters: whereas formal parameters are assumed to have anonymous types, actual parameters are not. In fact, requiring types of actuals to be anonymous in order to match types of formals would not make sense, because anonymity is relative to the `this`-binding, which changes at method calls. To deal with this difficulty, we need another substitutivity lemma that permits such mismatches between types of substituting values and substituted variables and adjusts types of substitution bodies to make up for the mismatch. The following operation for *deanonymizing* types is useful:

$$\mathsf{deanon}(T : \mathsf{ExpTy}, b : \mathsf{Bool}) \;\triangleq\; \begin{cases} T - \texttt{anon} & \text{if } b \\ T & \text{if } \neg b \end{cases}$$

Lemma 12 (Anonymous Substitutivity). *Suppose* $\vdash \bar{c} : \mathsf{ok}$.
If $(\Gamma[x{\leftarrow}o] \vdash o : T[x{\leftarrow}o]\ \mathsf{in}\ p, ar')$ *and* $(\Gamma, x : \texttt{anon}\,T \vdash e : U\ \mathsf{in}\ p, ar)$*,*
then $(\Gamma[x{\leftarrow}o] \vdash e[x{\leftarrow}o] : \mathsf{deanon}(U[x{\leftarrow}o], o = p)\ \mathsf{in}\ p, ar)$.

Proof.

(1) $\vdash \bar{c} : \mathsf{ok}$	assumption
(2) $\sigma = (x{\leftarrow}o)$	abbreviation
(3) $\Gamma[\sigma] \vdash o : T[\sigma]\ \mathsf{in}\ p, ar'$	assumption
(4) $\Gamma, x : \texttt{anon}\,T \vdash e : U\ \mathsf{in}\ p, ar$	assumption

Case 1, $o \neq p$: We invert $(\Gamma[\sigma] \vdash o : T[\sigma]\ \mathsf{in}\ p, ar')$ and obtain a type T' such that:

(1.1) $T' = \texttt{anon}(\Gamma[\sigma])(o)$	
(1.2) $\Gamma[\sigma] \vdash o : T'\ \mathsf{in}\ p, ar'$	
(1.3) $\Gamma[\sigma] \vdash T' \preceq T[\sigma]$	
(1.4) $\Gamma[\sigma] \vdash T' = \texttt{anon}\,T' \preceq \texttt{anon}\,T[\sigma]$	by (1.3), Lemma 7
(1.5) $\Gamma[\sigma] \vdash o : \texttt{anon}\,T[\sigma]\ \mathsf{in}\ p, ar'$	by (1.2), (1.4), (Sub)
(1.6) $\Gamma[\sigma] \vdash e[\sigma] : U[\sigma]\ \mathsf{in}\ p, ar$	by (1.5), (4), Lemma 9

This is what we want, because $\mathsf{deanon}(U[\sigma], o = p) = U[\sigma]$

Case 2, $o = p$: From (4) we obtain $(\Gamma, x : T \vdash e : (U - \texttt{anon})\ \mathsf{in}\ p, ar)$, by Lemma 7. Then $(\Gamma[\sigma] \vdash e[\sigma] : (U[\sigma] - \texttt{anon})\ \mathsf{in}\ p, ar)$, by (3) and value substitutivity (Lemma 9). This is what we want, because $\mathsf{deanon}(U[\sigma], o = p) = U[\sigma] - \texttt{anon}$. □

Lemma 13 (Inversion Lemma for Contexts). *If* $(\Gamma \vdash \mathscr{E}[e] : T\ \mathsf{in}\ p, ar)$*, then there exists a type* U *such that* $(\Gamma \vdash e : U\ \mathsf{in}\ p, ar)$ *and* $(\Gamma, x : U \vdash \mathscr{E}[x] : T\ \mathsf{in}\ p, ar)$*. Moreover,* U *can be chosen such that* $(\Gamma \vdash e : U\ \mathsf{in}\ p, ar)$*'s type derivation does not end in (Sub).*

Proof. By induction on the structure of $\mathscr{E}$. □

D Well-typed States

The top-level judgment for well-typed states has the following form:

$$\bar{o} \vdash \mathit{state} : T \quad \begin{cases} \text{“}\bar{o}\text{ are immutable objects under construction,} \\ \text{and } \mathit{state} \text{ evaluates to a } T\text{-value”} \end{cases}$$

Well-typed States, $\bar{o} \vdash \mathit{state} : T$

(State)

$$\frac{\Gamma;\bar{o} \vdash h : \mathsf{ok} \quad \Gamma;\bar{o} \vdash s : T \quad \bar{o} \subseteq \mathsf{dom}(\Gamma_{\mathsf{val}}) \quad \Gamma_{\mathsf{val}} = \mathtt{rdonly}\ \mathtt{wrlocal}\,\mathsf{ty}|\mathsf{dom}(\Gamma_{\mathsf{val}})}{\bar{o} \vdash h :: s : T}$$

Well-typed Heaps, $\Gamma;\bar{o} \vdash h : \mathsf{ok}$**:**

(Hp Obj) $\Gamma(p) = ea\,C\texttt{<}ar,q\texttt{>} \quad \mathsf{ty}(p) = C\texttt{<}ar,q\texttt{>} \quad C\texttt{<}ar,q\texttt{>}$ generative

$$\frac{\mathsf{fd}(C) = \bar{ty}\,\bar{f} \quad \sigma = \mathsf{self}(p,ar,q) \quad \Gamma \vdash \bar{v} : \bar{ty}[\sigma] \text{ in } \mathtt{world} \quad \bar{v} \cap \bar{o} = \emptyset}{\Gamma;\bar{o} \vdash p\{\bar{f} = \bar{v}\} : \mathsf{ok}}$$

(Hp)

$$\frac{(\forall obj \in h)(\Gamma;\bar{o} \vdash obj : \mathsf{ok})}{\Gamma;\bar{o} \vdash h : \mathsf{ok}}$$

For the stack judgments, we sometimes need to record whether a frame has been created as the result of a method call or a `new`-call. For this reason, we introduce *frame kinds*:

$$\kappa \in \mathsf{FrmKind} \ ::= \ \mathsf{call} \mid \mathsf{new} \qquad \text{frame kinds}$$

We have two judgments for stacks:

$$\begin{array}{rl} \Gamma;\bar{o} \vdash s : T & \text{“}s\text{ evaluates to a }T\text{-value”} \\ \Gamma;\bar{o} \vdash (s : T) \leftarrow (\Gamma';\bar{o}' \vdash^{\kappa} T' \text{ in } p) & \begin{cases} \text{“}(\Gamma;\bar{o} \vdash s : T)\text{, and } s \text{ needs a top frame} \\ \text{of type } (\Gamma';\bar{o}' \vdash^{\kappa} T' \text{ in } p)\text{”} \end{cases} \end{array}$$

In order to get an intuition for the purpose of the two environments Γ and Γ' in the second stack judgment, we describe how we will manipulate them in the upcoming type preservation proof: The environment Γ is the global type environment, whose domain equals the domain of the current heap. Γ records the types that object identifiers were generated with (i.e., their dynamic types). Γ and Γ' always agree on their value component, i.e., $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}}$. However, they differ on their owner component: Whereas Γ_{own} is always $\{(\mathtt{world}, \bullet)\}$, Γ'_{own} records those object identifiers that may be used as owner parameters. Γ'_{own} differs for each stack frame. When a method is entered, Γ'_{own} for the new stack frame contains `world` plus the objects that instantiate `this`, `myowner` and the method's owner parameters. Γ'_{own} remains fixed during the lifetime of a stack frame. Keeping track of permitted owner parameters is important, so that the type preservation proof can exploit the restrictions that the system imposes on the owners of write-references.

Corresponding to the two stack judgments, there are two judgments for stack frames. Let a *call frame* be a frame of the form $(\mathscr{E}[e] \,\mathsf{in}\, o)$, where e is a method call, `new`-call or constructor call, i.e., e has the form $q.m\texttt{<}\bar{u}\texttt{>}(\bar{v})$ or $\mathtt{new}\,C\texttt{<}ar,u\texttt{>}.k(\bar{v})$ or $e = C.k(\bar{v})$.

$$\Gamma;\bar{o} \vdash^{\kappa} fr : T \text{ in } p \quad \text{“}fr\text{ is a frame”}$$

$$\Gamma;\bar{o} \vdash^{\kappa} (fr : T \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} T' \text{ in } p') \quad \begin{cases} \text{“}fr\text{ is a call frame that} \\ \text{needs a top of type } (\bar{o}' \vdash^{\kappa'} T' \text{ in } p')\text{”} \end{cases}$$

Well-typed Stacks, $\Gamma;\bar{o} \vdash^{\kappa} s : T$ and $\Gamma;\bar{o} \vdash^{\kappa} (s : T) \leftarrow (\Gamma';\bar{o}' \vdash^{\kappa'} T' \text{ in } p)$:

(Stk Push)

$$\frac{\Gamma;\bar{o} \vdash (s : T) \leftarrow (\Gamma';\bar{o}' \vdash^{\kappa} T' \text{ in } p) \qquad \Gamma';\bar{o}' \vdash^{\kappa} fr : T' \text{ in } p}{\Gamma;\bar{o}' \vdash s,fr : T}$$

(Stk Nil)

$$\frac{}{\Gamma;\bar{o} \vdash (\varepsilon : T) \leftarrow (\Gamma';\bar{o} \vdash^{\kappa} T \text{ in } p)}$$

(Stk Call) Γ'' closed $\quad \Gamma''_{\mathsf{val}} = \Gamma_{\mathsf{val}}$

$$\frac{\Gamma;\bar{o} \vdash (s : T) \leftarrow (\Gamma',\bar{o}' \vdash^{\kappa} T' \text{ in } p) \qquad \Gamma';\bar{o}' \vdash^{\kappa} (fr : T' \text{ in } p) \leftarrow (\bar{o}'' \vdash^{\kappa'} T'' \text{ in } p')}{\Gamma;\bar{o}' \vdash (s,fr : T) \leftarrow (\Gamma'';\bar{o}'' \vdash^{\kappa'} T'' \text{ in } p')}$$

Perhaps the most interesting rules are the ones for well-typed stack frames, because these formalize crucial invariants. To state these invariants, we need the following functions:

$$\mathsf{acc}(ea C\texttt{<}ar,o\texttt{>}) \triangleq ar \qquad \mathsf{acc}(p) \triangleq \mathsf{acc}(\mathsf{ty}(p)) \qquad \mathsf{acc}(\texttt{world}) \triangleq \texttt{rdwr}$$

$$\mathsf{owner}(ea C\texttt{<}ar,o\texttt{>}) \triangleq o \qquad \mathsf{owner}(p) \triangleq \mathsf{owner}(\mathsf{ty}(p))$$

$$\mathsf{imm} \triangleq \{o \mid \texttt{immutable} \in \mathsf{atts}(o)\} \qquad \mathsf{eatts}(T) \triangleq \mathsf{atts}(T) \cap \{\texttt{anon},\texttt{rdonly},\texttt{wrlocal}\}$$

Lemma 14 (Ownerless Objects Have `rdwr` Access).

(a) *If* $\mathsf{owner}(o) = \texttt{world}$*, then* $\mathsf{acc}(o) = \texttt{rdwr}$.
(b) *If* $(\Gamma \vdash C\texttt{<}\mathsf{acc}(o),\mathsf{owner}(o)\texttt{>} \preceq D\texttt{<}\texttt{rdwr},v\texttt{>})$*, then* $\mathsf{acc}(o) = \texttt{rdwr}$.

Proof. Here we need our requirement on ty, namely, that it maps to types of the form $C\texttt{<}ar,p\texttt{>}$ *such that* $p = \texttt{world}$ *implies* $ar = \texttt{rdwr}$. □

Lemma 15 (Immutable Objects are Ownerless). *If* $o \in \mathsf{imm}$*, then* $\mathsf{owner}(o) = \texttt{world}$.

Proof. Here we need the other requirement on ty, namely, that it maps to types of the form $C\texttt{<}ar,p\texttt{>}$ *such that* $\texttt{immutable} \in \mathsf{atts}(C)$ *implies* $p = \texttt{world}$. □

Inside, $\mathsf{inside}(\bar{o})$, Outside, $\mathsf{outside}(\bar{o})$, and Owner Scope, $\mathsf{scope}(o)$:

(Just Inside)

$$\frac{o \in \bar{o}}{o \in \mathsf{inside}(\bar{o})}$$

(Strictly Inside)

$$\frac{\mathsf{owner}(p) \in \mathsf{inside}(\bar{o})}{p \in \mathsf{inside}(\bar{o})}$$

$$\mathsf{outside}(\bar{o}) \triangleq \mathsf{ObjId} - \mathsf{inside}(\bar{o})$$

$$\mathsf{scope}(o) \triangleq \{o, \mathsf{owner}(o), \texttt{world}\}$$

Here are some obvious facts that follow from these definitions.

Lemma 16 (Inside and Outside).

(a) *If* $\mathsf{owner}(p) \in \mathsf{inside}(\bar{o})$*, then* $p \in \mathsf{inside}(\bar{o})$.
(b) *If* $p \in \mathsf{outside}(\bar{o})$*, then* $\mathsf{owner}(p) \in \mathsf{outside}(\bar{o})$.
(c) *If* $p \in \mathsf{inside}(\bar{o}) - \bar{o}$*, then* $\mathsf{owner}(p) \in \mathsf{inside}(\bar{o})$.

(d) *If* $\mathsf{owner}(p) \in \mathsf{outside}(\bar{o})$, *then* $p \in \mathsf{outside}(\bar{o}) \cup \bar{o}$.

Lemma 17 (The Inside of Mutables is Mutable). *If* $o \notin \mathsf{imm} \cup \{\texttt{world}\}$, *then* $\mathsf{inside}(o) \cap (\mathsf{imm} \cup \{\texttt{world}\}) = \emptyset$.

Proof. By induction on $\mathsf{inside}(o)$ using Lemma 15. □

We define $\mathsf{vals}(e)$ as the set of all values that occur within e *in value positions*. *Value positions* are positions outside angle brackets. (Positions inside angle brackets could be termed *access right positions*, respectively, *owner positions*, but we will not use these terms in this paper.) Technically, $\mathsf{vals}(e)$ is defined by induction on the structure of e as follows:

$$\mathsf{vals}(e_1,\dots,e_n) \triangleq \mathsf{vals}(e_1) \cup \dots \cup \mathsf{vals}(e_n) \quad \mathsf{vals}(v) \triangleq \{v\} \quad \mathsf{vals}(v.f) \triangleq \mathsf{vals}(v)$$
$$\mathsf{vals}(v.m\texttt{<}\bar{u}\texttt{>}(\bar{e})) \triangleq \mathsf{vals}(v,\bar{e}) \quad \mathsf{vals}(\texttt{new}\,C\texttt{<}ar,b\texttt{>}.k(\bar{e})) \triangleq \mathsf{vals}(\bar{e}) \quad \mathsf{vals}((C)e) \triangleq \mathsf{vals}(e)$$
$$\mathsf{vals}(v.f\texttt{=}e) \triangleq \mathsf{vals}(v,e) \quad \mathsf{vals}(\texttt{let}\,x\texttt{=}e\,\texttt{in}\,e') \triangleq \mathsf{vals}(e,e') \quad \mathsf{vals}(C.k(\bar{e})) \triangleq \mathsf{vals}(\bar{e})$$

Lemma 18 (Value Occurrences).
(a) *If* $(\Gamma \vdash e : T \text{ in } v, ar)$, *then* $\mathsf{vals}(e) \subseteq \mathsf{dom}(\Gamma_{\mathsf{val}}) \cup \{\texttt{null}\}$.
(b) $\mathsf{vals}(e[\sigma]) \subseteq \{\sigma(x) \mid x \in \mathsf{vals}(e)\} \cup \{v \mid v \in \mathsf{vals}(e) - \mathsf{dom}(\sigma)\}$.

Well-typed Frames, $\Gamma;\bar{o} \vdash^{\kappa} fr : T \text{ in } p$:

(Frm)
$$\frac{\Gamma \vdash^{\kappa} e : T \text{ in } p, \mathsf{acc}(p) \quad \mathsf{vals}(e) \cap \bar{o} \subseteq \{p\} \quad \Gamma;\bar{o} \vdash \mathsf{eatts}(T) \text{ ok in } p \quad \bar{o} \subseteq \mathsf{imm}}{\Gamma;\bar{o} \vdash^{\kappa} e \,\mathsf{in}\, p : T \text{ in } p}$$

(Frm Imm Cons) $\mathsf{dom}(\Gamma_{\mathsf{own}}) \subseteq \{p, \texttt{world}\}$
$$\frac{p \in \bar{o} \quad \mathsf{owner}(p) = \texttt{world} \quad \texttt{anon} \in ea \quad \{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea \neq \emptyset}{\Gamma;\bar{o} \vdash ea \text{ ok in } p}$$

(Frm Rdwr) $\mathsf{dom}(\Gamma_{\mathsf{own}}) \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$
$$\frac{p \notin \bar{o}, \mathsf{imm} \quad p \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o}) \quad \mathsf{acc}(p) = \texttt{rdwr} \text{ or } \texttt{wrlocal} \in ea \quad \texttt{anon} \notin ea}{\Gamma;\bar{o} \vdash ea \text{ ok in } p}$$

(Frm Rdonly)
$$\frac{p \notin \bar{o} \quad \texttt{rdonly} \in ea \quad \texttt{anon} \notin ea}{\Gamma;\bar{o} \vdash ea \text{ ok in } p}$$

(Mth Frm Body)
$$\frac{\Gamma \vdash e : T \text{ in } p, ar}{\Gamma \vdash^{\mathsf{call}} e : T \text{ in } p, ar}$$

(New Frm Body)
$$\frac{\Gamma \vdash e, p : ea\,\texttt{void}, (ea - \texttt{anon})\,ty \text{ in } p, ar}{\Gamma \vdash^{\mathsf{new}} e\,\texttt{;}\,p : (ea - \texttt{anon})\,ty \text{ in } p, ar}$$

(New Frm Done)
$$\frac{\Gamma \vdash p : (T - \texttt{anon}) \text{ in } p, ar}{\Gamma \vdash^{\mathsf{new}} p : T \text{ in } p, ar}$$

Well-typed Call Frames, $\Gamma;\bar{o} \vdash^{\kappa} (fr : T \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} T' \text{ in } p')$:

(Call Frm) $\Gamma;\bar{o} \vdash^{\kappa} (\mathscr{E}[e] \,\mathsf{in}\, p) : T \text{ in } p \qquad e : (\bar{o} \vdash p) \leftarrow (\bar{o}' \vdash^{\kappa'} p')$
$$\frac{\Gamma, x : ea\,\texttt{rdonly}\,\texttt{wrlocal}\,T' \vdash \mathscr{E}[x] : T \text{ in } p, \mathsf{acc}(p) \quad \Gamma \vdash T' : \mathsf{ok} \quad ea = \{\texttt{anon} \mid p \neq p'\}}{\Gamma;\bar{o} \vdash^{\kappa} (\mathscr{E}[e] \,\mathsf{in}\, p : T \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} T' \text{ in } p')}$$

(Call Frm Mth)
$$\frac{e = p'.m\texttt{<}\bar{v}\texttt{>}(\bar{u})}{e : (\bar{o} \vdash p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} p')}$$

(Call Frm Cons)
$$\frac{e = C.k(\bar{u})}{e : (\bar{o} \vdash p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} p)}$$

(Call Frm New) $\bar{o} \subseteq \bar{o}'$
$$\frac{e = \texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{u}) \quad p' \notin p, \bar{o}}{e : (\bar{o} \vdash p) \leftarrow (\bar{o}' \vdash^{\mathsf{new}} p')}$$

E Type Preservation

To state the type preservation theorem, we label state reductions with *write effects*. Here is the intuition:

$$\mathit{state} \xrightarrow{\bar{o}} \mathit{state}' \quad \text{“}\mathit{state}\text{ reduces to }\mathit{state}'\text{ writing to fields of }\bar{o}\text{”}$$

Instrumented State Reductions, $\mathit{state} \xrightarrow{\bar{o}} \mathit{state}'$**:**

(Red Set)

$$h, o\{f = u, \bar{g} = \bar{w}\} :: s, \mathscr{E}[o.f\texttt{=}v] \,\mathsf{in}\, p \xrightarrow{o} h, o\{f = v, \bar{g} = \bar{w}\} :: s, \mathscr{E}[v] \,\mathsf{in}\, p$$

All other reductions are labeled with the empty set.

The reflexive, transitive closure of instrumented state reductions is defined as the least relation $\mathit{state} \xrightarrow{\bar{o}}{}^{*} \mathit{state}'$ that satisfies the following two rules:

$$\mathit{state} \xrightarrow{\bar{o}}{}^{*} \mathit{state} \qquad \mathit{state} \xrightarrow{\bar{o}}{}^{*} \mathit{state}', \mathit{state}' \xrightarrow{\bar{p}} \mathit{state}'' \Rightarrow \mathit{state} \xrightarrow{\bar{o},\bar{p}}{}^{*} \mathit{state}''$$

Theorem 2 (Type Preservation).
If $(\bar{o} \vdash h :: s : T)$ *and* $h :: s \xrightarrow{\bar{p}} \mathit{state}$*, then* $\bar{p} \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$ *and there exists* $\bar{o}'$ *such that* $(\bar{o}' \vdash \mathit{state} : T)$ *and* $(\bar{o}' - \bar{o}) \cap \mathsf{dom}(h) = \emptyset$*.*

The type preservation proof is lengthy. We postpone it to Section H.

F Immutability in a Closed World

As a corollary of type preservation, we can show that `immutable` classes are immutable in a closed world. In Section G, we will lift this result to an open world.

Definition 5 (Jimuva Programs). *A* Jimuva program *is a pair* $P = (\bar{c}; e)$ *such that* $(\vdash \bar{c} : \mathsf{ok})$ *and* $(\texttt{world} : \bullet \vdash e : T \;\mathsf{in}\; \texttt{world})$ *for some* T*.*

Theorem 3 (Immutability in a Closed World). *Let* $P = (\bar{c}; e)$ *be a Jimuva program,* C *be declared in* $\bar{c}$ *and* `immutable` $\in \mathsf{atts}(C)$*. Then* C *is immutable in* P*.*

To prove this theorem we first need to establish some facts about the topology of well-typed heaps. Because our system allows sharing of read-only objects, we have to deal with the fact that object states may overlap. This differs from a strict owner-as-dominator discipline, where two object states only overlap if one of them is fully included in the other one.

Example 2 (Overlapping Object States). Consider the well-typed heap $h \stackrel{\Delta}{=} \{o\{f = r\}, o'\{f = r\}, r\{\}\}$, where:

$$o, o' \in \mathsf{imm} \qquad \mathsf{rawty}(o) = \mathsf{rawty}(o') = C \qquad \mathsf{fd}(C) = \{\texttt{Object<rd,this>}\, f\}$$
$$\mathsf{ty}(r) = \texttt{Object<rd,}o'\texttt{>}$$

Our definitions of *object states* and *insides* yield:

$$\mathsf{state}(h)(o) = \{o\{f = r\}, r\{\}\} \qquad \mathsf{inside}(o) = \{o\}$$
$$\mathsf{state}(h)(o') = \{o'\{f = r\}, r\{\}\} \qquad \mathsf{inside}(o') = \{o', r\}$$

Note that the object states of o and o' overlap: $\mathsf{state}(h)(o) \cap \mathsf{state}(h')(o') = \{r\{\}\}$. Note also that the domain of o's object state is not contained in o's inside: $r \in \mathsf{dom}(\mathsf{state}(h)(o)) - \mathsf{inside}(o)$.

Lemma 19 (States of Mutable Objects are Inside Themselves). *If* $(\Gamma; \vdash h : \mathsf{ok})$ *and* $o \notin \mathsf{imm} \cup \{\mathtt{world}\}$*, then* $\mathsf{dom}(\mathsf{state}(h)(o)) \subseteq \mathsf{inside}(o)$.

Proof. By induction on the definition of $\mathsf{state}(h)$.

(1) $\Gamma; \vdash h : \mathsf{ok}$	assumption
(2) $o \notin \mathsf{imm} \cup \{\mathtt{world}\}$	assumption
(3) $p\{..\} \in \mathsf{state}(h)(o)$	assumption
(4) $p \in \mathsf{inside}(o)$	goal

We distinguish cases by the three possible reasons for (3). The proof case where this reason is rule (a) from Definition 2 is trivial. The proof case for rule (b) uses that subtyping leaves mutable owners fixed, i.e., a subtype of $C\texttt{<}ar, o\texttt{>}$, where $o \notin \mathsf{imm} \cup \{\mathtt{world}\}$, is of the form $D\texttt{<}ar, o\texttt{>}$.

Let's do the proof case for rule (c) in more detail. In this case, the reason for (3) is the following rule:

$$\frac{q \neq o \quad q\{..f = r..\} \in \mathsf{state}(h)(o) \quad C\texttt{<}ar, \mathtt{myowner}\texttt{>}\, f \in \mathsf{fd}(\mathsf{rawty}(q)) \quad p\{..\} \in \mathsf{state}(h)(r)}{p\{..\} \in \mathsf{state}(h)(o)}$$

By induction hypothesis, we know that:

(5) $q \in \mathsf{inside}(o)$

Moreover, because $(\Gamma; \vdash h : \mathsf{ok})$, we have:

(6) $\Gamma \vdash C_r\texttt{<}\mathsf{acc}(r), \mathsf{owner}(r)\texttt{>} \preceq C\texttt{<}.., \mathsf{owner}(q)\texttt{>}$

Because $q \in \mathsf{inside}(o)$ and $q \neq o$, we have:

(7) $\mathsf{owner}(q) \in \mathsf{inside}(o)$

By Lemma 17, we get:

(8) $\mathsf{owner}(q) \notin \mathsf{imm} \cup \{\mathtt{world}\}$

We now inspect the subtyping rules for possible reasons for judgment (6). Because we know (8), we find that:

(9) $\mathsf{owner}(q) = \mathsf{owner}(r)$

By induction hypothesis, we have:

(10) $p \in \mathsf{inside}(r) \subseteq \mathsf{inside}(\mathsf{owner}(r)) = \mathsf{inside}(\mathsf{owner}(q))$

From (10) and (7), we obtain $p \in \mathsf{inside}(o)$, by transitivity. $\square$

Lemma 20 (Outside Objects are Inside Rd-Objects). *If* $(\Gamma; \vdash h : \mathsf{ok})$ *and* $p\{..\} \in \mathsf{state}(h)(o)$*, then one of the following statements holds:*

(a) $p \in \mathsf{inside}(o)$

(b) $(\exists r \in \mathsf{dom}(\mathsf{state}(h)(o)))(\mathsf{acc}(r) = \mathtt{rd} \wedge \mathsf{owner}(r) \in \mathsf{imm} \wedge p \in \mathsf{inside}(r))$

Proof. By induction on the definition of $\mathsf{state}(h)$.

(1) $\Gamma;\vdash h : \mathsf{ok}$ — assumption

(2) $p\{..\} \in \mathsf{state}(h)(o)$ — assumption

In case $o = \mathtt{world}$, we have $p \in \mathsf{inside}(\mathtt{world}) = \mathsf{inside}(o)$, and we are done. So let's assume:

(3) $o \neq \mathtt{world}$ — assumption

We distinguish cases by the three possible reasons for assumption (2).

Case 1, the reason for (2) is rule (a): In this case $p = o \in \mathsf{inside}(o)$.

Case 2, the reason for (2) is rule (b):

$$\frac{o\{..f = q..\} \in h \quad C\texttt{<}ar,\mathtt{this}\texttt{>}\, f \in \mathsf{fd}(\mathsf{rawty}(o)) \quad p\{..\} \in \mathsf{state}(h)(q)}{p\{..\} \in \mathsf{state}(h)(o)}$$

From ($\Gamma;\vdash h : \mathsf{ok}$) we obtain:

(2.1) $\Gamma \vdash C_q\texttt{<}\mathsf{acc}(q),\mathsf{owner}(q)\texttt{>} \preceq C\texttt{<}..,o\texttt{>}$

On the other hand, by induction hypothesis we know that $p \in \mathsf{inside}(q)$ or $(\exists r)(\mathsf{acc}(r) = \mathtt{rd} \wedge \mathsf{owner}(r) \in \mathsf{imm} \wedge p \in \mathsf{inside}(r))$. In the latter case, we are done. So, let's suppose:

(2.2) $p \in \mathsf{inside}(q)$

We now distinguish cases by the possible reasons for subtyping judgment (2.1).

Case 2.1, (Sub Rep): In this case, we have $\mathsf{owner}(q) = o$. From this, we obtain $p \in \mathsf{inside}(q) \subseteq \mathsf{inside}(\mathsf{owner}(q)) = \mathsf{inside}(o)$.

Case 2.2, (Sub World): In this case, we have $o = \mathtt{world}$, in contradiction to assumption (3).

Case 2.3, (Sub Share): In this case, we have $\mathsf{acc}(q) = \mathtt{rd}$ and $\mathsf{owner}(q) \in \mathsf{imm}$. By (2.2), we also have $p \in \mathsf{inside}(q)$. Moreover, we have $q\{..\} \in \mathsf{state}(h)(q)$, by rule (a) from Definition 2, and then $q\{..\} \in \mathsf{state}(h)(o)$, by rule (b). Thus, condition (b) holds and we are done.

Case 3, the reason for (2) is rule (c):

$$\frac{q \neq o \quad q\{..f = q'..\} \in \mathsf{state}(h)(o) \quad C\texttt{<}ar,\mathtt{myowner}\texttt{>}\, f \in \mathsf{fd}(\mathsf{rawty}(q)) \quad p\{..\} \in \mathsf{state}(h)(q')}{p\{..\} \in \mathsf{state}(h)(o)}$$

Because we assume that the underlying class table is legal, we know that $ar = \mathtt{myaccess}$. From ($\Gamma;\vdash h : \mathsf{ok}$) we obtain:

(3.1) $\Gamma \vdash C_{q'}\texttt{<}\mathsf{acc}(q'),\mathsf{owner}(q')\texttt{>} \preceq C\texttt{<}\mathsf{acc}(q),\mathsf{owner}(q)\texttt{>}$

We apply the induction hypothesis to the second rule premise and obtain:

(3.2) $q \in \mathsf{inside}(o)$ or $(\exists r \in \mathsf{dom}(\mathsf{state}(h)(o)))(\mathsf{acc}(r) = \mathtt{rd} \wedge \mathsf{owner}(r) \in \mathsf{imm} \wedge q \in \mathsf{inside}(r))$

Because $q \neq o$, statement (3.2) can only hold if $\mathsf{owner}(q) \neq \mathtt{world}$. Then $q \notin \mathsf{imm}$, because immutable objects are owned by $\mathtt{world}$. Applying Lemma 19, we obtain:

(3.3) $\mathsf{dom}(\mathsf{state}(h)(q')) \subseteq \mathsf{inside}(q')$

By the last rule premise, we know that $p \in \mathsf{dom}(\mathsf{state}(h)(q'))$. So we have:

(3.4) $p \in \mathsf{dom}(\mathsf{state}(h)(q')) \subseteq \mathsf{inside}(q')$

Suppose that the reason for subtyping judgment (3.1) is (Sub Share). Then we have $\mathsf{acc}(q') = \mathtt{rd}$ and $\mathsf{owner}(q') \in \mathsf{imm}$. Furthermore, we have $p \in \mathsf{inside}(q')$, by (3.4), and $q' \in \mathsf{dom}(\mathsf{state}(h)(o))$, by rules (a) and (c) from Definition 2. Thus, we have established condition (b). The reason for subtyping judgment (3.1) cannot be (Sub World), because $\mathsf{owner}(p) \neq \mathtt{world}$ as we have already established. The remaining case is where the reason for subtyping judgment (3.1) is (Sub Rep). In this case, we have:

(3.5) $\mathsf{acc}(q') = \mathsf{acc}(q)$ and $\mathsf{owner}(q') = \mathsf{owner}(q)$

We get:

(3.6) $p \in \mathsf{inside}(q') \subseteq \mathsf{inside}(\mathsf{owner}(q')) = \mathsf{inside}(\mathsf{owner}(q))$

Now, we distinguish cases by the two possible disjuncts in statement (3.2).

Case 3.1, $q \in \mathsf{inside}(o)$: Because $q \neq o$, we know $\mathsf{owner}(q) \in \mathsf{inside}(o)$. From $p \in \mathsf{inside}(\mathsf{owner}(q))$ and $\mathsf{owner}(q) \in \mathsf{inside}(o)$, we obtain $p \in \mathsf{inside}(o)$, by transitivity.

Case 3.2, $r \in \mathsf{dom}(\mathsf{state}(h)(o)) \wedge \mathsf{acc}(r) = \mathtt{rd} \wedge \mathsf{owner}(r) \in \mathsf{imm} \wedge q \in \mathsf{inside}(r)$:

Case 3.2.1, $\mathsf{owner}(q) \in \mathsf{inside}(r)$: From $p \in \mathsf{inside}(\mathsf{owner}(q))$ and $\mathsf{owner}(q) \in \mathsf{inside}(r)$, we obtain $p \in \mathsf{inside}(r)$, by transitivity.

Case 3.2.2, $q = r$: In this case, we have $\mathsf{acc}(q') = \mathsf{acc}(q) = \mathsf{acc}(r) = \mathtt{rd}$ and $\mathsf{owner}(q') = \mathsf{owner}(q) = \mathsf{owner}(r) \in \mathsf{imm}$ and $p \in \mathsf{inside}(q')$. Moreover, we obtain $q' \in \mathsf{dom}(\mathsf{state}(h)(o))$, by rules (a) and (c) from Definition 2. Thus, we have established condition (b).

□

Lemma 21 (States of Immutabe Objects are Inside the Immutable World). *If* $(\Gamma; \vdash h : \mathsf{ok})$ *and* $q\{..\} \in \mathsf{state}(h)(\mathsf{imm})$*, then* $q \in \mathsf{inside}(\mathsf{imm})$.

Proof. This is a direct corollary of Lemma 20. □

Lemma 22 (Object State Invariance Under New Allocation).

(a) *If* $h \subseteq h'$*, then* $\mathsf{state}(h)(o) \subseteq \mathsf{state}(h')(o)$.
(b) *If* $h = (h', obj)$ *and* $obj \notin \mathsf{state}(h)(o)$*, then* $\mathsf{state}(h)(o) = \mathsf{state}(h')(o)$.
(c) *If* $p\{\bar{f} = \bar{v}\} \in \mathsf{state}(h)(o)$*, then* $p = o$ *or* $h = (h', q\{..g = p..\})$ *for some* h', g, q.
(d) *If* $\Gamma; \vdash h : \mathsf{ok}$*,* $o \in \mathsf{dom}(h)$ *and* $p \notin \mathsf{dom}(h)$*, then* $\mathsf{state}(h, p\{\bar{f} = \mathtt{null}\})(o) = \mathsf{state}(h)(o)$.

Proof. Part (a) is shown by induction on the definition of $\mathsf{state}(h)$. One inclusion of part (b) is a consequence of part (a), the other inclusion is shown by induction on the definition of $\mathsf{state}(h)$. Part (c) is shown by induction on the definition of $\mathsf{state}(h)$. For part (d), note that $p \notin \mathsf{dom}(h)$ and and $h = (h', q\{..g = p..\})$ is impossible in a well-typed heap. Therefore $p\{\bar{f} = \mathtt{null}\} \notin \mathsf{state}(h)(o)$, by part (c). Then $\mathsf{state}(h, p\{\bar{f} = \mathtt{null}\})(o) = \mathsf{state}(h)(o)$ is a consequence of part (b). □

Lemma 23 (No Writes to Fully Constructed Immutable Objects). *If* $(\bar{o} \vdash h_1 :: s_1 : T)$, $o \in (\mathsf{imm} - \bar{o}) \cap \mathsf{dom}(h_1)$ *and* $h_1 :: s_1 \;\rightarrow\; h_2 :: s_2$, *then* $\mathsf{state}(h_1)(o) = \mathsf{state}(h_2)(o)$.

Proof. Suppose $(\bar{o} \vdash h_1 :: s_1 : T)$, $o \in (\mathsf{imm} - \bar{o}) \cap \mathsf{dom}(h_1)$ and $h_1 :: s_1 \;\rightarrow\; h_2 :: s_2$. If the reason for $h_1 :: s_1 \;\rightarrow\; h_2 :: s_2$ is (Red New), then $\mathsf{state}(h_1)(p) = \mathsf{state}(h_2)(p)$ holds by Lemma 22(d). If the reason for $h_1 :: s_1 \;\rightarrow\; h_2 :: s_2$ is any other rule, except (Set), then $h_1 = h_2$ and $\mathsf{state}(h_1)(o) = \mathsf{state}(h_2)(o)$ trivially holds. So the only interesting case is the following:

Case 1, (Red Set):

$$\frac{}{h, q\{f = u, \bar{g} = \bar{w}\} :: s, \mathscr{E}[q.f\texttt{=}v]\,\mathsf{in}\,p \xrightarrow{q} h, q\{f = v, \bar{g} = \bar{w}\} :: s, \mathscr{E}[v]\,\mathsf{in}\,p}$$

(1.1) $h_1 = h, q\{f = u, \bar{g} = \bar{w}\}$
(1.2) $s_1 = s, \mathscr{E}[q.f\texttt{=}v]\,\mathsf{in}\,p$
(1.3) $h_2 = h, q\{f = v, \bar{g} = \bar{w}\}$
(1.4) $s_2 = s, \mathscr{E}[v]\,\mathsf{in}\,p$

If we can show $q\{f = u, \bar{g} = \bar{w}\} \notin \mathsf{state}(h_1)(o)$ and $q\{f = v, \bar{g} = \bar{w}\} \notin \mathsf{state}(h_2)(o)$ then we can apply Lemma 22 to get $\mathsf{state}(h_1)(o) = \mathsf{state}(h)(o) = \mathsf{state}(h_2)(o)$.

(1.5) $q\{f = u, \bar{g} = \bar{w}\} \notin \mathsf{state}(h_1)(o)$ goal
(1.6) $q\{f = v, \bar{g} = \bar{w}\} \notin \mathsf{state}(h_2)(o)$ goal

It turns out that the proof of goal (1.5) does not depend on the identity of u. So the proof of goal (1.6) is essentially identical to the proof of goal (1.5) and we only prove goal (1.5):

(1.7) $q\{f = u, \bar{g} = \bar{w}\} \in \mathsf{state}(h_1)(o)$ assumption towards contradiction

By type preservation (Theorem 2), we know:

(1.8) $q \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$

By Lemma 21, we have:

(1.9) $q \in \mathsf{inside}(\mathsf{imm})$

From (1.8) and (1.9), we get:

(1.10) $q \in \mathsf{inside}(o')$ for some $o' \in \bar{o}$

By assumption, we have $o \notin \bar{o}$. Therefore:

(1.11) $o \neq o'$

Because $o, o' \in \mathsf{imm}$, we know that neither $o \in \mathsf{inside}(o')$ nor $o' \in \mathsf{inside}(o)$. It follows that $\mathsf{inside}(o) \cap \mathsf{inside}(o') = \emptyset$. Then $q \notin \mathsf{inside}(o)$, because we know that $q \in \mathsf{inside}(o')$. Therefore we have:

(1.12) $q \in \mathsf{dom}(\mathsf{state}(h)(o)) - \mathsf{inside}(o)$

By Lemma 20, there exists an r such that:

(1.13) $r \in \mathsf{dom}(\mathsf{state}(h)(o)) \wedge \mathsf{acc}(r) = \texttt{rd} \wedge \mathsf{owner}(r) \in \mathsf{imm} \wedge q \in \mathsf{inside}(r)$

Because $h_1 :: s_1$ writes to r's inside and $\mathsf{acc}(r) = \mathtt{rd}$, we know that r must still be under construction.[7] On the other hand, we know that o's constructor has terminated, because $\bar{o}$ records the immutable objects under construction and $o \notin \bar{o}$. Because constructor calls are properly nested, we know that r did not exist yet when o's constructor terminated. Therefore, $r \in \mathsf{dom}(\mathsf{state}(h_1)(o))$ is impossible, because o's state did not change after termination of its constructor, by o's immutability. But by (1.13), we also have $r \in \mathsf{dom}(\mathsf{state}(h_1)(o))$. A contradiction.

$\square$

Lemma 24 (No Writes to Fully Constructed Immutable Objects, Transitive Version). *If* $(\bar{o}_1 \vdash h_1 :: s_1 : T)$, $o \in (\mathsf{imm} - \bar{o}_1) \cap \mathsf{dom}(h_1)$ *and* $h_1 :: s_1 \to^* h_2 :: s_2$, *then* $\mathsf{state}(h_1)(p) = \mathsf{state}(h_2)(p)$, $(\bar{o}_2 \vdash h_2 :: s_2 : T)$ *for some* $\bar{o}_2$ *such that* $o \notin \bar{o}_2$.

Proof. Follows from Lemma 23 and type preservation (Theorem 2), by induction on the length of the state reduction sequence. $\square$

Proof of Theorem 3. *Let* $P = (\bar{c}; e)$ *be a Jimuva program,* C *be declared in* $\bar{c}$ *and* $\mathtt{immutable} \in \mathsf{atts}(C)$. *Then* C *is immutable in* P.

Proof. Let $P = (\bar{c}; e_0)$ be a Jimuva program, C declared in $\bar{c}$ and $\mathtt{immutable} \in \mathsf{atts}(C)$. Then $(\vdash \bar{c} : \mathsf{ok})$ and $(\mathtt{world} : \bullet \vdash e : T \ \mathsf{in}\ \mathtt{world})$ for some T. Then $(\mathtt{world} : \bullet \vdash \emptyset :: e_0 \,\mathsf{in}\, \mathtt{world} : T)$, by (State).

(1) $\emptyset :: e_0 \,\mathsf{in}\, \mathtt{world} \to^* h_1 :: s_1 \to^* h_2 :: s_2$	assumption
(2) $h_1 :: s_1$ and $h_2 :: s_2$ are visible states for o	assumption
(3) $\mathsf{rawty}(o) <: C$	assumption

By type preservation, there exists $\bar{o}_1$ such that:

(4) $\bar{o}_1 \vdash h_1 :: s_1 : T$

Because $\mathsf{rawty}(o) <: C$, we have $o \in \mathsf{imm}$. Because $\mathsf{state}(h_1)(s_1)$ is a visible state for o, we have $o \notin \bar{o}$. Therefore, $\mathsf{state}(h_1)(o) = \mathsf{state}(h_2)(o)$, by Lemma 24. $\square$

G Immutability in an Open World

G.1 Core Java

Core Java is obtained from Core Jimuva by erasing all non-Java-annotations, namely, owner and access right parameters, as well as expression and class attributes:

Core Java — a Java-like Core Language:

c,d ::= $\mathit{fm}\,\mathtt{class}\,C\,\mathtt{ext}\,D\{\bar{F}\ \bar{K}\ \bar{M}\}$	class declaration (where $C \neq \mathtt{Object}$)
F ::= $C f$	field
K ::= $C.k(\bar{ty}\,\bar{x})\{e\}$	constructor (scope of $\bar{x}$ is e)
M ::= $\mathit{fm}\,\mathit{ty}\,m(\bar{ty}\,\bar{x})\{e\}$	method (scope of $\bar{x}$ is e)
$\mathit{ty} \in \mathsf{ValTy}$ ::= $C \mid \mathtt{void}$	value types
$u,v,w \in \mathsf{Val}$ ::= $\mathtt{null} \mid o \mid x$	values
$e \in \mathsf{Exp}$ ::=	expressions and statements
$\ldots \mid v.m(\bar{e}) \mid \mathtt{new}\,C.k(\bar{e}) \mid \ldots$ *other forms like in Jimuva*	

[7] Unfortunately, from this point on the proof gets a bit hand-wavy. A cleaner proof would keep track of additional information in the typing judgments for well-typed states.

The *operational semantics* of Core Java is almost identical to Core Jimuva's operational semantics, except that it does not track owners and access rights:

State Reductions, $state \to_{\mathsf{java},\bar{c}} state'$**:**

(Red Call) $\quad s = s', \mathscr{E}[o.m(\bar{v})] \,\mathsf{in}\, p \quad \mathsf{rawty}(o) = C \quad \mathsf{mbody}(C,m) = (\bar{x})(e)$
$\quad h :: s \to h :: s, e[\mathtt{this},\bar{x} \leftarrow o,\bar{v}] \,\mathsf{in}\, o$

(Red New) $\quad s = s', \mathscr{E}[\mathtt{new}\, C.k(\bar{v})] \,\mathsf{in}\, p \quad o \notin \mathsf{dom}(h) \quad \mathsf{rawty}(o) = C \quad \mathsf{fd}(C) = \bar{ty}\, \bar{f}$
$\quad h :: s \to h, o\{\bar{f} = \mathtt{null}\} :: s, C.k(\bar{v}) \,\mathtt{;}\, o \,\mathsf{in}\, o$

(Red Cons) $\quad s = s', \mathscr{E}[C.k(\bar{v})] \,\mathsf{in}\, p \quad \mathsf{cbody}(C.k) = (\bar{x})(e) \quad \mathsf{rawty}(p) = D$
$\quad h :: s \to h :: s, e[\mathtt{this},\bar{x} \leftarrow o,\bar{v}] \,\mathsf{in}\, p$

All other reductions are like in Jimuva.

The type erasure mapping $|\cdot|$ from Core Jimuva to Core Java is Core Jimuva to Core Java erases ownership information, access rights and expression and class attributes from Jimuva-programs. It is defined by induction on the syntax in the obvious way. We state the following lemma without proof.

Lemma 25 (Type Erasure Preserves and Reflects Reductions). *Suppose $\bar{c}$ is a Jimuva class table and state a Jimuva state such that* $(\vdash \bar{c} : \mathsf{ok})$ *and* $(\bar{o} \vdash state : \mathsf{ok})$*. Then* $(state \to_{\bar{c}} state')$ *iff* $(|state| \to_{\mathsf{java},|\bar{c}|} |state'|)$.

The "preserves" part of this lemma holds, because in well-typed Jimuva class tables the variables `myaccess` and `myowner` and the owner parameters for methods never occur outside angle brackets within method and constructor bodies. Therefore, Jimuva's operational semantics never propagates the actual access rights and owners that initialize these parameters into the "Java-part" of the language. The "reflects" part of Lemma 25 holds, because well-typed Jimuva programs do not get stuck.

Next we present Core Java's type system. Here, Γ ranges over functions from $\mathsf{Var} \cup \mathsf{ObjId} - \{\mathtt{world}\}$ to ValTy. The `this`-binding in typing judgment $(\Gamma \vdash_{\mathsf{java}} e : ty \,\mathsf{in}\, v)$ could, alternatively, be replaced by the class C of `this`: $(\Gamma \vdash_{\mathsf{java}} e : ty \,\mathsf{in}\, C)$. The class of `this` is needed, because we support protected fields, unlike some other Java-like language, e.g. FJ, where fields are public. We choose to keep track of the `this`-binding instead of only its class, because this relates more tightly to the Jimuva-type system.

$$\Gamma \vdash v : \mathsf{ok} \stackrel{\Delta}{=} (v \in \mathsf{dom}(\Gamma) \text{ or } v = \mathtt{world})$$

Good Class Declarations, $\vdash_{\mathsf{java}} c : \mathsf{ok}$**:**

(Cls Dcl) $\quad D$ is not `final`
$$\frac{\mathtt{this} : C \vdash \bar{F}, \bar{K}, \bar{M} : \mathsf{ok} \,\mathsf{in}\, C}{\vdash fm \,\mathtt{class}\, C \,\mathtt{ext}\, D \{\bar{F}\ \bar{K}\ \bar{M}\}}$$

(Fld Dcl)
$$\frac{C \,\mathtt{ext}\, D \Rightarrow f \notin \mathsf{fd}(D)}{\Gamma \vdash E\, f : \mathsf{ok} \,\mathsf{in}\, C}$$

(Cons Dcl)
$$\frac{\Gamma, \bar{x} : \bar{ty} \vdash e : \mathtt{void} \,\mathsf{in}\, \mathtt{this}}{\Gamma \vdash C.k(\bar{ty}\bar{x})\{e\} : \mathsf{ok} \,\mathsf{in}\, C}$$

(Mth Dcl) $\quad \Gamma, \bar{x} : \bar{ty} \vdash e : ty' \,\mathsf{in}\, \mathtt{this}$
$$\frac{C \,\mathtt{ext}\, D \Rightarrow \Gamma \vdash \mathsf{mtype}(m,C) \preceq \mathsf{mtype}(m,D)}{\Gamma \vdash fm\, ty'\, m(\bar{ty}\bar{x})\{e\} : \mathsf{ok} \,\mathsf{in}\, C}$$

Well-typed Expressions, $\mathit{\Gamma} \vdash_{\mathsf{java}} e : T \text{ in } v$:

(Var)
$$\frac{\mathit{\Gamma}(x) = C \quad v \in \mathsf{dom}(\mathit{\Gamma})}{\mathit{\Gamma} \vdash x : C \text{ in } v}$$

(Obj)
$$\frac{v \in \mathsf{dom}(\mathit{\Gamma})}{\mathit{\Gamma} \vdash o : \mathit{\Gamma}(o) \text{ in } v}$$

(Sub)
$$\frac{\mathit{\Gamma} \vdash e : ty \text{ in } v \quad \mathit{\Gamma} \vdash ty <: ty'}{\mathit{\Gamma} \vdash e : ty' \text{ in } v}$$

(Null)
$$\frac{\mathit{\Gamma} \vdash v : \mathsf{ok}}{\mathit{\Gamma} \vdash \mathtt{null} : ty \text{ in } v}$$

(Let)
$$\frac{\mathit{\Gamma} \vdash e : ty \text{ in } v \quad \mathit{\Gamma}, x : C \vdash e' : ty' \text{ in } v}{\mathit{\Gamma} \vdash \mathtt{let}\, x \mathtt{=} e\, \mathtt{in}\, e' : ty' \text{ in } v}$$

(Cast) C declared
$$\frac{\mathit{\Gamma} \vdash e : D \text{ in } v}{\mathit{\Gamma} \vdash (C)e : C \text{ in } v}$$

(Get)
$$\frac{C\, f \in \mathsf{fd}(D) \quad \mathit{\Gamma} \vdash u, v : D, D \text{ in } v}{\mathit{\Gamma} \vdash u.f : C \text{ in } v}$$

(Call)
$$\frac{\mathsf{mtype}(m, D) = \bar{ty} \to ty \quad \mathit{\Gamma} \vdash u, \bar{e} : D, \bar{ty} \text{ in } v}{\mathit{\Gamma} \vdash u.m(\bar{e}) : ty \text{ in } v}$$

(Set)
$$\frac{C\, f \in \mathsf{fd}(D) \quad \mathit{\Gamma} \vdash u, v, e : D, D, C \text{ in } v}{\mathit{\Gamma} \vdash u.f\mathtt{=}e : C \text{ in } v}$$

(New)
$$\frac{\mathsf{ctype}(C.k) = \bar{ty} \to \mathtt{void} \quad \mathit{\Gamma} \vdash \bar{e} : \bar{ty} \text{ in } v}{\mathit{\Gamma} \vdash \mathtt{new}\, C.k(\bar{e}) : C \text{ in } v}$$

(Cons)
$$\frac{\mathsf{ctype}(C.k) = \bar{ty} \to \mathtt{void} \quad \mathit{\Gamma} \vdash \bar{e}, v : \bar{ty}, C \text{ in } v}{\mathit{\Gamma} \vdash C.k(\bar{e}) : \mathtt{void} \text{ in } v}$$

Without proof we state that type erasure maps well-typed Jimuva class tables to well-typed Java class tables. This holds, because the Java typing rules have been obtained from Jimuva's rules by stripping off all Jimuva-related restrictions.

Lemma 26 (Type Erasure Preserves Typings). *If $\bar{c}$ is a Jimuva class table and* $(\vdash \bar{c} : \mathsf{ok})$*, then* $(\vdash_{\mathsf{java}} |\bar{c}| : \mathsf{ok})$.

G.2 Embedding Java into Jimuva

We sketch an embedding e that takes a Jimuva class table $\bar{c}$ and a Java class table $\bar{d}$ (which refers to $|\bar{c}|$) and returns a Jimuva class table $\mathsf{e}_{\bar{c}}(\bar{d})$ such that $|\mathsf{e}_{\bar{c}}(\bar{d})| = \bar{d}$. This embedding inserts into $\bar{d}$ the annotations `rdwr` and `world` wherever access or ownership parameters are required. Our Jimuva type system is designed so that $(\bar{c}, \mathsf{e}_{\bar{c}}(\bar{d}); \mathsf{e}_{\bar{c}}(e))$ is a well-typed Jimuva-program, provided $(|\bar{c}|, \bar{d}; e)$ is a Java-program and no class or method from $\bar{d}$ extends or overrides a Jimuva-annotated class or method from $\bar{d}$.

We omit the full definition of the embedding. Here are a few selected clauses:

For class declarations:

$$\mathsf{e}_{\bar{c}}(\mathit{fm}\,\mathtt{class}\,C\,\mathtt{ext}\,D\{\bar{F}\ \bar{K}\ \bar{M}\}) \stackrel{\Delta}{=} \mathit{fm}\,\mathtt{class}\,C\,\mathtt{ext}\,D\{\mathsf{e}_{\bar{c}}(\bar{F})\ \mathsf{e}_{\bar{c}}(\bar{K})\ \mathsf{e}_{\bar{c}}(\bar{M})\}$$

For method declarations:

$$\mathsf{e}_{\bar{c}}(\mathit{fm}\,ty\,m(\bar{ty}\,\bar{x})\{e\}) \stackrel{\Delta}{=} \mathit{fm}\,\mathtt{<>}\,\mathsf{e}_{\bar{c}}(ty)\,m(\mathsf{e}_{\bar{c}}(\bar{ty})\,\bar{x})\{\mathsf{e}_{\bar{c},\bar{x}:\bar{ty}}(e)\}$$

For types:

$$\mathsf{e}_{\bar{c}}(C) \stackrel{\Delta}{=} C\mathtt{<rdwr,world>}$$

For expressions:

$$\mathsf{e}_{\bar{c},\mathit{\Gamma}}(x.m(\bar{e})) \stackrel{\Delta}{=} x.m\mathtt{<world,}\ldots\mathtt{,world>}(\mathsf{e}_{\bar{c},\mathit{\Gamma}}(\bar{e})) \qquad \text{if } \mathit{\Gamma}(x) \in \bar{c}$$

where $\mathsf{mtype}(C, m) = \mathit{fm}\,\mathtt{<}\bar{y}\mathtt{>}\bar{ty} \to T$ and $|\bar{y}| = |\mathtt{world,}\ldots\mathtt{,world}|$

$$\mathsf{e}_{\bar{c},\mathit{\Gamma}}(x.m(\bar{e})) \stackrel{\Delta}{=} x.m\mathtt{<>}(\mathsf{e}_{\bar{c},\mathit{\Gamma}}(\bar{e})) \qquad \text{if } \mathit{\Gamma}(x) \notin \bar{c}$$

$$\mathsf{e}_{\bar{c},\mathit{\Gamma}}(\mathtt{new}\,C.k(\bar{e})) \stackrel{\Delta}{=} \mathtt{new}\,C\mathtt{<rdwr,world>}.k(\mathsf{e}_{\bar{c},\mathit{\Gamma}}(\bar{e}))$$

We state the following lemma without proof:

Lemma 27 (Java-Programs are Jimuva-Well-Typed). *Suppose $\bar{c}$ is Jimuva class table and ($\vdash \bar{c} : \mathsf{ok}$). If ($|\bar{c}|, \bar{d}; e$) is a Java-program and $\bar{d}$ validly subclasses $\bar{c}$, then ($\bar{c}, \mathsf{e}_{\bar{c}}(\bar{d}); \mathsf{e}_{\bar{c}}(e)$) is a Jimuva-program.*

It is obvious that Lemma 27 needs the restriction that $\bar{d}$ validly subclasses $\bar{c}$. For instance, the embedding maps a (possibly mutable) Java-class in $\bar{d}$ that extends an `immutable` class from $\bar{c}$ to a class without `immutable`-annotation, in violation to Jimuva's rule that subclasses of `immutable` classes must be `immutable`. For Lemma 27 to hold, it is important that Jimuva's typing rules do not restrict Java-clients of Jimuva-classes. This is the technical reason, why we need the typing rule (Sub World), which permits to ignore access right restrictions in context `world`, and the requirement that parameter and return types of methods do not mention `this`.

G.3 Immutability in an Open World

Proof of Theorem 1 (Soundness). *If ($\vdash \bar{c} : \mathsf{ok}$), then $\bar{c}$ is correct for immutability.*

Proof. Suppose ($\vdash \bar{c} : \mathsf{ok}$) and C is declared in $\bar{c}$. Suppose Java class table $\bar{d}$ validly subclasses $\bar{c}$ and ($|\bar{c}|, \bar{d}; e$) is a Java-program. Then ($\bar{c}, \mathsf{e}_{\bar{c}}(\bar{d}); \mathsf{e}_{\bar{c}}(e)$) is a Jimuva-program, by Lemma 27. Then C is immutable in ($\bar{c}, \mathsf{e}_{\bar{c}}(\bar{d}); \mathsf{e}_{\bar{c}}(e)$), by Theorem 3. □

H The Type Preservation Proof

Lemma 28 (Construction Set Grows with Stack).

(a) *If $\Gamma; \bar{o} \vdash^{\kappa} (e : T \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} T' \text{ in } p')$, then $\bar{o} \subseteq \bar{o}'$.*
(b) *If $\Gamma; \bar{o} \vdash^{\kappa} (s : T) \leftarrow (\Gamma'; \bar{o}' \vdash^{\kappa'} T' \text{ in } p)$, then $\bar{o} \subseteq \bar{o}'$.*

Proof. By inspection of the last rule. □

Lemma 29 (Weakening Construction Set for Heap). *If ($\Gamma; \bar{o} \vdash h : \mathsf{ok}$) and $\bar{o}' \subseteq \bar{o}$, then ($\Gamma; \bar{o}' \vdash h : \mathsf{ok}$).*

Lemma 30 (Heap Extension). *Let $o \notin \mathsf{dom}(\Gamma)$ and $T = \texttt{rdonly wrlocal}\,\mathsf{ty}(o)$. Then the following statements hold:*

(a) *If ($\Gamma; \bar{o} \vdash h : \mathsf{ok}$) and $\mathsf{fd}(\mathsf{rawty}(o)) = \bar{ty}\,\bar{f}$,*
then ($\Gamma, o : T; \bar{o}, o \vdash h, o\{\bar{f} = \texttt{null}\} : \mathsf{ok}$).
(b) *If ($\Gamma; \bar{o} \vdash^{\kappa} fr : U \text{ in } p$), then ($\Gamma, o : T; \bar{o} \vdash^{\kappa} fr : U \text{ in } p$).*
(c) *If $\Gamma; \bar{o} \vdash^{\kappa} (fr : U \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} U' \text{ in } p')$,*
then $\Gamma, o : T; \bar{o} \vdash^{\kappa} (fr : U \text{ in } p) \leftarrow (\bar{o}' \vdash^{\kappa'} U' \text{ in } p')$.
(d) *If $\Gamma; \bar{o} \vdash (s : U) \leftarrow (\Gamma'; \bar{o}' \vdash^{\kappa} U' \text{ in } p)$,*
then $\Gamma, o : T; \bar{o} \vdash (s : U) \leftarrow (\Gamma', o : T; \bar{o}' \vdash^{\kappa} U' \text{ in } p)$.
(e) *If ($\Gamma; \bar{o} \vdash s : U$), then ($\Gamma, o : T; \bar{o} \vdash s : U$).*

Let us say that *e is a call* whenever e has the form $o.m$<$\bar{v}$>($\bar{u}$) or $C.k(\bar{u})$ or `new`C<ar,w>$.k(\bar{u})$.

Lemma 31 (Return Lemma). *Suppose Γ and Γ' are closed, $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}} \subseteq \texttt{rdonly wrlocal}\,\mathsf{ty}$ and e is a call.*

If $\Gamma;\bar{o} \vdash^{\kappa} ((\mathscr{E}[e]\,\mathsf{in}\,p) : T\ \mathsf{in}\ p) \leftarrow (\bar{o}' \vdash^{\kappa'} T'\ \mathsf{in}\ p')$
and $(\Gamma';\bar{o}' \vdash^{\kappa'} (v\,\mathsf{in}\,p') : T'\ \mathsf{in}\ p')$, *then* $(\Gamma;\bar{o} \vdash^{\kappa} (\mathscr{E}[v]\,\mathsf{in}\,p) : T\ \mathsf{in}\ p)$.

Proof.

(1) Γ, Γ' are closed — assumption
(2) $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}} \subseteq \texttt{rdonly}\ \texttt{wrlocal}\,\mathsf{ty}$ — assumption
(3) e is a call — assumption
(4) $\Gamma;\bar{o} \vdash^{\kappa} ((\mathscr{E}[e]\,\mathsf{in}\,p) : T\ \mathsf{in}\ p) \leftarrow (\bar{o}' \vdash^{\kappa'} T'\ \mathsf{in}\ p')$ — assumption
(5) $\Gamma';\bar{o}' \vdash^{\kappa'} (v\,\mathsf{in}\,p') : T'\ \mathsf{in}\ p'$ — assumption

By inverting judgment (4), we obtain:

(6) $\Gamma;\bar{o} \vdash^{\kappa} (\mathscr{E}[e]\,\mathsf{in}\,p) : T\ \mathsf{in}\ p$
(7) $e : (\bar{o} \vdash p) \leftarrow (\bar{o}' \vdash^{\kappa'} p')$
(8) $ea = \{\texttt{anon} \mid p \neq p'\}$
(9) $U \stackrel{\Delta}{=} ea\ \texttt{rdonly}\ \texttt{wrlocal}\,T'$ — abbreviation
(10) $\Gamma, x : U \vdash^{\kappa} \mathscr{E}[x] : T\ \mathsf{in}\ p, \mathsf{acc}(p)$
(11) $\Gamma \vdash T' : \mathsf{ok}$

By further inverting judgment (6), we obtain:

(12) $\mathsf{vals}(\mathscr{E}[e]) \cap \bar{o} \subseteq \{p\}$
(13) $\Gamma;\bar{o} \vdash \mathsf{eatts}(T)\ \mathsf{ok\ in}\ p$
(14) $\bar{o} \subseteq \mathsf{imm}$
(15) $\mathsf{deanon}(T, p \notin \bar{o}) = T$

By inverting judgment (5), we obtain:

(16) $\Gamma' \vdash^{\kappa} v : T'\ \mathsf{in}\ p', \mathsf{acc}(p')$
(17) $\mathsf{vals}(v) \cap \bar{o}' \subseteq \{p'\}$

Case 1, $v = \texttt{null}$: Then $(\Gamma \vdash v : U\ \mathsf{in}\ p, \mathsf{acc}(p))$, by (Null). We apply substitutivity (Lemma 9) to this judgment and judgment (10) to obtain:

(1.1) $\Gamma \vdash \mathscr{E}[v] : T\ \mathsf{in}\ p, \mathsf{acc}(p)$

From (12) and $v = \texttt{null}$, we get:

(1.2) $\mathsf{vals}(\mathscr{E}[v]) \cap \bar{o} \subseteq \{p\}$

We obtain $(\Gamma;\bar{o} \vdash^{\kappa} (\mathscr{E}[v]\,\mathsf{in}\,p) : T\ \mathsf{in}\ p, \mathsf{acc}(p))$ by applying (Frm) to (1.1), (1.2), (13) and (14).

Case 2, $v \neq \texttt{null}$: We abbreviate:

(2.1) $U' \stackrel{\Delta}{=} U - \texttt{anon}$ — abbreviation

By inverting judgment (16), we obtain:

(2.2) $\Gamma' \vdash \Gamma'(v) \preceq U'$

By Lemma 6 and $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}}$, we obtain:

(2.3) $\Gamma \vdash \Gamma(v) \preceq U'$

By (Obj) and (Sub), we obtain:

(2.4) $\Gamma \vdash v : \mathsf{deanon}(\mathtt{anon}\, U', v = p) \text{ in } p, \mathsf{acc}(p)$

We want to prove $(\Gamma; \bar{o} \vdash^{\kappa} (\mathscr{E}[v] \,\mathsf{in}\, p) : T \text{ in } p)$. By (Frm), we need to show the following:

(2.5) $\Gamma \vdash^{\kappa} \mathscr{E}[v] : T \text{ in } p, \mathsf{acc}(p)$ goal
(2.6) $\mathsf{vals}(\mathscr{E}[v]) \cap \bar{o} \subseteq \{p\}$ goal
(2.7) $\Gamma; \bar{o} \vdash \mathsf{eatts}(T) \text{ ok in } p$ goal
(2.8) $\bar{o} \subseteq \mathsf{imm}$ goal

Goals (2.7) and (2.8) are exactly (13) and (14).

Proof of goal (2.5)*:*

Case 2.1, $p = p'$ and $\mathtt{anon} \notin \mathsf{atts}(T')$: Then $\mathtt{anon} \notin \mathsf{atts}(U)$, by definition of U, see (9), (8). Then, $\Gamma \vdash \mathsf{deanon}(\mathtt{anon}\, U', v = p) = \mathsf{deanon}(\mathtt{anon}\,(U - \mathtt{anon}), v = p) = \mathsf{deanon}(\mathtt{anon}\, U, v = p) \preceq U$. By applying substitutivity (Lemma 9) to judgments (2.4) and (10), we obtain $(\Gamma \vdash^{\kappa} \mathscr{E}[v] : T \text{ in } p, \mathsf{acc}(p))$.

Case 2.2, $p = p'$ and $\mathtt{anon} \in \mathsf{atts}(T')$: Because $p = p'$, the reason for judgment (16) cannot be (New Frm Done). Therefore $(\Gamma' \vdash v : T' \text{ in } p', \mathsf{acc}(p'))$. Because $\mathtt{anon} \in \mathsf{atts}(T')$, the judgment $(\Gamma' \vdash v : T' \text{ in } p', \mathsf{acc}(p'))$ is only derivable if $v \neq p'$. So we have $v \neq p'$. We also have $v \neq p$, because $p = p'$. Then $\Gamma \vdash \mathsf{deanon}(\mathtt{anon}\, U', v = p) = \mathtt{anon}\, U' = \mathtt{anon}\,(U - \mathtt{anon}) \preceq U$. By applying substitutivity (Lemma 9) to judgments (2.4) and (10), we obtain $(\Gamma \vdash^{\kappa} \mathscr{E}[v] : T \text{ in } p, \mathsf{acc}(p))$.

Case 2.3, $p \neq p'$: Applying anonymous substitutivity (Lemma 12) to judgments (2.4) and (10) results in:

(2.3.1) $\Gamma \vdash^{\kappa} \mathscr{E}[v] : \mathsf{deanon}(T, v = p) \text{ in } p, \mathsf{acc}(p)$

We are done, if we can show the following:

(2.3.2) $\Gamma \vdash \mathsf{deanon}(T, v = p) \preceq T$ goal

By (15), it suffices to show the following:

(2.3.3) $\Gamma \vdash \mathsf{deanon}(T, v = p) \preceq \mathsf{deanon}(T, p \notin \bar{o})$ goal

It suffices to show the following:

(2.3.4) $v = p \Rightarrow p \notin \bar{o}$ goal

There we go:

(2.3.5) $v = p$ assumption

Suppose towards a contradiction that $p \in \bar{o}$. By inspecting the last rule for judgment (7), we obtain $\bar{o} \subseteq \bar{o}'$. Thus, $p \in \bar{o}'$. We also have $p = v \in \mathsf{vals}(v)$. By (17), we obtain $p = p'$. But that contradicts assumption $p \neq p'$.

Proof of goal (2.6)*:* By (12), we already know that $\mathsf{vals}(\mathscr{E}) \cap \bar{o} \subseteq \{p\}$. Therefore, it suffices to show $(\{v\} - \mathsf{vals}(\mathscr{E})) \cap \bar{o} \subseteq \{p\}$.

(2.9) $v \notin \mathsf{vals}(\mathscr{E})$ — assumption
(2.10) $v = p$ or $v \notin \bar{o}$ — goal

By (17), we know that:

(2.11) $v = p'$ or $v \notin \bar{o}'$

We now distinguish cases by the last rule for judgment (7).

Case 2.4, (Call Frm New): In case $v = p'$, we obtain $v = p' \notin \bar{o}$ by premise of (Call Frm New). In case $v \notin \bar{o}'$, we obtain $v \notin \bar{o}$ because $\bar{o} \subseteq \bar{o}'$ by premise of (Call Frm New).

Case 2.5, (Call Frm Cons): In this case, $p = p'$ and $\bar{o} = \bar{o}'$. Therefore, our goal (2.10) is identical to fact (2.11).

Case 2.6, (Call Frm Mth): In this case, $\bar{o} = \bar{o}'$ and $p' \in \mathsf{vals}(e)$. If $v \notin \bar{o}$, we are done. So suppose $v \in \bar{o}$. Then $v = p'$, by (2.11). Because $p' \in \mathsf{vals}(e)$, we get $p' = p$, by (12). We have established that $v = p' = p$. □

We now present some corollaries of the substitutivity lemmas from Section C. They customize these earlier lemmas to the form that we need in the type preservation proof to deal with method, `new`- and constructor calls.

Lemma 32 (Customized Owner Substitutivity). *If Γ is closed, $(\Gamma \vdash \bar{o} : \bullet)$, $|\bar{x}| = |\bar{o}|$, $(\Gamma, \bar{x} : \bullet, \bar{y} : \bullet \vdash \bar{ty} : \mathsf{ok})$, $\bar{x} \cap \bar{z} = \emptyset$ and $(\Gamma, \bar{x} : \bullet, \bar{y} : \bullet, \bar{z} : \bar{ty} \vdash \mathscr{J})$, then $(\Gamma, \bar{y} : \bullet, \bar{z} : \bar{ty}[\bar{x} \leftarrow \bar{o}] \vdash \mathscr{J}[\bar{x} \leftarrow \bar{o}])$.*

Proof.

(1) Γ is closed — assumption
(2) $\Gamma \vdash \bar{o} : \bullet$ — assumption
(3) $|\bar{x}| = |\bar{o}|$ — assumption
(4) $\Gamma, \bar{x} : \bullet, \bar{y} : \bullet \vdash \bar{ty} : \mathsf{ok}$ — assumption
(5) $\Gamma, \bar{x} : \bullet, \bar{y} : \bullet, \bar{z} : \bar{ty} \vdash \mathscr{J}$ — assumption
(6) $\sigma = (\bar{x} \leftarrow \bar{o})$ — abbreviation

By weakening judgment (2), we obtain $(\Gamma, \bar{y} : \bullet \vdash \bar{o} : \bullet)$. By applying Lemma 10 to this judgment and judgment (4), we obtain $(\Gamma, \bar{y} : \bullet \vdash \bar{ty}[\sigma] : \mathsf{ok})$. Then $(\Gamma, \bar{y} : \bullet, \bar{z} : \bar{ty}[\sigma] \vdash \diamond)$. Then $(\Gamma, \bar{y} : \bullet, \bar{z} : \bar{ty}[\sigma] \vdash \bar{o} : \mathsf{ok})$, by weakening judgment (2). By applying Lemma 10 to this judgment and judgment (5), we obtain $(\Gamma, \bar{y} : \bullet, \bar{z} : \bar{ty}[\sigma] \vdash \mathscr{J}[\sigma])$. □

Lemma 33 (Customized Value Substitutivity). *If $\mathsf{dom}(\Gamma_{\mathsf{own}}) \cup \mathsf{ran}(\Gamma_{\mathsf{val}}) \subseteq \mathsf{ObjId}$, $(\Gamma \vdash \bar{v} : \bar{U}$ in $o, ar)$, $|\bar{x}| = |\bar{v}|$, $(\Gamma \vdash \bar{U} : \mathsf{ok})$ and $(\Gamma, \bar{x} : \mathtt{anon}\,\bar{U} \vdash e : T$ in $o, ar)$, then $(\Gamma \vdash e[\bar{x} \leftarrow \bar{v}] : \mathsf{deanon}(T[\bar{x} \leftarrow \bar{v}], o \in \bar{v})$ in $o, ar)$.*

Proof. By induction on $|\bar{v}|$. The base case for $|\bar{v}| = 0$ is trivial. So let us do the inductive case.

(1) $\bar{v} = (v, \bar{v}')$, $\bar{x} = (x, \bar{x}')$ and $\bar{U} = (U, \bar{U}')$ — assumption

(2) $\mathsf{dom}(\Gamma_{\mathsf{own}}) \cup \mathsf{ran}(\Gamma_{\mathsf{val}}) \subseteq \mathsf{ObjId}$ assumption
(3) $\Gamma \vdash \bar{v} : \bar{U}$ in o, ar assumption
(4) $|\bar{x}| = |\bar{v}|$ assumption
(5) $\Gamma \vdash \bar{U} : \mathsf{ok}$ assumption
(6) $\Gamma, \bar{x} : \mathtt{anon}\,\bar{U} \vdash e : T$ in o, ar assumption

By weakening, we obtain $(\Gamma, x : \mathtt{anon}\,U \vdash \bar{v}' : \bar{U}')$ and $(\Gamma, x : \mathtt{anon}\,U \vdash \bar{U}' : \mathsf{ok})$. By induction hypothesis, $(\Gamma, x : \mathtt{anon}\,U \vdash e[\bar{x}' \leftarrow \bar{v}'] : \mathsf{deanon}(T[\bar{x}' \leftarrow \bar{v}'], o \in \bar{v}')$ in $o, ar)$. By applying Lemma 9 to this judgment and $(\Gamma \vdash v : U$ in $o, ar)$, we obtain $(\Gamma \vdash e[\bar{x} \leftarrow \bar{v}] : \mathsf{deanon}(T[\bar{x} \leftarrow \bar{v}], o \in \bar{v})$ in $o, ar)$. □

Lemma 34 (Customized Substitutivity for Calls). *If Γ is closed, $\sigma = (\bar{y}, \mathtt{this}, \bar{x} \leftarrow \bar{o}, p, \bar{v})$, $(\Gamma \vdash \bar{o} : \bullet)$, $(\Gamma \vdash p, \bar{v} : (U, \bar{V})[\bar{y} \leftarrow \bar{o}]$ in $p, ar)$, $(\Gamma, \bar{y} : \bullet \vdash U, \bar{V} : \mathsf{ok})$ and $(\Gamma, \bar{y} : \bullet, \mathtt{this} : \bullet, \mathtt{this} : U, \bar{x} : \mathtt{anon}\,\bar{V} \vdash e : T$ in $\mathtt{this}, ar)$, then $(\Gamma \vdash e[\sigma] : \mathsf{deanon}(T[\sigma], p \in \bar{v})$ in $p, ar)$.*

Proof.

(1) Γ is closed assumption
(2) $\sigma = (\bar{y}, \mathtt{this}, \bar{x} \leftarrow \bar{o}, p, \bar{v})$ assumption
(3) $\Gamma \vdash \bar{o} : \bullet$ assumption
(4) $\Gamma \vdash p, \bar{v} : (U, \bar{V})[\bar{y} \leftarrow \bar{o}]$ in p, ar assumption
(5) $\Gamma, \bar{y} : \bullet \vdash U, \bar{V} : \mathsf{ok}$ assumption
(6) $\Gamma, \bar{y} : \bullet, \mathtt{this} : \bullet, \mathtt{this} : ty, \bar{x} : \mathtt{anon}\,\bar{V} \vdash e : T$ in $\mathtt{this}, ar$ assumption

We apply Lemma 32 to judgments (3), (5) and (6), obtaining:

(7) $\Gamma, \mathtt{this} : \bullet, \mathtt{this} : U[\bar{y} \leftarrow \bar{o}], \bar{x} : \mathtt{anon}\,\bar{V}[\bar{y} \leftarrow \bar{o}] \vdash e[\bar{y} \leftarrow \bar{o}] : T[\bar{y} \leftarrow \bar{o}]$ in p, ar

Next we apply Lemma 9 to $(\Gamma \vdash p : ty[\bar{y} \leftarrow \bar{o}]$ in $p, ar)$ and (7), obtaining:

(8) $\Gamma, \bar{x} : \mathtt{anon}\,\bar{V}[\bar{y} \leftarrow \bar{o}] \vdash e[\bar{y}, \mathtt{this} \leftarrow \bar{o}, p] : T[\bar{y}, \mathtt{this} \leftarrow \bar{o}, p]$ in p, ar

Finally we apply Lemma 33 to $(\Gamma \vdash \bar{v} : \bar{V}[\bar{y} \leftarrow \bar{o}]$ in $ar, p)$ and (8), obtaining:

(9) $\Gamma \vdash e[\sigma] : \mathsf{deanon}(T[\sigma], p \in \bar{v})$ in p, ar

□

Proof of Theorem 2 (Type Preservation). *If $(\bar{o} \vdash h :: s : T)$ and $h :: s \xrightarrow{\bar{p}} state$, then $\bar{p} \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$ and there exists $\bar{o}'$ such that $(\bar{o}' \vdash state : T)$ and $(\bar{o}' - \bar{o}) \cap \mathsf{dom}(h) = \emptyset$.*

Proof.

(1) $\bar{o} \vdash h :: s : T$ assumption
(2) $h :: s \xrightarrow{\bar{p}} state$ assumption

By inverting (1), we obtain a Γ such that:

(3) $\Gamma; \bar{o} \vdash h : \mathsf{ok}$
(4) $\Gamma; \bar{o} \vdash s : T$
(5) $\bar{o} \subseteq \mathsf{dom}(\Gamma_{\mathsf{val}})$
(6) $\Gamma_{\mathsf{val}} = \mathtt{rdonly}\ \mathtt{wrlocal}\ \mathtt{ty}|\mathsf{dom}(\Gamma_{\mathsf{val}})$

We distinguish cases by the possible reasons for (2).

Case 1, (Red Rtr):

$$\frac{e = q.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) \text{ or } e = \texttt{new}\,C\texttt{<}ar,u\texttt{>}.k(\bar{v}) \text{ or } e = C.k(\bar{v})}{h :: s', (\mathscr{E}[e] \,\mathsf{in}\, o), (v \,\mathsf{in}\, p) \;\rightarrow\; h :: s', \mathscr{E}[v] \,\mathsf{in}\, o}$$

(1.1) $s = s', (\mathscr{E}[e] \,\mathsf{in}\, o), (v \,\mathsf{in}\, p)$
(1.2) $state' = h :: s', \mathscr{E}[v] \,\mathsf{in}\, o$

By inverting judgment (4), we obtain the following:

(1.3) $\Gamma'; \bar{o} \vdash^{\kappa} (v \,\mathsf{in}\, p) : T' \;\mathsf{in}\; p$
(1.4) $\Gamma; \bar{o}' \vdash (s', (\mathscr{E}[e] \,\mathsf{in}\, o) : T) \leftarrow (\Gamma'; \bar{o} \vdash^{\kappa} T' \;\mathsf{in}\; p)$

By inverting judgment (1.4), we obtain the following:

(1.5) $\Gamma; \bar{o}'' \vdash (s' : T) \leftarrow (\Gamma''; \bar{o}' \vdash^{\kappa'} T'' \;\mathsf{in}\; o)$
(1.6) $\Gamma''; \bar{o}' \vdash^{\kappa'} ((\mathscr{E}[e] \,\mathsf{in}\, o) : T'' \;\mathsf{in}\; o) \leftarrow (\bar{o} \vdash^{\kappa} T' \;\mathsf{in}\; p)$
(1.7) Γ' is closed and $\Gamma'_{\mathsf{val}} = \Gamma_{\mathsf{val}}$

By inverting (1.5), we obtain that Γ'' is closed and $\Gamma''_{\mathsf{val}} = \Gamma_{\mathsf{val}}$. We can therefore apply Lemma 31 to (1.6) and (1.3) to obtain:

(1.8) $\Gamma''; \bar{o}' \vdash^{\kappa'} (\mathscr{E}[v] \,\mathsf{in}\, o) : T'' \;\mathsf{in}\; o$

From (1.5) and (1.8), we obtain $(\Gamma; \bar{o}' \vdash s', (\mathscr{E}[v] \,\mathsf{in}\, o) : T)$, by (Stk Push). Using (State), we get $(\bar{o}' \vdash state' : T)$. From judgment (1.4) we obtain $\bar{o}' \subseteq \bar{o}$, by Lemma 28. Therefore, $(\bar{o}' - \bar{o}) \cap \mathsf{dom}(h) = \emptyset \cap \mathsf{dom}(h) = \emptyset$, as desired.

Case 2, (Red Get):

$$\frac{h = h', o\{..f = v..\}}{h :: s', \mathscr{E}[o.f] \,\mathsf{in}\, p \;\rightarrow\; h :: s', \mathscr{E}[v] \,\mathsf{in}\, p}$$

(2.1) $s = s', \mathscr{E}[o.f] \,\mathsf{in}\, p$
(2.2) $state' = h :: s', \mathscr{E}[v] \,\mathsf{in}\, p$

By inverting judgment (4), we obtain the following:

(2.3) $\Gamma; \bar{o}' \vdash (s' : T) \leftarrow (\Gamma'; \bar{o} \vdash^{\kappa} T' \;\mathsf{in}\; p)$
(2.4) $\Gamma'; \bar{o} \vdash^{\kappa} (\mathscr{E}[o.f] \,\mathsf{in}\, p) : T' \;\mathsf{in}\; p$

By inverting judgment (2.4), we obtain:

(2.5) $\Gamma' \vdash^{\kappa} \mathscr{E}[o.f] : T' \;\mathsf{in}\; p, \mathsf{acc}(p)$
(2.6) $\mathsf{vals}(\mathscr{E}[o.f]) \cap \bar{o} \subseteq \{p\}$
(2.7) $\Gamma'; \bar{o} \vdash \mathsf{eatts}(T') \;\mathsf{ok\ in}\; p$
(2.8) $\bar{o} \subseteq \mathsf{imm}$
(2.9) $\mathsf{deanon}(T', p \notin \bar{o}) = T'$

By inverting judgment (3), we obtain:

(2.10) $v \notin \bar{o}$

From (2.6) and $v \notin \bar{o}$, we obtain:

(2.11) $\mathsf{vals}(\mathscr{E}[v]) \cap \bar{o} \subseteq \{p\}$

If we can show that $(\Gamma'; \bar{o} \vdash^{\kappa} \mathscr{E}[v] : T' \ \mathsf{in}\ p, \mathsf{acc}(p))$, then we can apply (Frm) to (2.11), (2.7) and (2.8) to obtain $(\Gamma' \vdash^{\kappa} (\mathscr{E}[v] \,\mathsf{in}\, p) : T' \ \mathsf{in}\ p)$, and then (Stk Push) and (State) to obtain $(\bar{o} \vdash \mathit{state}' : T)$. So we need to show the following statement:

(2.12) $\Gamma' \vdash^{\kappa} \mathscr{E}[v] : T' \ \mathsf{in}\ p, \mathsf{acc}(p)$ goal

By Lemma 13, we obtain a type U such that:

(2.13) $\Gamma', x : U \vdash^{\kappa} \mathscr{E}[x] : T' \ \mathsf{in}\ p, \mathsf{acc}(p)$
(2.14) $\Gamma' \vdash o.f : U \ \mathsf{in}\ p, \mathsf{acc}(p)$
(2.15) The last rule of (2.14)'s type derivation is (Get).

We now consider the last rule of (2.14)'s type derivation:

$$\frac{\begin{array}{c} \mathit{ty}\, f \in \mathsf{fd}(C_o) \quad \sigma = \mathsf{self}(o, ar_o, v_o) \\ \Gamma' \vdash o, p : ea_o\, C_o\texttt{<}ar_o, v_o\texttt{>}, C_o\texttt{<}\mathsf{acc}(p), w_p\texttt{>} \ \mathsf{in}\ p, \mathsf{acc}(p) \end{array}}{\Gamma' \vdash o.f : \texttt{anon}\ \texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}[\sigma] \ \mathsf{in}\ p, \mathsf{acc}(p)}$$

(2.16) $U = \texttt{anon}\ \texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}[\sigma]$

By inverting $(\Gamma' \vdash o : ea_o\, C_o\texttt{<}ar_o, v_o\texttt{>})$, we obtain:

(2.17) $\Gamma'(o) = ea'_o\, C'_o\texttt{<}ar'_o, v'_o\texttt{>}$
(2.18) $\Gamma' \vdash C'_o\texttt{<}ar'_o, v'_o\texttt{>} \preceq C_o\texttt{<}ar_o, v_o\texttt{>}$

We define:

(2.19) $\sigma' \triangleq \mathsf{self}(o, ar'_o, v'_o)$

By inverting (3), we obtain $(\Gamma \vdash v : \mathit{ty}[\sigma'] \ \mathsf{in}\ \texttt{world})$. Because $\Gamma_{\mathsf{val}} = \Gamma'_{\mathsf{val}}$, we can replace Γ by Γ' (Lemma 6) to obtain:

(2.20) $\Gamma' \vdash v : \mathit{ty}[\sigma'] \ \mathsf{in}\ \texttt{world}$

By (6), $\Gamma'_{\mathsf{val}}(o)$ is of the form $\texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}'$ for all o in $\mathsf{dom}(\Gamma'_{\mathsf{val}})$ and some ty'. Using this fact, we obtain:

(2.21) $\Gamma' \vdash v : \texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}[\sigma'] \ \mathsf{in}\ p, \mathsf{acc}(p)$

By (Fld Dcl), we know that ty is legal. We apply Lemma 3(e) to obtain:

(2.22) $\Gamma' \vdash \texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}[\sigma'] \preceq \texttt{rdonly}\ \texttt{wrlocal}\, \mathit{ty}[\sigma] = U - \texttt{anon}$

We apply anonymous substitutivity (Lemma 12) to (2.21)/(2.22) and (2.13):

(2.23) $\Gamma' \vdash^{\kappa} \mathscr{E}[v] : \mathsf{deanon}(T', v = p) \ \mathsf{in}\ p, \mathsf{acc}(p)$

Using (2.10) and (2.9) we obtain:

(2.24) $\Gamma' \vdash \mathsf{deanon}(T', v = p) \preceq \mathsf{deanon}(T', p \notin \bar{o}) = T'$

Then proof goal (2.12) holds by (Sub).

Case 3, (Red Set):

$$\frac{}{h', o\{f = u, \bar{g} = \bar{w}\} :: s', \mathscr{E}[o.f\texttt{=}v]\,\mathsf{in}\,p \xrightarrow{o} h', o\{f = v, \bar{g} = \bar{w}\} :: s', \mathscr{E}[v]\,\mathsf{in}\,p}$$

(3.1) $h = h', o\{f = u, \bar{g} = \bar{w}\}$
(3.2) $s = s', \mathscr{E}[o.f\texttt{=}v]\,\mathsf{in}\,p$
(3.3) $state' = h', o\{f = v, \bar{g} = \bar{w}\} :: s', \mathscr{E}[v]\,\mathsf{in}\,p$

We need to show three things: firstly, that the heap component of *state′* is ok, secondly, that the stack component of *state′* has type T and, thirdly, that the write effect o is in $\mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$:

(3.4) $\Gamma; \bar{o} \vdash o\{f = v, \bar{g} = \bar{w}\} : \mathsf{ok}$ goal
(3.5) $\Gamma; \bar{o} \vdash (s', \mathscr{E}[v]\,\mathsf{in}\,p) : T$ goal
(3.6) $o \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$ goal

First, we derive some facts that are useful for several of these proof goals: By inverting judgment (4), we obtain:

(3.7) $\Gamma; \bar{o}' \vdash (s' : T) \leftarrow (\Gamma'; \bar{o} \vdash^{\kappa} T' \;\mathsf{in}\; p)$
(3.8) $\Gamma'; \bar{o} \vdash^{\kappa} (\mathscr{E}[o.f\texttt{=}v]\,\mathsf{in}\,p) : T' \;\mathsf{in}\; p$

By inverting judgment (3.8), we obtain:

(3.9) $\Gamma' \vdash^{\kappa} \mathscr{E}[o.f\texttt{=}v] : T' \;\mathsf{in}\; p, \mathsf{acc}(p)$
(3.10) $\mathsf{vals}(\mathscr{E}[o.f\texttt{=}v]) \cap \bar{o} \subseteq \{p\}$
(3.11) $\Gamma'; \bar{o} \vdash \mathsf{eatts}(T') \;\mathsf{ok\ in}\; p$
(3.12) $\bar{o} \subseteq \mathsf{imm}$
(3.13) $\mathsf{deanon}(T', p \notin \bar{o}) = T'$

Note that `rdonly` $\notin \mathsf{eatts}(T')$ because if `rdonly` were in $\mathsf{eatts}(T')$ then $\mathscr{E}[o.f\texttt{=}v]$ would not contain a subexpression $o.f\texttt{=}v$ that writes to the heap.

(3.14) $\texttt{rdonly} \notin \mathsf{eatts}(T')$

By applying Lemma 13 to (3.9), we obtain a type U such that:

(3.15) $\Gamma', x : U \vdash^{\kappa} \mathscr{E}[x] : T' \;\mathsf{in}\; p, \mathsf{acc}(p)$
(3.16) $\Gamma' \vdash o.f\texttt{=}v : U \;\mathsf{in}\; p, \mathsf{acc}(p)$
(3.17) The last rule of (3.16)'s type derivation is (Set).

We spell out the last rule of (3.16)'s type derivation:

$$\frac{\begin{array}{l} ty\,f \in \mathsf{fd}(C_o) \quad \Gamma' \vdash v_o : \bullet \\ ea = \bigcap(\{x \;\mathsf{as}\; \texttt{wrlocal} \mid (\{x\}, o, ar_o, v_o) \;\mathsf{wrloc\ in}\; p\} \cup \{\texttt{anon}\}, ea_v) \quad ar_o \;\mathsf{wrsafe\ in\ acc}(p) \\ \Gamma' \vdash o, p, v : ea_o\, C_o\texttt{<}ar_o, v_o\texttt{>}, C_o\texttt{<}\mathsf{acc}(p), w_p\texttt{>}, ea_v\, ty[\sigma] \;\mathsf{in}\; p, \mathsf{acc}(p) \quad \sigma = \mathsf{self}(o, ar_o, v_o) \end{array}}{\Gamma' \vdash o.f\texttt{=}v : ea\,ty[\sigma] \;\mathsf{in}\; p, \mathsf{acc}(p)}$$

(3.18) $U = ea\,ty[\sigma]$

By inverting $(\Gamma' \vdash v_o : \bullet)$. Therefore:

(3.19) $v_o \in \mathsf{dom}(\Gamma'_{\mathsf{own}})$

Let C'_o, C'_p be such that:

(3.20) $\Gamma'(o) = \texttt{rdonly wrlocal}\, C'_o\texttt{<}\mathsf{acc}(p), \mathsf{owner}(o)\texttt{>}$
(3.21) $\Gamma'(p) = \texttt{rdonly wrlocal}\, C'_p\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>}$

By inverting $(\Gamma' \vdash o, p : ea_o\, C_o\texttt{<}ar_o, v_o\texttt{>}, C_p\texttt{<}\mathsf{acc}(p), w_p\texttt{>}\ \mathsf{in}\ p, ar_p)$, we obtain:

(3.22) $\Gamma' \vdash C'_o\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>} \preceq C_o\texttt{<}ar_o, v_o\texttt{>}$
(3.23) $\Gamma' \vdash C'_p\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>} \preceq C_o\texttt{<}\mathsf{acc}(p), w_p\texttt{>}$

Proof of goal (3.4)*:* By inverting judgment (3), we obtain:

(3.24) $(u, \bar{w}) \cap \bar{o} = \emptyset$
(3.25) $\mathsf{fd}(C) = ty\, f, \bar{ty}\, \bar{g}$
(3.26) $\sigma' = \mathsf{self}(o, \mathsf{acc}(p), \mathsf{owner}(o))$
(3.27) $\Gamma \vdash u, \bar{w} : ty[\sigma'], \bar{ty}[\sigma']\ \mathsf{in}\ \texttt{world}$

It suffices to show the following:

(3.28) $v \notin \bar{o}$ subgoal for (3.4)
(3.29) $\Gamma \vdash v : ty[\sigma']\ \mathsf{in}\ \texttt{world}$ subgoal for (3.4)

Proof of subgoal (3.28)*:* Suppose towards a contradiction that $v \in \bar{o}$. Then $v = p$, by (3.10). The reason for (3.11) cannot be (Frm Imm Cons), because if `anon` were in $\mathsf{eatts}(T')$ then $\mathscr{E}[o.f\texttt{=}v]$ would not contain a subexpression $o.f\texttt{=}v$ (which is identical to $o.f\texttt{=}p$) that writes the `this`-binding p to the heap. Then $p \notin \bar{o}$. But then $v = p \notin \bar{o}$, in contradiction to our assumption $v \in \bar{o}$.

Proof of subgoal (3.29)*:* We know that $(\Gamma' \vdash v : ty[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p))$, by premise of the last rule of $(\Gamma' \vdash o.f\texttt{=}v : ea\, ty[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p))$'s type derivation, as displayed above. By Lemma 8, we obtain $(\Gamma' \vdash v : ty[\sigma]\ \mathsf{in}\ \texttt{world})$. It therefore suffices to show the following:

(3.30) $\Gamma' \vdash ty[\sigma] \preceq ty[\sigma']$ subgoal towards (3.29)

By Lemma 3(e), it suffices to show the following:

(3.31) $\Gamma' \vdash C'_o\texttt{<}ar_o, v_o\texttt{>} \preceq C_o\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>}$ subgoal towards (3.30)

But this holds because we can swap the class parameters in subtyping judgment (3.22), by Lemma 3(d).

Proof of goal (3.5)*:* By the last rule of $(\Gamma' \vdash o.f\texttt{=}v : ea\, ty[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p))$'s type derivation, as displayed above, we obtain:

(3.32) $\Gamma' \vdash v : ea_v\, ty[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p)$
(3.33) $\texttt{anon} \in ea \Rightarrow \texttt{anon} \in ea_v$

Because $\Gamma'_{\mathsf{val}} = \texttt{rdonly wrlocal}\,\mathsf{ty}|\mathsf{dom}(\Gamma'_{\mathsf{val}})$, we obtain:

(3.34) $\Gamma' \vdash v : \texttt{rdonly wrlocal}\ ea_v\, ty[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p)$

Using (3.33), we obtain:

(3.35) $\Gamma' \vdash \texttt{rdonly wrlocal}\ ea_v\, ty[\sigma] \preceq ea\, ty[\sigma] = U$

We apply substitutivity (Lemma 9) to judgments (3.34)/(3.35) and (3.15) to obtain $(\Gamma' \vdash^{\kappa} \mathscr{E}[v] : T'\ \mathsf{in}\ p, \mathsf{acc}(p))$. We apply (Frm) to this judgment and (3.9), (3.10) and (3.11) to obtain:

(3.36) $\Gamma'; \bar{o} \vdash^{\kappa} (\mathscr{E}[v]\,\mathsf{in}\,p) : T'\ \mathsf{in}\ p$

We then apply (Stk Push) to (3.36) and (3.7) to obtain our proof goal (3.5).

Proof of goal (3.6)*:* We distinguish cases by the possible reasons for (3.11). We have already established that this is not (Frm Rdonly).

Case 3.1, (Frm Rdwr), $\mathsf{acc}(p) = \mathtt{rdwr}$: Because ($ar_o$ $\mathsf{wrsafe\ in}$ $\mathsf{acc}(p)$), we have $ar_o = \mathtt{rdwr}$. Then the reason for subtyping judgment (3.22) cannot be (Sub Share) and, thus, $v_o = \mathsf{owner}(o)$. Then, $\mathsf{owner}(o) = v_o \in \mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$, by premise of (Frm Rdwr). Then $o \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm}) \cup \mathsf{imm}$. If $o \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$, we are done. We will show that $o \in \mathsf{imm}$ is impossible: To this end, we assume, towards a contradiction, that $o \in \mathsf{imm}$. Then $\mathtt{immutable} \in \mathsf{atts}(C'_o)$. Then $\mathtt{immutable} \in \mathsf{atts}(C_o)$, because superclasses (except `Object`) of immutable classes are immutable. Then $\mathtt{immutable} \in \mathsf{atts}(C'_p)$, because subclasses of immutable classes are immutable. Then $p \in \mathsf{imm}$, by definition of imm. But that contradicts $p \notin \mathsf{imm}$, which is a premise of (Frm Rdwr).

Case 3.2, (Frm Rdwr), $\mathtt{wrlocal} \in \mathsf{eatts}(T')$: Because T' is $\mathscr{E}[o.f\texttt{=}v]$'s type, the subexpression $o.f\texttt{=}v$ must also be `wrlocal`. That is, $\mathtt{wrlocal} \in \mathsf{eatts}(U) = ea$. By inspecting the last rule of ($\Gamma' \vdash o.f\texttt{=}v : U$ in $p, \mathsf{acc}(p)$)'s derivation, see above, we get $(\{\mathtt{wrlocal}\}, o, ar_o, v_o)$ $\mathsf{wrloc\ in}$ p. By definition, this means that either $o = p$ or $(ar_o, v_o) = (\mathtt{rdwr}, p)$. If $o = p$, we get $o = p \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$, by premise of (Frm Rdwr). So suppose that $(ar_o, v_o) = (\mathtt{rdwr}, p)$. Then the reason for subtyping judgment (3.22) cannot be (Sub Share) and, thus, $\mathsf{owner}(o) = v_o$. So we have $\mathsf{owner}(o) = v_o = p$. By premise of (Frm Rdwr), we have $p \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$. So $\mathsf{owner}(o) = p \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$ Then $o \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm}) \cup \mathsf{imm}$. If $o \in \mathsf{inside}(\bar{o}) \cup \mathsf{outside}(\mathsf{imm})$, we are done. So suppose $o \in \mathsf{imm}$. Then $\mathtt{immutable} \in \mathsf{atts}(C'_o)$. Then $\mathtt{immutable} \in \mathsf{atts}(C_o)$, because superclasses (except `Object`) of immutable classes are immutable. Then $\mathtt{immutable} \in \mathsf{atts}(C'_p)$, because subclasses of immutable classes are immutable. Then $p \in \mathsf{imm}$, by definition of imm. But that contradicts $p \notin \mathsf{imm}$, which is a premise of (Frm Rdwr).

Case 3.3, (Frm Imm Cons): In this case, $\mathtt{wrlocal} \in \mathsf{eatts}(T')$. Because T' is $\mathscr{E}[o.f\texttt{=}v]$'s type, the subexpression $o.f\texttt{=}v$ must also be `wrlocal`. That is, $\mathtt{wrlocal} \in \mathsf{eatts}(U) = ea$. By inspecting the last rule of ($\Gamma' \vdash o.f\texttt{=}v : U$ in $p, \mathsf{acc}(p)$)'s derivation, see above, we get $(\{\mathtt{wrlocal}\}, o, ar_o, v_o)$ $\mathsf{wrloc\ in}$ p. By definition, this means that either $o = p$ or $(ar_o, v_o) = (\mathtt{rdwr}, p)$. If $o = p$, we get $o = p \in \bar{o}$, by premise of (Frm Imm Cons). So suppose that $(ar_o, v_o) = (\mathtt{rdwr}, p)$. Then the reason for subtyping judgment (3.22) cannot be (Sub Share) and, thus, $\mathsf{owner}(o) = v_o$. So we have $\mathsf{owner}(o) = v_o = p$. But then $o \in \mathsf{inside}(\bar{o})$, because $p \in \bar{o}$ by premise of (Frm Imm Cons).

Case 4, (Red Call):

$$\frac{s = s', \mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})]\,\mathsf{in}\,p \quad \mathsf{ty}(o) = C\texttt{<}ar'_o, v'_o\texttt{>} \quad \mathsf{mbody}(C,m) = \texttt{<}\bar{y}\texttt{>}(\bar{x})(e)}{h :: s \rightarrow h :: s, e[\mathsf{self}(o, ar'_o, v'_o), \bar{y} \leftarrow \bar{u}, \bar{x} \leftarrow \bar{v}]\,\mathsf{in}\,o}$$

(4.1) $\sigma' \overset{\Delta}{=} \mathsf{self}(o, ar'_o, v'_o), \bar{y} \leftarrow \bar{u}, \bar{x} \leftarrow \bar{v}$ abbreviation
(4.2) $state = h :: s, e[\sigma']\,\mathsf{in}\,o$

By inverting judgment (4), we obtain:

(4.3) $\Gamma; \bar{o}' \vdash (s' : T) \leftarrow (\Gamma'; \bar{o} \vdash^{\kappa} T'\ \mathsf{in}\ p)$
(4.4) $\Gamma'; \bar{o} \vdash^{\kappa} (\mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})]\,\mathsf{in}\,p) : T'\ \mathsf{in}\ p$

By inverting judgment (4.4), we obtain:

(4.5) $\Gamma' \vdash^{\kappa} \mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})] : T' \ \mathsf{in}\ p, \mathsf{acc}(p)$
(4.6) $\mathsf{vals}(\mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})]) \cap \bar{o} \subseteq \{p\}$
(4.7) $\Gamma'; \bar{o} \vdash \mathsf{eatts}(T')\ \mathsf{ok\ in}\ p$
(4.8) $\bar{o} \subseteq \mathsf{imm}$
(4.9) $\mathsf{deanon}(T', p \notin \bar{o}) = T'$

By applying Lemma 13 to (4.5), we obtain a type U such that:

(4.10) $\Gamma', x : U \vdash^{\kappa} \mathscr{E}[x] : T' \ \mathsf{in}\ p, \mathsf{acc}(p)$
(4.11) $\Gamma' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U \ \mathsf{in}\ p, \mathsf{acc}(p)$
(4.12) The last rule of (4.11)'s type derivation is (Call).

An inspection of the possible reasons for (4.7) shows that $\texttt{anon} \notin \mathsf{eatts}(T')$ unless $p \in \bar{o}$. We can therefore assume that $\texttt{anon} \notin \mathsf{eatts}(U)$ unless $p \in \bar{o}$. (If $\texttt{anon}$ is in $\mathsf{eatts}(U)$ but not in $\mathsf{eatts}(T')$, we can remove $\texttt{anon}$ from $\mathsf{eatts}(U)$ without violating judgment(4.10), by Lemma 7, or judgment (4.11), by (Sub).)

(4.13) $\texttt{anon} \in \mathsf{eatts}(U) \Rightarrow p \in \bar{o}$

We spell out the last rule of (4.11)'s type derivation:

$$\frac{\begin{array}{l}\mathsf{mtype}(m, C_o) = \mathit{fm}\texttt{<}\bar{y}\texttt{>}\bar{ty} \rightarrow ea_m\, ty' \\ (\texttt{rdonly} \in ea_m)\ \text{or}\ (ar_o\ \mathsf{wrsafe\ in\ acc}(p)) \qquad \sigma = \mathsf{self}(o, ar_o, v_o), \bar{y} \leftarrow \bar{u} \\ ea = \bigcap \bar{ea}_{\bar{v}} \cap \bigcup(\ \{\texttt{anon}\} \cap (ea_m \cup ea_o),\ \{x\ \mathsf{as}\ \texttt{rdonly} \mid x \in ea_m\ \text{or}\ v_o, \bar{u} = \texttt{world}\}, \\ \quad \{\texttt{wrlocal} \mid (ea_m, o, ar_o, v_o)\ \mathsf{wrloc\ in}\ p\ \text{or}\ \texttt{rdonly} \in ea_m\ \text{or}\ v_o = \texttt{world}\}\ ) \\ \Gamma' \vdash o, \bar{v} : ea_o\, C_o\texttt{<}ar_o, v_o\texttt{>}, \bar{ea}_{\bar{v}}\, \bar{ty}[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p) \quad \Gamma' \vdash \bar{u} : \bullet \quad (ar_o = \texttt{rd}\ \text{or}\ \Gamma' \vdash v_o : \bullet)\end{array}}{\Gamma' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : ea\, ty'[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p)}$$

(4.14) $U = ea\, ty'[\sigma]$

We define:

(4.15) $U'' \triangleq \begin{cases} ea_m\, ty'[\sigma'] & \text{if } p = o \in \bar{o} \\ (ea_m - \texttt{anon})\, ty'[\sigma'] & \text{otherwise} \end{cases}$
(4.16) $ea' \triangleq \{\texttt{anon} \mid p \neq o\}$
(4.17) $U' \triangleq ea'\ \texttt{rdonly}\ \texttt{wrlocal}\, U''$

The following holds:

(4.18) $\Gamma, x : U' \vdash U' \preceq U$

We show statement (4.18): To this end, it suffices to show that $\mathsf{eatts}(U) \subseteq \mathsf{eatts}(U')$. To this end, it suffices to show that $\texttt{anon} \in \mathsf{eatts}(U)$ implies $\texttt{anon} \in \mathsf{eatts}(U')$. So suppose that $\texttt{anon} \in \mathsf{eatts}(U)$. Then $p \in \bar{o}$, by (4.13). Then either $\texttt{anon} \in \mathsf{eatts}(U'')$ or $p \neq o$, by definition of U''. In the former case, we have $\texttt{anon} \in \mathsf{eatts}(U'') \subseteq \mathsf{eatts}(U')$ and, in the latter case, we have $\texttt{anon} \in ea' \subseteq \mathsf{eatts}(U')$.

Now that we have established statement (4.18), we can apply type specialization (Lemma 5) to judgment (4.10):

(4.19) $\Gamma', x : U' \vdash^{\kappa} \mathscr{E}[x] : T' \ \mathsf{in}\ p, \mathsf{acc}(p)$

Moreover, we have:

(4.20) $o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : (\bar{o} \vdash p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} o)$ by (Call Frm Mth)
(4.21) $\Gamma' \vdash U'' : \mathsf{ok}$ follows from (4.11)

We now apply (Call Frm) to (4.4), (4.20), (4.19) and (4.21). We obtain:

(4.22) $\Gamma';\bar{o} \vdash ((\mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})]\,\mathsf{in}\,p) : T' \ \mathsf{in}\ p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} U'' \ \mathsf{in}\ o)$

We define:

(4.23) $\Gamma'' \triangleq (\emptyset, (\texttt{world}, \bar{u}, o, \mathsf{owner}(o) : \bullet), \Gamma'_{\mathsf{val}})$ abbreviation

By applying (Stk Call) to (4.3) and (4.22), we obtain:

(4.24) $\Gamma;\bar{o} \vdash (s', \mathscr{E}[o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})]\,\mathsf{in}\,p : T) \leftarrow (\Gamma'';\bar{o} \vdash^{\mathsf{call}} U'' \ \mathsf{in}\ o)$

We want to show the following:

(4.25) $\bar{o} \vdash \mathit{state} : T$ goal

We know that $(\Gamma;\bar{o} \vdash h : \mathsf{ok})$, by (3). By (State), (5) and (6), it therefore suffices to show the following:

(4.26) $\Gamma;\bar{o} \vdash s, e[\sigma']\,\mathsf{in}\,o : T$ subgoal towards (4.25)

By (Stk Push) and (4.24), it suffices to show the following:

(4.27) $\Gamma'';\bar{o} \vdash^{\mathsf{call}} e[\sigma']\,\mathsf{in}\,o : U'' \ \mathsf{in}\ o$ subgoal towards (4.26)

By (Frm) and (Mth Frm Body), we need to show the following four statements:

(4.28) $\Gamma'' \vdash e[\sigma'] : U'' \ \mathsf{in}\ o, \mathsf{acc}(o)$ subgoal towards (4.27)
(4.29) $\mathsf{vals}(e[\sigma']) \cap \bar{o} \subseteq \{o\}$ subgoal towards (4.27)
(4.30) $\Gamma'';\bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o$ subgoal towards (4.27)
(4.31) $\bar{o} \subseteq \mathsf{imm}$ subgoal towards (4.27)

Subgoal (4.31) is identical to fact (4.8); only subgoals (4.28), (4.29) and (4.30) remain open. We first derive some facts that are useful for several of these open proof goals. By construction, we have:

(4.32) $\Gamma''(o) = \texttt{rdonly}\ \texttt{wrlocal}\,\mathsf{ty}(o) = \texttt{rdonly}\ \texttt{wrlocal}\,C\texttt{<}ar'_o, v'_o\texttt{>}$
(4.33) $\sigma' = \mathsf{self}(o, ar'_o, v'_o), \bar{y}\leftarrow\bar{u}, \bar{x}\leftarrow\bar{v}$

By definition of $\mathsf{acc}(o)$ and $\mathsf{owner}(o)$, we have $\mathsf{acc}(o) = ar'_o$ and $\mathsf{owner}(o) = v'_o$. We can therefore restate (4.32) and (4.33) like this:

(4.34) $\Gamma''(o) = \texttt{rdonly}\ \texttt{wrlocal}\,C\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>}$
(4.35) $\sigma' = \mathsf{self}(o, \mathsf{acc}(o), \mathsf{owner}(o)), \bar{y}\leftarrow\bar{u}, \bar{x}\leftarrow\bar{v}$

By premise of the last rule of $(\Gamma' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U\ \mathsf{in}\ p, \mathsf{acc}(p))$'s derivation, as displayed above, we have $(\Gamma' \vdash o : ea_o\,C_o\texttt{<}ar_o, v_o\texttt{>}\ \mathsf{in}\ p, \mathsf{acc}(p))$. By inverting this judgment and using Lemma 6 in order to replace Γ' by Γ'', we obtain the following:

(4.36) $\Gamma'' \vdash C\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>} \preceq C_o\texttt{<}ar_o, v_o\texttt{>}$

Proof of goal (4.28)*:* We define:

(4.37) $ea_{\mathsf{rw}} \overset{\Delta}{=} \texttt{rdonly wrlocal}$ abbreviation
(4.38) $\Gamma_1 \overset{\Delta}{=} (\texttt{myaccess}, \texttt{world}, \texttt{myowner}, \texttt{this} : \bullet,$
$\texttt{this} : ea_{\mathsf{rw}}\, C\texttt{<myaccess,myowner>})$

Because $C <: C_o$, we have $(\Gamma_1 \vdash \mathsf{mtype}(C,m) \preceq \mathsf{mtype}(C_o,m))$, by premise of (Mth Dcl). By definition of method subtyping, this means that there are fm', V such that:

(4.39) $\mathsf{mtype}(C,m) = fm'\, \texttt{<}\bar{y}\texttt{>}\bar{ty} \to V$
(4.40) $\Gamma_1 \vdash V \preceq ea_m\, ty'$

By Lemma 3(d), we can swap the class parameter in subtyping judgment (4.36) and obtain $(\Gamma'' \vdash C\texttt{<}ar_o, v_o\texttt{>} \preceq C_o\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>})$. We then apply Lemma 3(e) to obtain:

(4.41) $\Gamma'' \vdash \bar{ty}[\sigma] \preceq \bar{ty}[\sigma']$

Because the underlying class table is well-typed, we know that m's declaration in C is well-typed. By premise of (Mth Dcl), there are ar_1, σ_1 such that the following statements hold:

(4.42) $ea_{\mathsf{arw}} \overset{\Delta}{=} \texttt{anon rdonly wrlocal}$ abbreviation
(4.43) $\Gamma_1[\sigma_1], \bar{y} : \bullet, \bar{x} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma_1] \vdash e[\sigma_1] : V[\sigma_1]$ in $\texttt{this}, ar$
(4.44) $ar_1 = \texttt{myaccess}$ or $(\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea_m = \emptyset,\ ar_1 = \texttt{rdwr})$
(4.45) $\sigma_1 = (\texttt{myaccess} \leftarrow ar_1)$

We define σ'_1 as the restriction of σ' to the singleton domain $\{\texttt{myaccess}\}$:

(4.46) $\sigma'_1 \overset{\Delta}{=} (\texttt{myaccess} \leftarrow \mathsf{acc}(o)) = \sigma'|\texttt{myaccess}$ abbreviation

We will now show the following statement:

(4.47) $\Gamma_1[\sigma'_1], \bar{y} : \bullet, \bar{x} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma'_1] \vdash e[\sigma'_1] : V[\sigma'_1]$ in $\texttt{this}, \mathsf{acc}(o)$

We show statement (4.47): Suppose first that $\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea_m \neq \emptyset$. Then σ_1 is the identity, by (4.44)/(4.45). We obtain (4.47) by applying substitutivity (Lemma 11) to judgment (4.43). Suppose now that $\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea_m = \emptyset$. Then $\mathsf{acc}(p) = \texttt{rdwr}$, by inspection of the possible reasons for (4.7). Then $ar_o = \texttt{rdwr}$, because ar_o wrsafe in $\mathsf{acc}(p)$. Applying Lemma 14 to $(\Gamma'' \vdash C_o\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>} \preceq C\texttt{<}ar_o, v_o\texttt{>})$, we obtain $\mathsf{acc}(o) = \texttt{rdwr}$. Therefore, $\sigma'_1 = \sigma_1$, and judgment (4.47) is identical to judgment (4.43).

We have now completed the proof of statement (4.47) and resume the proof of goal (4.28): By expanding the definition of Γ_1 and weakening judgment (4.47), we obtain the following judgment:

(4.48) $\Gamma'', \texttt{myowner}, \bar{y}, \texttt{this} : \bullet, \texttt{this} : ea_{\mathsf{rw}}\, C\texttt{<}\mathsf{acc}(o)\texttt{,myowner>}, \bar{x} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma'_1]$
$\vdash e[\sigma'_1] : V[\sigma'_1]$ in $\texttt{this}, \mathsf{acc}(o)$

On the other hand, we have the following judgments:

(4.49) $\sigma'_2 \overset{\Delta}{=} \sigma'|\texttt{myowner}, \texttt{myaccess}, \bar{y}$ abbreviation
(4.50) $\Gamma'' \vdash \mathsf{owner}(o), \bar{u} : \bullet$
(4.51) $\Gamma'' \vdash o : ea_{\mathsf{rw}}\, C\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>}$ in $o, \mathsf{acc}(o)$
(4.52) $\Gamma'' \vdash \bar{v} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma] \preceq ea_{\mathsf{arw}}\, \bar{ty}[\sigma'] = ea_{\mathsf{arw}}\, \bar{ty}[\sigma'_2]$ in $o, \mathsf{acc}(o)$

Judgment (4.50) holds by construction of $\mathit{\Gamma}''$. Judgment (4.51) holds by (Obj). Judgment (4.52) follows from ($\mathit{\Gamma}' \vdash \bar{v} : \bar{ty}[\sigma]$ in $p, \mathsf{acc}(p)$) by subtyping judgment (4.41) and because we can replace $\mathit{\Gamma}', p, \mathsf{acc}(p)$ by $\mathit{\Gamma}'', o, \mathsf{acc}(o)$, by Lemma 6 and Lemma 8.

We now apply the customized substitutivity lemma 34 to judgments (4.50), (4.51), (4.52) and (4.47) to obtain the following:

(4.53) $\mathit{\Gamma}'' \vdash e[\sigma'] : \mathsf{deanon}(V[\sigma'], o \in \bar{v})$ in $o, \mathsf{acc}(o)$

We will have established goal (4.28), if we can show the following:

(4.54) $\mathit{\Gamma}'' \vdash \mathsf{deanon}(V[\sigma'], o \in \bar{v}) \preceq U''$ subgoal towards (4.28)

We expand the definition of U'':

(4.55) $\mathit{\Gamma}'' \vdash \mathsf{deanon}(V[\sigma'], o \in \bar{v}) \preceq \begin{cases} ea_m\, ty'[\sigma'] & \text{if } p = o \in \bar{o} \\ (ea_m - \mathtt{anon})\, ty'[\sigma'] & \text{otherwise} \end{cases}$ goal

By applying weakening and substitutivity to subtyping judgment (4.40), we get ($\mathit{\Gamma}'' \vdash V[\sigma'] \preceq ea_m\, ty'[\sigma']$). Moreover, we have ($\mathit{\Gamma}'' \vdash ty'[\sigma'] \preceq ty'[\sigma]$), by Lemma 3(e). So, by transitivity, ($\mathit{\Gamma}'' \vdash V[\sigma'] \preceq ea_m\, ty[\sigma]$). It now suffices to show the following:

(4.56) $p = o \in \bar{o} \Rightarrow o \notin \bar{v}$ subgoal towards (4.28)

To show (4.56), let $p = o \in \bar{o}$. By inspecting the possible reasons for (4.7), we see that $p \in \bar{o}$ is only possible if $\mathtt{anon} \in \mathsf{eatts}(T')$. Then it must also be the case that $\mathtt{anon} \in \mathsf{eatts}(U)$ (because subexpressions of anonymous expressions are anonymous). Then $\mathtt{anon} \in \bigcap \bar{ea}_{\bar{v}}$, by premise of the last rule of ($\mathit{\Gamma}' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U$ in $p, \mathsf{acc}(p)$)'s derivation, as displayed above. Then $p \notin \bar{v}$, by inverting judgment ($\mathit{\Gamma}' \vdash \bar{v} : \bar{ea}_{\bar{v}}\, \bar{ty}[\sigma]$ in $p, \mathsf{acc}(p)$) from the premise of the same rule. Then $o \notin \bar{v}$, because $o = p$.

Proof of goal (4.29)*:* By well-typedness of the underlying class table, m's body e in C is well-typed in an environment $\mathit{\Gamma}$ such that $\mathsf{dom}(\mathit{\Gamma}_{\mathsf{val}}) = \{\mathtt{this}, \bar{x}\}$. Then $\mathsf{vals}(e) \subseteq \{\mathtt{null}, \mathtt{this}, \bar{x}\}$, by Lemma 18(a). Then $\mathsf{vals}(e[\sigma']) \subseteq \{\mathtt{null}, o, \bar{v}\}$, by Lemma 18(b). We need to show the following:

(4.57) $(q \in \{o, \bar{v}\} \cap \bar{o}) \Rightarrow (q = o)$ subgoal towards (4.29)

Let $q \in \{o, \bar{v}\} \cap \bar{o}$. Then $q = p$, by (4.6). An inspection of the possible reasons for (4.7) shows that $p \in \bar{o}$ is only possible if $\mathtt{anon} \in \mathsf{eatts}(T')$. Then also $\mathtt{anon} \in \mathsf{eatts}(U)$, because subexpressions of anonymous expressions are again anonymous. Then all arguments $\bar{v}$ for $o.m\texttt{<}\bar{u}\texttt{>}(\bar{v})$ are anonymous in p, by premise of the last rule of ($\mathit{\Gamma}' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U$ in $p, \mathsf{acc}(p)$)'s derivation, as displayed above. That means $q = p \notin \bar{v}$. But then $q = o$, because $q \in \{o, \bar{v}\}$ by assumption.

Proof of goal (4.30)*:* We distinguish cases by the possible reasons for (4.7).

Case 4.1, (Frm Rdonly): In this case, $p \notin \bar{o}$ and $\mathtt{rdonly} \in \mathsf{eatts}(T')$. By (4.6) and because $p \notin \bar{o}$, we obtain:

(4.1.1) $o \notin \bar{o}$

By definition of U'' and because $p \notin \bar{o}$, we have:

(4.1.2) $\mathtt{anon} \notin \mathsf{eatts}(U'')$

From $\mathtt{rdonly} \in \mathsf{eatts}(T')$ we obtain $\mathtt{rdonly} \in \mathsf{eatts}(U)$, because subexpressions of read-only expressions are read-only. We now inspect the last rule of ($\mathit{\Gamma}' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U$ in $p, \mathsf{acc}(p)$)'s derivation, as displayed above. Because $\mathtt{rdonly} \in \mathsf{eatts}(U)$, there are two possibilities: Either $\mathtt{rdonly} \in ea_m$ or $v_o, \bar{u} = \mathtt{world}$.

Case 4.1.1, $\texttt{rdonly} \in ea_m$: Then $(\Gamma'';\bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$, by (Frm Rdonly).

Case 4.1.2, $\texttt{rdonly} \notin ea_m$, $v_o, \bar{u} = \texttt{world}$: Because $\texttt{rdonly} \notin ea_m$ and all methods on immutable objects are read-only, we have:

(4.1.2.1) $o \notin \mathsf{imm}$

By inspection of the possible last rules for subtyping judgment (4.36) and because $v_o = \texttt{world}$, we obtain $\mathsf{owner}(o) = \texttt{world}$, We also have $\bar{u} = \texttt{world}$, by assumption. Therefore, $\mathsf{dom}(\Gamma''_{\mathsf{own}}) = \{o, \texttt{world}\}$, by definition of Γ''. Because $\mathsf{owner}(o) = \texttt{world}$ and $o \notin \mathsf{imm}$, we obtain:

(4.1.2.2) $o \in \mathsf{outside}(\mathsf{imm})$
(4.1.2.3) $\mathsf{dom}(\Gamma''_{\mathsf{own}}) \subseteq \{o, \texttt{world}\} \subseteq \mathsf{outside}(\mathsf{imm})$

By Lemma 14 and because $\mathsf{owner}(o) = \texttt{world}$, we have:

(4.1.2.4) $\mathsf{acc}(o) = \texttt{rdwr}$

We can now apply (Frm Rdwr) to these facts in order to obtain $(\Gamma'';\bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.2, (Frm Rdwr): In this case, $\mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$ and $p \notin \bar{o}, \mathsf{imm}$ and $p \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$. By (4.6) and because $p \notin \bar{o}$, we obtain:

(4.2.1) $o \notin \bar{o}$

By definition of U'' and because $p \notin \bar{o}$, we have:

(4.2.2) $\mathsf{anon} \notin \mathsf{eatts}(U'')$

Case 4.2.1, $\texttt{rdonly} \in ea_m$: Then $(\Gamma'';\bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$, by (Frm Rdonly).

Case 4.2.2, $\texttt{rdonly} \notin ea_m$: Because $\texttt{rdonly} \notin ea_m$ and all methods on immutable objects are read-only, we have:

(4.2.2.1) $o \notin \mathsf{imm}$

In the following, we will inspect the premises of (Call)—the last rule of $(\Gamma' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U\ \mathsf{in}\ p, \mathsf{acc}(p))$'s derivation, as displayed above. By premise of (Frm Rdwr), we know that either $\mathsf{acc}(p) = \texttt{rdwr}$ or $\texttt{wrlocal} \in \mathsf{eatts}(T')$, thus, $\texttt{wrlocal} \in \mathsf{eatts}(U)$. In case $\texttt{wrlocal} \in \mathsf{eatts}(U)$, we know that either $v_o = \texttt{world}$ or $(ea_m, o, ar_o, v_o)\ \mathsf{wrloc\ in}\ p$, by premise of (Call). By definition, the statement $(ea_m, o, ar_o, v_o)\ \mathsf{wrloc\ in}\ p$ holds whenever $(\texttt{wrlocal} \in ea_m, o = p)$ or $(ar_o, v_o) = (\texttt{rdwr}, p)$. So we have to consider the following four cases: $\mathsf{acc}(p) = \texttt{rdwr}$ or $v_o = \texttt{world}$ or $(\texttt{wrlocal} \in ea_m, o = p)$ or $(ar_o, v_o) = (\texttt{rdwr}, p)$.

Case 4.2.2.1, $\mathsf{acc}(p) = \texttt{rdwr}$: By premise of (Call), we get $ar_o\ \mathsf{wrsafe\ in}\ \mathsf{acc}(p)$. Because we also have $\mathsf{acc}(p) = \texttt{rdwr}$, we obtain $ar_o = \texttt{rdwr}$ by definition of wrsafe. We also have $(\Gamma' \vdash v_o : \bullet)$, by premise of (Call) and because $\mathsf{acc}(p) = \texttt{rdwr}$. Moreover $(\Gamma' \vdash \bar{u} : \bullet)$, by premise of (Call). By expanding the definition of $(\Gamma' \vdash v_o, \bar{u} : \bullet)$, we obtain $v_o, \bar{u} \in \mathsf{dom}(\Gamma'_{\mathsf{own}})$. By inspection of the possible last rules for subtyping judgment (4.36) and because $ar_o = \texttt{rdwr}$, we obtain $\mathsf{owner}(o) = v_o$. We know that $\mathsf{owner}(o) = v_o \in \mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$. Moreover, we know that $o \notin \mathsf{imm}$. It follows that:

(4.2.2.1.1) $o \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$
(4.2.2.1.2) $\mathsf{dom}(\Gamma''_{\mathsf{own}}) = \{\mathtt{world}, o, \mathsf{owner}(o), \bar{u}\} \subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$

Applying Lemma 14 to $\Gamma' \vdash C\texttt{<}\mathsf{acc}(o), \mathsf{owner}(o)\texttt{>} \preceq C_o\texttt{<}ar_o, \mathsf{owner}(o)\texttt{>}$, we obtain:

(4.2.2.1.3) $\mathsf{acc}(o) = \mathtt{rdwr}$

We can now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.2.2.2, $v_o = \mathtt{world}$: Then $\mathsf{owner}(o) = \mathtt{world}$, by inspection of the possible reasons for subtyping judgment (4.36). Because we already know that $o \notin \mathsf{imm}$, we obtain:

(4.2.2.2.1) $o \in \mathsf{outside}(\mathsf{imm})$
(4.2.2.2.2) $\mathsf{dom}(\Gamma''_{\mathsf{own}}) = \{\mathtt{world}, o, \mathsf{owner}(o), \bar{u}\}$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{dom}(\Gamma'_{\mathsf{own}})$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$

By Lemma 14 and because $\mathsf{owner}(o) = \mathtt{world}$, we obtain:

(4.2.2.2.3) $\mathsf{acc}(o) = \mathtt{rdwr}$

We can now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.2.2.3, $\mathtt{wrlocal} \in ea_m$, $o = p$: Then $o = p \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$. Because we already know that $o \notin \bar{o}$, we also have $\mathsf{owner}(o) \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$.

(4.2.2.3.1) $\mathsf{dom}(\Gamma''_{\mathsf{own}}) = \{\mathtt{world}, o, \mathsf{owner}(o), \bar{u}\}$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o}) \cup \mathsf{dom}(\Gamma'_{\mathsf{own}})$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$

We can now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.2.2.4, $(ar_o, v_o) = (\mathtt{rdwr}, p)$: By inspection of the last rule of subtyping judgment (4.36), we get that $\mathsf{owner}(o) = p$. Then $\mathsf{owner}(o) = p \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$. Because we already know that $o \notin \mathsf{imm}$, we also get:

(4.2.2.4.1) $o \in \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$
(4.2.2.4.2) $\mathsf{dom}(\Gamma''_{\mathsf{own}}) = \{\mathtt{world}, o, \mathsf{owner}(o), \bar{u}\}$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o}) \cup \mathsf{dom}(\Gamma'_{\mathsf{own}})$
$\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})$

Using Lemma 14, we obtain:

(4.2.2.4.3) $\mathsf{acc}(o) = \mathtt{rdwr}$

We can now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.3, (Frm Imm Cons): We have $\mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \{p, \mathtt{world}\}$ and $p \in \bar{o}$ and $\mathsf{owner}(p) = \mathtt{world}$ and $\mathtt{anon} \in \mathsf{eatts}(T')$ and $\{\mathtt{rdonly}, \mathtt{wrlocal}\} \cap \mathsf{eatts}(T') \neq \emptyset$. Then also $\mathtt{anon} \in \mathsf{eatts}(U)$ and $\{\mathtt{rdonly}, \mathtt{wrlocal}\} \cap \mathsf{eatts}(U) \neq \emptyset$, because if an expression has these attributes then each subexpression (except perhaps values) also have these attributes.

Case 4.3.1, $\mathtt{rdonly} \in ea_m$, $o \neq p$: By (4.6) and because $o \neq p$, we obtain $o \notin \bar{o}$. By definition of U'' and because $o \neq p$, we obtain $\mathtt{anon} \notin \mathsf{eatts}(U'')$. We can therefore apply (Frm Rdonly) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ o)$.

Case 4.3.2, $\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea_m \neq \emptyset$, $o = p$: We inspect the premises of (Call)—the last rule of $(\Gamma' \vdash o.m\texttt{<}\bar{u}\texttt{>}(\bar{v}) : U \text{ in } p, \mathsf{acc}(p))$'s derivation, as displayed above. By premise of (Call) and because $o = p$, we have $\texttt{anon} \in ea_m$. Because $o = p \in \bar{o} \subseteq \mathsf{imm}$ and all methods on immutable objects are read-only, we obtain $\texttt{rdonly} \in ea_m$. By $\texttt{anon}, \texttt{rdonly} \in ea_m$ and definition of U'', we obtain:

(4.3.2.1) $\texttt{anon}, \texttt{rdonly} \in \mathsf{eatts}(U'')$

Because $o = p$, $\mathsf{owner}(p) = \texttt{world}$, $\bar{u} \in \mathsf{dom}(\Gamma'_{\mathsf{own}})$ and $\mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \{p, \texttt{world}\}$, we obtain:

$$\begin{aligned}\mathsf{dom}(\Gamma''_{\mathsf{own}}) &= \{\texttt{world}, o, \mathsf{owner}(o), \bar{u}\} \\ &\subseteq \{\texttt{world}, p, \mathsf{owner}(p)\} \cup \mathsf{dom}(\Gamma'_{\mathsf{own}}) \subseteq \{p, \texttt{world}\}\end{aligned}$$

We now apply (Frm Imm Cons) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'') \text{ ok in } o)$.

Case 4.3.3, $v_o = \texttt{world}$, $\texttt{rdonly} \notin ea_m$: Because $p \in \bar{o} \subseteq \mathsf{imm}$ and methods on immutable objects are read-only, we have $o \neq p$. Then $o \notin \bar{o}$, by (4.6). Moreover, $o \notin \mathsf{imm}$, because $\texttt{rdonly} \notin ea_m$. Because $v_o = \texttt{world}$, we have $\mathsf{owner}(o) = \texttt{world}$, by inspection of the possible last rules for subtyping judgment (4.36). Because $\mathsf{owner}(o) = \texttt{world}$ and $o \notin \mathsf{imm}$, it follows that $o \in \mathsf{outside}(\mathsf{imm})$.

$$\begin{aligned}\mathsf{dom}(\Gamma''_{\mathsf{own}}) &= \{\texttt{world}, o, \mathsf{owner}(o), \bar{u}\} \\ &\subseteq \{\texttt{world}, o, \mathsf{owner}(o)\} \cup \mathsf{dom}(\Gamma'_{\mathsf{own}}) \\ &\subseteq \mathsf{outside}(\mathsf{imm}) \cup \{p, \texttt{world}\} \\ &\subseteq \mathsf{outside}(\mathsf{imm}) \cup \{p\} \\ &\subseteq \mathsf{outside}(\mathsf{imm}) \cup \bar{o} \\ &\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})\end{aligned}$$

Because $\mathsf{owner}(o) = \texttt{world}$, we have $\mathsf{acc}(o) = \texttt{rdwr}$, by Lemma 14. Because $o \neq p$, we have $\texttt{anon} \notin \mathsf{eatts}(U'')$, by definition of U''. We can now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'') \text{ ok in } o)$.

Case 4.3.4, $(ar_o, v_o) = (\texttt{rdwr}, p)$, $\texttt{rdonly} \notin ea_m$: Because $p \in \bar{o} \subseteq \mathsf{imm}$ and methods on immutable objects are read-only, we have $o \neq p$. Then $o \notin \bar{o}$, by (4.6). Moreover, $o \notin \mathsf{imm}$, because $\texttt{rdonly} \notin ea_m$. Because $ar_o = \texttt{rdwr}$, we have $\mathsf{owner}(o) = p$, by inspection of the possible last rules for subtyping judgment (4.36). Because $\mathsf{owner}(o) = p \in \bar{o}$, we have that $o \in \mathsf{inside}(\bar{o})$.

$$\begin{aligned}\mathsf{dom}(\Gamma''_{\mathsf{own}}) &= \{\texttt{world}, o, \mathsf{owner}(o), \bar{u}\} \\ &\subseteq \{\texttt{world}, o, \mathsf{owner}(o)\} \cup \mathsf{dom}(\Gamma'_{\mathsf{own}}) \\ &\subseteq \{\texttt{world}, o, \mathsf{owner}(o)\} \cup \{p, \texttt{world}\} \\ &\subseteq \mathsf{inside}(\bar{o}) \cup \{\texttt{world}\} \\ &\subseteq \mathsf{outside}(\mathsf{imm}) \cup \mathsf{inside}(\bar{o})\end{aligned}$$

Then $\mathsf{acc}(o) = \texttt{rdwr}$, by Lemma 14. Because $o \neq p$, we have $\texttt{anon} \notin \mathsf{eatts}(U'')$, by definition of U''. We now apply (Frm Rdwr) to obtain $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'') \text{ ok in } o)$.

Case 5, (Red New):

$$\frac{s = s', \mathscr{E}[\texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v})]\,\mathsf{in}\,p \quad o \notin \mathsf{dom}(h) \quad \mathsf{ty}(o) = C\texttt{<}ar,w\texttt{>} \quad \mathsf{fd}(C) = \bar{ty}\,\bar{f}}{h :: s \,\rightarrow\, h, o\{\bar{f} = \texttt{null}\} :: s, C.k(\bar{v})\,;o\,\mathsf{in}\,o}$$

(5.1) $\mathit{state} = h, o\{\bar{f} = \texttt{null}\} :: s, C.k(\bar{v})\,;o\,\mathsf{in}\,o$

By definition of $\mathsf{owner}(o)$ and $\mathsf{acc}(o)$, we have:

(5.2) $(ar, w) = (\mathsf{acc}(o), \mathsf{owner}(o))$

We define:

(5.3) $ea_{\mathsf{rw}} \overset{\Delta}{=} \texttt{rdonly}\;\texttt{wrlocal}$
(5.4) $\Gamma_o \overset{\Delta}{=} (\Gamma, o : ea_{\mathsf{rw}}\,C\texttt{<}ar,w\texttt{>})$

By applying Lemma 30 to judgment (3), we obtain:

(5.5) $\Gamma_o; \bar{o}, o \vdash h, o\{\bar{f} = \texttt{null}\} : \mathsf{ok}$

By inverting judgment (4), we obtain:

(5.6) $\Gamma; \bar{o}' \vdash (s' : T) \leftarrow (\Gamma'; \bar{o} \vdash^{\kappa} T'\ \mathsf{in}\ p)$
(5.7) $\Gamma'; \bar{o} \vdash^{\kappa} (\mathscr{E}[\texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v})]\,\mathsf{in}\,p) : T'\ \mathsf{in}\ p$

By inverting judgment (4.4), we obtain:

(5.8) $\Gamma' \vdash^{\kappa} \mathscr{E}[\texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v})] : T'\ \mathsf{in}\ p, \mathsf{acc}(p)$
(5.9) $\mathsf{vals}(\mathscr{E}[\texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v})]) \cap \bar{o} \subseteq \{p\}$
(5.10) $\Gamma'; \bar{o} \vdash \mathsf{eatts}(T')\ \mathsf{ok\ in}\ p$
(5.11) $\bar{o} \subseteq \mathsf{imm}$
(5.12) $\mathsf{deanon}(T', p \notin \bar{o}) = T'$

By applying Lemma 13 to (5.8), we obtain a type U such that:

(5.13) $\Gamma', x : U \vdash^{\kappa} \mathscr{E}[x] : T'\ \mathsf{in}\ p, \mathsf{acc}(p)$
(5.14) $\Gamma' \vdash \texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v}) : U\ \mathsf{in}\ p, \mathsf{acc}(p)$
(5.15) The last rule of (5.14)'s type derivation is (New).

We spell out the last rule of (5.14)'s type derivation:

$$\frac{\begin{array}{l}\mathsf{ctype}(C.k) = \bar{ty} \rightarrow ea_k\,\texttt{void} \quad (ar = \texttt{rdwr})\ \text{or}\ (\texttt{wrlocal}, \texttt{anon} \in ea_k) \\ ea = \bigcap(\{\texttt{rdonly} \mid w = \texttt{world}\} \cup \{\texttt{wrlocal}, \texttt{anon}\}, \bar{ea}_{\bar{v}}) \quad \Gamma \vdash ar, w : \mathsf{ok}, \bullet \\ \Gamma' \vdash \bar{v} : \bar{ea}_{\bar{v}}\,\bar{ty}[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p) \quad \sigma = \mathsf{self}(\texttt{null}, ar, w) \quad C\texttt{<}ar,w\texttt{>}\ \mathsf{generative}\end{array}}{\Gamma' \vdash \texttt{new}\,C\texttt{<}ar,w\texttt{>}.k(\bar{v}) : ea\,C\texttt{<}ar,w\texttt{>}\ \mathsf{in}\ p, \mathsf{acc}(p)}$$

(5.16) $U = ea\,C\texttt{<}ar,w\texttt{>}$

We define:

(5.17) $U'' \overset{\Delta}{=} \begin{cases} ea_k\,C\texttt{<}ar,w\texttt{>} & \text{if } \texttt{immutable} \in \mathsf{atts}(C) \\ (ea_k - \texttt{anon})\,C\texttt{<}ar,w\texttt{>} & \text{otherwise} \end{cases}$
(5.18) $ea_{\mathsf{arw}} \overset{\Delta}{=} \texttt{anon}\;\texttt{rdonly}\;\texttt{wrlocal}$
(5.19) $U' \overset{\Delta}{=} ea_{\mathsf{arw}}\,U''$

(5.20) $\bar{o}' \triangleq \begin{cases} (\bar{o}, o) & \text{if } \texttt{immutable} \in \mathsf{atts}(C) \\ \bar{o} & \text{otherwise} \end{cases}$

The following holds because $\mathsf{eatts}(U) \subseteq \{\texttt{anon}, \texttt{rdonly}, \texttt{wrlocal}\} = \mathsf{eatts}(U')$:

(5.21) $\Gamma, x : U' \vdash U' \preceq U$

We apply type specialization (Lemma 5) to judgment (5.13):

(5.22) $\Gamma', x : U' \vdash^{\kappa} \mathscr{E}[x] : T' \text{ in } p, \mathsf{acc}(p)$

Moreover, we have:

(5.23) $\texttt{new}C\texttt{<}ar\texttt{,}w\texttt{>}.k(\bar{v}) : (\bar{o} \vdash p) \leftarrow (\bar{o}' \vdash^{\mathsf{new}} o)$ by (Call Frm Mth)
(5.24) $\Gamma' \vdash U'' : \mathsf{ok}$ follows from (5.14)

We now apply (Call Frm) to (5.7), (5.23), (5.22) and (5.24). We obtain:

(5.25) $\Gamma'; \bar{o} \vdash ((\mathscr{E}[\texttt{new}C\texttt{<}ar\texttt{,}w\texttt{>}.k(\bar{v})] \,\mathsf{in}\, p) : T' \text{ in } p) \leftarrow (\bar{o}' \vdash^{\mathsf{new}} U'' \text{ in } o)$

We define:

(5.26) $\Gamma'' \triangleq (\emptyset, (\texttt{world}, o, w : \bullet), \Gamma'_{\mathsf{val}})$

By applying (Stk Call) to (5.6) and (5.25), we obtain:

(5.27) $\Gamma; \bar{o} \vdash (s', \mathscr{E}[\texttt{new}C\texttt{<}ar\texttt{,}w\texttt{>}.k(\bar{v})] \,\mathsf{in}\, p : T) \leftarrow (\Gamma''; \bar{o}' \vdash^{\mathsf{new}} U'' \text{ in } o)$

We want to show the following:

(5.28) $\bar{o}' \vdash \mathit{state} : T$ goal

We know that $(\Gamma_o; \bar{o}' \vdash h, o\{\bar{f} = \texttt{null}\} : \mathsf{ok})$ by (5.5) and Lemma 29. By (State), (5) and (6), it therefore suffices to show the following:

(5.29) $\Gamma_o; \bar{o}' \vdash s, (C.k(\bar{v})\,; o \,\mathsf{in}\, o) : T$ subgoal towards (5.28)

By (Stk Push), (5.27) and Lemma 30, it suffices to show the following:

(5.30) $\Gamma''_o \triangleq (\Gamma'', o : ea_{\mathsf{rw}} C\texttt{<}ar, w\texttt{>})$
(5.31) $\Gamma''_o; \bar{o}' \vdash^{\mathsf{new}} (C.k(\bar{v})\,; o \,\mathsf{in}\, o) : U'' \text{ in } o$ subgoal towards (5.29)

By (Frm), we need to show the following four statements:

(5.32) $\Gamma''_o \vdash^{\mathsf{new}} C.k(\bar{v})\,; o : U'' \text{ in } o, \mathsf{acc}(o)$ subgoal towards (5.31)
(5.33) $\mathsf{vals}(C.k(\bar{v})\,; o) \cap \bar{o}' \subseteq \{o\}$ subgoal towards (5.31)
(5.34) $\Gamma''_o; \bar{o}' \vdash \mathsf{eatts}(U'') \text{ ok in } o$ subgoal towards (5.31)
(5.35) $\bar{o}' \subseteq \mathsf{imm}$ subgoal towards (5.31)

Subgoal (5.35) follows from $\bar{o} \subseteq \mathsf{imm}$, see (5.11), by definition of $\bar{o}'$. So only subgoals (5.32), (5.33) and (5.34) remain open.

Proof of goal (5.32)*:* By (New Frm Body), it suffices to show the following statements:

(5.36) $\Gamma''_o \vdash o : ea_{\mathsf{rw}}\, C\text{<}ar,w\text{>} \preceq (ea_k - \texttt{anon})\, C\text{<}ar,w\text{>}$ in $o, \mathsf{acc}(o)$
(5.37) $\Gamma''_o \vdash C.k(\bar{v}) : ea_k\, \texttt{void}$ in $o, \mathsf{acc}(o)$ subgoal towards (5.32)

Statement (5.36) holds by (Obj) and (Sub). So we need to show (5.32): By premise of the last rule of ($\Gamma' \vdash \texttt{new}C\text{<}ar,w\text{>}.k(\bar{v}) : U$ in $p, \mathsf{acc}(p)$)'s type derivation, as displayed above, we have:

(5.38) $\mathsf{ctype}(C.k) = \bar{ty} \to ea_k\, \texttt{void}$
(5.39) $\sigma = \mathsf{self}(\texttt{null}, ar, w)$
(5.40) $\Gamma' \vdash \bar{v} : \bar{ty}[\sigma]$ in $p, \mathsf{acc}(p)$

We define:

(5.41) $\sigma' \triangleq \mathsf{self}(o, ar, w)$

The parameter types of well-typed constructors do not contain occurrences of `this`, by (Cons Dcl). Therefore, ($\Gamma' \vdash \bar{v} : \bar{ty}[\sigma']$ in $p, \mathsf{acc}(p)$). Because $ea_{\mathsf{rw}} \subseteq \mathsf{eatts}(\Gamma'_{\mathsf{val}}(o))$ for all o in $\mathsf{dom}(\Gamma_{\mathsf{val}})'$, we obtain ($\Gamma' \vdash \bar{v} : ea_{\mathsf{rw}}\, \bar{ty}[\sigma']$ in $p, \mathsf{acc}(p)$). We then apply Lemmas 6 and 8 to replace $\Gamma', p, \mathsf{acc}(p)$ by $\Gamma'', o, \mathsf{acc}(o)$ and weakening to replace Γ'' by Γ''_o. This way, we obtain ($\Gamma''_o \vdash \bar{v} : ea_{\mathsf{rw}}\, \bar{ty}[\sigma']$ in $o, \mathsf{acc}(o)$). Because $\bar{v} \subseteq \mathsf{dom}(\Gamma')$, by (5.40), but $o \notin \mathsf{dom}(\Gamma')$, we have $\bar{v} \cap \{o\} = \emptyset$. That is $\bar{v}$ are anonymous in o. We obtain:

(5.42) $\Gamma''_o \vdash \bar{v} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma']$ in $o, \mathsf{acc}(o)$

We now apply to (Cons) to (5.38), (5.41), and (5.42), obtaining our goal (5.37).

Proof of goal (5.33)*:* It suffices to show the following:

(5.43) $\{\bar{v}\} \cap \bar{o} = \emptyset$ subgoal towards (5.33)

Suppose, towards a contradiction, that $q \in \{\bar{v}\} \cap \bar{o}$. Then $q = p$, by (5.10). An inspection of the possible reasons for (5.11) shows that $p \in \bar{o}$ is only possible if $\texttt{anon} \in \mathsf{eatts}(T')$. Then arguments $\bar{v}$ for $\texttt{new}C\text{<}ar,w\text{>}.k(\bar{v})$ are anonymous in p. That means $p \notin \bar{v}$. Then $q \notin \bar{v}$, because $q = p$. This contradicts our assumption.

Proof of goal (5.34)*:*

Case 5.1, $\texttt{immutable} \in \mathsf{atts}(C)$: It is straightforward to check that ($\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')$ ok in o), by (Frm Imm Cons).

We now distinguish cases by the possible reasons for (4.7):

Case 5.2, (Frm Rdwr), $\texttt{immutable} \notin \mathsf{atts}(C)$: It is straightforward to check that ($\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')$ ok in o), by (Frm Rdwr).

Case 5.3, (Frm Rdonly), $\texttt{immutable} \notin \mathsf{atts}(C)$: It is straightforward to check that ($\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')$ ok in o), by (Frm Rdwr).

Case 5.4, (Frm Imm Cons), $\texttt{immutable} \notin \mathsf{atts}(C)$: It is straightforward to check that ($\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')$ ok in o), by (Frm Rdwr).

Case 6, (Red Cons):

$$\frac{s = s', \mathscr{E}[C.k(\bar{v})]\,\mathsf{in}\,p \quad \mathsf{cbody}(C.k) = (\bar{x})(e) \quad \mathsf{ty}(p) = D\texttt{<}ar,w\texttt{>}}{h :: s \;\rightarrow\; h :: s, e[\mathsf{self}(p,ar,w), \bar{x}\leftarrow\bar{v}]\,\mathsf{in}\,p}$$

(6.1) $\sigma' \stackrel{\Delta}{=} \mathsf{self}(p,ar,w), \bar{x}\leftarrow\bar{v}$ abbreviation
(6.2) $state = h :: s, e[\sigma']\,\mathsf{in}\,o$

By inverting judgment (4), we obtain:

(6.3) $\Gamma;\bar{o}' \vdash (s' : T) \leftarrow (\Gamma';\bar{o} \vdash^{\kappa} T' \text{ in } p)$
(6.4) $\Gamma';\bar{o} \vdash^{\kappa} (\mathscr{E}[C.k(\bar{v})]\,\mathsf{in}\,p) : T' \text{ in } p$

By inverting judgment (6.4), we obtain:

(6.5) $\Gamma' \vdash^{\kappa} \mathscr{E}[C.k(\bar{v})] : T' \text{ in } p, \mathsf{acc}(p)$
(6.6) $\mathsf{vals}(\mathscr{E}[C.k(\bar{v})]) \cap \bar{o} \subseteq \{p\}$
(6.7) $\Gamma';\bar{o} \vdash \mathsf{eatts}(T') \text{ ok in } p$
(6.8) $\bar{o} \subseteq \mathsf{imm}$
(6.9) $\mathsf{deanon}(T', p \notin \bar{o}) = T'$

By applying Lemma 13 to (6.5), we obtain a type U such that:

(6.10) $\Gamma', x : U \vdash^{\kappa} \mathscr{E}[x] : T' \text{ in } p, \mathsf{acc}(p)$
(6.11) $\Gamma' \vdash C.k(\bar{v}) : U \text{ in } p, \mathsf{acc}(p)$
(6.12) The last rule of (6.11)'s type derivation is (Cons).

An inspection of the possible reasons for (6.7) shows that $\texttt{anon} \notin \mathsf{eatts}(T')$ unless $p \in \bar{o}$. We can therefore assume that $\texttt{anon} \notin \mathsf{eatts}(U)$ unless $p \in \bar{o}$. (If $\texttt{anon}$ is in $\mathsf{eatts}(U)$ but not in $\mathsf{eatts}(T')$, we can remove $\texttt{anon}$ from $\mathsf{eatts}(U)$ without violating judgment(6.10), by Lemma 7, or judgment (6.11), by (Sub).)

(6.13) $\texttt{anon} \in \mathsf{eatts}(U) \Rightarrow p \in \bar{o}$

We spell out the last rule of (6.11)'s type derivation:

$$\frac{\begin{array}{c}\mathsf{ctype}(C.k) = \bar{ty} \rightarrow ea_k\,\texttt{void} \quad \sigma = \mathsf{self}(p, \mathsf{acc}(p), w_p) \\ \Gamma' \vdash \bar{v}, p : \bar{ea}_{\bar{v}}\,\bar{ty}[\sigma], C\texttt{<}\mathsf{acc}(p), w_p\texttt{>} \text{ in } p, \mathsf{acc}(p) \quad ea = \bigcap(ea_k, \bar{ea}_{\bar{v}})\end{array}}{\Gamma' \vdash C.k(\bar{v}) : ea\,\texttt{void} \text{ in } p, \mathsf{acc}(p)}$$

(6.14) $U = ea\,\texttt{void}$

We define:

(6.15) $U'' \stackrel{\Delta}{=} \begin{cases} ea_k\,\texttt{void} & \text{if } p \in \bar{o} \\ (ea_k - \texttt{anon})\,\texttt{void} & \text{otherwise} \end{cases}$
(6.16) $U' \stackrel{\Delta}{=} \texttt{rdonly wrlocal}\,U''$

The following holds because $\texttt{anon} \in \mathsf{eatts}(U)$ implies $p \in \bar{o}$, by (6.13).

(6.17) $\Gamma, x : U' \vdash U' \preceq U$

We apply type specialization (Lemma 5) to judgment (6.10):

(6.18) $\Gamma', x : U' \vdash^{\kappa} \mathscr{E}[x] : T' \text{ in } p, \mathsf{acc}(p)$

Moreover, we have:

(6.19) $C.k(\bar{v}) : (\bar{o} \vdash p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} p)$ by (Call Frm Cons)
(6.20) $\Gamma' \vdash U'' : \mathsf{ok}$ follows from (6.11)

We now apply (Call Frm) to (6.4), (6.19), (6.18) and (6.20). We obtain:

(6.21) $\Gamma'; \bar{o} \vdash ((\mathscr{E}[C.k(\bar{v})] \,\mathsf{in}\, p) : T' \;\mathsf{in}\; p) \leftarrow (\bar{o} \vdash^{\mathsf{call}} U'' \;\mathsf{in}\; o)$

We define:

(6.22) $\Gamma'' \stackrel{\Delta}{=} (\emptyset, (\mathtt{world}, p, w : \bullet), \Gamma'_{\mathsf{val}})$ abbreviation

By applying (Stk Call) to (6.3) and (6.21), we obtain:

(6.23) $\Gamma; \bar{o} \vdash (s', \mathscr{E}[C.k(\bar{v})] \,\mathsf{in}\, p : T) \leftarrow (\Gamma''; \bar{o} \vdash^{\mathsf{call}} U'' \;\mathsf{in}\; o)$

We want to show the following:

(6.24) $\bar{o} \vdash \mathit{state} : T$ goal

We know that $(\Gamma; \bar{o} \vdash h : \mathsf{ok})$, by (3). By (State), (5) and (6), it therefore suffices to show the following:

(6.25) $\Gamma; \bar{o} \vdash s, e[\sigma'] \,\mathsf{in}\, p : T$ subgoal towards (6.24)

By (Stk Push) and (6.23), it suffices to show the following:

(6.26) $\Gamma''; \bar{o} \vdash^{\mathsf{call}} e[\sigma'] \,\mathsf{in}\, p : U'' \;\mathsf{in}\; p$ subgoal towards (6.25)

By (Frm) and (Mth Frm Body), we need to show the following four statements:

(6.27) $\Gamma'' \vdash e[\sigma'] : U'' \;\mathsf{in}\; p, \mathsf{acc}(p)$ subgoal towards (6.26)
(6.28) $\mathsf{vals}(e[\sigma']) \cap \bar{o} \subseteq \{p\}$ subgoal towards (6.26)
(6.29) $\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'') \;\mathsf{ok\ in}\; p$ subgoal towards (6.26)
(6.30) $\bar{o} \subseteq \mathsf{imm}$ subgoal towards (6.26)

Subgoal (6.30) is identical to fact (6.8); only subgoals (6.27), (6.28) and (6.29) remain open. We first derive some facts that are useful for several of these open proof goals. By construction, we have:

(6.31) $\Gamma''(p) = \mathtt{rdonly}\ \mathtt{wrlocal}\ \mathsf{ty}(p) = \mathtt{rdonly}\ \mathtt{wrlocal}\, D\texttt{<}ar, w\texttt{>}$

By definition of $\mathsf{acc}(p)$ and $\mathsf{owner}(p)$, we have $\mathsf{acc}(p) = ar$ and $\mathsf{owner}(p) = w$. We can therefore restate (6.31) and (4.1) like this:

(6.32) $\Gamma''(o) = \mathtt{rdonly}\ \mathtt{wrlocal}\, D\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>}$
(6.33) $\sigma' = \mathsf{self}(p, \mathsf{acc}(p), \mathsf{owner}(p)), \bar{x} \leftarrow \bar{v}$

By premise of the last rule of $(\Gamma' \vdash C.k(\bar{v}) : U \;\mathsf{in}\; p, \mathsf{acc}(p))$'s derivation, as displayed above, we have $(\Gamma' \vdash p : C\texttt{<}\mathsf{acc}(p), w_p\texttt{>} \;\mathsf{in}\; p, \mathsf{acc}(p))$. By inverting this judgment and using Lemma 6 in order to replace Γ' by Γ'', we obtain the following:

(6.34) $\Gamma'' \vdash D\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>} \preceq C\texttt{<}\mathsf{acc}(p), w_p\texttt{>}$

By Lemma 3(d), we can swap the class parameters in subtyping judgment (6.34) to obtain:

(6.35) $\Gamma'' \vdash D\texttt{<}\mathsf{acc}(p), w_p\texttt{>} \preceq C\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>}$

Because the types $\bar{ty}$ of $C.k()$'s formal parameters are legal, by premise of (Cons Dcl), we can apply Lemma 3(e) to obtain:

(6.36) $\Gamma'' \vdash \bar{ty}[\sigma] \preceq \bar{ty}[\sigma']$

Proof of goal (6.27)*:* We define:

(6.37) $ea_{\mathsf{rw}} \overset{\Delta}{=} \texttt{rdonly wrlocal}$ abbreviation

(6.38) $\Gamma_1 \overset{\Delta}{=} (\texttt{myaccess}, \texttt{world}, \texttt{myowner}, \texttt{this} : \bullet,$
$\texttt{this} : ea_{\mathsf{rw}}\, C\texttt{<myaccess,myowner>})$

Because the underlying class table is well-typed, we know, in particular, that *k*'s declaration in *C* is well-typed. By premise of (Cons Dcl), we have:

(6.39) $ea_{\mathsf{arw}} \overset{\Delta}{=} \texttt{rdonly wrlocal}$ abbreviation

(6.40) $\Gamma_1, \bar{x} : ea_{\mathsf{arw}}\, \bar{ty} \vdash e : ea_k\, \texttt{void}\ \mathsf{in}\ \texttt{this}, \texttt{myaccess}$

We define σ_1' as the restriction of σ' to the singleton domain $\{\texttt{myaccess}\}$:

(6.41) $\sigma_1' \overset{\Delta}{=} (\texttt{myaccess} \leftarrow \mathsf{acc}(p)) = \sigma'|\texttt{myaccess}$ abbreviation

By applying access right substitutivity (Lemma 11) to judgment (6.40), we obtain the following statement:

(6.42) $\Gamma_1[\sigma_1'], \bar{x} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma_1'] \vdash e[\sigma_1'] : ea_k\, \texttt{void}\ \mathsf{in}\ \texttt{this}, \mathsf{acc}(o)$

By expanding the definition of Γ_1 and weakening judgment (6.42), we obtain the following judgment:

(6.43) $\Gamma'', \texttt{myowner}, \texttt{this} : \bullet, \texttt{this} : ea_{\mathsf{rw}}\, C\texttt{<}\mathsf{acc}(p), \texttt{myowner>}, \bar{x} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma_1']$
$\vdash e[\sigma_1'] : V[\sigma_1']\ \mathsf{in}\ \texttt{this}, \mathsf{acc}(o)$

On the other hand, we have the following judgments:

(6.44) $\sigma_2' \overset{\Delta}{=} \sigma'|\texttt{myowner}, \texttt{myaccess}$ abbreviation

(6.45) $\Gamma'' \vdash \mathsf{owner}(o) : \bullet$

(6.46) $\Gamma'' \vdash p : ea_{\mathsf{rw}}\, C\texttt{<}\mathsf{acc}(p), \mathsf{owner}(p)\texttt{>}\ \mathsf{in}\ p, \mathsf{acc}(p)$

(6.47) $\Gamma'' \vdash \bar{v} : ea_{\mathsf{arw}}\, \bar{ty}[\sigma] \preceq ea_{\mathsf{arw}}\, \bar{ty}[\sigma'] = ea_{\mathsf{arw}}\, \bar{ty}[\sigma_2']\ \mathsf{in}\ o, \mathsf{acc}(o)$

Judgment (6.45) holds by construction of Γ''. Judgment (6.46) holds by (Obj). Judgment (6.47) follows from $(\Gamma' \vdash \bar{v} : \bar{ty}[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p))$ by subtyping judgment (6.36) and because we can replace Γ' by Γ'', by Lemma 6.

We now apply the customized substitutivity lemma 34 to judgments (6.45), (6.46), (6.47) and (6.42) to obtain the following:

(6.48) $\Gamma'' \vdash e[\sigma'] : \mathsf{deanon}(ea_k\, \texttt{void}, p \in \bar{v})\ \mathsf{in}\ p, \mathsf{acc}(p)$

We will have established goal (6.27), if we can show the following:

(6.49) $\Gamma'' \vdash \mathsf{deanon}(ea_k\, \texttt{void}, p \in \bar{v}) \preceq U''$ subgoal towards (6.27)

We expand the definition of U'':

(6.50) $\Gamma'' \vdash \mathsf{deanon}(ea_k\, \texttt{void}, p \in \bar{v}) \preceq \begin{cases} ea_k\, \texttt{void} & \text{if } p \in \bar{o} \\ (ea_k - \texttt{anon})\, \texttt{void} & \text{otherwise} \end{cases}$ goal

To this end, it suffices to show the following:

(6.51) $p \in \bar{o} \Rightarrow p \notin \bar{v}$ subgoal towards (6.27)

To show (6.51), let $p \in \bar{o}$. By inspecting the possible reasons for (6.7), we see that $p \in \bar{o}$ is only possible if $\texttt{anon} \in \mathsf{eatts}(T')$. Then it must also be the case that $\texttt{anon} \in \mathsf{eatts}(U)$ (because subexpressions of anonymous expressions are anonymous). Then $\texttt{anon} \in \bigcap \bar{ea}_{\bar{v}}$, by premise of the last rule of $(\Gamma' \vdash C.k(\bar{v}) : U\ \mathsf{in}\ p, \mathsf{acc}(p))$'s derivation, as displayed above. Then $p \notin \bar{v}$, by inverting judgment $(\Gamma' \vdash \bar{v} : \bar{ea}_{\bar{v}}\, \bar{ty}[\sigma]\ \mathsf{in}\ p, \mathsf{acc}(p))$ from the premise of the same rule.

Proof of goal (6.28)*:* By well-typedness of the underlying class table, k's body e in C is well-typed in an environment Γ such that $\mathsf{dom}(\Gamma_{\mathsf{val}}) = \{\mathtt{this}, \bar{x}\}$. Then $\mathsf{vals}(e) \subseteq \{\mathtt{null}, \mathtt{this}, \bar{x}\}$, by Lemma 18(a). Then $\mathsf{vals}(e[\sigma']) \subseteq \{\mathtt{null}, p, \bar{v}\}$, by Lemma 18(b). We need to show the following:

(6.52) $(q \in \{p, \bar{v}\} \cap \bar{o}) \Rightarrow (q = p)$ subgoal towards (6.28)

This is a consequence of (6.6).

Proof of goal (6.29)*:* We distinguish cases by the possible reasons for (6.7).

Case 6.1, (Frm Rdwr): In this case we get $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ p)$, by (Frm Rdwr). This is straightforward to check.

Case 6.2, (Frm Rdonly): In this case we get $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ p)$, by (Frm Rdonly). This is straightforward to check.

Case 6.3, (Frm Imm Cons): In this case we get $(\Gamma''; \bar{o} \vdash \mathsf{eatts}(U'')\ \mathsf{ok\ in}\ p)$, by (Frm Imm Cons). This is straightforward to check.

Case 7, (Let): This case is unproblematic. We omit the details.

Case 8, (Cast): This case is unproblematic. We omit the details. □

References

[BE04] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA'04*, pages 35–49, October 26–28, 2004.

[Blo01] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL'03*, pages 213–223, 2003.

[CD02] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA'02*, pages 292–310, 2002.

[CPN98] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA'98*, pages 48–64, 1998.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[Goe02] Brian Goetz. Java theory and practice: Safe construction techniques–don't let the "this" reference escape during construction. *IBM DevelopersWork*, 2002.

[IPW01] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[LP06a] Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP'06*, volume 4067 of *LNCS*, pages 99–123. Springer-Verlag, 2006.

[LP06b] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *POPL'06*, pages 359–371. ACM Press, 2006.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[PBKM00] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON'02*. IBM Press, 2000.

[PS05] I. Pechtchanski and V. Sarkar. Immutability specification and applications. *Concurrency and Computation: Practice and Experience*, 17:639–662, 2005.

[Sch04] J. Schäfer. Encapsulation and specification of object-oriented runtime components. Master's thesis, Technische Universität Kaiserslautern, 2004.

[SR05] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI'05*, pages 199–215, 2005.

[TE05] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA'05*, pages 211–230, October 18–20, 2005.

[VB01] J. Vitek and B. Bokowski. Confined types in Java. *Softw. Pract. Exper.*, 31(6):507–532, 2001.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[ZPV06] T. Zhao, J. Palsberg, and J. Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, January 2006.