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 declara-

tions $\bar{F}$, constructor declarations $\bar{K}$, method declarations $\bar{M}$ and parameter declarations $\bar{t}\bar{y}\,\bar{x}$ 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 $fm\,ca\,\texttt{class}\,C\,\texttt{ext}\,D\,\{..\} \in \bar{c}$. 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`)
ca ::= `immutable?`	class attributes
ea ::= `anon? rdonly? wrlocal?`	expression attributes
ar ::= `rd` | `rdwr` | `myaccess`	access rights for objects
fm ::= `final?`	final modifier
$u,v,w \in \mathsf{Val}$ ::= `null` | o | x	values
$ty \in \mathsf{ValTy}$ ::= C`<`ar,v`>` | `void`	value types
$T \in \mathsf{ExpTy}$ ::= $ea\,ty$	expression types
c,d ::= $fm\,ca\,\texttt{class}\,C\,\texttt{ext}\,D\,\{\bar{F}\ \bar{K}\ \bar{M}\}$	class declaration (where $C \neq$ `Object`)
F ::= C`<`ar,v`>` f`;`	field
K ::= $ea\,C.k(\bar{t}\bar{y}\bar{x})\{e\}$	constructor (scope of $\bar{x}$ is e)
M ::= fm`<`$\bar{y}$`>` $T\,m(\bar{t}\bar{y}\bar{x})\{e\}$	method (scope of $\bar{y}$ is $(T,\bar{t}\bar{y},e)$, of $\bar{x}$ is e)
$e \in \mathsf{Exp}$ ::=	expressions and statements
$v \mid v.f \mid v.f$`=`$e \mid v.m$`<`$\bar{v}$`>`$(\bar{e}) \mid$ `new`C`<`ar,v`>`$.k(\bar{e}) \mid$ `let`x`=`e `in`$e \mid (C)e \mid C.k(\bar{e})$	

Derived Forms:

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

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

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

$e.m(\bar{e}) \triangleq e.m\texttt{<>}(\bar{e})$ $fm\,T\,m(\bar{t}\bar{y}\bar{x})\{e\} \triangleq fm\texttt{<>}\,T\,m(\bar{t}\bar{y}\bar{x})\{e\}$

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

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 $C\texttt{<}ar,v\texttt{>}$, 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 $\texttt{new}\,C\texttt{<}ar,v\texttt{>}.k(\bar{e})$ creates a new object of type $C\texttt{<}ar,v\texttt{>}$ 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 $\texttt{C<rd,}p\texttt{>}$, then access to *o* is read-restricted. Furthermore, access to all objects in the transitive reach of *o* is read-restricted, too: $o.\texttt{get}()$, $o.\texttt{get}().\texttt{get}()$, etc., all have type $C\texttt{<rd,}p\texttt{>}$ 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 $C\texttt{<}ar,o\texttt{>}$ 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, we can define the Universe type system's `rep` and `peer` types [MPH01] as syntax sugar: `rep`$C \triangleq C$<`rdwr`,`this`> and `peer`$C \triangleq C$<`rdwr`,`myowner`>.

Jimuva has *owner-polymorphic methods*: In a method declaration <$\bar{y}$>$T\ m(\bar{t}y\,\bar{x})\{e\}$, the scope of owner parameters $\bar{y}$ includes the types $T, \bar{t}y$ 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
$h ::= \overline{obj} \in \mathsf{Heap} = \mathsf{ObjId} \to (\mathsf{FieldId} \to \mathsf{Val})$	heaps
$obj ::= o\{\bar{f} = \bar{v}\} \in \mathsf{Obj} = \mathsf{ObjId} \times (\mathsf{FieldId} \to \mathsf{Val})$	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
$\mathscr{E} ::= [\,] \mid v.f{=}\mathscr{E} \mid v.m{<}\bar{v}{>}(\bar{v}, \mathscr{E}, \bar{e}) \mid \texttt{new}\, C{<}ar,v{>}.k(\bar{v}, \mathscr{E}, \bar{e}) \mid$ `let` $x{=}\mathscr{E}$ `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 $C{<}ar, p{>}$ such that $p = \texttt{world}$ implies $ar = \texttt{rdwr}$. We define $\mathsf{rawty}(o) \stackrel{\Delta}{=} C$, if $\mathsf{ty}(o) = C{<}ar, p{>}$.

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.

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

We use several auxiliary functions that are essentially as in FJ [IPW01] (see also [HPSS07] for details): 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 $\bar{c}$. We omit the subscript $\bar{c}$ if it is clear from the context.

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

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

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

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

(Red New) $s=s',\mathscr{E}[\texttt{new}C\texttt{<}ar,w\texttt{>}.k(\bar{v})]\,\mathsf{in}\,p \quad o\notin\mathsf{dom}(h) \quad \mathsf{ty}(o)=C\texttt{<}ar,w\texttt{>} \quad \mathsf{fd}(C)=\bar{t}\,\bar{f}$
$h::s \rightarrow h,o\{\bar{f}=\texttt{null}\}::s,C.k(\bar{v})\,\texttt{;}\,o\,\mathsf{in}\,o$

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

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

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

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

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 $C\texttt{<}ar,\texttt{this}\texttt{>}$ then the state of the object that f refers to is included in `this`'s state; if f's type annotation has the form $C\texttt{<}ar,\texttt{myowner}\texttt{>}$ 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:

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

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

Example 1 (Object State).

```
class C ext Object { D<this> x; D<world> y; constructors and methods }
class D ext Object { E<myowner> x; E<this> y; constructors and methods }
class E ext Object { Object<myowner> x; constructors and 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 [HPSS07] 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 [HPSS07]. 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 [HPSS07] 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 supports *sealed optional packages*, which prohibit subclassing from outside the package.[5]

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] Out-of-package subclassing 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 $C\texttt{<}ar,\texttt{world>}$, and (3) all its `new`-calls have the form $\texttt{new}C\texttt{<}ar,\texttt{world>}.k(\bar{e})$.

`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 = $ `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 = $ `this` or (c) m is `wrlocal` and v has a type C<`rdwr`,`this`> or (c) 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, $\Gamma \vdash T \preceq U$:

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

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

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 $\Gamma_{\mathsf{acc}} \in \{\texttt{myaccess}\} \rightarrow \{\bullet\}$, $\Gamma_{\mathsf{own}} \in \mathsf{Var} \cup \mathsf{ObjId} \rightarrow \{\bullet\}$ and $\Gamma_{\mathsf{val}} \in \mathsf{Var} \cup \mathsf{ObjId} \rightarrow \mathsf{ExpTy}$. 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"
$\Gamma \vdash e : T$ in v, ar	"if `this` $= v$ and v has access rights ar, then e has type T"

In useful judgments $(\Gamma \vdash e : T \text{ in } v, ar)$, 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: $(\Gamma \vdash_{\bar{c}} e : T \text{ in } v, ar)$. 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 [HPSS07] for more details.

Auxiliary Predicates and Judgments:

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

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

$(ea, u, ar_u, v_u)\ \mathsf{wrloc\ in}\ v \triangleq (u = v, \texttt{wrlocal} \in ea) \text{ or } (ar_u, v_u) = (\texttt{rdwr}, v)$

$ar\ \mathsf{wrsafe\ in}\ ar' \triangleq (ar = \texttt{rdwr} \text{ or } ar' = \texttt{rd} \text{ or } ar = ar')$

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

$(\Gamma \vdash e : T \text{ in } ar) \triangleq (\Gamma \vdash e : T \text{ in } \texttt{this}, ar) \qquad (\Gamma \vdash e : T \text{ in } v) \triangleq (\Gamma \vdash e : T \text{ in } v, \texttt{rdwr})$

$(\Gamma \vdash \bar{e}, e : \bar{T}, T \text{ in } v, ar) \triangleq (\Gamma \vdash \bar{e} : \bar{T} \text{ in } v, ar \text{ and } \Gamma \vdash e : T \text{ in } v, ar)$

$(\Gamma \vdash e : T \preceq U \text{ in } v, ar) \triangleq (\Gamma \vdash e : T \text{ in } v, ar \text{ and } \Gamma \vdash T \preceq U)$

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

$(\Gamma \vdash v : \bullet) \triangleq (\Gamma \vdash \diamond \text{ and } \Gamma(v) = \bullet) \quad (\Gamma \vdash v : \mathsf{ok}) \triangleq (\Gamma \vdash \diamond \text{ and } v \in \mathsf{dom}(\Gamma) \cup \{\texttt{null}\})$

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

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

$(\Gamma \vdash ty\ \mathsf{legal}) \triangleq (\Gamma \vdash ty : \mathsf{ok} \text{ and } ty\ \mathsf{legal})$

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

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

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

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

(Mth Dcl) $C\,\texttt{ext}\,D \Rightarrow \Gamma \vdash \mathsf{mtype}(m, C) \preceq \mathsf{mtype}(m, D) \quad \mathsf{atts}(C) \neq \emptyset \Rightarrow \texttt{rdonly} \in \mathsf{atts}(T)$
$ar = \texttt{myaccess} \text{ or } (\{\texttt{rdonly}, \texttt{wrlocal}\} \cap ea = \emptyset, ar = \texttt{rdwr}) \quad \texttt{this} \notin \mathsf{vars}(\bar{ty}, T)$
$$\frac{\Gamma[\sigma], \bar{y} : \bullet, \bar{x} : \texttt{anon rdonly wrlocal}\, \bar{ty}[\sigma] \vdash e[\sigma] : T[\sigma] \text{ in } ar \qquad \sigma = (\texttt{myaccess} \leftarrow ar)}{\Gamma \vdash fm\texttt{<}\bar{y}\texttt{>}\, ea\, ty'\, m(\bar{ty}\,\bar{x})\{e\} : \mathsf{ok} \text{ in } C}$$

Well-typed Expressions, $\Gamma \vdash e : T$ in v, ar:

$$\text{(Var)}\ \frac{\Gamma(x) = ea\texttt{C<}ar_x, v_x\texttt{>}}{\Gamma \vdash x : ea\texttt{C<}ar_x, v_x\texttt{>}\ \text{in}\ v, ar_v} \qquad \text{(Obj)}\ \frac{ea = \{\texttt{anon} \mid o \neq p\}}{\Gamma \vdash o : ea\,\Gamma_{\mathsf{val}}(o)\ \text{in}\ p, ar_p} \qquad \text{(Sub)}\ \frac{\Gamma \vdash e : T \preceq U\ \text{in}\ v, ar_v}{\Gamma \vdash e : U\ \text{in}\ v, ar_v}$$

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

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

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

$$\text{(Call)}\ \frac{\begin{array}{l} \mathsf{mtype}(m, C_u) = fm\texttt{<}\bar{y}\texttt{>}\bar{t}y \to ea_m\, ty' \\ (\texttt{rdonly} \in ea_m)\ \text{or}\ (ar_u\ \text{wrsafe in}\ ar_v) \qquad \sigma = \mathsf{self}(u, ar_u, v_u), \bar{y} \leftarrow \bar{w} \\ ea = \bigcap \bar{e}a_{\bar{e}} \cap \bigcup(\ \{\texttt{anon}\} \cap (ea_m \cup ea_u),\ \{x\ \text{as}\ \texttt{rdonly} \mid x \in ea_m\ \text{or}\ v_u, \bar{w} = \texttt{world}\}, \\ \quad \{\texttt{wrlocal} \mid (ea_m, u, ar_u, v_u)\ \text{wrloc in}\ v\ \text{or}\ \texttt{rdonly} \in ea_m\ \text{or}\ v_u = \texttt{world}\}\ ) \\ \Gamma \vdash u, \bar{e} : ea_u C_u\texttt{<}ar_u, v_u\texttt{>}, \bar{e}a_{\bar{e}}\, \bar{t}y[\sigma]\ \text{in}\ v, ar_v \quad \Gamma \vdash \bar{w} : \bullet \quad (ar_u = \texttt{rd}\ \text{or}\ \Gamma \vdash v_u : \bullet) \end{array}}{\Gamma \vdash u.m\texttt{<}\bar{w}\texttt{>}(\bar{e}) : ea\, ty'[\sigma]\ \text{in}\ v, ar_v}$$

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

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

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.

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.

[HPSS07] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. Technical report, Radboud University Nijmegen, 2007. Forthcoming.

[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.