Type-based Object Immutability with Flexible Initialization

Christian Haack[1,2⋆] and Erik Poll[1⋆]

[1] Radboud University, Nijmegen [2] aicas GmbH, Karlsruhe

Abstract. We present a type system for checking object immutability, read-only references, and class immutability in an open or closed world. To allow object initialization outside object constructors (which is often needed in practice), immutable objects are initialized in lexically scoped regions. The system is simple and direct; its only type qualifiers specify immutability properties. No auxiliary annotations, e.g., ownership types, are needed, yet good support for deep immutability is provided. To express object confinement, as required for class immutability in an open world, we use qualifier polymorphism. The system has two versions: one with explicit specification commands that delimit the object initialization phase, and one where such commands are implicit and inferred. In the latter version, all annotations are compatible with Java's extended annotation syntax, as proposed in JSR 308.

1 Introduction

1.1 Motivation

Immutable data structures greatly simplify programming, program maintenance, and reasoning about programs. Immutable structures can be freely shared, even between concurrent threads and with untrusted code, without the need to worry about modifications, even temporary ones, that could result in inconsistent states or broken invariants. In a nutshell, immutable data structures are simple. It is therefore not surprising that favoring immutability is a recommended coding practice for Java [3].

Unfortunately, statically checking object immutability in Java-like languages is not easy, unless one settles for supporting only a restricted programming style that can be enforced through `final` fields. Clearly, objects are immutable if all their fields are `final` and of primitive type. Additionally, one can allow `final` fields of immutable types, this way supporting immutable recursive data structures. Thus, Java's `final` fields support a style of programming immutable objects that mimics datatypes in functional languages and is advocated, for instance, by Felleisen and Friedman [15].

Many immutable objects, however, do not follow this style. A prominent example are Java's immutable strings. An immutable string is a wrapper around a character array. While `final` fields can prevent that a string's internal character array is replaced by another character array, `final` fields cannot prevent that the array elements themselves are mutated. Moreover, Java's type system provides no means for preventing representation exposure of the character array, which would allow indirect mutation of a string through aliases to its (supposedly) internal character array. Preventing this, not just for

⋆ Supported by IST-FET-2005-015905 Mobius project.

arrays but for any internal mutable data structures, requires a richer type system with support for object confinement.

It is also quite common to have immutable data structures that are not instances of immutable classes. Examples include immutable arrays, immutable collections that are implemented in terms of Java's mutable collection classes (but are never mutated after initialization), and immutable cyclic data structures, e.g., doubly linked lists, graphs or trees with parent references. Concrete examples are given on pages 8, 10 and Figure 3.

This article presents the design of a pluggable type system for Java to specify and statically check various immutability properties. A pluggable type checker operates on Java's abstract syntax trees and is optionally invoked after the standard type checker, to ensure additional properties. A pluggable checker for object immutability guarantees that immutable objects never mutate.

Syntactically, our immutability type system can be handled with Java's extended annotation syntax as proposed by JSR 308 [19], to be included in Java 7, which allows annotations on all occurrences of types. While in this paper we slightly deviate from legal annotation syntax (for explanatory reasons), all proposed annotations are in syntactic positions allowed by JSR 308.

1.2 Kinds of Immutability

The following classification of immutability properties has been used in various places in the literature [34,22]:

- *Object immutability:* An object is immutable if its state cannot be modified.
- *Class immutability:* A class is immutable if all its instances in all programs are immutable objects.
- *Read-only references:* A reference is read-only if the state of the object it refers to cannot be modified through this reference.

Examples of *immutable classes* are Java's `String` class and the wrapper classes for primitive types, e.g., `Integer` and `Boolean`. All instances of immutable classes are immutable objects.

Conversely, *immutable objects* need not be instances of immutable classes. For example, immutable arrays are not instances of an immutable class, and neither are immutable collections that are implemented in terms of Java's mutable collection libraries. Immutable objects that are not instances of immutable classes typically have public, non-final fields or public mutator methods, but the pluggable type system disallows assignments to these fields and calls to these methods.

An example for a *read-only reference* is the reference created by Java's static method `Collection unmodifiableCollection(Collection c)`, which generates a wrapper around collection `c`. This wrapper refers to `c` through a read-only reference.

For class immutability, we further distinguish between an open and a closed world [25]:

- Class immutability *in a closed world* assumes that all program components follow the rules of the pluggable type system.
- Class immutability *in an open world* assumes that immutable classes and the classes they depend on follow the rules of the pluggable type system, but clients of immutable classes are unchecked (i.e., they only follow Java's standard typing rules).

Unchecked class clients may for instance be untrusted applets. Note that the closed world assumption only makes sense if *all* code is checked with the additional type rules. Java's classes `String`, `Integer` and `Boolean` are immutable in an open world. For class immutability in an open world it is essential that instances of immutable classes encapsulate their representation objects. Open-world-immutable classes necessarily have to initialize their instances inside constructors or factory methods, and they should not provide accessible mutator methods or fields. Note also that, in an open world, object immutability without class immutability can only be achieved for objects that are never exposed to unchecked clients, because unchecked clients cannot be prevented from calling mutator methods or assigning to accessible fields if these exist. Similarly, in an open world, read-only references can only be achieved for references that are never exposed to unchecked clients.

1.3 Specifying Immutability with Type Qualifiers

Following our earlier work [18], we support the distinction between mutable and immutable objects through *access qualifiers* on types:

Access qualifiers:

p,q	::=	`RdWr`	read-write access (default)
		`Rd`	read-only access
		...	

Types:

T	::=	$q\,C$	C-object with q-access
C	$\in$	ClassId	class identifiers

Objects of type `Rd` C are called `Rd`-objects, and have immutable fields. Our type system is designed to guarantee the following soundness property (see Theorem 2):

Well-typed programs never write to fields of `Rd`*-objects*.

For instance, the method `bad()` attempts an illegal write to a `Rd`-object and is forbidden by our type system. On the other hand, `good()` legally writes to a `RdWr`-object:

```
class C { int f; }
static void bad(Rd C x) {
    x.f = 42; // TYPE ERROR
}
```

```
static void good(RdWr C x) {
    x.f = 42; // OK
}
```

An additional type qualifier, `Any`, represents the least upper bound of `Rd` and `RdWr`:

p,q	::=	...	
		`Any`	"either `Rd` or `RdWr`"

Subqualifying:

`Rd` $<:$ `Any` `RdWr` $<:$ `Any`

Subtyping:

$$\frac{p <: q \qquad C <: D}{p\,C <: q\,D}$$

A reference of a type `Any` C may refer to a `Rd`-object or a `RdWr`-object, so writes through `Any`-references are forbidden. Beware of the difference between `Rd` and `Any`. A reference of type `Any` C is a *read-only reference*, meaning you cannot write to the object through this particular reference. A reference of type `Rd` C is a reference to a read-only object, i.e. to an object that *nobody* has write-access to.[1]

[1] IGJ [34] uses the same three qualifiers, calling them `@Mutable`, `@Immutable`, and `@ReadOnly` instead of `Rd`, `RdWr` and `Any`.

The following example shows how Any-references can be useful. The method m() creates a RdWr-array and then applies the method foo() to the array. From the type of foo() we can tell that foo() does not mutate the array: [2]

```
interface Util {
    void foo(int Any [] a);
}
```

```
static void m(Util util) {
    int[] a = new int RdWr [] {42,43,44};
    util.foo(a);
    assert a[0] == 42;
}
```

In this example, we assume a closed world. In an open world, where there may be unchecked classes that do not play by the additional rules our type system imposes, there is still the possibility that foo() writes a to some heap location of type Any, so that unchecked class could modify a[0] concurrently. Preventing foo() from writing its parameter to the heap can be achieved by a more general method type that uses qualifier polymorphism, as will be discussed in Section 2.3.

1.4 Flexible Object Initialization With Stack-local Regions

A common problem of type systems for object immutability [4,18,34,22] and for non-nullness (more generally, object invariants) [13,14,28] is object initialization. Whereas in traditional type systems, values have the same types throughout program execution, this is not quite true for these systems. Type systems for non-nullness face the difficulty that all fields are initially null; type systems for object immutability face the difficulty that even immutable objects mutate while being initialized. In these systems, each object starts out in an uninitialized state and only obtains its true type at the end of its initialization phase. Thus, objects go through a *typestate transition* from "uninitialized" to "initialized".

Object initialization is often the most complicated aspect of otherwise simple type systems, see for instance Fähndrich and Leino's non-nullness type system [13]. Some of the above type systems require that initialization takes place inside object constructors [13,18,34]. Unfortunately, this does not really simplify matters because object constructors in Java-like languages can contain arbitrary code (which may, for instance, leak self-references or call dynamically dispatched methods). Moreover, initialization inside constructors is often too restrictive in practice. For instance, cyclic data structures often get initialized outside constructors, and array objects do not even have constructors.

One contribution of this paper is a simple but flexible object initialization technique for immutability, using stack-local memory regions. Object initialization with stack-local regions supports a programming style that is natural for programmers in mainstream OO languages. In particular, programmers do not have to mimic destructive reads, as required by type systems where object initialization is based on unique references [4,22]. Statically checking object initialization with stack-local regions is simple, as it does not require tracking aliasing on the heap, which is needed in more general typestate-like systems based on static capabilities [10,29,6,11,5,7,2]. In order to facilitate modular static checking, these systems use additional program annotations in the form of constraints, effects, or pre/postconditions. Our system, on the other hand, only uses standard type annotations, largely hiding the typestate change from "uninitialized"

[2] Following JSR 308 syntax, the qualifier of an array type C[] is written before the [].

to "initialized" from programmers. To this end, we have designed an inference algorithm that automatically infers the end of object initialization phases (see Section 3.4).

1.5 Object Confinement with Qualifier-polymorphic Methods

A type system for class immutability in an open world must enforce several confinement properties [3]. Specifically, it must guarantee that instances of immutable classes encapsulate their representation objects and that their object constructors do not leak self-references. In our earlier paper [18], we enforced these properties using two type-based confinement techniques (in addition to the access qualifiers `Rd` and `RdWr`), namely a dedicated ownership type system for enforcing encapsulation of representation objects, and so-called anonymous methods [32] for confining self-references during object construction. Unfortunately, the resulting type system was more complex than one would desire. One of the insights of this article is that, when combined with flexible object initialization, the various confinement properties for class immutability can be expressed in terms of methods that are polymorphic in access qualifiers.

To get an idea how polymorphism helps with confinement, consider the following qualifier-polymorphic method signature:

```
<q> void foo(char q [] arg)
```

where `<q>` denotes universal quantification of the qualifier variable `q`, making the method polymorphic in `q`. For a qualifier hierarchy without greatest element, this signature tells us that `foo()` does not write its parameter to a heap location, because the type of such a location would need a single qualifier annotation that is greater than all other qualifiers.[3] This observation can be exploited to confine representation objects of immutable objects and to confine self-references to constructors of immutable objects.

To support *deep immutability* we treat the access qualifier as an implicit class parameter. It is interesting that this single class parameter in combination with qualifier-polymorphic methods and flexible object initialization suffices for satisfactorily encoding class immutability. In particular, we do not need separate ownership annotations, because the required confinement properties can be expressed in terms of these primitives, in a similar way as in ownership type systems. Flexible initialization is a crucial ingredient, as it allows us, for instance, to treat the internal character array of a string as an *immutable* object (rather than as a *mutable* object that is owned by an immutable one). This would not be possible if object initialization was tied to object constructors, because then all arrays would necessarily be mutable[4]. As a result of treating the character array inside a string as immutable, our type system can, for instance, easily support different strings sharing the same, immutable, character array for their representation, which is often problematic with ownership types.

---

[3] `Any` is actually not the greatest element of our qualifier hierarchy, but the greatest qualifier for *initialized* objects. We still name this qualifier `Any` (rather than `Initialized`). Fortunately, qualifiers for uninitialized objects are inferred and never need to be written by programmers.

[4] Supporting immutable arrays initialized by array initializers is not enough for the constructor `String(char[] c)` of Java's `String` class, because the length of `c` is not known statically.

1.6 Summary of Contributions

Based on the ideas sketched in this introduction, we have designed a pluggable immutability type system for Java-like languages. The primitives of the type language are the type qualifiers `Rd`, `RdWr` and `Any` for specifying object access rights. The features of the system are:

- *expressiveness:* the system supports object immutability, read-only references, and class immutability in a closed and open world;
- *simplicity and directness:* the system only needs the type qualifiers `Rd`, `RdWr` and `Any` plus qualifier polymorphism; its formal typing rules are simple; annotations are only required on field types and in method signatures; no annotations are required inside method bodies;
- *flexible initialization:* object initialization is not tied to object constructors; while the type system is necessarily flow-sensitive in order to support object initialization, it works for concurrency, too, because it enforces that threads only share initialized objects and because types of initialized objects are persistent.

On the technical side, our contributions are:

- *type system formalization and proof of soundness for object immutability:* we formalize a subset of the type system for a small model language; this subset focuses on what we believe is the most critical part of the system, namely, the initialization phase; we prove that the system is sound for object immutability: *well-typed programs never write to* `Rd`*-objects*;
- *a local annotation inference algorithm:* we present a local annotation inference algorithm that automatically infers the end of object initialization phases; we have formalized this algorithm for our model language and proven it sound.

Outline. The rest of the paper has two parts. Section 2 informally discusses the type system design. Section 3 contains the technical contributions: it formalizes the type system for a small model language, presents the annotation inference algorithm, and states soundness theorems, whose detailed proofs are contained in the companion report [17]. Section 4 compares to related work and Section 5 concludes.

Acknowledgments. We thank the anonymous ECOOP referees and James Noble for their careful reviews, and comments and critique that helped improve the paper.

2 Informal Presentation

We carry on with the informal presentation, as started in Section 1.3.

2.1 Access Qualifier as Class Parameter

For aggregate object structures, it is desirable to associate a single access qualifier with the entire aggregate, especially if the internal structure of the aggregate is hidden from object clients. In order to support *access control for aggregates through single access qualifiers*, we treat the access qualifier as an implicit class parameter. We have already proposed this in [18] and so has IGJ [34]. Technically, we introduce a *special access variable* `myaccess` that refers to the access qualifier of `this`. The scope of this variable is the entire class body. In particular, the `myaccess` variable can be used in field types

and signatures of methods and constructors. In the `Square` class below, `myaccess` annotates the type `Point` of its fields. Method `m()` takes an `Any`-square, so can neither write to the `Point`-fields of the square, nor to the `int`-fields of its points.

```
class Point { int x; int y; }
class Square { myaccess Point upperleft; myaccess Point lowerright; }
static void m(Any Square s) {
    s.upperleft = s.lowerright; // TYPE ERROR
    s.upperleft.x = 42; // TYPE ERROR
}
```

It is also possible to assign a single access right to a cyclic structure. For instance:

```
class Person { myaccess Person partner; }
class Couple { myaccess Person husband; myaccess Person wife; }
```

Old-fashioned couples stick with each other forever: they have type `Rd Couple`. Modern couples can divorce and the partners can re-marry: they have type `RdWr Couple`.

The access qualifier is a *covariant class parameter*. Generally, covariant class parameters are unsound, because upcasting a class parameter allows ill-typed writes to fields whose types depend on this class parameter. Here, treating the access qualifier covariantly is sound, because access qualifiers that permit write-access are minimal elements of the qualifier hierarchy. Thus, *upcasting access qualifiers makes object references read-only*.

2.2 Flexible Initialization

For sound object initialization, we adapt a technique from region-based memory management [30], allowing initialization of immutable objects inside *stack-local memory regions* (closely related to *lexically scoped regions*). A stack-local region is a part of the heap that cannot be reached from the rest of the heap. All references into a stack-local region are on the stack. Each stack-local region is *owned* by a method (or a constructor), namely, the lowest method on the call stack that holds references into this region. All objects inside a stack-local region have the same special type qualifier. The method that owns the region (and only this method) is permitted to change this type qualifier to some other qualifier, uniformly for all objects in the same region. When this typestate change is performed, the owning method is on the top of the call stack, so all references into the stack-local region come from local variables of this owning method. This means that all references into the stack-local region at the time of the typestate change are statically known: the static type system can easily modify the type qualifiers of these references.

Technically, to support flexible initialization, we add `Fresh`-qualifiers. These have a name as an argument, which we call an *initialization token*.

p,q	$::=$	$\cdots$	
		`Fresh(`n`)`	fresh object under initialization
n	$\in$	Name	token for initializing a set of related objects

An initialization token can be viewed as an identifier for a stack-local region that contains `Fresh(`n`)`-objects. The token n is secret to the method that owns the associated region and grants permission to commit `Fresh(`n`)` to q, for any q. To syntactically capture this semantics, we introduce two *specification commands*:

`newtoken` *n*	create a new initialization token
`commit Fresh(`*n*`) as` *q*	globally convert `Fresh(`*n*`)` to *q*

These are specification commands, i.e., they operate on auxiliary state ("ghost state") and have no runtime effect on concrete state or control flow. Our inference algorithm can infer all specification commands, so they need not be written by the programmer. In fact, all annotations inside method bodies can be inferred, so that programmers only have to write qualifiers in field declarations and method signatures. In the examples below, all inferred annotations are shaded gray.

The following method, for instance, creates an immutable array; it uses the flexible initialization technique, to initialize the array `r` outside a constructor.

```
static char Rd [] copy (char Any [] a) {
    newtoken n;
    char[] r = new char Fresh(n) [a.length];
    for (int i=0; i++; i < a.length) r[i] = a[i];
    commit Fresh(n) as Rd;
    return r;
}
```

To initialize immutable cyclic data structures, we use the same initialization token for all members of the structure. Using the flexible initialization technique, we can set cross-references (here `husband` and `wife`) *after* the constructors have been called:[5]

```
newtoken n;
Person alice = new <Fresh(n)>Person();
Person bob = new <Fresh(n)>Person();
alice.partner = bob; bob.partner = alice;
Couple couple = new <Fresh(n)>Couple();
couple.husband = bob; couple.wife = alice;
commit Fresh(n) as Rd;
```

Note that field types and method signatures cannot contain `Fresh(`*n*`)`-annotations, because *n* is out-of-scope in field types and method signatures:

```
class C {
    Fresh(n) D x; // TYPE ERROR: n out of scope
    static Rd C commit(Fresh(n) C x) { // TYPE ERROR: n out of scope
        commit Fresh(n) as Rd; return x; }
}
```

Because we do not allow methods that are parametrized by initialization tokens, each initialization token is confined to a single method. As a result, only the method that "owns" a `Fresh(`*n*`)`-region can commit it, which is crucial for the soundness of `commit`.

Figure 1 sketches a runtime configuration before a `commit`-statement. In this configuration, the heap has three regions: a region of initialized objects, and two `Fresh` regions with associated initialization tokens `n1` and `n2`. The picture shows possible inter-region references. Importantly, the type system ensures that there are *no incoming references from the heap into* `Fresh` *regions*. Furthermore, when the top of the stack

[5] `Person()` is a qualifier-polymorphic constructor, hence the angle brackets. See Section 2.4.

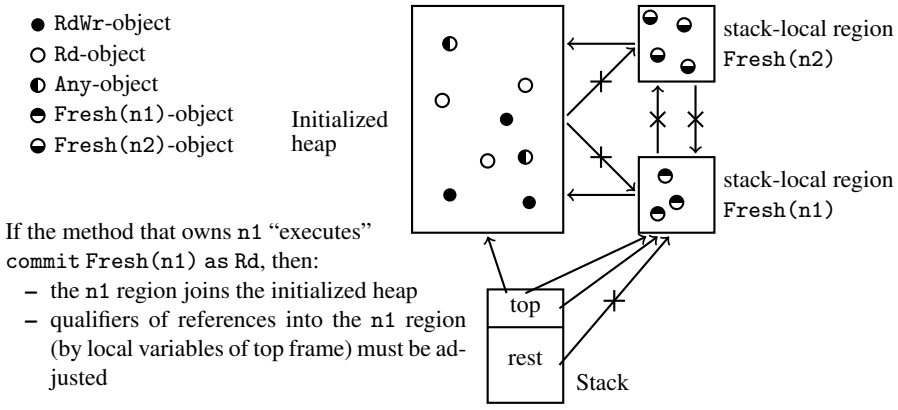

Fig. 1. Committing the fresh region owned by the top stack frame.

owns region `n1`, there are no references from the rest of the stack into this region. When the `commit`-statement is executed, region `n1` is merged with the initialized region. The type system then has to adjust the qualifiers of all references into region `n1`. Fortunately, this can be done statically, because all references into this region come from local variables in its owning method.

2.3 Qualifier Polymorphism for Methods

Consider the following method:

```
static void copy(Point src, Point dst) {
    dst.x = src.x; dst.y = src.y;
}
```

This method could accept both `RdWr`-points and `Fresh`-points as `dst`-parameters. To facilitate this, we introduce *bounded qualifier polymorphism for methods*. The Hasse diagram in Figure 2.3 depicts the qualifier hierarchy, including qualifier bounds. The syntax for qualifier-polymorphic methods is as in Java Generics:

$$\texttt{<}\bar{\alpha}\ \texttt{extends}\ \bar{B}\texttt{>}\ T\ m(\bar{T}\ \bar{x})\ q\,\{\ldots\}\quad \textit{(method declaration)}$$

We usually omit the qualifier bound `Qual`, writing `<a extends Qual>` as `<a>`. The qualifier q is associated with the receiver parameter, that is, $e.m()$ can only be called if e's access qualifier is a subqualifier of q. Receiver qualifiers are not present in `static` methods. For subclassing, method types are treated contravariantly in the qualifiers on input types (including the receiver qualifier) and covariantly in the qualifier on the output type. These variances are as in IGJ [34]. We can now type `copy()` as follows:

```
static <a, b extends Writeable> void copy(a Point src, b Point dst) {
    dst.x = src.x; dst.y = src.y;
}
```

Note that `Writeable` can only be used as a qualifier bound, but not as a qualifier. Allowing `Writeable` as qualifier would lead to unsoundness for two reasons: Firstly,

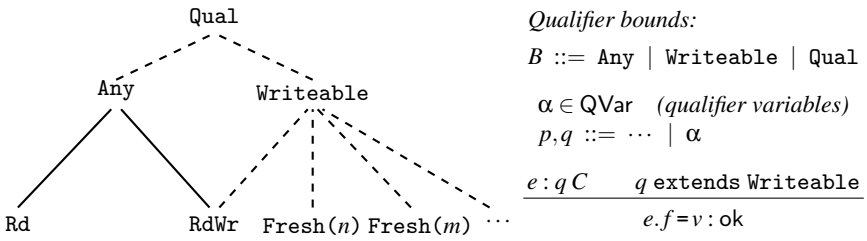

Fig. 2. The qualifier hierarchy. `Qual` and `Writable` are qualifier *bounds*, not qualifiers, so they cannot be used as type qualifiers, only in `extends`-clauses.

`Writeable` would be a *non-minimal qualifier that allows writes*, which would make covariance of the `myaccess` class parameter unsound. Secondly, `Writeable` could be used as an annotation on field types. This would open the door for violating stack locality of `Fresh`-regions, which would make the typestate transition at `commit`s unsound.

Signatures of qualifier-polymorphic methods tell us which method parameters are potentially mutated by the method. In addition, they also provide information about which method parameters are potentially written to the heap. For instance:

- `static <a> void foo(int a [] x);`
  - does not write to object `x` through reference `x`
  - does not write object `x` to the heap
- `static void faa(int Any [] x);`
  - does not write to object `x` through reference `x`
  - may write object `x` to the heap (into `Any`-fields)
- `static <a extends Writeable> void fee(int a [] x);`
  - may write to object `x` through reference `x`
  - does not write object `x` to the heap

The method `foo`(x) cannot write x to the heap, because the qualifier hierarchy does not have a greatest element, which would be needed as the type of a location that x can be written to. Similarly, `fee`(x) cannot write x to the heap, because there is no qualifier that bounds all writeable qualifiers.

In the following example, we use the qualifier for the receiver parameter to distinguish between inspector and mutator methods. Inspectors can be called on any receivers, whereas mutators can only be called on writeable receivers:

```
class Hashtable<K,V> {
    <a> V get(K key) a { ... }  // inspector
    <a extends Writeable> V put(K key, V value) a { ... }  // mutator
}
```

To create an immutable hash table we can use flexible initialization outside the constructor:

```
newtoken n;
Hashtable<String,String> t = new <Fresh(n)>Hashtable<String,String>();
t.put("Alice", "Female"); t.put("Bob", "Male");
commit Fresh(n) as Rd;
t.get("Alice"); // OK
t.put("Charly", "Male"); // TYPE ERROR
```

2.4 Constructors

Constructor declarations have one of the following two forms:

`<`$\bar{\alpha}$ `extends` $\bar{B}$`>` $q\ C(\bar{T}\ \bar{x})\ p$`{` *body* `}`	*(caller-commit constructor)*
`<`$\bar{\alpha}$ `extends` $\bar{B}$`>` $q\ C(\bar{T}\ \bar{x})$`{` `newtoken` n; *body* `}`	*(constructor-commit constructor)*

Caller-commit constructors are more common. In their signature, p represents the qualifier of `this` when the constructor body starts executing. The typechecker assumes this qualifier initially when checking the constructor body, and enforces that constructor callers, through `super()` or `this()`, establish this precondition. The postcondition q represents the qualifier of `this` when the constructor terminates.

A typical instance of caller-commit constructors looks like this:

$$\texttt{<}\alpha\ \texttt{extends Writeable>}\ \alpha\, C(\bar{T}\ \bar{x})\ \alpha\texttt{\{ ... \}}$$

In particular, the default no-arg constructors have this form. Note that, if in the above constructor signature α does not occur in any of the parameter types $\bar{T}$, then we know that the constructor does not leak references to `this`[6]. This is often desired for constructors. Constructors that deliberately leak `this` could have the following form (which prevents the creation of immutable class instances):

$$\texttt{RdWr}\ C(\bar{T}\ \bar{x})\ \texttt{RdWr\{ ... \}}$$

Constructor-commit constructors enforce that the object is committed inside the constructor. This is useful in an open world to prevent object clients from ever seeing an uninitialized object. In constructor-commit constructors, the precondition is omitted. Instead, the constructor begins by generating a fresh token n. The body then initially assumes that `this` has qualifier `Fresh(`n`)`. The scope of n is the constructor body, and therefore n cannot be mentioned in the constructor postcondition. To establish the postcondition, the body is forced to commit `Fresh(`n`)` before it terminates. The type system disallows calling constructor-commit constructors through `super()` or `this()`. Therefore, constructor-commit constructors are particularly suited for `final` classes.

Figure 3 shows an example with a caller-commit constructor. An immutable tree with parent pointers is constructed from the bottom up. A single initialization token is used for all nodes and is committed only after the root node has been initialized. This example is interesting because Qi and Myers [28] identify it as a problematic initialization pattern for other type systems [14]. It causes no problems for our system.

2.5 Class Immutability in an Open World

In his book "Effective Java" [3], Bloch presents rules that ensure class immutability. These rules require that fields of immutable classes are private and final, that public methods are inspectors, that methods and constructors do not leak representation objects, that public constructors do not leak `this`, and that the behaviour of instances of immutable classes does not depend on overridable methods. Some of these rules (e.g., that all fields are private and final) can very easily be checked automatically. The conditions that methods of immutable classes are inspectors, that instances of immutable

[6] If α occurs in $\bar{T}$, the constructor could for instance leak `this` to a field $x.f$ of a constructor parameter $\alpha\, D\, x$, in case f's type in C is annotated with `myaccess`.

```
class Tree {
    myaccess Tree parent, left, right;
    <a extends Writeable> a Tree (a Tree left, a Tree right) a {
        this.left = left; this.right = right;
        if (left != null) left.parent = this;
        if (right != null) right.parent = this;
    }
}
newtoken n;
Tree left_leaf = new <Fresh(n)>Tree(null, null);
Tree right_leaf = new <Fresh(n)>Tree(null, null);
Tree root = new <Fresh(n)>Tree(left_leaf, right_leaf);
root.parent = root;
commit Fresh(n) as Rd;
```

Fig. 3. Bottom-up initialization of a tree with parent pointers

classes do not leak representation, and that constructors of immutable classes do not leak `this` can be expressed and checked by our type system.

If we specify class immutability with a class annotation `Immutable`, we could for instance declare an immutable `String` class like this:

```
Immutable final class String {
    private final char myaccess [] value;
    ...
}
```

Semantically, the `Immutable` annotation is meant to specify that `String` is an immutable class in an open world, i.e., that all instances of `String` are `Rd`-objects that cannot be mutated by possibly unchecked clients. In order to tie the access modifier for the `value` array to the access modifier for the enclosing string, it is important that we annotate the `value` field with `myaccess` instead of `Rd`. In combination with the requirements on method and constructor signatures below, this prevents representation exposure of the character array.

The following rules guarantee class immutability:

- immutable classes must be final and direct subclasses of `Object`
- methods and constructors may only call static or final methods or methods of final classes (transitively)
- all fields must be final
- public constructors must have the following form:

  $\texttt{<}\bar{\alpha}\ \texttt{extends}\ \bar{B}\texttt{>}\ \texttt{Rd}\ C(\bar{T}\ \bar{x})\texttt{\{}\ \texttt{newtoken}\ n\texttt{;}\ \ldots\texttt{;}\ \texttt{commit}\ \texttt{Fresh(}n\texttt{)}\ \texttt{as Rd;}\ \texttt{\}}$
  where `myaccess` does not occur in $\bar{T}$

- types of public methods must have the following form:

  $$\texttt{<}\alpha,\bar{\beta}\ \texttt{extends}\ \bar{B}\texttt{>}\ U\ m(\bar{T}\ \bar{x})\ \alpha\texttt{\{}\ldots\texttt{\}}$$

```
static <a, b extends Writeable>
void arraycopy(a Object src, int srcPos, b Object dst, int dstPos, int l);

public <a> Rd String(char a value[]) {
    newtoken n;
    int size = value.length;
    char[] v = new char Fresh(n) [size];
    System.arraycopy(value, 0, v, 0, size);
    this.offset = 0; this.count = size; this.value = v;
    commit Fresh(n) as Rd;
}
```

Fig. 4. A constructor of Java's immutable `String` class

We use the `String` example to explain the constructor rule: The rule ensures that public constructors do not assign previously existing character arrays to the string's `value` field. This would only be possible, if the class parameter `myaccess` occurred in one of the parameter types $\bar{T}$, which is forbidden. For instance, the constructor `String(char value[])` is forced to make a defensive copy of its input parameter, as shown in Figure 4. Furthermore, constructors can not assign `this` or `this.value` to heap locations outside the stack-local `Fresh(`n`)`-region. This would only be possible if one of the parameter types $\bar{T}$ mentioned `myaccess`, or if the `commit`-statement were executed somewhere in the middle of the constructor, in which case the constructor could write `this.value` or `this` to the heap as a `Rd`-object after the `commit`.

As for the method rule, we have already argued that the above method type enforces that m is an inspector. Furthermore, the type forbids that m assigns the `value` array to the heap, because the qualifier hierarchy does not have a greatest element. Note that method types of the form $U\ m($`(`$\bar{T}\ \bar{x}$`)` `Any{...}` do not prevent representation exposure, because they enable writing the `value` array to `Any`-fields, which is dangerous in an open-world. Similarly, if the `value` field were annotated with `Rd` instead of `myaccess`, the `value` array could be written to `Rd`-fields or `Any`-fields.

2.6 Threads

For type soundness in multi-threaded programs, we must ensure that thread-shared objects are initialized, i.e., they must have types `Rd`, `RdWr` or `Any`, but not `Fresh`. This suffices for soundness, because types of initialized objects never change. As all thread-shared objects are reachable from the sharing `Thread`-objects and as the initialized region is closed under reachability[7], it suffices to require that `Thread`-objects are initialized when threads get started. Furthermore, we must assume this fact as the precondition for verifying the body of `Thread.run()`:

```
class Thread {
    void run() RdWr { }
    void start(); // Treated specially. Type system uses run()'s type.
}
```

[7] In this discussion, we ignore Java Generics. See [17] for a discussion of generics.

Subclasses of `Thread` may override `run()` with receiver qualifier `RdWr` or `Any` (by contravariance)[8]. Calling `start()` on a receiver o, whose static type is a subtype `MyThread` of `Thread`, requires that o has `run()`'s receiver qualifier from `MyThread`. Note that treating `Thread.start()` specially is not a random special case, because conceptually `Thread.start()` is a *concurrency primitive for dynamic thread creation* (a.k.a. `fork` or `spawn`), which is always treated specially in verification systems for concurrency.

3 The Formal Model

We formalize our system for a model language that is deliberately simple. The main objective is to prove soundness of the flexible initialization technique in a very simple setting, to describe the local inference algorithm in the small as a high-level blueprint for an implementation, and to prove soundness of the inference algorithm. Our simple language is based on recursively defined records with nominal types, recursive function definitions, and a simple command language. We include conditionals and while-loops, because the type system and the associated inference algorithm are flow-sensitive, and so branching and repetition are interesting.

Mathematical Notation. Let $X \to Y$ be the set of functions from X to Y, and $X \rightharpoonup Y$ the set of partial functions, and $\mathsf{SetOf}(X)$ the set of all subsets of X. Functions $f \in X \rightharpoonup Y$ induce functions in $\hat{f} \in \mathsf{SetOf}(X) \to \mathsf{SetOf}(Y)$: $\hat{f}(X') = \{f(x) \mid x \in X' \cap \mathsf{dom}(f)\}$. We usually omit the hat when the context resolves ambiguities. For $f \in X \rightharpoonup Y$ and Z some set, let $f|Z$ be the restriction of f to Z: $f|Z = \{(x,y) \in f \mid x \in Z\}$. For $f \in X \rightharpoonup Y$ and $g \in Y \to Z$, let $g \circ f = \{(x, g(f(x))) \mid x \in \mathsf{dom}(f)\}$. Note that $g \circ f \in X \rightharpoonup Z$. For $f, g \in X \rightharpoonup Y$, let $f[g] = g \cup (f | \{x \mid x \notin \mathsf{dom}(g)\})$. Let $x \mapsto y = \{(x,y)\}$. We write $f, x \mapsto y$ instead of $f[x \mapsto y]$ when we want to indicate that $x \notin \mathsf{dom}(f)$. If f is a type environment, we write $f[x : y]$ and $f, x : y$ instead of $f[x \mapsto y]$ and $f, x \mapsto y$. We write π_1 and π_2 for the first and second projection that map pairs to their components.

3.1 A Model Programming Language with Access Qualifiers

Our model is based on records. We refer to named record types as classes, and to records as objects. Record types are of the form $q\,C$, where q is an access qualifier and C a class identifier. The `void`-type has only one element, namely `null`. We define a mapping that erases qualifiers from types: $|q\,C| = C$ and $|\texttt{void}| = \texttt{void}$. *Subqualifying* is the least partial order such that `Rd` $<:$ `Any` and `RdWr` $<:$ `Any`. *Subtyping* is the least partial order such that $p\,C <: q\,C$ for all $p <: q$. A *class table* is a set of class declarations for distinct class identifiers. Class declarations may be (mutually) recursive. A *method table* is a set of (mutually) recursive function declarations for distinct identifiers. The syntax of the model language is shown below. The identifiers x and n in the forms $(C\ x; e)$ and $(\texttt{newtoken}\ n; e)$ are binders with scope e, and we identify expressions up to renaming of bound identifiers[9]. The judgment in Figure 5 formalizes boundedness (writing $\triangleleft$ for `extends`) and ensures that arguments n of `Fresh(`n`)` represent initialization tokens.

[8] It would also be sound to use `Rd` as the receiver qualifier for `Thread.run()`. However, this would be too restrictive, because it would globally enforce that threads never write to fields of their `Thread`-objects.

[9] See also the remark on the operational semantics of `newtoken` at the end of Section 3.2.

$\Delta ::= \varepsilon \mid \Delta, \alpha \triangleleft B \mid \Delta, n : \texttt{Token}$ *(qualifier environments)*

$$\frac{}{\Delta, \alpha \triangleleft B, \Delta' \vdash \alpha \triangleleft B} \qquad \frac{}{\Delta, n : \texttt{Token}, \Delta' \vdash n : \texttt{Token}}$$

$$\frac{q <: \texttt{Any}}{\Delta \vdash q \triangleleft \texttt{Any}} \qquad \frac{\Delta \vdash q \triangleleft B}{\Delta \vdash q \triangleleft \texttt{Qual}} \qquad \frac{}{\Delta \vdash \texttt{RdWr} \triangleleft \texttt{Writeable}} \qquad \frac{\Delta \vdash n : \texttt{Token}}{\Delta \vdash \texttt{Fresh}(n) \triangleleft \texttt{Writeable}}$$

Fig. 5. Qualifier typing, $\Delta \vdash q \triangleleft B$ and $\Delta \vdash n : \texttt{Token}$

$n, o \in \mathsf{Name}$ *(names)* $\quad \alpha, \beta \in \mathsf{QVar}$ *(qualifier variables, including* `myaccess`*)*

$p, q \in \mathsf{Qual} ::= \texttt{Rd} \mid \texttt{RdWr} \mid \texttt{Any} \mid \texttt{Fresh}(n) \mid \alpha$ *(access qualifiers)*

$f, g \in \mathsf{FieldId}$ *(field identifiers)* $\quad C, D \in \mathsf{ClassId}$ *(class identifiers)*

$class ::= \texttt{class}\, C\, \{\bar{T}\, \bar{f}\}$ *(class declarations)* $\quad T \in \mathsf{Ty} ::= q\, C \mid \texttt{void}$ *(types)*

$B \in \mathsf{QualBound} ::= \texttt{Writeable} \mid \texttt{Any} \mid \texttt{Qual}$ *(qualifier bounds)*

$m \in \mathsf{MethodId}$ *(method identifiers)* $\quad x \in \mathsf{Var}$ *(local variables)*

$method ::= \texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\, T\, m(\bar{T}\, \bar{x})\{e\}$ *(method declarations)*

$v \in \mathsf{OpenVal} ::= \texttt{null} \mid n \mid x$ *(open values)*

$e \in \mathsf{Exp} ::= v \mid C\, x; e \mid \texttt{newtoken}\, n; e \mid h; e$ *(expressions)*

$h \in \mathsf{HdExp} ::= x\texttt{=}v \mid x\texttt{=}v.f \mid v.f\texttt{=}v \mid x\texttt{=<}\bar{q}\texttt{>}m(\bar{v}) \mid x\texttt{=new}\, q\, C \mid \texttt{if}\, v\, e\, e \mid \texttt{while}\, v\, e \mid \texttt{commit}\, \texttt{Fresh}(n)\, \texttt{as}\, q$ *(head expressions)*

Derived form, $e; e'$: $\quad v; e \triangleq e \quad (h; e); e' \triangleq h; (e; e') \quad (C\, x; e); e' \triangleq C\, x; (e; e')$ if x not free in e'

$(\texttt{newtoken}\, n; e); e' \triangleq \texttt{newtoken}\, n; (e; e')$ if n not free in e'

Derived form, $e;$: $\quad e; \triangleq e; \texttt{null}$

Note that declarations of local variables associate a class C with the variable, but no access qualifier q. The reason for this design choice is that local variables may change their qualifier at commit-statements. We would find it misleading if our system fixed an access qualifier for a local variable at its declaration site, even though later the variable refers to objects with incompatible access qualifiers.

Our system also permits qualifier changes at assignments to local variables. This seems a natural design choice, given that we have flexible qualifiers for local variables anyway. When a local variable x is used, the type system assumes the access qualifier of the object that most recently got assigned to x. For instance, assuming a context where local variables `r` and `w` have types `Rd Point` and `RdWr Point`, respectively:

```
Point p; p=w;      // now p has type RdWr Point
p.x=42;            // this typechecks
p=r;               // now p has type Rd Point
p.x=42;            // type error: illegal write to Rd-object
```

3.2 Operational Semantics

Heaps are functions from names to objects. Each object is tagged with an access qualifier. These tags are auxiliary state in the sense that they have no effect on concrete program state or control flow, that is, they are erasable. The operational semantics also tracks the pool of tokens that have so far been generated. Token pools are erasable.

$$\mathsf{v} \in \mathsf{Val} ::= \texttt{null} \mid n \qquad obj \in \mathsf{Object} \triangleq \mathsf{Qual} \times (\mathsf{FieldId} \rightharpoonup \mathsf{Val}) ::= q\{\bar{f}\texttt{=}\bar{\mathsf{v}}\}$$

$$h \in \mathsf{Heap} \triangleq \mathsf{Name} \rightharpoonup \mathsf{Object} \qquad t \in \mathsf{TokenPool} \triangleq \mathsf{SetOf}(\mathsf{Name})$$

Commit-environments are functions from names to access qualifiers. They are used to track `Fresh`-qualifiers that have been committed.

(Red Dcl)
$(\sigma, C\ x; e) :: s,h,t \to ((\sigma, x \mapsto \texttt{null}), e) :: s,h,t$

(Red New Token) $n \notin t$
$(\sigma, \texttt{newtoken}\ n; e) :: s,h,t \to (\sigma, e) :: s,h,t \cup \{n\}$

(Red Set Local)
$(\sigma, x\texttt{=}v; e) :: s,h,t \to (\sigma[x \mapsto \sigma(v)], e) :: s,h,t$

(Red Get) $v \neq \texttt{null} \quad \sigma(v) = n$
$(\sigma, x\texttt{=}v.f; e) :: s,h,t \to (\sigma[x \mapsto \pi_2(h(n))(f)], e) :: s,h,t$

(Red Set) $v \neq \texttt{null} \quad \sigma(v) = n$
$(\sigma, v.f\texttt{=}w; e) :: s,h,t \to (\sigma, e) :: s, h[n \mapsto (\pi_1(h(n)), \pi_2(h(n))[f \mapsto \sigma(w)]\,)], t$

(Red Call) $\texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\ U\ m(\bar{T}\ \bar{x})\texttt{\{}e'\texttt{\}}$
$(\sigma, x\texttt{=<}\bar{q}\texttt{>}m(\bar{v}); e) :: s,h,t \to (\bar{x} \mapsto \sigma(\bar{v}), e'[\bar{q}/\bar{\alpha}]) :: (\sigma, x\texttt{=<}\bar{q}\texttt{>}m(\bar{v}); e) :: s,h,t$

(Red Return)
$(\sigma, w) :: (\sigma', x\texttt{=<}\bar{q}\texttt{>}m(\bar{v}); e) :: s,h,t \to (\sigma'[x \mapsto \sigma(w)], e) :: s,h,t$

(Red New) $\texttt{class}\ C\ \texttt{\{}\bar{T}\ \bar{f}\texttt{\}} \quad n \notin \mathsf{dom}(h)$
$(\sigma, x\texttt{=new}\ q\ C; e) :: s,h,t \to (\sigma[x \mapsto n], e) :: s, (h, n \mapsto q\texttt{\{}\bar{f}\texttt{=null\}}), t$

(Red If True) $\sigma(v) = \texttt{null}$
$(\sigma, (\texttt{if}\ v\ e\ e'); e'') :: s,h,t \to (\sigma, e; e'') :: s,h,t$

(Red If False) $\sigma(v) \neq \texttt{null}$
$(\sigma, (\texttt{if}\ v\ e\ e'); e'') :: s,h,t \to (\sigma, e'; e'') :: s,h,t$

(Red While True) $\sigma(v) = \texttt{null}$
$(\sigma, (\texttt{while}\ v\ e); e') :: s,h,t \to (\sigma, e; (\texttt{while}\ v\ e); e') :: s,h,t$

(Red While False) $\sigma(v) \neq \texttt{null}$
$(\sigma, (\texttt{while}\ v\ e); e') :: s,h,t \to (\sigma, e') :: s,h,t$

(Red Commit) $\delta = (n \mapsto q)$
$(\sigma, \texttt{commit Fresh(}n\texttt{) as}\ q; e) :: s,h,t \to (\sigma, e) :: s, (\delta \circ h), t$

Fig. 6. Operational semantics

$$\delta \in \mathsf{CommitEnv} \triangleq \mathsf{Name} \rightharpoonup \mathsf{Qual}$$

Commit-environments δ induce functions $\hat{\delta}$ in $\mathsf{Qual} \to \mathsf{Qual}$, $\mathsf{Ty} \to \mathsf{Ty}$ and $\mathsf{Object} \to \mathsf{Object}$: $\hat{\delta}(\texttt{Fresh(}n\texttt{)}) = q$ if $\delta(n) = q$, $\hat{\delta}(q) = q$ otherwise; $\hat{\delta}(q\,C) = \hat{\delta}(q)\,C$, $\hat{\delta}(\texttt{void}) = \texttt{void}$; $\hat{\delta}(q\texttt{\{}\bar{f}\texttt{=}\bar{v}\texttt{\}}) = \hat{\delta}(q)\texttt{\{}\bar{f}\texttt{=}\bar{v}\texttt{\}}$. If the context resolves ambiguities, we omit the hat.

A *stack frame* is a pair of a local store σ and an expression e:

$$\sigma \in \mathsf{Var} \rightharpoonup \mathsf{Val} \qquad fr \in \mathsf{Frame} \triangleq (\mathsf{Var} \rightharpoonup \mathsf{Val}) \times \mathsf{Exp} \qquad s \in \mathsf{Stack} ::= \mathsf{nil} \mid fr :: s$$

We extend the domain of functions σ to $\mathsf{OpenVal}$, by setting $\sigma(\nu) = \nu$ for $\nu \in \mathsf{Val}$.

Configurations are triples of stacks, heaps and token pools.

$$cfg \in \mathsf{Configuration} \triangleq \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{TokenPool}$$

The rules in Figure 6 define the small-step operational semantics on configurations. In the rules (Red Dcl) and (Red New Token), we implicitly use a bound-variable convention that allows us to rename bound variables and names appropriately.

3.3 Type System

A *type environment* is a function from variables and names to types.

$$\iota \in \mathsf{Var} \cup \mathsf{Name} \qquad \Gamma \in \mathsf{TyEnv} \triangleq (\mathsf{Var} \cup \mathsf{Name}) \rightharpoonup \mathsf{Ty}$$

Let $\Gamma <: \Gamma'$ whenever $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ and $\Gamma(\iota) <: \Gamma'(\iota)$ for all ι in $\mathsf{dom}(\Gamma)$. We extend the domain of type environments to include $\texttt{null}$: $\Gamma(\texttt{null}) = \texttt{void}$.

We define: $\Delta \vdash q : \mathsf{ok}$ iff $\Delta \vdash q \triangleleft \texttt{Qual}$; $C : \mathsf{ok}$ iff C is declared; $\Delta \vdash q\,C : \mathsf{ok}$ iff $\Delta \vdash q : \mathsf{ok}$ and $C : \mathsf{ok}$; $\Delta \vdash \texttt{void} : \mathsf{ok}$ always; $\Delta \vdash \Gamma : \mathsf{ok}$ iff $\Delta \vdash \Gamma(\iota) : \mathsf{ok}$ for all ι in $\mathsf{dom}(\Gamma)$; $\Delta \vdash \delta : \mathsf{ok}$ iff $\Delta \vdash n : \texttt{Token}$ and $\Delta \vdash \delta(n) : \mathsf{ok}$ for all x in $\mathsf{dom}(\delta)$.

Typing judgments for expressions have the following formats:

$$\Sigma \vdash \{\Gamma, \delta\} e : T \{\Gamma', \delta'\} \qquad \Sigma \vdash \{\Gamma, \delta\} h \{\Gamma', \delta'\}$$

(Γ,δ) represents the configuration before executing the expression, and (Γ',δ') the one afterwards. We refer to (Γ,δ) as the precondition of the expression, and to (Γ',δ') as its postcondition. Recall that we permit local variables to change the qualifier components of their types. This is why we need to include type environments in postconditions. We write $\Delta;\Gamma \vdash v : T$ to abbreviate $\Delta \vdash \{\Gamma,\emptyset\}v : T\{\Gamma,\emptyset\}$.

Now we can present the typing rules for expressions:

$$\text{(Null)}\ \frac{\Delta \vdash \Gamma,\delta,T : \mathtt{ok}}{\Delta \vdash \{\Gamma,\delta\}\mathtt{null} : T\{\Gamma,\delta\}} \qquad \text{(Id)}\ \frac{\Delta \vdash \Gamma,\delta : \mathtt{ok}}{\Delta \vdash \{\Gamma,\delta\}\iota : \Gamma(\iota)\{\Gamma,\delta\}} \qquad \text{(Sub)}\ \frac{\Delta \vdash U,\Gamma'' : \mathtt{ok} \quad T <: U \quad \Delta \vdash \{\Gamma,\delta\}e : T\{\Gamma',\delta'\} \quad \Gamma' <: \Gamma''}{\Delta \vdash \{\Gamma,\delta\}e : U\{\Gamma'',\delta'\}}$$

$$\text{(Dcl)}\ \frac{\Delta \vdash q\,C : \mathtt{ok} \quad \delta(q) = q \quad \Delta \vdash \{(\Gamma,x : q\,C),\delta\}e : T\{(\Gamma',x : U),\delta'\}}{\Delta \vdash \{\Gamma,\delta\}C\ x;e : T\{\Gamma',\delta'\}} \qquad \text{(Seq)}\quad \Delta \vdash \Gamma,\delta : \mathtt{ok}\ \ \frac{\Delta \vdash \{\Gamma,\delta\}h\{\Gamma',\delta'\} \quad \Delta \vdash \{\Gamma',\delta'\}e : T\{\Gamma'',\delta''\}}{\Delta \vdash \{\Gamma,\delta\}h;e : T\{\Gamma'',\delta''\}}$$

$$\text{(New Token)}\ \frac{\Delta \vdash \Gamma,\delta,\Gamma',\delta' : \mathtt{ok} \quad \Delta,n : \mathtt{Token} \vdash \{\Gamma,(\delta,n \mapsto \mathtt{Fresh}(n))\}e : T\{\Gamma',(\delta',n \mapsto q)\}}{\Delta \vdash \{\Gamma,\delta\}\mathtt{newtoken}\ n;e : T\{\Gamma',\delta'\}}$$

In the rule (Dcl), we assume that the newly declared local variable initially has type $q\,C$, where q can be chosen appropriately. An automatic typechecker needs to delay the choice of an appropriate q until the new variable first gets assigned to. This delayed choice of q is subsumed by the inference algorithm in Section 3.4. The premise $\delta(q) = q$ ensures that q is not a previously committed `Fresh`-qualifier.

In the typing rules for head expressions, note that we update the qualifiers of local variables after assignments, implementing flexible qualifiers of local variables, as discussed earlier. Crucially, the rule (Set) checks that the object is writeable:

$$\text{(Set Local)}\ \frac{|\Gamma(v)| = |\Gamma(x)|}{\Delta \vdash \{\Gamma,\delta\}x\mathtt{=}v\{\Gamma[x : \Gamma(v)],\delta\}} \qquad \text{(Get)}\quad \mathtt{class}\ C\ \mathtt{\{..}\,T\,f\,\mathtt{..\}}\ \ \frac{\Gamma(v) = q\,C \quad U = T[q/\mathtt{myaccess}] \quad |U| = |\Gamma(x)|}{\Delta \vdash \{\Gamma,\delta\}x\mathtt{=}v.f\{\Gamma[x : U],\delta\}}$$

$$\text{(Set)}\quad \mathtt{class}\ C\ \mathtt{\{..}\,T\,f\,\mathtt{..\}}\ \ \frac{\Gamma(v) = q\,C \quad \Delta \vdash q \lhd \mathtt{Writeable} \quad \Delta;\Gamma \vdash w : T[q/\mathtt{myaccess}]}{\Delta \vdash \{\Gamma,\delta\}v.f\mathtt{=}w\{\Gamma,\delta\}}$$

$$\text{(Call)}\quad \mathtt{<}\bar{\alpha} \lhd \bar{B}\mathtt{>}\ U\ m(\bar{T}\ \bar{x})\mathtt{\{}e\mathtt{\}}\ \ \frac{\delta(\bar{q}) = \bar{q} \quad \Delta \vdash \bar{q} \lhd \bar{B} \quad \Delta;\Gamma \vdash \bar{v} : \bar{T}[\bar{q}/\bar{\alpha}] \quad V = U[\bar{q}/\bar{\alpha}] \quad |V| = |\Gamma(x)|}{\Delta \vdash \{\Gamma,\delta\}x\mathtt{=<}\bar{q}\mathtt{>}m(\bar{v})\,\{\Gamma[x : V],\delta\}} \qquad \text{(New)}\ \frac{\Delta \vdash q\,C : \mathtt{ok} \quad \delta(q) = q \quad C = |\Gamma(x)|}{\Delta \vdash \{\Gamma,\delta\}x\mathtt{=new}\ q\,C\{\Gamma[x : q\,C],\delta\}}$$

$$\text{(If)}\ \frac{\Delta;\Gamma \vdash v : T \quad \Delta \vdash \{\Gamma,\delta\}e : \mathtt{void}\{\Gamma',\delta'\} \quad \Delta \vdash \{\Gamma,\delta\}e' : \mathtt{void}\{\Gamma',\delta'\}}{\Delta \vdash \{\Gamma,\delta\}\mathtt{if}\ v\ e\ e'\{\Gamma',\delta'\}} \qquad \text{(While)}\ \frac{\Delta;\Gamma \vdash v : T \quad \Delta \vdash \{\Gamma,\delta\}e : \mathtt{void}\{\Gamma,\delta\}}{\Delta \vdash \{\Gamma,\delta\}\mathtt{while}\ v\ e\{\Gamma,\delta\}}$$

$$\text{(Commit)}\ \frac{\delta(n) = \mathtt{Fresh}(n) \quad \Delta \vdash q : \mathtt{ok} \quad \delta(q) = q \quad \delta' = n \mapsto q}{\Delta \vdash \{\Gamma,\delta\}\mathtt{commit\ Fresh}(n)\ \mathtt{as}\ q\{\delta' \circ \Gamma,\delta' \circ \delta\}}$$

In the (While) rule, note that the environments are an invariant for the loop body. Consequently, it is disallowed to commit inside a loop body a token that was generated outside the loop body (as this would modify the commit-environment). On the other hand, it is allowed to commit tokens that were generated inside the loop body, because the rule (New Token) removes such tokens from pre- and postcondtions.

For checking class and method declarations, we use the following rules:

$$\text{(Class)}\ \frac{\mathtt{myaccess} \lhd \mathtt{Qual} \vdash \bar{T} : \mathtt{ok}}{\mathtt{class}\ C\ \mathtt{\{}\,\bar{T}\ \bar{f}\,\mathtt{\}} : \mathtt{ok}} \qquad \text{(Method)}\ \frac{\bar{\alpha} \lhd \bar{B} \vdash U,\bar{T} : \mathtt{ok} \quad \bar{\alpha} \lhd \bar{B} \vdash \{\bar{x} : \bar{T},\emptyset\}e : U\{\Gamma,\emptyset\}}{\mathtt{<}\bar{\alpha} \lhd \bar{B}\mathtt{>}\ U\ m(\bar{T}\ \bar{x})\mathtt{\{}e\mathtt{\}} : \mathtt{ok}}$$

Well-typed stack frames, $\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T$ **and** $\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T \to U$**:**

$$\frac{\Delta,\Delta';\Gamma,\Gamma' \vdash \sigma : \Gamma'' \quad \Delta,\Delta' \vdash \{\Gamma'',\delta\} e : T\{\Gamma''',\delta'\} \quad \mathsf{dom}(\delta) \subseteq \mathsf{dom}(\Delta') \quad \delta \circ \Gamma'' = \Gamma'''}{\Delta;\Delta';\Gamma;\Gamma' \vdash (\sigma, e) : T}$$

$$\frac{fr = (\sigma, x\texttt{=<}\bar{q}\texttt{>}m(\bar{v}); e) \quad \Delta;\Delta';\Gamma;\Gamma' \vdash fr : U \quad \texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\ T\ m(\bar{V}\ \bar{x})\{e'\}}{\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T[\bar{q}/\bar{\alpha}] \to U} \qquad \frac{\Delta \vdash \Gamma : \mathsf{ok} \quad (\forall x \in \mathsf{dom}(\sigma))(\Delta;\Gamma \vdash \sigma(x) : \Gamma'(x))}{\Delta;\Gamma \vdash \sigma : \Gamma'}$$

Well-typed stacks, $\Delta;\Gamma \vdash s : \mathsf{ok}$ **and** $\Delta;\Gamma \vdash s : T \to \mathsf{ok}$**:**

$$\frac{\Delta \vdash \Gamma, T : \mathsf{ok}}{\Delta;\Gamma \vdash \mathsf{nil} : T \to \mathsf{ok}} \qquad \frac{\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T \quad \Delta;\Gamma \vdash s : T \to \mathsf{ok}}{\Delta,\Delta';\Gamma,\Gamma' \vdash fr :: s : \mathsf{ok}} \qquad \frac{\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T \to U \quad \Delta;\Gamma \vdash s : U \to \mathsf{ok}}{\Delta,\Delta';\Gamma,\Gamma' \vdash fr :: s : T \to \mathsf{ok}}$$

Well-typed objects, $\Delta;\Gamma \vdash obj : T$**:**

$$\frac{\texttt{class}\ C\ \{\bar{T}\ \bar{f}\} \quad \Delta;\Gamma \vdash \bar{v} : \bar{T}[q/\texttt{myaccess}]}{\Delta;\Gamma \vdash q\{\bar{f}\texttt{=}\bar{v}\} : q\ C}$$

Well-typed heaps, $\Delta;\Gamma \vdash h : \mathsf{ok}$**:**

$$\frac{\mathsf{dom}(\Gamma) = \mathsf{dom}(h) \quad (\forall n \in \mathsf{dom}(h))(\Delta;\Gamma \vdash h(n) : \Gamma(n))}{\Delta;\Gamma \vdash h : \mathsf{ok}}$$

Well-typed token pools, $\Delta \vdash t : \mathsf{ok}$**:**

$$\frac{\mathsf{dom}(\Delta) = \mathsf{dom}(t) \quad (\forall n \in t)(\Delta \vdash n : \texttt{Token})}{\Delta \vdash t : \mathsf{ok}}$$

Well-typed configurations, $cfg : \mathsf{ok}$**:**

$$\frac{\Delta;\Gamma \vdash s : \mathsf{ok} \quad \Delta;\Gamma \vdash h : \mathsf{ok} \quad \Delta \vdash t : \mathsf{ok}}{s,h,t : \mathsf{ok}}$$

Fig. 7. Typing rules for configurations

Soundness. We extend the type system to configurations, as shown in Figure 7. The judgment for stack frames has the format $\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T$. The type T is the type of the return value. Whereas Δ and Γ account for tokens and objects that are known to stack frames below *fr*, the environments Δ' and Γ' account for tokens and objects that have been generated in *fr* or in stack frames that were previously above *fr* and have been popped off the stack. The premise $\mathsf{dom}(\delta) \subseteq \mathsf{dom}(\Delta')$ in the first typing rule for stack frames captures formally that the commit-environment for the top frame never contains initialization tokens that have been generated in the rest of the stack. This is important for the soundness of (Commit). Another judgment for stack frames has the form $\Delta;\Delta';\Gamma;\Gamma' \vdash fr : T \to U$. Intuitively, it holds when $\Delta;\Delta';\Gamma;\Gamma' \vdash fr : U$ and in addition *fr* currently waits for the termination of a method call that returns a value of type T.

We can now prove the following preservation theorem:

Theorem 1 (Preservation). *If* $cfg : \mathsf{ok}$ *and* $cfg \to cfg'$*, then* $cfg' : \mathsf{ok}$*.*

The proof of the preservation theorem is mostly routine and contained in the companion report [17]. The following theorem says that the type system is sound for object immutability: *well-typed programs never write to fields of* `Rd`*-objects*. The theorem is a simple corollary of the preservation theorem and the fact that a configuration is ill-typed when the head expression of its top frame instructs to write to a field of a `Rd`-object.

Theorem 2 (Soundness for Object Immutability). *If* $cfg : \mathsf{ok}$*,* $cfg \to^* (\sigma, v.f\texttt{=}w; e) :: s,h,t$ *and* $\sigma(v) = n$*, then* $\pi_1(h(n)) \neq \texttt{Rd}$*.*

3.4 Local Annotation Inference

Figure 8 presents the syntax for annotation-free expressions E, as obtained from the expression syntax by omitting the specification statements `newtoken` and `commit`, as

well as the qualifier arguments at call sites and the qualifier annotations at object creation sites. The function $e \mapsto |e|$ erases specification commands and annotations from annotated expressions. This section presents an algorithm that infers the erased information, deciding the following question: Given Δ, Γ, E, T such that $\Delta \vdash \Gamma, T : \mathsf{ok}$. Are there e, Γ' such that $|e| = E$ and $\Delta \vdash \{\Gamma, \emptyset\} e : T \{\Gamma', \emptyset\}$?

We have proven that our algorithm answers this question soundly: if the inference algorithm answers "yes", then the answer to this question is indeed "yes". We believe that the converse also holds (completeness), but cannot claim a rigorous proof. The algorithm constructs an annotated expression e whose erasure is E. An implementation does not have to really construct e, because knowing that e exists suffices. There are, of course, many annotated expressions that erase to the same annotation-free expression. So what is the strategy for inserting the specification commands without restricting generality? Conceptually, the algorithm parses the unannotated E from left to right, inserting specification commands `newtoken` and `commit` as needed.

Inserting Commits. For commits, we use a lazy strategy and only insert a `commit` if this is strictly necessary. For instance, we never insert commits in front of local variable assignment, because commits and local variable assignments can always be commuted without breaking well-typedness or changing the erasure. The spots where commits do get inserted are: (1) in front of field assignments when a value of type `Fresh(`n`)` is assigned to a field of type q where $q \neq$ `Fresh(`n`)`, (2) in front of method calls when the method signature forces to commit types of arguments, (3) in front of the return value when the return type forces to commit the type of the return value, (4) at the end of conditional branches to match commits that have been performed in the other branch, (5) at the end of loop bodies (for tokens generated inside the loop) to establish the loop invariant, and (6) in front of loop entries (for tokens generated outside the loop) to establish the loop invariant. Consider the following example with a while-loop:

```
void r(Rd C x);     void w(RdWr C x);     <a ◁ Writeable> f (a C x);
C x; x = new C; while x ( f(x); w(x); );
```

Generated annotated expression:

```
newtoken m; newtoken n; C x; x = new Fresh(n) C;
commit Fresh(n) as RdWr; while x ( <RdWr>f(x); w(x); );
commit Fresh(m) as Any;
```

In the above expression, the method call `w(x)` inside the loop body forces a commit in front of the loop.[10] In contrast, the following expression does not typecheck, because the loop body forces `x` to have both a `Writeable` type and type `Rd`, which is impossible.

```
C x; x = new C; while x ( f(x); r(x); ); // TYPE ERROR
```

One could deal with while-loops by a fixed point computation that requires two iterations over the loop body, one to discover a candidate loop invariant and another one to check if the candidate grants the access permissions required by the loop body. Our algorithm is syntax-directed, because this is simpler to implement on top of the JSR 308 checkers framework [23].

[10] Technically, the inference algorithm delays the generation of the prefix `newtoken m;newtoken n;` and the postfix `commit Fresh(m) as Any`. These get inserted at the top level, see Theorem 3.

$E \in \mathsf{AfreeExp} ::= v \mid C\ x;E \mid H;E$ *(annotation-free expressions)*

$H \in \mathsf{AfreeHdExp} ::= x{=}v \mid x{=}v.f \mid v.f{=}v \mid x{=}m(\bar{v}) \mid x{=}\texttt{new}\ C \mid \texttt{if}\ v\ E\ E \mid \texttt{while}\ v\ E$ *(annotation-free head expressions)*

$|\cdot| : \mathsf{Exp} \to \mathsf{AfreeExp}$

$|v| \triangleq v \quad |C\ x;e| \triangleq C\ x;|e| \quad |\texttt{newtoken}\ n;e| \triangleq |e| \quad |\texttt{commit Fresh}(n)\ \texttt{as}\ q;e| \triangleq |e|$

$|h;e| \triangleq |h|;|e|$, if $h \neq \texttt{commit Fresh}(\_)\ \texttt{as}\ \_$

$|\cdot| : \mathsf{HdExp} \to \mathsf{AfreeHdExp}$

$|x{=}\texttt{<}\bar{q}\texttt{>}m(\bar{v})| \triangleq x{=}m(\bar{v}) \quad |x{=}\texttt{new}\ q\ C| \triangleq x{=}\texttt{new}\ C \quad |\texttt{if}\ v\ E\ E'| \triangleq \texttt{if}\ v\ |E|\ |E'|$

$|\texttt{while}\ v\ E| \triangleq \texttt{while}\ v\ |E| \quad |h| \triangleq h$, otherwise

Fig. 8. Annotation-free expressions and erasure

Generating Tokens. Concerning the generation of initialization tokens, there are two questions to answer. Firstly, when does the algorithm generate new initialization tokens, and secondly, where does the algorithm insert the `newtoken` statements that bind the tokens. Generation happens (1) at variable declaration sites, (2) at object creation sites, and (3) at call sites for instantiation of qualifier parameters that occur in the method return type but not in the method parameter types. At such sites, the algorithm generates a new token n and uses `Fresh(`n`)` as the type of the newly declared variable, the newly created object or the method return value. In the above example, `m` and `n` are the tokens that were generated at the variable declaration site for `x` and at the object creation site that follows it. Note that tokens generated at variable creation sites often do not occur in the program text. Using `Fresh(`n`)` as the qualifier for newly created objects (and similarly for variable declarations and method returns) is no restriction, because the following type- and erasure-preserving transformation replaces qualifiers q at object creation sites by `Fresh(`n`)`:

$$x{=}\texttt{new}\ q\ C \ \rightarrow\ \texttt{newtoken}\ n;x{=}\texttt{new Fresh}(n)\ C;\texttt{commit Fresh}(n)\ \texttt{as}\ q$$

As for where to insert `newtoken`, observe that these can always be pulled out of conditional branches by the following type- and erasure-preserving transformation:

$\texttt{if}\ v\ (\texttt{newtoken}\ n;e)\ e' \ \rightarrow\ \texttt{newtoken}\ n;\texttt{if}\ v\ e\ (e';\texttt{commit Fresh}(n)\ \texttt{as}\ \delta(n);)$
where δ is the commit environment in the postcondition of e (as found in the type derivation)

We cannot pull `newtoken` out of loops, though, because the typing rules prevent loop bodies to commit tokens that were generated outside the loop. Consider the following variation of the earlier example:

```
C x; while x ( x = new C; f(x); r(x); );
```

In contrast to the erroneous expression further up, this expression is well-typed. The inference algorithm generates the following annotated expression for it:

```
newtoken m; C x; commit Fresh(m) as Rd; while x (
  newtoken n; x = new Fresh(n) C; <Fresh(n)>f(x);
  commit Fresh(n) as Rd; r(x); );
```

The `newtoken` command commutes with all other commands, and therefore the inference algorithm generates `newtoken` at the beginning of loop bodies only (leaving token generation at the beginning of method bodies implicit).

$\boxed{f;g}$ $\quad f;g \triangleq (g \circ f) \cup g \quad$ if $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset$

$\boxed{ts \in \mathsf{Scopes} ::= t \mid t :: ts}$ $\quad |t| \triangleq t \quad |t :: ts| \triangleq t \cup |ts| \quad \mathsf{rest}(t) \triangleq \emptyset \quad \mathsf{rest}(t :: ts) \triangleq |ts|$

$$\begin{array}{rcll} \mathsf{newtokens}(t);e & \triangleq & \texttt{newtoken}\, n_1;\ldots;\texttt{newtoken}\, n_k;e & \text{if } t = \{n_1,\ldots,n_k\} \\ \mathsf{commit}(\delta) & \triangleq & \texttt{commit Fresh}(n_1)\ \texttt{as}\ q_1;\ldots;\texttt{commit Fresh}(n_k)\ \texttt{as}\ q_k; & \text{if } \delta = \{n_1 \mapsto q_1,\ldots,n_k \mapsto q_k\} \end{array}$$

Fig. 9. Helpers

Subqualifying Constraints. To deal with subqualifying the inference algorithm generates subqualifying constraints. We extend qualifiers by existential variables:

$$?\alpha \in \mathsf{ExVar} \quad \textit{(existential variables)} \qquad p,q \in \mathsf{Qual} ::= \cdots \mid ?\alpha \qquad \Delta \vdash ?\alpha \triangleleft \texttt{Qual}$$

We partition the set of qualifiers into the sets PQual of *persistent qualifiers* and TQual of *transient qualifiers*:

$$\mathsf{TQual} \triangleq \{\texttt{Fresh}(n) \mid n \in \mathsf{Name}\} \qquad \mathsf{PQual} \triangleq \mathsf{Qual} \setminus \mathsf{TQual}$$

A *substitution* is a function from existential variables to closed persistent qualifiers:

$$\rho \in \mathsf{Subst} \triangleq \mathsf{ExVar} \rightharpoonup (\mathsf{PQual} \setminus \mathsf{ExVar})$$

Note that existential variables range over persistent qualifiers only. Substitutions ρ induce functions $\hat{\rho}$ in $\mathsf{PQual} \to \mathsf{PQual}$: $\hat{\rho}(?\alpha) = \rho(?\alpha)$ if $?\alpha \in \mathsf{dom}(\rho)$; $\hat{\rho}(q) = q$ otherwise. Let $\hat{\rho}(T)$ (resp. $\hat{\rho}(e)$) denote the type (resp. expression) obtained by substituting all qualifier occurrences q by $\hat{\rho}(q)$. We omit the hat when no ambiguities arise.

A *constraint set* contains pairs of the forms (q,B) and (p,q):

$$\mathcal{C} \in \mathsf{Constraints} \triangleq \mathsf{SetOf}(\mathsf{PQual} \times \mathsf{QualBound} \cup \mathsf{PQual} \times \mathsf{PQual})$$

A Δ-*solution* of a constraint set $\mathcal{C}$ is substitution ρ such that $\Delta \vdash \rho(q) \triangleleft B$ and $\rho(p) <: \rho(q)$ for all (q,B), (p,q) in $\mathcal{C}$.

Inference Algorithm. The inference judgment has the following format, where ts, Γ, δ_{pre} and T are inherited attributes, and the other attributes are synthesized.

$$ts;\Gamma \vdash E : T \Downarrow (\Gamma', \delta, ts', t, \mathcal{C})^{\,\mathsf{for}\,(\delta_{pre} \vdash e)}$$

The synthesized annotated expression e is such that $|E| = e$. An implementation does not need to compute e or track δ_{pre}, as the other attributes do not depend on them.

- (Γ, δ_{pre}) represents the precondition for e.
- $(\Gamma', (\delta_{pre}; \delta))$ represents the postcondition for e.
- ts contains the tokens in scope before e. ts has a stack structure that reflects the nesting of enclosing while loops.
- ts' contains the tokens in scope after e.
- t contains all tokens n in $\mathsf{rest}(ts')$ such that the type derivation for e has a leaf of the form $\Delta \vdash \texttt{Fresh}(n) \triangleleft \texttt{Writeable}$. These tokens must be tracked because they cannot be committed to Rd in front of enclosing while-loops. (See the example on page 19.)
- $\mathcal{C}$ are the subqualifying constraints required for well-typedness of e.

For the details of the inference algorithm we refer to our report [17], where the following soundness theorem is proven:

Theorem 3 (Soundness of Inference). *Suppose* $\mathsf{ran}(\Delta) \subseteq \mathsf{QualBound}$, $(\Delta \vdash \Gamma, T : \mathsf{ok})$, Γ, *T do not contain existential variables,* $\emptyset; \Gamma \vdash E : T \Downarrow (\Gamma', \_, t, \_, \mathcal{C})^{\mathsf{for}\,(\emptyset \vdash e)}$ *and* ρ Δ-*solves* $\mathcal{C}$. *Then* $(\Delta \vdash \{\Gamma, \emptyset\}\mathsf{newtokens}(t); \rho(e); \mathsf{commit}(\delta) : T\{(\delta; \rho) \circ \Gamma', \emptyset\})$ *for* $\delta = \{(n, \texttt{Any}) \mid n \in t, \hat{\delta}(n) = \texttt{Fresh(}n\texttt{)}\}$.

4 Related Work

Immutability. Our type system supports class immutability, object immutability, and read-only references, allows flexible object initialization, and is simple and direct (building only on the access qualifiers `Rd`, `RdWr` and `Any`). To the best of our knowledge, no existing type system for a Java-like language meets all these goals at once: Our earlier system Jimuva [18] supports object immutability and open-world class immutability, but requires immutable objects to be initialized inside constructors and does not meet the goal of simplicity and directness, as it requires ownership types, effect annotations and anonymity annotations in addition to access qualifiers. IGJ [34] is simple, direct and supports both object immutability and read-only references, but requires immutable objects to be initialized inside constructors and its support for deep immutability is limited. For instance, IGJ has no way of enforcing that the character array inside an immutable string is part of the string and should thus be immutable. This would either require immutable arrays or a special treatment of owned mutable subobjects, neither of which IGJ supports[11]. SafeJava [4] and Joe_3 [22] are ownership type systems that support immutable objects with long initialization phases, where the transition from "uninitialized" to "initialized" is allowed through unique object references. In order to maintain uniqueness they use destructive reads, which is a rather unnatural programming style in Java-like languages. These systems build on top of expressive ownership type systems, thus violating our design goals of simplicity and directness. Frozen objects [20] support immutable objects with long initialization phases, but builds on the Boogie verification methodology [1], so is not suitable for an independent pluggable type system. The Universe type system [21] features read-only references. In particular, Generic Universe Types [12] support covariant class parameters if the main modifier of the supertype is `Any` (which is essentially what we and IGJ [34] do).

Unkel and Lam [31] automatically infer stationary fields, i.e., fields that may turn immutable outside constructors and after previous assignments, and thus are not necessarily `final`. Their fully automatic analysis requires the whole program. It only detects fields that turn stationary before their objects have been written to the heap, and is in this respect more restrictive than our system, which can deal with stack-local *regions*, as needed for initializing cyclic structures. On the other hand, our system only works at the granularity of objects. Interestingly, non-`final` stationary fields are reportedly much more common than `final` fields.

Our system does not address *temporary immutability*, which would require heavier techniques in order to track aliasing on the heap. On an experimental level, statically

[11] IGJ supports immutable arrays initialized by array initializers. This is not enough to check the `String`-constructor `String(char[] c)`, because the length of `c` is not known statically.

checking temporary immutability has been addressed by Pechtchanski and Sarkar [24]. On a theoretical level, it is very nicely supported by fractional permissions [5].

Object confinement and ownership. For open-world class immutability, we use qualifier polymorphism to express several confinement properties. Firstly, we express a variant of so-called anonymous methods [32] in terms of qualifier polymorphism. Anonymous methods do not write `this` to the heap. Our variant of anonymity for constructors of immutable classes is slightly weaker and forbids that `this` is written to the heap outside the `Fresh` region in which the instance of the immutable class is constructed. Secondly, by combining the `myaccess` class parameter with conditions on method types, we can express that representation objects of immutable objects are encapsulated, thus avoiding the need to include both access qualifiers *and* ownership annotations in the system. To this end, we make use of qualifier-polymorphic methods, similar to owner-polymorphic methods in ownership type systems [9,4,33,27,18].

It is not clear if the `myaccess` parameter alone is enough to express tree-structured ownership hierarchies in general, as facilitated in parametric ownership type systems (e.g., [8], [4]) through instantiating the `owner` class parameter by `rep` or `this`, and in the Universe type system [21] through the `rep`-modifier. Potanin's system `FGJ+c` for package-level confinement [26] is based on a static set of owner constants (formally similar to `Rd` and `RdWr` but without the additional access semantics). It seems that very similar confinement properties as in `FGJ+c` could be expressed purely in terms of qualifier-polymorphic methods and without the owner constants. A subtle difference, however, is this: `FGJ+c`, as most ownership type systems, allows methods to return confined objects, ensuring safety by preventing "outside" class clients from calling such methods. Our system, on the other hand, prevents methods from returning confined objects in the first place. In an open world, where class clients may not follow the rules of the pluggable type system, the latter is the only safe choice.

Type systems for flexible object initialization. There are several articles on initialization techniques for non-nullness type systems [13,14,28]. Fähndrich and Xia's system of "delayed types" [14] is most closely related to our work, like us using lexically scoped regions for safe typestate changes, and using a class parameter representing a "delay time", similar to our `myaccess` parameter. Unlike us, Fähndrich and Xia do not address local annotation inference. Our system is considerably simpler than theirs, because the initialization problem for immutability seems inherently simpler than the initialization problem for object invariants. Intuitively, there are two reasons for this: Firstly, whereas for object immutability the end of the initialization phase is merely associated with the disposal of a write permission, for object invariants it is associated with an *obligation* to prove the invariant. Secondly, a major complication in [14] is the need to permit inserting uninitialized objects into initialized data structures. This is essential to satisfactorily support cyclic data structures, but requires the use of existential types. Fortunately, this complication does not arise for immutability, because no objects (whether uninitialized or not) ever get inserted into *immutable* data structures.

J\mask [28] is a type-and-effect system for reasoning about object initialization. It is based on a rich language for specifying partial object initialization, including primitives for expressing that fields may or must be uninitialized, as well as conditional assertions. It is designed to guarantee that well-typed programs never read uninitial-

ized fields. It is not designed for immutability, and consequently offers no support for specifying deep immutability or object confinement, as needed for object and class immutability. J\mask (based on a rich specification language for partial object initialization) is quite different in nature to Fähndrich and Xia's delayed types (based on a variant of lexically scoped regions combined with dependent types). Qi and Myers rightly claim that J\mask supports some initialization patterns that delayed types do not, giving bottom-up initialization of trees with parent pointers as an example where delayed types cannot establish object invariants in the required order. This example causes no problems for our immutability system, see Figure 3. In fact, our annotations for this example avoid conditional assertions and are thus simpler than J\mask's (but this comparison is not quite fair, as J\mask and our system have different goals).

Lexically scoped regions. Stack-local regions are closely related to lexically scoped regions [30] for region-based memory management (see also [16]). Whereas, in region-based memory management, lexical scoping is used to statically determine when memory regions can safely be deallocated, here we use it to statically determine when the types of memory regions can safely be changed. Lexically scoped regions do not have a separate `commit`-statement, but associate the end of region lifetimes with the end of region name scopes. We opted for a separate `commit`-statement, because it simplifies the description of our inference algorithm, which works by a left-to-right pass over the abstract syntax tree, inserting `commit`s when field or method types enforce this.

5 Conclusion

We presented a pluggable type system for immutable classes, immutable objects, and read-only references. The system supports flexible initialization outside constructors by means of stack-local regions. Our system shows, for the first time, that support for the various forms of immutability, including open-world class immutability, is possible without building on top of an expressive ownership type system (though the class parameter `myaccess` effectively provides some notion of confinement) and without using effect annotations or unique references. A lesson we have learned is that parametric qualifier polymorphism is a very expressive tool, both for flexibility and confinement.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
2. K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. In *OOPSLA*, pages 301–320, 2007.
3. J. Bloch. *Effective Java*. Addison-Wesley, 2001.
4. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.
5. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer-Verlag, 2003.
6. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, London, UK, 2001. Springer-Verlag.
7. J. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *POPL*, pages 283–295, 2005.

8. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
9. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.
10. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, pages 262–275, 1999.
11. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.
12. W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *ECOOP*, pages 28–53, 2007.
13. M. Fähndrich and K.R.M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM Press, 2003.
14. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM, 2007.
15. M. Felleisen and D. Friedman. *A Little Java, A Few Patterns*. MIT Press, 1997.
16. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, pages 282–293, 2002.
17. C. Haack and E. Poll. Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University, Nijmegen, January 2009.
18. C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In *ESOP*, volume 4421 of *LNCS*, pages 347–362. Springer, 2007.
19. JSR 308 Expert Group. Annotations on Java types. Java specification request, Java Community Process, December 2007.
20. K.R.M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *VSTTE*, pages 192–208, 2008.
21. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
22. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In *TOOLS Europe*, pages 178–197, 2008.
23. M. Papi, M. Ali, T. Correa, J. Perkins, and M. Ernst. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis*, pages 201–212, 2008.
24. I. Pechtchanski and V. Sarkar. Immutability specification and applications. *Concurrency and Computation: Practice and Experience*, 17:639–662, 2005.
25. S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON'02*. IBM Press, 2000.
26. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight generic confinement. *J. Funct. Program.*, 16(6):793–811, 2006.
27. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, pages 311–324, 2006.
28. X. Qi and A. Myers. Masked types for sound object initialization. In *POPL*. ACM, 2009.
29. F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, 2000.
30. M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
31. C. Unkel and M. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *POPL*, pages 183–195. ACM, 2008.
32. J. Vitek and B. Bokowski. Confined types in Java. *Softw. Pract. Exper.*, 31(6):507–532, 2001.
33. T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, KTH Stockholm, 2006.
34. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007*, pages 75–84. ACM, 2007.