Generic Abstract Immutability Types

Aosen Xiong
aosen.xiong@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Werner Dietl
wdietl@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Abstract

Generic data structures are representation boundaries: an immutable list may protect its own representation while still carrying mutable payload objects, or it may treat those payloads as part of its protected state. Existing immutability systems support pieces of this story, but the interaction with generics and viewpoint adaptation is subtle. The difficulty is to distinguish protected representation from independently mutable payload while preserving soundness under generic substitution and viewpoint adaptation. This paper presents GAIT (Generic Abstract Immutability Types), an extension of an abstract immutability type system with parametric polymorphism. GAIT uses type-parameter mutability bounds to distinguish representation from payload, and organizes its guarantees into four preservation properties: abstract-state preservation (AS), concrete-state preservation (CS), readonly-state preservation (RS), and transitive-state preservation (TS). We are mechanizing the design in Lean 4 and have implemented a prototype type checker in the EISOP Checker Framework. We evaluate the checker on approximately 26,000 non-comment lines of Java code, including the Java Collections Framework.

Keywords: immutability, type system, generics

1 Introduction

Immutability properties have been studied through readonly references, object immutability, and class immutability for object-oriented (OO) languages [1, 5, 7–9, 11–13, 16–20]. Transitive immutability protects every object reachable from a reference, so its reachable state remains unchanged. However, real programs often need a more flexible guarantee. For example, a logically immutable object may still update a cache, or an access counter; such mutations should not invalidate the immutability guarantee. Abstract immutability [15] captures this distinction by protecting only the object's *abstract state* [17]: its logical fields and the subgraph reachable through them, while auxiliary state may remain mutable.

Generic data structures introduce an additional design choice. An immutable container may protect its own representation while carrying mutable payload objects [14], or it may make those payloads part of its protected representation. The ambiguity appears when the type argument and the container have different mutability types. Consider two possible designs, the first of which GAIT deliberately rejects:

```
@Imm CoupledList<@Mut Person> protectedList = ...;
@Imm Person p1 = protectedList.get(0); // adapts the
     argument

@Imm PayloadList<@Mut Person> payloadList = ...;
@Mut Person p2 = payloadList.get(0);   // element remains
     mutable
```

The container and element qualifiers are the same in both cases; the difference is whether access treats the element as protected representation or as payload. Some immutability type systems and reference-capability systems specify the first behavior: accessing an element from an immutable container adapts the type argument and produces an immutable element view [7, 12]. The second behavior treats the type argument and container independently [4, 19], so the container can be immutable while the payload remains mutable. Existing immutability systems support pieces of this story [17–20], but they do not make the guarantee clear for generic data structures. If an immutable container protects only its representation and permits mutable payloads to escape, what is the formal immutability guarantee? If it also protects the payload, what distinguishes the two behaviors semantically?

GAIT[1] (Generic Abstract Immutability Types) is an extension of PICO, a non-generic abstract immutability type system [18], with parametric polymorphism. It uses type-parameter mutability bounds to express whether the payload is protected. It also defines how generic types interact with viewpoint adaptation. In this paper, we present the design, the core type rules, and four protection modes obtained by choosing different viewpoint adaptation rules. Their corresponding properties are abstract-state preservation (AS), concrete-state preservation (CS), readonly-state preservation (RS), and transitive-state preservation (TS). A prototype type checker implements the design in the EISOP Checker Framework [3]. We evaluate it on approximately 26,000 non-comment lines of Java code, including the Java Collections Framework.

2 Mutability, Assignability, and Abstract State

We introduce the mutability type qualifiers of PICO that GAIT builds on through small examples. The examples use Java-like syntax, but the distinction applies to object-oriented languages more generally. Mutability qualifiers are type-use

[1] Pronounced "gate."

annotations: a reference is mutable (`@Mut`) when it may be used to update an object, immutable (`@Imm`) when it protects the object's state, and readonly (`@RO`) when it may point to either kind of object. Assignability modifiers, introduced below, instead annotate field declarations and govern the field slot.

```
@Mut Box mutBox = new @Mut Box();
@Imm Box immBox = new @Imm Box();
@RO Box box = mutBox;
box = immBox;              // also accepted
immBox = mutBox;           // rejected
```

Classes also declare a mutability bound. Declaring `@Mut class Box` permits only mutable instances, while `@Imm class Box` permits only immutable instances. `@RDM` stands for *receiver dependent mutable* and leaves that choice to the receiver: `@RDM class Box` may be used as `@Mut Box`, `@Imm Box`, or `@RDM Box`. At allocation, `@RDM` is adapted through the active receiver and becomes the receiver's concrete runtime mutability.

Assignability separately controls whether a field slot may be updated. A `@Final` field corresponds to a Java `final` field and cannot be reassigned after initialization, whereas an `@Assignable` field can be reassigned:

```
@RDM class Box {
  @Final @Imm int value = 1;
  @Assignable @Imm int cache = 0;
}

@Mut Box mb = new @Mut Box(...);
mb.value = 2;                    // rejected
mb.cache = 2;                    // accepted
```

Mutability and assignability therefore answer different questions: the former describes the referenced object, while the latter describes the field slot. To introduce transitive immutability, the qualifier system also includes a receiver-dependent qualifier:

```
@RDM class OuterBox {
  @RDA @RDM Box box;
}

@Mut OuterBox mo = new @Mut OuterBox(...);
mo.box = new @Mut Box(...);      // accepted

@Imm OuterBox io = new @Imm OuterBox(...);
@Imm Box b = io.box;
io.box = new @Imm Box(...);      // rejected
```

The `@RDM` qualifier makes the mutability of `box` depend on the receiver: access through `mo` produces a mutable box, whereas access through `io` produces an immutable box. The `@RDA` modifier similarly makes the field slot assignable through a mutable receiver and final otherwise. These receiver-dependent qualifiers are governed by *viewpoint adaptation* [4]. Through a readonly receiver, an `@RDM` field becomes `@Lost`, because the receiver's concrete mutability is unknown.

These declarations also identify abstract state. The non-assignable receiver-dependent `box` field and the subgraph reachable through it define the abstract state protected by an immutable `OuterBox`. An explicitly `@Assignable` cache slot would instead be auxiliary state and could be updated. Thus, immutability protects the logical value without requiring auxiliary state to remain unchanged.

3 Generics and Viewpoint Adaptation

Section 2 showed how viewpoint adaptation interprets `@RDM` and `@RDA` fields through a receiver. Viewpoint adaptation also applies to method parameters and return types. Generics add a second question: when a declared type contains a type parameter, which parts should follow the receiver and which should retain the actual type supplied by the client?

Qualified generic bounds. GAIT extends this story to generics. In particular, type-parameter bounds are declared with mutability qualifiers:

```
@RDM class RDMBox<E extends @RDM Person> { ... }
@RDM class ROBox <E extends @RO  Person> { ... }
```

An `@RDM` bound says that the element is part of the receiver-dependent protected representation. A readonly bound says that the actual type argument is payload: the container structure can be protected while the element object keeps the mutability supplied by the client.

```
@Imm RDMBox<@Imm Person> c1;  // accepted; get exposes @Imm
     Person
@Imm RDMBox<@Mut Person> bad; // rejected by the adapted
     @Imm bound
@Imm ROBox <@Mut Person> c2;  // accepted; get exposes @Mut
     Person
```

The mutability bound restricts which instances are allowed. The `@RDM` bound admits only type arguments compatible with the bound after adapting it through the receiver; under an `@Imm` receiver, the `@RDM` bound becomes an `@Imm` bound and rejects `@Mut` type arguments. The readonly bound permits mutable payload type arguments because the element is not treated as part of the container's protected representation. Thus, GAIT does not obtain the protected-element behavior from Section 1 by silently changing a client-supplied type argument. The representation-coupled `RDMBox` instead rejects `@Mut Person`; the client must supply `@Imm Person`. The payload-oriented `ROBox` admits `@Mut Person` and preserves its mutability.

Protection modes. Qualified bounds identify the representation boundary, but modules may need to protect different amounts of state around that boundary. Consider a store with protected representation, an assignable cache, a generic payload, and an explicitly mutable field:

```
@RDM class Store<E extends @RO Object> {
  @RDA @RDM Rep rep;
  @Assignable @Imm Cache c;  // auxiliary cache
  @Final E payload;
  @Final @Mut Rep alias;      // may alias rep
}
```

The AS mode provides *abstract-state preservation*: it preserves an object's abstract state. An abstract-state field contributes its field slot and, when that slot contains an object reference, the abstract state of the referenced object. A field is classified this way when its slot is non-`@Assignable` and its declared type is headed by `@Imm` or `@RDM`. For a type-variable field, its declared bound must instead be headed by `@Imm` or `@RDM`. Recursively following such fields yields the subgraph included in the receiver's abstract state.

In the example, the `rep` slot and the abstract state of the referenced representation object belong to the store's abstract state. The cache `c` is excluded because it is `@Assignable`. The `payload` field is excluded because the declaration-site `@RO` bound classifies `E` as payload, independently of the actual type argument.

An immutable store is *abstractly immutable* when its abstract state does not change. Thus the ordinary `@Final` modifier prevents reassignment of `payload`, while the AS classification does not protect the payload object; the assignable cache slot may also change. The AS viewpoint adaptation rules enforce this boundary: `@RDM` follows the receiver's mutability, whereas the payload keeps the mutability supplied by the client.

The CS mode provides *concrete-state preservation* by restricting only the assignability rules. It uses the same mutability adaptation, and hence the same representation boundary, as AS, but treats every field as `@Final` if the receiver is not `@Mut`. Thus `c` can no longer be reassigned. The difference between AS and CS is whether explicitly assignable auxiliary slots may change.

The RS mode provides *readonly-state preservation* for a scope entered through a readonly receiver and argument references. An `@RO` reference hides whether its receiver is mutable or immutable, but AS alone does not prevent it from exposing a field that is explicitly `@Mut`. For example, suppose an `@RO` store reference denotes a mutable store whose `alias` field points to the same object as `rep`, or to another object in the store's abstract state. Under AS, reading `alias` would recover an `@Mut` capability to that object. The scope could then mutate the readonly receiver's abstract state even though it started with no mutable root.

To rule out this capability recovery, RS restricts the mutability rules while retaining AS's assignability rules: an explicitly `@Mut` field such as `alias` is weakened to `@Lost` when read through a non-`@Mut` receiver. The resulting immutability guarantee protects objects already present in the method-entry heap and reachable from the readonly receiver and arguments, rather than only the declared `@RDM` representation. Explicitly `@Assignable` slots such as `c` may still be reassigned.

Finally, the TS mode provides *transitive-state preservation* by combining the two restrictions: it uses RS's mutability rules and CS's assignability rules. It therefore neither exposes mutable capabilities into that entry-heap reachable graph nor permits its field slots to be reassigned. The four modes are thus the four combinations of two independent choices: abstract versus transitive mutability protection, and permissive versus strict assignability.

Each protected module scope selects one of these modes. In the core calculus, a method is the representative scoped module: its body is checked in the mode on its declaration. The corresponding preservation property is stated relative to the entry heap; objects allocated during the scope may still be mutated freely.

When a member's declared type is a type-variable use, member access substitutes the actual type argument without adapting that argument through the receiver. The resulting type is then used for mutability and assignability checks.

Summary. Qualified bounds determine which generic fields extend an object's abstract state. Starting from AS, CS restricts assignment and RS restricts mutability through readonly references. TS combines both restrictions; Section 4 formalizes these modes.

4 Programming Language

A Lean 4 development [2] formalizes GAIT's CESK machine [6] and proves progress, mutation safety, and the four preservation properties for a selected core fragment. Its remaining generic cases are being aligned with the presentation here.

Formal overview. Appendix A gives the selected core calculus: generic class and method syntax, static and runtime object subtyping, a CESK transition system, dynamization, mode-indexed viewpoint adaptation, representative typing rules, and continuation typing. A machine state $M = \langle c, \mathrm{T}, \iota_s, h, \kappa \rangle$ records the current control, runtime environment, active receiver, heap, and continuation. Execution may produce a mutation error when a field write fails the runtime assignability check, or an NPE when a member is accessed through null.

Write $M\ \mathsf{OK}_{\geq p}$ when the complete configuration is well typed relative to mode p: the current activation and every activation stored in its continuation are checked in p or a stronger mode, and their static and runtime environments agree with the heap. This condition is preserved across calls and returns, including nested and nonterminating calls. Mutation safety is:

$$M_0\ \mathsf{OK}_{\geq p} \ \wedge\ M_0 \longrightarrow^* M \quad \Longrightarrow \quad \mathsf{control}(M) \neq \mathsf{error}(\mathsf{Mutation}). \tag{Mutation-Safety}$$

GAIT does not prevent null pointer exceptions: an NPE may be reached and becomes terminal after its continuation is unwound. The preservation properties compare an initial heap h with the heap h' at any machine state in the execution. Fresh objects may be allocated and mutated.

AS and CS preservation. Write $\mathsf{ASField}_h(\ell, f)$ when field f has declared modifier `@Final` or `@RDA` and its declared type is headed by `@Imm` or `@RDM`; for a type-variable field, its declared bound must have one of those qualifiers. Those fields and their reachable subgraph define the abstract state of the object at ℓ. An abstract-state edge $\ell \rightsquigarrow_{\mathsf{AS},h} \ell'$ follows a field f when $\mathsf{ASField}_h(\ell, f)$ and $h(\ell.f) = \ell'$. Write $\ell \rightsquigarrow^*_{\mathsf{AS},h} \ell'$ for the reflexive-transitive closure of these edges in the initial heap.

For the theorem statements, $\mathsf{rtype}(h, \ell) = h(\ell)\downarrow_1$. Fix the initial runtime environment $^{\prime}\mathrm{T}$, active receiver ι_s, and continuation κ, and let $M_s = \langle s, {}^{\prime}\mathrm{T}, \iota_s, h, \kappa\rangle$. Then

$$\mathsf{Exec}_p(s, h, h') \triangleq M_s\ \mathsf{OK}_{\geq p} \wedge \exists M'.\ M_s \longrightarrow^* M' \wedge \mathsf{heap}(M') = h'.$$

Thus Exec ranges over all machine states in an execution, whether or not the execution terminates. AS preserves the abstract-state fields, while CS preserves every field of each object in the abstract state:

$$\begin{aligned}&\mathsf{Exec}_{\mathsf{AS}}(s, h, h') \wedge \mathsf{rtype}(h, \ell) = \mathsf{Imm}_r\ _ \\ &\quad \wedge \ell \rightsquigarrow^*_{\mathsf{AS},h} \ell' \wedge \mathsf{ASField}_h(\ell', f) \qquad \text{(AS-Preservation)} \\ &\quad \Longrightarrow h(\ell'.f) = h'(\ell'.f)\end{aligned}$$

$$\begin{aligned}&\mathsf{Exec}_{\mathsf{CS}}(s, h, h') \wedge \mathsf{rtype}(h, \ell) = \mathsf{Imm}_r\ _ \\ &\quad \wedge \ell \rightsquigarrow^*_{\mathsf{AS},h} \ell' \wedge f \in \mathsf{dom}(h(\ell')) \qquad \text{(CS-Preservation)} \\ &\quad \Longrightarrow h(\ell'.f) = h'(\ell'.f)\end{aligned}$$

RS and TS preservation. Write $\ell \rightsquigarrow^*_{\mathsf{TS},h} \ell'$ for ordinary transitive reachability through all object fields in the selected method's entry heap, and let $\mathsf{Roots}({}^{\prime}\mathrm{T}, y, \overline{z})$ be the non-null locations stored in receiver y and arguments $\overline{z}$. RS and TS follow all fields from these roots. RS protects the non-`@Assignable` fields of the reached old objects; TS protects every field. The predicate $\mathsf{NoMut}_{^s\Delta}$ inspects the receiver and parameter types in the generic declaration, before substituting actual type arguments:

$$\begin{aligned}\mathsf{NoMut}_{^s\Delta}(q\ _\langle\overline{T}\rangle) &\iff q \neq \mathsf{Mut} \wedge \bigwedge_i \mathsf{NoMut}_{^s\Delta}(T_i), \\ \mathsf{NoMut}_{^s\Delta}(X) &\iff {}^s\Delta(X) = T_B \wedge \mathsf{NoMut}_{^s\Delta}(T_B).\end{aligned} \qquad \text{(NoMut)}$$

Here ${}^s\Delta$ contains both class and method type-parameter bounds. The definition follows the acyclic bound dependencies guaranteed by class-table well-formedness and is lifted pointwise to sequences and tuples. Thus an X bounded by `@RO` passes even when a caller instantiates it with `@Mut`: the modularly checked body sees only the readonly bound. A bound headed by `@Mut` fails because it lets the body write through X:

```
@RDM class Cell { @RDA @RO Object value; }
@RS <X extends @Mut Cell>
void replace(X x, @RO Object y) { x.value = y; }
```

The assignment changes an old `@RDA` field and therefore violates both the RS and TS guarantees.

RS and TS concern a selected method invocation. Let $s_c = x := y.m\langle\overline{T_m}\rangle(\overline{z})$, where an empty $\overline{T_m}$ denotes an ordinary call, and let $M_c = \langle s_c, {}^{\prime}\mathrm{T}, \iota_s, h, \kappa\rangle$. The call transition $M_c \longrightarrow M_e$ pushes this invocation's call frame. Write $\mathsf{within}(M_e, M')$ when M' occurs before that frame is popped. This includes states inside nested and recursive calls, and an invocation that never returns, but excludes states after the selected invocation returns.

The call frame stores the resolved declaration I. Write $M_c \xrightarrow{\text{call } I} M_e$ when the call-entry transition resolves I. The label exposes the static declaration selected by dynamic dispatch; its mode and context do not affect execution. Write $I.\mathsf{ctx} = ({}^s\Delta, T_r, \overline{T_p})$ for its bound environment, receiver type, and parameter types. Because AS is the weakest mode, $M_c\ \mathsf{OK}_{\geq \mathsf{AS}}$ requires only a generally well-typed caller. The condition $I.\mathsf{mode} = p$ identifies p with the selected method's statically declared mode; it is not a runtime check.

$$\begin{aligned}\mathsf{CallExec}_p(s_c, {}^s\Delta, T_r, \overline{T_p}, h, h') \triangleq \exists I, M_e, M'.\ M_c\ \mathsf{OK}_{\geq \mathsf{AS}} \\ \wedge M_c \xrightarrow{\text{call } I} M_e \wedge I.\mathsf{mode} = p \\ \wedge I.\mathsf{ctx} = ({}^s\Delta, T_r, \overline{T_p}) \\ \wedge \mathsf{within}(M_e, M') \\ \wedge \mathsf{heap}(M') = h'.\end{aligned}$$

For the RS property, ${}^r\mathsf{FType}(h, \ell, f)$ selects f's modifier and runtime type from $\mathsf{rFields}(\mathsf{rtype}(h, \ell))$.

$$\begin{aligned}&\mathsf{NoMut}_{^s\Delta}(T_r, \overline{T_p}) \\ &\quad \wedge \mathsf{CallExec}_{\mathsf{RS}}(s_c, {}^s\Delta, T_r, \overline{T_p}, h, h') \wedge \ell_0 \in \mathsf{Roots}({}^{\prime}\mathrm{T}, y, \overline{z}) \\ &\quad \wedge \ell_0 \rightsquigarrow^*_{\mathsf{TS},h} \ell' \wedge {}^r\mathsf{FType}(h, \ell', f) = a\ _ \\ &\quad \wedge a \neq \mathsf{Assignable} \Longrightarrow h(\ell'.f) = h'(\ell'.f)\end{aligned} \qquad \text{(RS-Preservation)}$$

$$\begin{aligned}&\mathsf{NoMut}_{^s\Delta}(T_r, \overline{T_p}) \\ &\quad \wedge \mathsf{CallExec}_{\mathsf{TS}}(s_c, {}^s\Delta, T_r, \overline{T_p}, h, h') \wedge \ell_0 \in \mathsf{Roots}({}^{\prime}\mathrm{T}, y, \overline{z}) \\ &\quad \wedge \ell_0 \rightsquigarrow^*_{\mathsf{TS},h} \ell' \wedge f \in \mathsf{dom}(h(\ell')) \\ &\qquad \Longrightarrow h(\ell'.f) = h'(\ell'.f).\end{aligned} \qquad \text{(TS-Preservation)}$$

5 Conclusion

GAIT uses qualified bounds to distinguish representation from payload and defines generic viewpoint adaptation across AS/CS/RS/TS. A Lean 4 development mechanizes a selected core calculus; remaining generic cases are being aligned. Our EISOP Checker Framework prototype is evaluated on about 26,000 non-comment lines of Java, including the Java Collections Framework.

References

[1] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 496–506. https://doi.org/10.1109/ICSE.2017.52

[2] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, Cham, Switzerland, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[3] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, Waikiki, Hawaii, USA, 681–690. https://doi.org/10.1145/1985793.1985889

[4] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *European Conference on Object-Oriented Programming* (Berlin, Germany) *(ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 28–53.

[5] Vlastimil Dort and Ondřej Lhoták. 2020. Reference Mutability for DOT. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.18

[6] Mattias Felleisen and D. P. Friedman. 1987. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 314. https://doi.org/10.1145/41625.41654

[7] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. https://doi.org/10.1145/2384616.2384619

[8] Christian Haack and Erik Poll. 2009. Type-Based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, Springer, Berlin, Heidelberg, 520–545. https://doi.org/10.1007/978-3-642-03013-0_24

[9] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 879–896. https://doi.org/10.1145/2384616.2384680

[10] Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/378795.378797

[11] Elad Kinsbruner, Shachar Itzhaky, and Hila Peleg. 2024. Constrictor: Immutability as a Design Concept. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:29. https://doi.org/10.4230/LIPIcs.ECOOP.2024.22

[12] Edward Lee and Ondřej Lhoták. 2023. Simple Reference Immutability for System F. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 857–881.

[13] K Rustan M Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible Immutability with Frozen Objects. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, Springer, Berlin, Heidelberg, 192–208. https://doi.org/10.1007/978-3-540-87873-5_17

[14] Wolfram Pfeifer, Werner Dietl, and Mattias Ulbrich. 2026. A Framework for the Interoperable Specification and Verification of Encapsulated Data Structures. In *Formal Methods*, Augusto Sampaio and Marielle Stoelinga (Eds.). Springer Nature Switzerland, Cham, 90–110.

[15] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, Berlin, Heidelberg, 233–269. https://doi.org/10.1007/978-3-642-36946-9_9

[16] Tobias Runge, Marco Servetto, Alex Potanin, and Ina Schaefer. 2023. Immutability and Encapsulation for Sound OO Information Flow Control. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 3 (mar 2023), 35 pages. https://doi.org/10.1145/3573270

[17] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 211–230. https://doi.org/10.1145/1094811.1094828

[18] Aosen Xiong, Yudi Bai, Haifeng Shi, Lian Sun, Mier Ta, and Werner Dietl. 2026. Transitive, Abstract, and Class Polymorphic Immutability. (2026). Under submission.

[19] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. 2007. Object and Reference Immutability Using Java Generics. In *Foundations of Software Engineering (FSE)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/1287624.1287637

[20] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and Immutability in Generic Java. *ACM Sigplan Notices* 45, 10 (2010), 598–617.

A Supplementary Formal Details

This appendix gives the selected core calculus underlying the guarantees in Section 4. It proceeds from syntax and the runtime model to static viewpoint adaptation, subtyping, the CESK operational semantics, runtime typing, representative static typing rules, and runtime configuration well-formedness.

A.1 Syntax and Runtime Model

Programs consist of class declarations whose method bodies are statements. Statements include variable assignment, field reads, writes, allocation, and method calls. Type-variable names are globally unique throughout a program. We use overlines for sequences and lift judgments pointwise.

$$
\begin{array}{rcl}
q_c & ::= & \mathsf{Mut} \mid \mathsf{Imm} \mid \mathsf{RDM} \\
q & ::= & q_c \mid \mathsf{RO} \mid \mathsf{Lost} \mid \mathsf{Bottom} \\
a & ::= & \mathsf{Assignable} \mid \mathsf{Final} \mid \mathsf{RDA} \\
p & ::= & \mathsf{AS} \mid \mathsf{CS} \mid \mathsf{RS} \mid \mathsf{TS} \\
N & ::= & q\, C\langle \overline{T} \rangle \qquad \text{(non-variable type)} \\
T & ::= & N \mid X \\
P & ::= & \overline{cd} \\
cd & ::= & q_c \ \mathsf{class}\ C\langle \overline{X \ \mathsf{extends}\ N} \rangle \ \mathsf{extends}\ D\langle \overline{T} \rangle\ \{\, \overline{fd}\ \overline{md} \,\} \\
fd & ::= & a\ T\ f \\
md & ::= & p\ \langle \overline{X \ \mathsf{extends}\ N} \rangle\ T\ m(T_r\ \mathsf{this}, \overline{T_p\ x}) \\
 & & \{\overline{T\ x}; s; \mathsf{return}\ x\} \\
s & ::= & \mathsf{skip} \mid x := y \mid x := \mathsf{null} \mid x := y.f \mid y.f := z \\
 & \mid & x := \mathsf{new}\ q_c\, C\langle \overline{T} \rangle(\overline{y}) \mid x := y.m\langle \overline{T} \rangle(\overline{z}) \mid s; s
\end{array}
$$

The distinguished root class `Object` omits the extends clause. Superclass type arguments are therefore explicit in the class table and are available to inherited member lookup. We assume a well-formed class table P OK: inheritance is acyclic, superclasses exist, superclass arguments satisfy their bounds, type-variable bound dependencies are acyclic, inherited field names do not conflict, and overriding methods have compatible generic signatures. Type-parameter bounds are non-variable class types, so a declaration such as X extends Y is not in the core language. A method's protection mode is invariant under overriding, so static signature lookup and dynamic method resolution select the same mode.

We define execution using a CESK-style machine [6]. At runtime, heap objects store a concrete object mutability qualifier and runtime type arguments, following the .NET generics formalization [10].

$$
\begin{array}{rcl}
q_r & ::= & \mathsf{RO}_r \mid \mathsf{Imm}_r \mid \mathsf{Mut}_r \mid \mathsf{Bottom}_r \\
q_{rc} & ::= & \mathsf{Imm}_r \mid \mathsf{Mut}_r \\
R & ::= & q_r\, C\langle \overline{R} \rangle \\
v & ::= & \ell \mid \mathsf{null} \\
o & ::= & (q_{rc}\, C\langle \overline{R} \rangle,\ \overline{f \mapsto v}) \\
h & ::= & \overline{\ell \mapsto o} \\
{}^{r}\Gamma & ::= & \overline{x \mapsto v} \\
E & ::= & \mathsf{Mutation} \mid \mathsf{NPE} \\
c & ::= & s \mid \mathsf{error}(E) \\
\kappa & ::= & \mathsf{halt} \mid \mathsf{seq}(s, \kappa) \mid \mathsf{call}({}^{r}\Gamma, \iota_c, \iota_d, x, I, \kappa) \\
M & ::= & \langle c, {}^{r}\Gamma, \iota_s, h, \kappa \rangle
\end{array}
$$

A machine state records the current control c, runtime environment, receiver address of the current frame ι_s, heap, and continuation κ. ι_s is an ordinary heap location; the subscript s means *self*. A call frame stores the caller environment and receiver, the callee receiver, the destination variable, and the resolved method declaration I. The declaration's mode is static metadata; the CESK transition relation never inspects it. The error Mutation records a failed runtime assignability check. The error NPE records an attempt to read or write a field, or invoke a method, through null. An error performs no further statement effects and propagates through pending sequence and call frames until it reaches halt. The runtime type arguments $\overline{R}$ are produced by dynamization from well-formed static type arguments. ${}^{s}\Gamma$ maps static variables to static types, ${}^{r}\Gamma$ maps runtime variables to values, and ${}^{s}\Delta$ maps static type variables to their bounds. During runtime lookup and call resolution, the auxiliary substitution ${}^{r}\Delta$ maps type variables to concrete runtime types.

A.2 Static Viewpoint Adaptation

The modes vary mutability and assignability adaptation along two independent axes. In $\triangleright_{m,p}$, m selects mutability adaptation and p is the protection mode; $\triangleright_{a,p}$ analogously selects assignability adaptation. We present AS first. Its mutability rules resolve `@RDM` from precise receivers and lose that precision through a readonly receiver:

$$
\begin{array}{rcll}
q \triangleright_{m,\mathsf{AS}} \mathsf{RDM} & = & q & q \neq \mathsf{RO} \\
\mathsf{RO} \triangleright_{m,\mathsf{AS}} \mathsf{RDM} & = & \mathsf{Lost} &
\end{array}
\qquad \text{(AS-Mut-VPA)}
$$

Unlisted AS mutability cases preserve the declared qualifier. Its assignability rules permit writes to `@Assignable` fields and to `@RDA` fields through mutable receivers:

$$
\begin{array}{rcll}
\mathsf{Mut} \triangleright_{a,\mathsf{AS}} \mathsf{RDA} & = & \mathsf{Assignable} & \\
q \triangleright_{a,\mathsf{AS}} \mathsf{Assignable} & = & \mathsf{Assignable} & \\
q \triangleright_{a,\mathsf{AS}} a & = & \mathsf{Final} & \text{otherwise.}
\end{array}
\qquad \text{(AS-Assign-VPA)}
$$

CS changes only assignability, RS changes only mutability, and TS combines both changes:

$$\begin{aligned} \triangleright_{m,\mathrm{CS}} &= \triangleright_{m,\mathrm{AS}} \\ \mathsf{Mut} \triangleright_{a,\mathrm{CS}} a &= \mathsf{Assignable} && (a \in \{\mathsf{Assignable}, \mathsf{RDA}\}) \\ q \triangleright_{a,\mathrm{CS}} a &= \mathsf{Final} && \text{otherwise} \\ q \triangleright_{m,\mathrm{RS}} \mathsf{Mut} &= \mathsf{Lost} && (q \neq \mathsf{Mut}) \\ \triangleright_{a,\mathrm{RS}} &= \triangleright_{a,\mathrm{AS}} \\ \triangleright_{m,\mathrm{TS}} &= \triangleright_{m,\mathrm{RS}} \\ \triangleright_{a,\mathrm{TS}} &= \triangleright_{a,\mathrm{CS}}. \end{aligned}$$

(Mode-VPA)

Full type-as-viewpoint adaptation lifts qualifier adaptation recursively and pointwise to types and sequences. The judgment $q\,C\langle\overline{T}\rangle \triangleright_p S = S'$ reads "adapt S with respect to receiver type $q\,C\langle\overline{T}\rangle$ in mode p." It first recursively adapts the qualifiers written in S through q, leaving a type-variable occurrence X unchanged, and then substitutes the receiver's actual arguments for C's type parameters. Consequently, when a declared member type is X, its actual type argument is substituted without adapting that argument.

A.3 Subtyping

Static qualifier subtyping, written $q \leq_s q'$, is the reflexive closure of

$$\mathsf{Mut}, \mathsf{RDM}, \mathsf{Imm}, \mathsf{Lost} \;\leq_s\; \mathsf{RO}.$$

In addition, $\mathsf{Bottom} \leq_s q$ for every q; bottom is the static type of null and is never a class declaration or allocation qualifier.

Subclassing is separate from qualified-type subtyping. Write $C\langle\overline{X}\rangle \sqsubseteq D\langle\overline{S}\rangle$ for the following three rules, where $\mathsf{super}(C)$ returns the type in C's declared `extends` clause:

$$\frac{\mathsf{classParams}(C) = \overline{X} \qquad \mathsf{super}(C) = D\langle\overline{S}\rangle}{C\langle\overline{X}\rangle \sqsubseteq D\langle\overline{S}\rangle}\ (\textsc{SC-Extends})$$

$$\frac{\mathsf{classParams}(C) = \overline{X}}{C\langle\overline{X}\rangle \sqsubseteq C\langle\overline{X}\rangle}\ (\textsc{SC-Reflexivity})$$

$$\frac{C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{S'}\rangle \qquad C'\langle\overline{X'}\rangle \sqsubseteq D\langle\overline{S''}\rangle}{C\langle\overline{X}\rangle \sqsubseteq D\langle\overline{S''[\overline{S'}/\overline{X'}]}\rangle}\ (\textsc{SC-Transitivity})$$

The transitive rule substitutes the intermediate arguments $\overline{S'}$ for C''s parameters $\overline{X'}$. Subclassing itself does not perform qualifier widening or check class bounds.

Type-argument subtyping is structural:

$$\frac{}{X <:_l X}\ (\textsc{STA-Var})$$

$$\frac{q' = q \;\vee\; (q' = \mathsf{Lost} \wedge q \in \{\mathsf{Mut}, \mathsf{Imm}\}) \qquad \overline{T <:_l U}}{q\,C\langle\overline{T}\rangle <:_l q'\,C\langle\overline{U}\rangle}\ (\textsc{STA-Class})$$

Thus type-variable arguments are invariant. Class-shaped arguments preserve their class and recursively relate their arguments; their head qualifier is preserved, except that `@Mut` and `@Imm` may widen to `@Lost`.

Qualified-type subtyping is the reflexive and transitive relation generated by the following rules:

$$\frac{C\langle\overline{X}\rangle \sqsubseteq D\langle\overline{S}\rangle \qquad q\,C\langle\overline{T}\rangle \triangleright_{\mathrm{AS}} \overline{S} = \overline{V}}{q\,C\langle\overline{T}\rangle <:_s q\,D\langle\overline{V}\rangle}\ (\textsc{ST-Hier})$$

$$\frac{q \leq_s q' \qquad \overline{T <:_l U}}{q\,C\langle\overline{T}\rangle <:_s q'\,C\langle\overline{U}\rangle}\ (\textsc{ST-Qual})$$

$$X <:_s X \qquad X <:_s {}^{s}\Delta(X).$$

(Static-Sub)

In particular, type-argument subtyping does not widen `@RDM`, `@RO`, or the bottom qualifier to `@Lost`.

Runtime object-qualifier subtyping relates the concrete qualifier stored on a heap object to an exposed view:

$$\mathsf{Mut}_r \leq_r \mathsf{Mut}_r, \quad \mathsf{Mut}_r \leq_r \mathsf{RO}_r,$$
$$\mathsf{Imm}_r \leq_r \mathsf{Imm}_r, \quad \mathsf{Imm}_r \leq_r \mathsf{RO}_r.$$

The null value has a Bottom_r-headed runtime type, and Bottom_r is below every runtime reference qualifier. Heap objects still store only Mut_r or Imm_r at their heads. Runtime subtyping reuses the static subclassing relation. Its address-based dynamization form $\mathsf{dyn}_{h,\ell}$ resolves the static superclass arguments using the subclass object's concrete qualifier and runtime arguments. Then

$$\frac{C\langle\overline{X}\rangle \sqsubseteq D\langle\overline{S}\rangle \qquad q_{rc} \leq_r q_r \qquad \exists\overline{q_r'}.\ \mathsf{dyn}_{h,\ell}(\overline{S}, q_{rc}C\langle\overline{R}\rangle, \overline{q_r'}) = \overline{R'}}{q_{rc}\,C\langle\overline{R}\rangle <:_r q_r\,D\langle\overline{R'}\rangle}.\ (\textsc{Runtime-Sub})$$

A.4 Operational Semantics

The transition relation rewrites CESK states. Sequencing stores the second statement in a continuation frame and restores it after the first statement completes:

$$\frac{}{\langle s_1; s_2, \mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle s_1, \mathrm{T}, \iota_s, h, \mathsf{seq}(s_2, \kappa)\rangle}\ (\text{S-Seq})$$

$$\frac{}{\langle \mathsf{skip}, \mathrm{T}, \iota_s, h, \mathsf{seq}(s_2, \kappa)\rangle \longrightarrow \langle s_2, \mathrm{T}, \iota_s, h, \kappa\rangle}\ (\text{S-Seq-Pop})$$

Errors unwind continuations without executing the pending statements. For example, an error discards a sequence frame:

$$\frac{}{\langle \mathsf{error}(E), \mathrm{T}, \iota_s, h, \mathsf{seq}(s_2, \kappa)\rangle \longrightarrow \langle \mathsf{error}(E), \mathrm{T}, \iota_s, h, \kappa\rangle}$$

(S-Error-Seq)

An error crossing a call frame analogously restores the caller environment and receiver while preserving the error. It becomes terminal when the continuation is halt. Allocation dynamizes static type arguments into runtime object type arguments and stores the resulting runtime object type in

the heap. Let $T_n = q_c\, C\langle\overline{T}\rangle$:

$$\frac{{}^{r}\mathrm{T}(\overline{y}) = \overline{v} \qquad \mathsf{dyn}_h(T_n, {}^{r}\mathrm{T}, \emptyset) = q_{rc}\, C\langle\overline{R}\rangle \qquad \mathsf{rFields}(q_{rc}\, C\langle\overline{R}\rangle) = \overline{__f} \qquad o = (q_{rc}\, C\langle\overline{R}\rangle, \overline{f \mapsto v}) \qquad h + o = (h', \ell)}{\langle x := \mathsf{new}\ q_c\, C\langle\overline{T}\rangle(\overline{y}), {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle \mathsf{skip}, {}^{r}\mathrm{T}[x \mapsto \ell], \iota_s, h', \kappa\rangle} \quad \text{(S-New)}$$

$h + o = (h', \ell)$ allocates o at a fresh location ℓ and returns the extended heap h'. The runtime field-layout premise supplies the complete inherited field sequence; $\overline{f \mapsto v}$ is its pointwise initialization. The pointwise initializer check in T-New implicitly requires the initializer and field sequences to have the same arity. Dynamization converts a static type to a runtime type in three stages:

$$\frac{{}^{r}\mathrm{T}(\mathsf{this}) = \iota_s \qquad h(\iota_s) = (q_{rc}\, C\langle\overline{R}\rangle, _) \qquad \mathsf{classParams}(C) = \overline{X} \qquad q_{rc} \triangleright_{m,r} \mathsf{RDM} = q \qquad T[q/\mathsf{RDM}] = T' \qquad \overline{q'_r} \in \{\mathsf{Mut}_r, \mathsf{Imm}_r\}^* \qquad R' = T'[\overline{R}/\overline{X}, \mathsf{Imm}_r/\mathsf{Imm}, \mathsf{Mut}_r/\mathsf{Mut}, \mathsf{RO}_r/\mathsf{RO}, \mathsf{Bottom}_r/\mathsf{Bottom}, \overline{q'_r/\mathsf{Lost}}]}{\mathsf{dyn}_h(T, {}^{r}\mathrm{T}, \overline{q'_r}) = R'} \quad \text{(DYN)}$$

The rule obtains the current receiver from the runtime environment and resolves `@RDM` using that receiver's runtime mutability. Its final bracketed operation is a simultaneous static-to-runtime translation: it replaces the receiver's class parameters by their runtime arguments while mapping the remaining static qualifiers to runtime qualifiers. It is not an ordinary same-sort substitution through a mixed intermediate type. The sequence $\overline{q'_r}$ supplies runtime qualifiers for occurrences of `@Lost`. Each such qualifier is concrete mutable or immutable: `@Lost` records lost static precision about an object, not a runtime readonly or bottom object. Static Bottom maps directly to Bottom_r for typing null; it is not a witness for `@Lost`. DYN is applied only after method resolution and inherited-member lookup have instantiated all non-receiver type variables; the remaining free variables are precisely the active receiver class's parameters $\overline{X}$. Resolution guarantees this precondition before invoking DYN. Runtime adaptation is mode-independent because heap objects store concrete mutability. Mutability adaptation converts that concrete tag back to the corresponding static qualifier when resolving `@RDM`:

$$\mathsf{Mut}_r \triangleright_{m,r} \mathsf{RDM} = \mathsf{Mut} \qquad \mathsf{Imm}_r \triangleright_{m,r} \mathsf{RDM} = \mathsf{Imm}. \quad \text{(Runtime-Mut-VPA)}$$

Runtime assignability adaptation permits an `@RDA` slot only on a mutable object, preserves an explicitly `@Assignable` slot, and otherwise produces `@Final`:

$$\begin{aligned} \mathsf{Mut}_r \triangleright_{a,r} \mathsf{RDA} &= \mathsf{Assignable} \\ q_{rc} \triangleright_{a,r} \mathsf{Assignable} &= \mathsf{Assignable} \\ q_{rc} \triangleright_{a,r} a &= \mathsf{Final} \qquad \text{otherwise.} \end{aligned} \quad \text{(Runtime-Assign-VPA)}$$

The projection $h(\ell)\!\downarrow_1$ returns the runtime type of the object at ℓ, and $\mathsf{fType}(C, f)$ returns the declared modifier and type of field f, including inherited fields:

$$\frac{h(\ell)\!\downarrow_1 = q_{rc}\, C\langle\overline{R}\rangle \qquad \mathsf{fType}(C, f) = a\ _ \qquad q_{rc} \triangleright_{a,r} a = a'}{\mathsf{canAssign}(h, \ell, f) \iff a' = \mathsf{Assignable}} \quad \text{(Can-Assign)}$$

Field write has a normal step when the runtime assignability check succeeds and a mutation-error step when it fails. The target mutation-safety theorem rules out this error branch for well-typed configurations:

$$\frac{{}^{r}\mathrm{T}(y) = \ell \quad {}^{r}\mathrm{T}(z) = v \quad \mathsf{canAssign}(h, \ell, f)}{\langle y.f := z, {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle \mathsf{skip}, {}^{r}\mathrm{T}, \iota_s, h[\ell.f \mapsto v], \kappa\rangle} \quad \text{(S-Write)}$$

$$\frac{{}^{r}\mathrm{T}(y) = \ell \quad \neg\mathsf{canAssign}(h, \ell, f)}{\langle y.f := z, {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle \mathsf{error}(\mathsf{Mutation}), {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle} \quad \text{(S-Write-Err)}$$

$h[\ell.f \mapsto v]$ changes only field f of the object at location ℓ; error(Mutation) is the mutation-error control. It becomes terminal only after the pending continuation frames have been discarded. For example, field access through null produces the other error without inspecting or replacing the pending continuation:

$$\frac{{}^{r}\mathrm{T}(y) = \mathsf{null}}{\langle x := y.f, {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle \mathsf{error}(\mathsf{NPE}), {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle} \quad \text{(S-Read-NPE)}$$

Field writes and calls through null have similar rules.

Method calls use the continuation to retain the caller. The resolved method declaration I supplies the parameter and local-variable names, body, and return variable. The resolution helper obtains the receiver's concrete class and runtime class arguments from $h(\ell)$, searches that class and its superclasses for m, and instantiates the selected declaration with the receiver's runtime class arguments and the dynamized method arguments $\overline{R_m}$, forming ${}^{r}\Delta$. The method arguments are dynamized in the caller state before resolution, because they may mention the caller's class parameters or `@RDM`. Resolution then returns a closed declaration I whose parameter, local, and return types and body are ready for execution. It also retains the selected declaration's mode, type-parameter bound environment, and declaration-site receiver and parameter types. An empty $\overline{T_m}$ denotes an ordinary call. The callee environment is created at method entry:

$$\mathsf{callEnv}(\ell, \overline{v}, I) = [\mathsf{this} \mapsto \ell,\ I.\mathsf{params} \mapsto \overline{v},\ I.\mathsf{locals} \mapsto \overline{\mathsf{null}}].$$

The I.locals mapping contains one null for each declared local. Thus all local bindings exist before the method body takes its first step. Let $s_c = x := y.m\langle\overline{T_m}\rangle(\overline{z})$:

$$\frac{{}^{r}\mathrm{T}(y) = \ell \quad {}^{r}\mathrm{T}(\overline{z}) = \overline{v} \qquad \mathsf{dyn}_h(\overline{T_m}, {}^{r}\mathrm{T}, \emptyset) = \overline{R_m} \qquad \mathsf{resolve}(h, \ell, m, \overline{R_m}) = I \qquad {}^{r}\mathrm{T}_m = \mathsf{callEnv}(\ell, \overline{v}, I) \qquad \kappa' = \mathsf{call}({}^{r}\mathrm{T}, \iota_s, \ell, x, I, \kappa)}{\langle s_c, {}^{r}\mathrm{T}, \iota_s, h, \kappa\rangle \longrightarrow \langle I.\mathsf{body}, {}^{r}\mathrm{T}_m, \ell, h, \kappa'\rangle} \text{ (S-Call)}$$

Write $M_c \xrightarrow{\text{call } I} M_e$ for an instance of S-Call whose resolution premise returns I. This label exposes the selected static declaration to the metatheory; it does not change the machine's execution.

$$\frac{{}^{r}\mathrm{T}(I.\mathsf{return}) = v \qquad \kappa' = \mathsf{call}({}^{r}\mathrm{T}_c, \iota_c, \ell, x, I, \kappa)}{\langle \mathsf{skip}, {}^{r}\mathrm{T}, \ell, h, \kappa'\rangle \longrightarrow \langle \mathsf{skip}, {}^{r}\mathrm{T}_c[x \mapsto v], \iota_c, h, \kappa\rangle} \text{ (S-Return)}$$

Thus nested calls reduce one step at a time: call entry pushes the caller state, and return restores it while copying the result into the destination variable.

A.5 Runtime Typing

Runtime value typing uses dynamization: a static type is converted to a runtime type, and a heap location is valid only when the object's runtime type is a runtime subtype of that target:

$$\frac{\mathsf{Bottom}_r \le_r q_r}{h \vdash \mathsf{null} : q_r\, C\langle\overline{R}\rangle} \text{ (R-Null)}$$

$$\frac{h(\ell) = (q_{rc}\, C\langle\overline{R}\rangle, \overline{f \mapsto v}) \qquad \exists\overline{q_r'}.\ \mathsf{dyn}_h(T, {}^{r}\mathrm{T}, \overline{q_r'}) = R' \qquad q_{rc}\, C\langle\overline{R}\rangle <:_r R'}{{}^{r}\mathrm{T}, h \vdash \ell : T} \text{ (R-Val)}$$

The existential witnesses allow a static `@Lost` view to correspond to the concrete runtime qualifiers of the value being typed. Heap well-formedness checks stored field values against the runtime field layout computed from the object's runtime type. Thus an immutable object with an `@RDM` field stores that field at an immutable runtime type, while a mutable object stores the corresponding mutable runtime type:

$$\frac{h(\ell) = (q_{rc}\, C\langle\overline{R}\rangle, \overline{f \mapsto v}) \qquad \mathsf{rFields}(q_{rc}\, C\langle\overline{R}\rangle) = \overline{a_f\, R_f\, f} \qquad h \vdash \overline{v} : \overline{R_f}}{h \vdash \ell\ \mathsf{OK}} \text{ (WF-Obj)}$$

rFields computes the runtime field layout from the runtime object type; $h \vdash v : R$ is runtime value typing against a runtime type.

A.6 Static Well-Formedness and Typing

Well-formed type uses. Class type well-formedness checks both the use of the class and the compatibility of type arguments against adapted bounds:

$$\frac{\mathsf{lookup}(C) = _ \qquad \mathsf{Bound}(C) = q_b \qquad q \triangleright_{m,\mathrm{AS}} q_b \le_s q \qquad {}^{s}\Delta \vdash \overline{T}\ \mathsf{OK} \qquad \mathsf{aBound}(q\, C\langle\overline{T}\rangle) = \overline{N} \qquad \overline{T} <:_s \overline{N}}{{}^{s}\Delta \vdash q\, C\langle\overline{T}\rangle\ \mathsf{OK}} \text{ (WF-Type)}$$

The lookup fetches the class declaration, whose qualifier is $\mathsf{Bound}(C)$. The premise $q \triangleright_{m,\mathrm{AS}} q_b \le_s q$ checks that the use-site qualifier is a valid view of that class bound; for example, a mutable or immutable class may still be viewed through `@RO`. The premise ${}^{s}\Delta \vdash \overline{T}\ \mathsf{OK}$ checks the actual type arguments pointwise, and $\mathsf{aBound}(q\, C\langle\overline{T}\rangle)$ returns the class's type parameter bounds after adapting them through the use-site qualifier q.

Typing field read and write. Adapted field lookup returns both the adapted field assignability modifier and the field mutability qualifier. Reads check the adapted field type against the destination variable x; writes separate assignability from subtyping:

$$\frac{{}^{s}\Gamma(x) = T_x \qquad {}^{s}\Gamma(y) = T_y \qquad \mathsf{aFType}_p(T_y, f) = _\ T_f \qquad T_f <:_s T_x}{{}^{s}\Gamma \vdash_p x := y.f} \text{ (T-Read)}$$

$$\frac{{}^{s}\Gamma(y) = T_y \qquad {}^{s}\Gamma(z) = T_z \qquad \mathsf{aFType}_p(T_y, f) = a_f\, T_f \qquad a_f = \mathsf{Assignable} \qquad \mathsf{Lost} \notin T_f \qquad T_z <:_s T_f}{{}^{s}\Gamma \vdash_p y.f := z} \text{ (T-Write)}$$

$\mathsf{aFType}_p(T, f)$ looks up field f in the class of T and adapts the field's mutability qualifier and assignability modifier through T's qualifier in mode p. Lookup records the field's declaring class, so after traversing inheritance it instantiates the field with the type arguments corresponding to that owner rather than those of the receiver's immediate class. If the declared field type is a type-variable use, lookup substitutes its actual type argument without adapting that argument; otherwise, the member type is adapted through the receiver. In both cases the field modifier is adapted separately. In particular, static assignability is defined by

$$\mathsf{canAssign}_p(q, a) \iff q \triangleright_{a,p} a = \mathsf{Assignable}. \quad \text{(Static-Can-Assign)}$$

Adapted lookup returns `@Assignable` precisely when the declared modifier passes $\mathsf{canAssign}_p$ for the receiver qualifier. The field-write rule therefore need not adapt it again.

Equation (Can-Assign) defines the runtime counterpart. It is mode-independent because the mode restricts writes statically. The predicate $\mathsf{Lost} \notin T$ means that `@Lost` does not occur anywhere in the type.

Typing allocation and calls. Allocation checks the creation qualifier, adapts field types through the new object's qualifier, and checks the initializer values against those adapted field types:

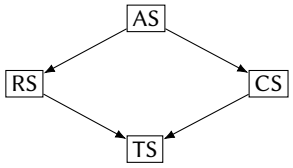
$$\frac{\begin{array}{c}{}^{s}\Gamma(x) = T_x \qquad T_n = q_c\, C\langle\overline{T}\rangle \qquad \mathsf{Lost} \notin T_n \\ q_c \triangleright_{m,\mathsf{AS}} \mathsf{Bound}(C) = q_c \qquad {}^{s}\Delta \vdash T_n\ \mathsf{OK} \\ \mathsf{adaptFields}(T_n) = _\ \overline{T'_f}\ _ \qquad \mathsf{Lost} \notin \overline{T'_f} \\ {}^{s}\Gamma(\overline{y}) <:_s \overline{T'_f} \qquad T_n <:_s T_x\end{array}}{{}^{s}\Gamma \vdash_p x := \mathsf{new}\ q_c\, C\langle\overline{T}\rangle(\overline{y})}\ \text{(T-New)}$$

Here $\mathsf{Bound}(C)$ is the class declaration qualifier of C, and $q_c \triangleright_{m,\mathsf{AS}} \mathsf{Bound}(C) = q_c$ checks that the allocation qualifier is compatible with the class declaration. This allocation-bound adaptation is mode-independent because it determines the runtime mutability of the fresh object rather than access to old protected state. The premise $\mathsf{Lost} \notin T_n$ enforces the allocation-only restriction on `@Lost`. T_n abbreviates the allocated class type $q_c\, C\langle\overline{T}\rangle$. $\mathsf{adaptFields}(T_n)$ maps the complete inherited field layout of C, after substituting superclass type arguments, to the adapted initializer types for a fresh object allocated with qualifier q_c. The pointwise premise ${}^{s}\Gamma(\overline{y}) <:_s \overline{T'_f}$ implicitly requires exactly one initializer for every field in that layout. For a well-formed, `@Lost`-free allocation type, dynamizing the types returned by adaptFields yields the same field names, modifiers, and runtime types as rFields of the resulting runtime object type. This correspondence connects the initializer checks in T-New to the layout installed by S-New.

Method calls use one adapted signature lookup. Empty method-type arguments give ordinary calls; nonempty method-type arguments instantiate method-generic calls:

$$\frac{\begin{array}{c}{}^{s}\Gamma(x) = T_x \qquad {}^{s}\Gamma(y) = T_y \\ \mathsf{aMSig}_p(T_y, m\langle\overline{T_m}\rangle) = (\overline{N_m},\ p_m,\ T_o\ m(T_r\ \mathsf{this}, \overline{T_p})) \\ \mathsf{allowsCall}(p, p_m) \qquad {}^{s}\Delta \vdash \overline{T_m}\ \mathsf{OK} \qquad \overline{T_m} <:_s \overline{N_m} \\ \mathsf{Lost} \notin \overline{T_m} \qquad \mathsf{Lost} \notin \overline{T_p} \\ T_y <:_s T_r \qquad {}^{s}\Gamma(\overline{z}) <:_s \overline{T_p} \qquad T_o <:_s T_x\end{array}}{{}^{s}\Gamma \vdash_p x := y.m\langle\overline{T_m}\rangle(\overline{z})}\ \text{(T-Call)}$$

The first premises obtain the destination and receiver types. aMSig_p performs one method lookup, instantiates the method type arguments, and adapts the selected declaration through the receiver type. It returns the adapted method bounds $\overline{N_m}$, selected mode p_m, and adapted receiver, parameter, and result types. $\mathsf{allowsCall}(p, p_m)$ checks the mode boundary. The next premises require well-formed method type arguments that satisfy their adapted bounds; both sequences are empty for an ordinary call. Neither explicit method arguments nor adapted parameter types may contain `@Lost`, because call resolution must dynamize them without an additional runtime qualifier witness. Finally, subtyping checks the receiver, arguments, and result against the adapted signature and destination. The mode-specific $\mathsf{NoMut}_{s\Delta}$ assumptions are not call-typing premises; they appear directly in the RS and TS preservation theorems. $\mathsf{allowsCall}(p, p_m)$ requires the callee to be checked in the same or a stronger mode. The arrows show the allowed strengthening steps; a caller may invoke a method at its own node or at any node reachable by following arrows:



A.7 Complete Configuration Typing

A well-typed complete configuration requires both a well-formed current activation and a continuation that can safely resume its saved activations. Write

$$\mathsf{ActOK}_{p_i}(c, {}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h)$$

when the current control is well typed in mode p_i and its static environment, runtime environment, and heap agree. Write $\mathsf{FrameOK}_{p_c}$ when a saved call frame contains a well-typed caller activation and its destination accepts the resolved method's return type. Continuations are checked relative to a baseline mode p:

$$\frac{}{\mathsf{KontOK}_{\succeq p}({}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h, p_i, \mathsf{halt})}\ \text{K-Halt}$$

$$\frac{{}^{s}\Gamma \vdash_{p_i} s_2 \qquad \mathsf{KontOK}_{\succeq p}({}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h, p_i, \kappa)}{\mathsf{KontOK}_{\succeq p}({}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h, p_i, \mathsf{seq}(s_2, \kappa))}\ \text{K-Seq}$$

$$\frac{\begin{array}{c}\mathsf{FrameOK}_{p_c}({}^{s}\Gamma_c, {}^{r}\Gamma_c, \iota_c, \iota_d, x, I, h) \\ \mathsf{allowsCall}(p, p_c) \\ \mathsf{KontOK}_{\succeq p}({}^{s}\Gamma_c, {}^{r}\Gamma_c, \iota_c, h, p_c, \kappa)\end{array}}{\mathsf{KontOK}_{\succeq p}({}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h, p_i, \mathsf{call}({}^{r}\Gamma_c, \iota_c, \iota_d, x, I, \kappa))}\ \text{K-Call.}$$

The caller-to-callee mode condition already follows from T-Call when the enclosing method body is checked. Therefore, K-Call records only that the saved caller is in the baseline mode p or a stronger one. For $M = \langle c, {}^{r}\Gamma, \iota_s, h, \kappa\rangle$, the complete configuration judgment is

$$\begin{aligned} M\ \mathsf{OK}_{\succeq p} \iff \exists {}^{s}\Gamma, p_i.\ & \mathsf{allowsCall}(p, p_i) \\ & \wedge\ \mathsf{ActOK}_{p_i}(c, {}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h) \\ & \wedge\ \mathsf{KontOK}_{\succeq p}({}^{s}\Gamma, {}^{r}\Gamma, \iota_s, h, p_i, \kappa). \end{aligned}\ \text{(Config-OK)}$$

The modes in these judgments are typing indices, not components of a machine state. At a call, configuration typing uses the callee's statically declared mode; at return, it again uses the saved caller's mode. Hence every activation in an arbitrary, possibly infinite execution remains checked in the baseline mode or a stronger one.