

# Type-Safe Monotonic Object Evolution

ALEXANDRA MIRRLEES-BLACK and HAoyu WU, Australian National University, Australia

GREGOR RICHARDS, University of Waterloo, Canada

FABIAN MUEHLBOECK, Australian National University, Australia

Object evolution is a monotonic approach to typestate and object reclassification, enforcing that objects may gain, but not lose properties, to permit aliasing. We present a formalization and prototype implementation of our new language MAY, featuring inheritance-based evolution that changes the run-time class of an object to a subclass. To statically guarantee evolution succeeds, we introduce a simple affine permission system for ensuring evolvable references match the run-time type of an object. Furthermore, we demonstrate that our system provides an effective and type-safe way of expressing staged operations and complex initialization procedures.

CCS Concepts: • **Software and its engineering** → **Classes and objects**; *Inheritance*; **Object oriented languages**; • **Theory of computation** → **Object oriented constructs**.

Additional Key Words and Phrases: Object Evolution, Object-Oriented, Class-Based, Affine Types

## ACM Reference Format:

Alexandra Mirrlees-Black, Haoyu Wu, Gregor Richards, and Fabian Muehlboeck. 2026. Type-Safe Monotonic Object Evolution. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 144 (April 2026), 27 pages. <https://doi.org/10.1145/3798252>

## 1 Introduction

Typestate [Strom and Yemini 1986] extends the notion of an object’s static type to include its state. An object’s state may change across its lifetime, which can manifest as various properties or methods being available or unavailable according to some specification. Typestate systems have hence been used to model programming patterns that naturally transition between states, such as object protocols [DeLine and Fähndrich 2004b]. Many real-world programs have state transitions:

- Compilers transform abstract syntax trees in stages.
- GUI programs and games switch between views, enable/disable controls, etc.
- APIs for files and network connections have state transitions, e.g., from “open” to “closed”.
- Complex data structures may be initialized in multiple passes.

Typestate enforces and documents state transitions as part of the static type system and allows for checking of state-related program errors, such as invoking a method not currently available on an object. But, because in typestate systems, an object’s changing state can affect its type, aliasing creates a potential problem for type-soundness, typically requiring some form of aliasing control in the type system, such as affine types [Bierhoff and Aldrich 2007; Garcia et al. 2014; Girard 1987; Tov and Pucella 2011]. This in turn makes the type system more complicated and restricts code patterns that would otherwise be valid.

---

Authors’ Contact Information: [Alexandra Mirrlees-Black](mailto:alex.mirrlees-black@anu.edu.au), alex.mirrlees-black@anu.edu.au; [Haoyu Wu](mailto:haoyu.wu@anu.edu.au), Australian National University, Canberra, Australia, haoyu.wu@anu.edu.au; [Gregor Richards](mailto:gregor.richards@uwaterloo.ca), University of Waterloo, Waterloo, Canada, gregor.richards@uwaterloo.ca; [Fabian Muehlboeck](mailto:fabian.muehlboeck@anu.edu.au), Australian National University, Canberra, Australia, fabian.muehlboeck@anu.edu.au.



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/).

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/4-ART144

<https://doi.org/10.1145/3798252>

In this paper, we focus on *monotonic* typestate [Fähndrich and Leino 2003], in the form of *object evolution* [Cohen and Gil 2009]. In a class-based object-oriented language, object evolution allows the typestate of an object to change by reclassifying the object to a subclass of its current class. Since this reclassification maintains any existing object invariants, we retain type soundness. Furthermore, monotonicity in this direction significantly simplifies the aliasing problem: while our technique still requires affine types to track a single *evolvable* reference to an object, any number of *non-evolvable* references may exist concurrently. These evolvable references are still required for type-safe evolution, as evolution is a property of the run-time type, which we need to know statically to evolve an object safely.

While it is a restricted form of typestate, object evolution can model a number of particularly interesting applications, among them AST transformations in compilers, the state of a file handle, and, in particular, more complex initialization patterns. The latter is particularly timely, as null values—often a key language feature used to fill properties before initialization is completed—are falling out of favor. Nullable and optional types do not address the initialization problem well, as when null is used as a stand-in for a yet-to-be-initialized value, that value will become non-null and remain so forever, and yet with nullable or optional types, would still require run-time checks every time it is used. This pattern can be expressed with object evolution by evolving an object from a class without some field, to the subclass that has it. Null no longer needed!

As pointed out by Muehlboeck and Tate [2021], gradual typing, specifically viewed through the migratory typing [Tobin-Hochstadt et al. 2017] lens, needs to take into account the idiomatic differences between common dynamically-typed and statically-typed languages. Although nothing in this paper relates to gradual typing directly, and the language we present is not gradually typed, we believe that its capabilities match the idioms of dynamically typed languages in a way that may be suitable to gradual typing, and this observation was one motivation for this work.

We present a formalization and prototype implementation of MAY, a Java-like class-based object-oriented language with single class inheritance, multiple interface inheritance, and type-safe monotonic object evolution. Key additional elements are special *evolvers*, a variant on constructors enabling the evolution from an instance of a superclass to an instance of the defining class, an affine type system for tracking *evolvable* references, and a way to evolve the type of fields. Evolvable references are interesting because they also provide an escape hatch from subtyping and inheritance in a way that turns out to be useful for state transitions: they allow accessing members and operations that only exist for a particular class, and are not inherited or required for subtyping between classes. This is safe since only non-evolvable references may experience subtyping. In addition, because of the aforementioned similarity between evolving types and initializing them, the resulting system provides a formal basis for discussing safe object initialization.

In summary, we make the following contributions:

- We present a type system (Section 5) and semantics (Section 6) for MAY (Section 3), a language featuring type-safe monotonic object evolution, and prove standard type-safety properties for it (Section 7)
- We describe a prototype implementation of MAY (Section 8)
- We evaluate the usability of the language and its model through a case study (Section 9)

Before those contributions, we will discuss the relevant background, examples, and motivation.

## 2 Background

In current statically-typed class-based object-oriented languages like Java, C#, and Kotlin, the class of an object is fixed at creation for its entire lifetime, and so are its fields and methods. This typically implies that its fields need to be initialized completely at object creation, and at the end

of the constructor, the object's invariants must be fully established. In contrast, dynamic object re-classification allows objects to be created as instances of some class A and later to be changed to instances of some other class B, and potentially further to yet some other class C, and so on.

The consequences of this behavior in an object-oriented language fundamentally revolves around two somewhat intertwined considerations. First: aliasing. There may be multiple typed references to the same object. Changing its class (and thereby its type) requires avoiding the unsoundness that would arise if the new type was now incompatible with the types of existing references to the object. Second, the precise semantics of state transitions and when they are valid. This may be a less obvious problem than the first, but the essential question here is if state transitions are, for example, idempotent, or whether it matters that every individual transition is done only exactly once from a correct state (and in that case, finding a way to somehow guarantee this).

## 2.1 Object Re-Classification

Fickle<sub>II</sub> [Drossopoulou et al. 2002] provides language-level support for the *state pattern* [Gamma et al. 1995]. It designates some classes as *root* classes and others as *state* classes. Root classes are meant to address the aliasing problem: they are the types meant for (potentially aliased) references, and re-classification can never go from a root class or any of its subtypes to a supertype or unrelated type of a root class. Any instance of a root class can always be re-classified as an instance of another state class extending the root class, thereby losing all fields of its current state class and gaining any additional fields of the new state class. State classes cannot be used as types of fields, but can be used as types of local variables. An effect system tracks the occurrence of re-classification events occurring for each root class; the response of the type system to such an effect is that all local types at a subclass of the relevant root class are now typed at the root class. This is because aliasing is not otherwise tracked, so any reference to a subclass of the same root could point to the object whose re-classification caused the effect. This comparatively heavy imprecision loses a lot of potentially useful type information, forcing programmers into specific orders of operations that avoid losing type information as long as possible, and in any case makes it impossible to keep precise state type information around for very long.

Later work used substructural type systems to track aliasing more precisely, but otherwise follows the same ideas. For example, featherweight typestate [Garcia et al. 2014] uses types of the form  $k(D) C$ , where  $k$  is one of `full`, `shared`, or `pure`, and  $C$  and  $D$  are classes such that  $C <: D$ . Here,  $k(D)$  is called the *state guarantee*, with  $D$  being the class that no state transition can go above, and  $k$  being the substructural annotations that control different forms of aliasing and level of access to an object's members [Bierhoff and Aldrich 2007].

## 2.2 Object Evolution

The systems discussed so far allowed objects to change classes arbitrarily, at least within some bounds. Our paper is about the less-deeply explored idea of *object evolution* [Cohen and Gil 2009; Serrano 1999], a monotonic form of typestate where objects can only *evolve* to a subclass of their current class. This immediately takes care of most soundness concerns with respect to aliasing, as any existing references to an object that underwent an evolution step must see it as an instance of one of its supertypes, which is perfectly safe. However, the second consideration above now plays a more significant role: what is the precise semantics of an evolution step?

Assume a class hierarchy where a class A has two direct subclasses it could evolve to, B1 and B2, and B1 has another direct subclass, C. If the declared type of an object is A, we might be able to evolve it into an instance of B1. But what if, at run time, the object is already an instance of B1? Or, worse, an instance of B2? Or an instance of C? Plausible answers for the dynamic semantics of the relevant evolution operations are *failure*, *reset* (except for the B2 case), or *no-op*. For MAY, we

Class Name $C$	Interface Name $I$	Field Name $f$	Method Name $m$	Variable Name $x$
Program	$\mathcal{P} ::= \langle \Delta \mid e \rangle$			
Type	$\tau ::= \mathbb{B} \mid M \mid !M$	Nominal Type	$M ::= C \mid I$	
Argument List	$A ::= \emptyset \mid A, x : \tau$	Program Definitions	$\Delta ::= \emptyset \mid \Delta, c \mid \Delta, i$	
Method Signature	$s ::= g \ m(A) : \tau$	Method Definition	$d ::= s(A) \mapsto e$	
Evolution Attribute	$g ::= \cdot \mid \text{evo}(x)$			
Constructor	$cm ::= \text{constructor}(A)(A) \{e \mid (e, \dots) \mid (e, \dots) \mid e\}$			
Evolution	$em ::= \text{evolver}(A)(A) \{e \mid (e, \dots) \mid e\}$			
Interface Definition	$i ::= \text{interface } I \text{ extends } I, \dots \{s; \dots\}$			
Class Definition	$c ::= \text{class } C(g \ f : \tau, \dots; g \ f : \tau, \dots)_x \text{ extends } C$ implements $I, \dots \{cm; \dots; em; \dots; d; \dots\}$			
Expression	$e ::= x \mid !x \mid \text{true} \mid \text{false} \mid e.f \mid \text{new } C(e, \dots) \mid e == e$ $\mid x = e \mid e.f = e \mid e.m(e, \dots) \mid e; e \mid \text{if } e \text{ then } e \text{ else } e \mid e \Downarrow C(e, \dots)$			

Fig. 1. The abstract syntax for core MAY

choose *failure*; that is, an evolution operation is only valid if the current class of the object is a direct superclass of the target class of the evolution. This matches the design by [Cohen and Gil \[2009\]](#). However, [Cohen and Gil](#) compared evolution to a dynamic cast operation, and thus provided no static way of ensuring that evolution failures would not occur. In contrast, MAY uses affine types to track *evolvability*, guaranteeing statically that every evolution operation will succeed. To our knowledge, this is the first type system for monotonic object evolution with this property.

### 3 Language Introduction and Syntax

The core design of MAY is similar to Java, or any other statically typed class-based object-oriented language with single class inheritance and multiple interface inheritance. But, in addition, in certain circumstances, objects are permitted to *evolve* to a sub-class of their current class at run-time, changing their class, and thus their fields and methods. Objects are referenced as is usual in languages of this family, but in addition, each object may have a single *evolvable* reference, on which *evolution constructors* may be called to evolve the object's class.

The uniqueness of the evolvable reference is ensured statically through affine typing, which has the further implication that an evolvable reference is always given the exact type that the run-time object will have, and not any supertype. This allows us to ensure that the changes to the type that occur at run-time are monotonic, and so we do not incur multiple inheritance, or otherwise complicate typing relationships.

#### 3.1 Syntax

Figure 1 shows the core abstract syntax for MAY. Programs  $\mathcal{P}$  consist of a set of class and interface definitions  $\Delta$ , and a main expression  $e$ . Types are one of: the primitive boolean type  $\mathbb{B}$ , a nominal type  $M$  (an interface name  $I$  or class name  $C$ ), or an evolvable nominal type  $!M$ . The first two are mostly standard, except that classes have two field lists (one for normal fields and one for overridable fields), but the latter provides affine tracking of evolvability, which requires ensuring at most one evolvable reference to an object may exist. Argument lists  $A$  are type-variable pairs, and program definitions  $\Delta$  are lists of class and interface definitions (see below).

Method signatures  $s$  consist of a method name  $m$ , an argument list  $A$ , and a return type  $\tau$ . Crucially, they also contain an evolution attribute  $g$ , either  $\text{nil}(\cdot)$  or  $\text{evo}(x)$ , requiring the method be called on an evolvable reference. The evolvable reference is passed via the local variable  $x$ .

```

class Foo {
    b : bool;
    constructor(b : bool) { this.b = b; }
}

```

(a) A simple base class.

```

class Bar extends Foo {
    c : bool;
    evolver(c : bool) { this.c = c; }
}

```

(b) A simple subclass with an evolver.

Fig. 2. A basic setup for object evolution

To conveniently cope with local variable scoping in our formalization, methods have a second argument list  $A$  which describes all local variables used in the method's body  $e$ .

MAY constructor definitions  $cm$  similarly have one argument list for actual parameters and another for local variables. Their code comes in four parts, the middle two being lists of expressions. The first component, a single  $e$ , may contain initializing code used to define local variables shared between the other expressions. The second component lists an expression for every field defined in this class, initializing them. The third component contains arguments for a superclass constructor call. The last single  $e$  contains any code to be run after the superclass constructor.

Evolver definitions  $em$  are very similar to constructor definitions, but omit the list of superclass constructor arguments; since they are evolving an existing object, it can be safely assumed that the superclass has already been constructed. Interface definitions  $i$  are standard: interfaces may extend several other interfaces, and may describe several method signatures. Class definitions  $c$  define two lists of fields  $f$ : the first is the list of fields introduced by the class; the second is the list of fields overridden by the class. The evolution attribute  $g$  of a field definition indicates whether a field is evolvable, that is, whether it can be overridden to a new type in a subclass. If a field is evolvable, it is restricted to be read-only by default. It can only be changed within an evolver, and the parameter  $x$  of the  $\text{evo}(x)$  attribute defines the name of a local variable containing the value of the field within an evolver; the reason for this will become more clear when talking about other restrictions around evolvability in Section 5. The subscripted variable  $x$  after the field lists defines the name of the variable used for the `this`-argument of methods, constructors, and evolvers. Classes must extend a single other class  $C$  (up to the special predefined class `Object`), and may implement arbitrarily many interfaces. They may also define arbitrarily many constructors  $cm$ , evolvers  $em$ , and methods  $d$ , subject to there only being one of each for each number of arguments in a given class/interface, allowing a simple form of overloading.

Finally, expressions  $e$  are standard with two exceptions. The standard expressions are variable references  $x$ , boolean literals `true` and `false`, field access  $e.f$ , constructor calls  $\text{new } C(e, \dots)$ , polymorphic reference equality  $e == e$ , local variable assignment  $x = e$ , field assignment  $e.f = e$ , method calls  $e.m(e, \dots)$ , sequencing  $e; e$ , and conditional branches `if-then-else`. The two non-standard expressions are evolvable variable references  $!x$  and evolution expressions  $e \Downarrow C(e, \dots)$ , described more in the following examples.

### 3.2 Evolution by Example

The implementation of MAY has a surface language that is less constrained than the core syntax presented here. In the examples here and in Section 4, we adopt syntax closer to that surface language. We omit syntax with obvious defaults, such as `extends Object`, always use `this` as the `this`-argument, allow inline variable declarations, infer evolvable variable references  $!x$ , and use standard field assignment syntax in constructors and evolvers.

**3.2.1 Simple Object Evolution.** Figure 2 shows two classes in a basic setup for object evolution. Class `Bar` extends class `Foo` and defines an evolver.

```

class Zip {
    evolves f : !Foo;
    constructor(f : !Foo) {
        this.f = f;
    }
}

class Zorp extends Zip {
    overrides f : !Bar;
    evolver(c : bool) {
        this.f = this.f evolve Bar(c);
    }
}

```

Fig. 3. Field evolution based on the classes of Figure 2

Based on the definitions in Figure 2, we can write the following code:

```

f1 : !Foo = new Foo(false); // creates a new Foo object
f2 : Foo = f1; // a non-evolvable alias to f1
b : !Bar = f1 evolve Bar(true); // evolves f1 to a Bar object; f1 now inaccessible
r1 : bool = f2 == b; // true (still the same object, reference equality)
r2 : bool = b.b == b.c; // false (b was assigned false, c was assigned true)

```

The above example shows the key feature of MAY: calling an evolver defined in a class on an evolvable reference of the direct superclass. It also shows that we can obtain arbitrary non-evolvable references from evolvable ones, as well as the preservation of object identity through evolution.

What the example does not show so far is behaviors that are statically prevented. For example, including a new line `b2 : !Bar = f1 evolves Bar(true);` would fail, since `f1` becomes inaccessible (or, in our surface language, downgraded to a non-evolvable reference) after being used in the first evolution expression. This reflects the affine behavior of evolvable references.

Furthermore, accessing `f2.c` wherever `f2` is defined also fails, simply because `Foo` does not have such a field. Typically, where an object may only need a field in a later stage, the language necessitates it be there when created, either filled with a default value or given a nullable type. Object evolution makes it possible to do this in a more type-safe way.

**3.2.2 Evolvable Fields.** MAY allows the type of an object's fields to evolve when the object itself evolves. Fields annotated with the `evolves` attribute, like `Zip.f` in Figure 3, may be overridden in a subclass, as it is in the class `Zorp` in the same figure. Omitting the `evolves` attribute in a subclass prevents the field's type from evolving in any further subclasses. This is necessary as evolvability comes at a cost: such fields may not be assigned outside of constructors and evolvers. This is because in any other context (i.e. without an evolvable reference to the object) we cannot know whether the field has been evolved further to a subclass. It would be unsound to allow writing a value of type `!Foo` to field `f` through a reference typed at class `Zip`, because the underlying object might also be an instance of `Zorp`, which requires `f` to be the more specific `!Bar`. Within constructors and evolvers, the current instance type, and thus the required field type, are precisely known, permitting updates. Conversely, if there is any hope of evolving an object's field, we must be able to mutate it somewhere, so that we may provide an instance of the correct subtype during evolution.

For a field to be evolvable, it is not necessary for the field's type to be an evolvable reference. For example, our MAY implementation also features nullable types (written as `?τ`), and a field might evolve from being nullable to not being nullable in a subclass.

**3.2.3 Evolution and Subtyping.** Being able to evolve an object depends on knowing its exact run-time type at the point of evolution. Any variable typed at an evolvable type `!C` must reference an object of precisely that class, not any subclasses. To ensure this invariant holds in the presence of evolution, at most one evolvable reference to an object is permitted to exist at a given time. Only that evolvable reference is permitted to evolve, and so the precise type is known at the point that it changes. By contrast, non-evolvable references still follow regular subtyping as in, e.g., Java.

```

interface AstNode {
    evolves fun typecheck() : !TypedNode;
}
class AddNode implements AstNode {
    evolves left : !AstNode;
    evolves right : !AstNode;
    ...
    evolves fun typecheck() : !TypedNode {
        return this evolves TypedAddNode();
    }
}

interface TypedNode extends AstNode {}
class TypedAddNode extends AddNode
    implements TypedNode {
    overrides left : !TypedNode;
    overrides right : !TypedNode;
    type : Type;
    evolver() {
        this.left = this.left.typecheck();
        this.right = this.right.typecheck();
        this.type = ...;
    }
}

```

Fig. 4. An example of the interplay of subtyping, interfaces, and evolution.

We take care to ensure that data structures using subtype polymorphism cooperate with the requirement that evolution needs known run-time types. MAY allows the `evolves` attribute on methods and method signatures for this purpose. Such methods are special in two ways: their `this`-argument is typed as `!C` instead of `C`, and they are only available on evolvable receivers. In turn, they are not inherited via regular subclassing, since a subclass’s receiver is not of the same exact type at run-time. They can, however, be part of an interface specification, in which case they must be implemented by any classes directly implementing the interface. The method may then be called on evolvable interface references with type `!I`, and any `!C` directly implementing the interface may be typed a `!I`. We use this word *directly* carefully here: classes may implement interfaces through the transitivity of subtyping, but not explicitly state that they do so. These classes *indirectly* implement the interface and are not required to implement `evolves` methods of the interface. In exchange, a `!C` may not be treated as a `!I` despite `C` transitively implementing `I`.

## 4 Motivating Examples

In this section, we present additional example problems that motivated the design of MAY and demonstrate useful cases for object evolution. These examples are not fully developed case studies; for that, see Section 9 and the artifact [Mirrlees-Black et al. 2026b].

### 4.1 Pipelines

Data often moves through a number of analyses in a pipeline. As an example, we use AST nodes in a compiler, and discuss the specific pipeline phase of type checking. This example has been fully implemented as a case study, discussed in Section 9. Here we discuss the language design choices that were made to support it.

Figure 4, based on the case study discussed in Section 9, demonstrates evolvable interface types. On the left, an interface `AstNode` prescribes an `evolves` method to turn instances of the interface into instances of the sub-interface `TypedNode` on the right. Each interface is followed by an example class implementing it, representing an AST node for addition operations and its typed version, respectively. The operands of an addition operation can be arbitrary AST nodes, but those fields must be evolvable if we wish to evolve the whole tree into a typed version of itself. This necessitates evolvable references typed at interfaces. These evolvable references permit calling interface `evolves` methods, dispatching to a concrete method which can invoke the appropriate evolution constructor. In this example, `AddNode` evolves into a `TypedAddNode`, with the evolution constructor evolving and type-checking its operands, to compute its new `type` field.

```

class A {
    public int foo;
    public A(int f) {
        this.foo=f;
        IO.print(get());
    }
    public int get() {
        return foo;
    }
}
class B extends A {
    public int bar;
    public B(int b) {
        super(b);
        this.bar=b+1;
        IO.print(get());
    }
    public int get() {
        return bar;
    }
}

```

(a) Java

```

class A {
    public:
    int foo;
    A(int f): foo(f) {
        std::cout << this->get();
    }
    virtual int get() {
        return foo;
    }
};
class B : public A {
    public:
    int bar;
    B(int b): A(b), bar(b+1) {
        std::cout << this->get();
    }
    virtual int get() override {
        return bar;
    }
};

```

(b) C++

```

class A {
    foo : int;
    constructor(f : int) {
        this.foo = f;
        print(this.get());
    }
    int get() {
        this.foo;
    }
}
class B extends A {
    bar : int;
    constructor(b : int) {
        this.bar = b+1;
        super(b);
        Int.print(this.get());
    }
    int get() {
        this.bar;
    }
}

```

(c) MAY

Fig. 5. A simple superclass constructor example in three languages.

## 4.2 Sound Object Initialization

Consider the three versions of the same program in Figure 5. In each language, we define two classes, where **B** is a subclass of **A**. Each class declares a field: **A.foo**, and **B.bar**, and **A** also defines a method `get`, which **B** overrides. The constructor for each class assigns its own field and prints the result of calling `get` on the current object. In **B**'s case, it also calls the superclass constructor.

Running the code `new B(5)` in each language prints “06” in Java, “56” in C++, and “66” in MAY. This is because of differences in their constructor semantics.

Like MAY, Java sets the run-time class of the object to be the most concrete class of the object being created, in this case **B**. But unlike MAY, Java does not require the fields of the subclasses to be initialized before calling the superclass constructor. In Java, this is not necessarily a soundness problem, as every type has a default value: some variant of `0`, `false`, or `null`. Kotlin [JetBrains 2011], as well as many other newer languages, requires types to explicitly be nullable in order to have a null value, but its constructors work the same as in Java, and hence one can write code where a null pointer exception is thrown after trying to access a field that has a non-nullable type. Fundamentally, the problem here is that it is unsound to allow unrestricted access to the `this` pointer before all fields have been assigned.

C++ does something slightly different: it effectively *evolves* the type of the object as it goes through the constructor hierarchy. The superclass constructor is called first, but while it executes, the object has the type of the current superclass, and so dispatches to its method. MAY can emulate this behavior with *evolvers*, but is more general as the *evolvers* can run at any time. C++'s behavior is equivalent to first constructing an instance of **A**, and then evolving it to an instance of **B**.

As ever more languages move towards null-safe types, sound object initialization deserves more attention. Java's behavior in initialization made it possible to create an unsound program, where the unsoundness depends on `null` values, without any mention of `null` in the code [Amin and Tate 2016]. Safe monotonic object evolution could be a lightweight way to strengthen safety properties

```

class Demuxer {
  url : string; // file name or URL
  constructor(url: string) { this.url = url; }
  evolves fun open() : ?!OpenDemuxer {
    let handle = /* open file */;
    if handle >= 0 {
      this evolves OpenDemuxer(handle);
    } else {
      null
    }
  }
}
// audio streams have sample rates, etc
interface IAudioStream {
  fun sampleRate() : int;
  fun audioFormat() : AudioFormat; //enum
}
// video streams have width, height, etc
interface IVideoStream {
  fun width() : int;
  fun height() : int;
  fun fps() : IntPair;
  fun videoFormat() : VideoFormat; //enum
}

class OpenDemuxer extends Demuxer {
  handle : int; // handle to libav demuxer data
  evolver(handle : int) {
    this.handle = handle;
  }
  evolves fun identifyStream() : unit {
    let audio=/*find audio stream, or -1*/;
    let video=/*find video stream, or -1*/;
    // Evolve to the appropriate types
    if audio >= 0 and video >= 0 {
      this evolves AVStream(audio, video);
    } else if audio >= 0 {
      this evolves AudioStream(audio);
    } else if video >= 0 {
      this evolves VideoStream(video);
    }
  }
  fun readRawPacket() : ?Packet {
    // read an undifferentiated packet...
  }
}
class AudioStream extends OpenDemuxer
  implements IAudioStream {...}
class VideoStream extends OpenDemuxer
  implements IVideoStream {...}
class AVStream extends OpenDemuxer
  implements IAudioStream, IVideoStream {...}

```

Fig. 6. Core components of a binding to the libav libraries using object evolution

around object initialization, while still allowing mutually recursive data structures to be initialized, just in a safe way with explicit phases - at least in the core calculus. A more advanced surface language may automatically derive evolution phases for typical constructors in a way that hides some of the complexity from the user.

### 4.3 Data-Driven Types

Many file types can contain diverse data. Typically, representing such files requires either shuffling data between a variety of objects, or bespoke techniques for representing when certain data is absent or present. As an example, we discuss multimedia files, and compare to the popular libav libraries, a part of FFmpeg [The FFmpeg Project 2000].

An MPEG-4 file, and many other media file types, may contain a video stream, an audio stream, or both. libav's structures have to suffice for any kind of data, so, for example, libav's types for streams, packets, and frames always have a width field, which is simply set to 0 for non-video data. The type system does not aid in correctly distinguishing different types of files, and code that does not correctly check will behave unexpectedly or crash.

Using object evolution, one can express both the stages of initialization and the distinct kinds of data in a more precise, and type-safe, way, by evolving an object as more information becomes known about the multimedia file it represents. Figure 6 shows a basic example of this structure.

Evolvable methods that are not inherited allow an object to decide how to evolve based on other data it has already gathered. In a first step, a `Demuxer` is given a path to a file that it might at

some point open. Opening it through the open method will turn it into an `OpenDemuxer`, which can then work to determine the kind of multimedia stream contained in the file, and turn itself into a corresponding class—`AudioStream`, `VideoStream`, or `AVStream`—that implements interfaces specific to audio, video, or both: `IAudioStream` for audio and/or `IVideoStream` for video.

Because objects of either of these interfaces always represent a file with the right sort of packets, they can expose specialized packet reading methods that pass the same advantage onto those.

For example, the `AudioStream` class might implement `readAudioPacket` as follows:

```
fun readAudioPacket() : ?AudioPacket {
  let ret = this.readRawPacket();
  if (AVPacket_stream_idx(ret) == this.audioStream) {
    // we can be typesafe in derived objects
    return ret evolves AudioPacket(this.sampleRate, this.audioFormat);
  }
  return null;
}
```

For brevity, we have not included the complete type hierarchy for packets or other stages in the multimedia pipeline. As shown in this example, the evolution of types can allow the type system to reflect information about data that can only be determined at run-time while reading the data. To create a differentiated `AudioStream` type otherwise would require transferring the file’s state to a new object, thus losing references.

## 5 Typing Object Evolution

The idea of object evolution and its basic semantics are not new. Our novel contribution is a type system that prevents failed attempts at evolution, as well as the controlled evolution of field types. In this section, we introduce the surface-level typing rules for MAY. In order to discuss these rules concretely, Figures 7 and 8 together define a number of smaller examples, some of which should and some of which should not type-check. Some examples may not make sense immediately, in which case we encourage the reader to move on to the definitions of the typing rules, and come back to the examples after going through the rules.

### 5.1 Notation

We use square brackets  $[\ ]$  to indicate different kinds of substitution. For evaluation contexts, this means inserting an expression into the hole  $\bullet$ . For all kinds of maps, the notation  $M[K \text{ OP } V]$  means to update the entry with key  $K$  to entry  $V$  in map  $M$ ;  $OP$  represents the map’s association symbol, which might be  $:$ ,  $=$ , or  $\mapsto$ . The order of items in lists is generally relevant, except for heaps and heap typings, which are seen as unordered maps.

Nominal types, variables, and locations are atomic constructs that each have an evolvable counterpart, indicated by a prepended “!”. In many places, we need a “stripped” version of that same construct, for which we use the operator  $[\cdot]$ . It simply strips the ! if present, e.g.,  $[\! \ell] = \ell$ . On terms not containing a !, it is the identity function (i.e., a no-op), e.g.,  $[\ell] = \ell$ .

### 5.2 Type-Checking Examples

Figure 7 shows example definitions in our core grammar from Figure 1. Class `A` has three non-evolvable fields `b`, `z`, and `y`, and an evolvable field `x`. The left middle part of its constructor initializes those fields, and the right middle part is empty because the constructor for the superclass `Object` takes no arguments; the outer two parts are the expression `true` because the core of MAY lacks a skip expression, and our example needs no code to be executed before or after initializing fields. The `evo` method `makeB` is prescribed by the interface `I`; it consumes an evolvable reference to the

```

class A(· b : B, · z : Z, · y : !Y,
      evo(e) x : !X;)this
  extends Object
  implements I {
    constructor(z : Z, y : !Y)()
      {true|(true, z, !y, new X())|()|true};
    evo(t) makeB(c : B) : !B () ↦
      !t ↓ B(c);
  }

interface I extends {
  evo(t) makeB(c : B) : !B;
}

class B(· c : B; · x : !Y)this
  extends A
  implements {
    evolver(c : B)()
      {true|(c, !e ↓ Y())|true};
  }

```

Fig. 7. Two classes and an interface; classes  $Z$  extends  $Y$  extends  $X$  with constructors and evolvers.

```

class E() extends Object implements {
  · t(a : !A) : Object (i : I, ie : !I, b : !B, z : Z, y : Y, x : X, ye : !Y, xe : !X) ↦
  z = a.z;      x = a.x;      ye = new Y();      xe = new X();      ye = new Y();
  y = a.y      ye = a.y ⚡      ye = (a.y = !ye)  xe = (a.x = !xe) ⚡  xe = (a.y = !ye) ⚡
      (a) ✓          (b) ⚡          (c) ✓          (d) ⚡          (e) ⚡

  ie = !a;      ie = !a;      b = !a ↓ B(true);  b = !a.makeB(true);
  b = !ie.makeB(true)  b = !a.makeB(true) ⚡  !b.makeB(true) ⚡  ye = (b.x = new Y())
      (f) ✓          (g) ⚡          (h) ⚡          (i) ✓

```

Fig. 8. Example programs using the definitions in Figure 7. ⚡ marks type-checking failure, ✓ success.

receiver object and evolves it to the subclass  $B$ , which adds an additional field  $c$  and overrides the evolvable field  $x$ . Its evolver therefore assigns both these fields, and again does nothing before or after these field assignments. Notably, the  $x$  is assigned the result of evolving its previous value, accessible via the special variable  $e$  available at this point, specified through  $evo(e)$  in  $A$ .

Based on the definitions in Figure 7, Figure 8 shows a variety of smaller examples that demonstrate various issues to keep in mind with respect to object and field evolution. What we will see across these examples is that it is critical to ensure the uniqueness of evolvable references, while there may simultaneously exist arbitrarily many non-evolvable references to the same object. Any evolvable reference may also be used to retrieve a non-evolvable reference to the same object, without invalidating the evolvable reference.

A key distinction is how evolvable references are handled for local variables versus fields: after an object is constructed, a field is always assumed to contain a valid value of its type. Since evolvable references cannot be copied, that means retrieving an evolvable reference from a field requires swapping it with some other valid evolvable reference. Local variables, on the other hand, track their initialization status. They may already start uninitialized (write-only), and return to this status when an evolvable reference is retrieved from them via the  $!$ -operator. The header of Figure 8 shows the signature of a method whose code the individual examples are supposed to be. The argument  $a$  is given to the method and thus forms an initialized local variable, while the second list of local variable definitions creates uninitialized local variables.

Example (a) shows two straightforward local variable assignments whose type is not evolvable - and even though the field  $y$  in class  $A$  has type  $!Y$ , a field access only ever returns a non-evolvable reference. Example (b) demonstrates this in a second line, where the non-evolvable reference returned from the field access cannot be used to assign a value to a local variable with an evolvable type. Instead, retrieving an evolvable reference from a field can only be done via field assignment, which essentially acts as a swap operation, as shown in Example (c). Swapping in this way requires that a field be writable, and outside of an evolver, the evolvable field  $x$  can only be read, causing

$$\boxed{\Delta \vdash \tau <: \tau} \quad \frac{}{\Delta \vdash \tau <: \tau} \quad \frac{}{\Delta \vdash !M <: M} \quad \frac{}{\Delta \vdash I <: \text{Object}} \quad \frac{\Delta \vdash \tau'' <: \tau' \quad \Delta \vdash \tau' <: \tau}{\Delta \vdash \tau'' <: \tau}$$

$$\frac{M' \text{ extends } M \in \Delta}{\Delta \vdash M' <: M} \quad \frac{C \text{ implements } I \in \Delta}{\Delta \vdash C <: I} \quad \frac{C \text{ implements } I \in \Delta}{\Delta \vdash !C <: !I}$$

Fig. 9. Subtyping

a failure in line 2 of Example (d). In Example (e), the swapping out of field  $y$  in the second line is successful, but the resulting value of type  $!Y$  cannot be assigned to local variable  $x_e$  of type  $!X$ , despite  $Y$  being a subclass of  $X$ . The reason is that an evolvable reference exists to allow calling an evolver, but that is only reasonable if we know that an object has exactly the superclass of the class of the evolver being called. Therefore, it would be wrong to assign a value of type  $!Y$  to a type that allows evolution *towards*  $Y$ .

In the second line of examples, Example (f) shows a use of evolvable references to instances of interfaces. Interfaces themselves are not superclasses of anything, and so cannot be directly used in an evolver call. However, they can be used to call `evo` methods such as `makeB`, which require a receiver of an evolvable type. This only works on classes that directly implement an interface that prescribe such a method, which `A` does in this case. Of course, one could also call such a method on an evolvable instance of the class directly, which would make the call in the second line in Example (g) valid in principle. However, we already used the evolvable reference `!a` in the line above; it is now stored in the variable `ie`, and `a` is unassigned, which makes the call (or really, the evolvable dereference) fail. Another way in which the type system prevents conflicting evolutions is shown in Example (h). This time, we already evolved an instance of `A` into an instance of `B` in the first line, making `!b` a valid evolvable dereference. However, evolvers and `evo` methods are not inherited, and so `!B` does not have a `makeB` method. Finally, in Example (i), we again evolve `a`, this time via `makeB`, and now we see that, in contrast to Example (d), the field `x` can be updated. This is because `B` overrode the field in such a way that it was not evolvable anymore, turning it into a regular field for anyone who knows a precise-enough type. This also shows that while evolvable fields may be somewhat restrictive during the evolution process, if evolution is used for some form of staged initialization that has a clear end, those restrictions can fall away at the end of the process.

### 5.3 Evolvability Types and Subtyping

As is common for substructural type systems, our type system needs to precisely track the locations of the evolvable references in a program, guaranteeing that there is always at most one evolvable reference to any object. The required precision in typing somewhat clashes with the abstraction typically afforded by subtyping. Here, we will discuss how this affects the subtyping relation.

Besides the primitive `B` type, our type system has reference types for interfaces  $I$  and classes  $C$ . Together, we syntactically categorize them as nominal types  $M$ . Both categories of nominal types have evolvable variants,  $!I$  and  $!C$ . Evolvable nominal references ( $!I$  and  $!C$ ) allow calling methods of the interface or class with evolution attribute `evo(x)`, which is not allowed on non-evolvable references. In addition, evolvable class references ( $!C$ ) can be the targets of evolution expressions.

This has important ramifications for subtyping: an evolver expects an object whose current exact class  $C$  is the direct superclass of the class  $C'$  that contains the evolver. There is only a single such class. Behavioral subtyping [Liskov and Wing 1994] then tells us that in order to be a subtype of  $!C$ , a type's instance would have to be able to be applied to an evolver of  $C$ 's direct subclasses, which is not true for any other class, and so not true for any other type<sup>1</sup>. Thus,  $!C$  has no subtypes but itself.

<sup>1</sup>In a type system with a bottom type ( $\perp$ ), this would also be true for  $\perp$ , but our system has no bottom type.

$$\boxed{\Delta \vdash M :^g s, \dots; s, \dots; fd, \dots; (A), \dots; (A), \dots} \quad \frac{}{\Delta \vdash \text{Object} : \emptyset; \emptyset; \emptyset; () ; \emptyset}$$

$$\begin{array}{c}
\text{interface } I \text{ extends } I_1, \dots \{s_1^I; \dots\} \in \Delta \\
\forall i. \vdash s_i \mathbf{dsj} s^r, \dots \quad \forall i. \Delta \vdash I_i : s^I, \dots; s^{I_i}, \dots; \emptyset; \emptyset; \emptyset \quad s^l, \dots < s_1^I; \dots > s^r, \dots \\
\Delta \mid s^r, \dots \mid \emptyset \vdash s^r, \dots \quad \Delta \mid s^l, \dots \mid s^I, \dots, \dots \vdash s_1, \dots \quad \forall i. \Delta \vdash s_i^I \\
\hline
\Delta \vdash I : s_1, \dots; s^r, \dots; \emptyset; \emptyset; \emptyset
\end{array}$$

$$\begin{array}{c}
\text{class } C(fd_1, \dots; fd'_1, \dots)_x \text{ extends } C' \text{ implements } I_1, \dots \{cm_1; \dots; em_1; \dots; d_1 \dots\} \in \Delta \\
\Delta \vdash C' :^{g^C} s^C, \dots; s^{C'}, \dots; fd_1^C, \dots; (A^C), \dots; (A'^C), \dots \\
\Delta \mid fd_1^l, \dots \mid fd_1^C, \dots \vdash_x^g fd_1^r, \dots \quad \Delta \vdash fd_1, \dots, fd_1'', \dots \quad g^C = \mathbf{evo}(x^g) \Rightarrow g = \mathbf{evo}(x) \\
\forall i. \Delta \vdash x^C.d_i \quad \forall i. \Delta \mid fd_1, \dots \mid (A^C), \dots \vdash x^C.cm_i :^g A_i^{cm} \\
\forall i. \Delta \mid fd_1, \dots, fd_1', \dots \mid fd_1^C, \dots \vdash x^{C <: C'} .em_i : A_i^{em} \quad \forall i. d_i = s_i^d(A^{d_i}) \mapsto e^{d_i} \\
s^l, \dots < s_1^d, \dots > s_1^r, \dots \quad \Delta \mid s^r, \dots \mid \emptyset \vdash s^r, \dots \quad \forall i. \Delta \vdash I_i : s^I, \dots; s^{I_i}, \dots; \emptyset; \emptyset; \emptyset \\
s_1^I, \dots = s_1^I, \dots, \dots \quad s_1^{I'}, \dots = s_1^{I'}, \dots, \dots \quad \Delta \mid s^l, \dots \mid s^C, \dots \vdash s_1, \dots \quad \forall i. \exists j. s_j <: s_i^I \\
\forall i. \exists j. s_j^r <: s_i^{I'} \quad \forall i. s_i \mathbf{dsj} s_1^r, \dots \quad \forall i. \cdot m'(A_i^{cm}) : C \mathbf{dsj} \cdot m'(A_{i+1}^{cm}) : C, \dots \\
\forall i. \cdot m'(A_i^{em}) : C \mathbf{dsj} \cdot m'(A_{i+1}^{em}) : C, \dots \quad m' \mathbf{any} \\
\hline
\Delta \vdash C :^g s_1, \dots; s_1^r, \dots; fd_1, \dots, fd_1'', \dots; (A_1^{cm}), \dots; (A_1^{em}), \dots
\end{array}$$

Fig. 10. Class and Interface Validation

The other use of evolvable types is to be able to call methods marked with the `evo` attribute, which indicates an evolvable receiver. Such methods cannot easily be inherited for the same reason: we must be sure about the exact class of the receiver. Blocking this sort of inheritance is fine for classes, because the subtyping relationship that would enable a call to such a method inherited from a superclass does not exist between the necessary receiver types.

Interfaces, however, cannot implement methods on their own. This means that any class directly implementing an interface but not passing on its own `evo` methods can still assume that it is the only possible receiver type, even if a method call to an `evo` method is received through an interface. For this reason, evolvable interface types partly mirror their non-evolvable subtyping relationships, and we can offer at least some level of subtyping-based abstraction over evolvable references. We limit the subtyping of evolvable types to classes that directly declare to implement an interface. Any subclass can also explicitly declare to be implementing the interface again, rather than relying on transitive inheritance in order for this subtyping to work out.

The subtyping relation shown in Figure 9 formalizes these ideas and is otherwise standard.

#### 5.4 Definition Validation

A MAY program consists of a number of class and interface definitions and a main expression. In order for any part of the program to make sense, those class and interface definitions need to be valid, in that they need to contain valid member definitions who refer only to valid types and only use them appropriately. This subsection discusses the key parts of checking program definition validity, starting with the top-level definitions.

**5.4.1 Class and Interface Validation.** The top-level nominal definition validation relation shown in Figure 10 doubles as a relation that describes precisely the methods, fields, constructors, and evolvers available at a particular type, in their most precise (i.e. modulo overriding) form. Methods are split into non-`evo` and `evo` methods, in that order, followed by fields, constructor argument lists, and evolver argument lists. The superscript  $g$  argument tracks whether a class or any of its

superclasses have overridden an evolvable field, which disallows having constructors - instances of such classes can only be reached via evolution. This is because the order in which constructors execute would otherwise override the values of evolved fields in super-constructors, making the process unsound.

For **Object**, there are no methods or fields of any kind, no evolvers, and a single empty constructor. To validate an interface, we retrieve the methods of all super-interfaces, and split the method declarations  $s_1^I, \dots$  of the current interface into the non-**evo** methods  $s^I, \dots$  and the **evo** methods  $s^r, \dots$ . **evo** methods are not inherited across interfaces, mirroring the fact that evolvable interface types are not subtypes of each other, so the **evo** methods  $s^{I_i}$  are ignored. The non-**evo** methods from the current interface and the super-interfaces are merged such that there remains a list of signatures where there is only one signature for every combination of method name and number of arguments (allowing a limited form of overloading) using the most specific signature (contravariant in argument types, covariant in the return type) for overridden methods. This merging may fail if multiple super-interfaces contain variants of methods of the same name and number of arguments, but no clear best signature. Where there is a signature that can override all of them, an interface can manually declare such a method (otherwise, type-checking fails due to a name conflict). For the **evo** methods  $s^r, \dots$ , we essentially check that there are no overloading conflicts either among them or with the non-**evo** methods. Definitions of the relevant helper relations are further below.

For classes, the process for methods is similar, with the main addition that we have to check that a class has a method overriding every signature inherited from a super-interface (this time for both non-**evo** and **evo** methods, the latter due to the single hop in evolvable subtyping between classes and directly implemented interfaces), while only inheriting non-**evo** methods from the superclass (with respect to non-**evo** methods, this is standard). Class definitions already separate field definitions into fields that are new and fields that are overriding fields in a superclass. The combination of the latter with the fields in the superclass yields the inherited and overridden fields  $fd_1'', \dots$ , which together with the regular new fields  $fd_1, \dots$  forms the overall fields of a class, after checking that field names are unique and their types are well-formed. If any field in a superclass is overridden (or such an override already happened in the superclass), the **evo**-attribute  $g$  of the class is forced to be **evo**(x), while having a constructor forces it to be  $\cdot$  through constructor validation - thus both cannot be true at the same time. Method definitions  $d_i$ , constructor definitions  $cm_i$ , and evolver definitions  $em_i$  are validated using judgments explained further below, which are given the relevant pieces of context to validate them. Finally, we validate that there is only at most one constructor and evolver each for each number of arguments by making up some method name  $m'$  and reusing the judgment for method disjointness by constructing pseudo-signatures using  $m'$  as the method name.

**5.4.2 Member Validation.** Figure 11 shows the rules to validate class members, and introduces the grammar for frame types  $\Gamma$ . In addition to assigning types  $\tau$  to variables  $x$ , a frame type also records an availability  $a$  for the variable. Availabilities come in two overlapping groups: readable and writable—overlapping with the read-write availability RW, and separately also containing the read-only value R or the write-only value W, respectively. A normal local variable is usually in the writable group  $w$ : if it is initialized, it is both read and writable, and if it is uninitialized, it is write-only. A special case is the **this**-pointer, which is a read-only local variable that cannot be assigned. Intuitively, when checking expressions that access variables, reading a variable requires read access (R or RW), and assigning a variable requires write access (W or RW).

For validation, we start with fields. The first relation merges overriding fields (the first list of fields in the context) with fields inherited from the superclass (the second list of fields in the context). If there are no overriding fields, this simply results in the list of inherited fields. For every overriding field, we check that the non-evolvable versions of their types are subtypes of each other. This is

$$\begin{array}{c}
\text{Frame Typing } \Gamma ::= \cdot \mid \Gamma, x :^a \tau \quad \text{Read-Access } r ::= R \mid RW \\
\text{Availability } a ::= r \mid w \quad \text{Write-Access } w ::= W \mid RW \\
\hline
\boxed{\Delta \mid fd, \dots \mid fd, \dots \vdash^g fd, \dots} \quad \Delta \vdash fd, \dots \\
\hline
\Delta \mid \emptyset \mid fd, \dots \vdash_x^g fd, \dots \quad \Delta \vdash \emptyset \quad \Delta \vdash gf : \tau, fd'_1, \dots \\
\hline
\Delta \mid fd_1^o, \dots \mid fd_1^l, \dots, fd_1^r, \dots \vdash_x^g fd', \dots \quad \Delta \vdash [\tau'] <: [\tau] \quad \Delta \vdash \tau' \\
\hline
\Delta \mid g'f : \tau', fd_1^o, \dots \mid fd_1^l, \dots, \mathbf{evo}(x') f : \tau, fd_1^r, \dots \vdash_x^{\mathbf{evo}(x')} g'f : \tau', fd', \dots \\
\hline
\boxed{\Delta \mid fd, \dots \mid (A), \dots \vdash x^C.cm :^g A} \quad \forall i. fd_i = g_i f_i : \tau_i^{\text{fld}} \quad (x_1^{\text{sup}} : \tau_1^{\text{sup}}, \dots, x_k^{\text{sup}} : \tau_k^{\text{sup}}) \in (A^C), \dots \\
\Delta \mid x_{\text{this}} \mid \emptyset \mid \emptyset \vdash A > \Gamma < A' \quad \Delta \mid \Gamma \vdash e^{\text{pre}} : \tau^{\text{pre}} \dashv \Gamma_0^{\text{fld}} \quad \forall i. \Delta \mid \Gamma_{i-1}^{\text{fld}} \vdash e_i^{\text{fld}} <: \tau_i^{\text{fld}} \dashv \Gamma_i^{\text{fld}} \\
\Gamma_0^{\text{sup}} = \Gamma_n^{\text{fld}} \quad \forall i. \Delta \mid \Gamma_{i-1}^{\text{sup}} \vdash e_i^{\text{sup}} <: \tau_i^{\text{sup}} \dashv \Gamma_i^{\text{sup}} \quad \Delta \mid \Gamma_k^{\text{sup}}, x_{\text{this}} :^R C \vdash e^{\text{post}} : \tau^{\text{post}} \dashv \Gamma' \\
\hline
\Delta \mid fd_1, \dots, fd_n \mid (A^C), \dots \vdash x_{\text{this}}^C.\text{constructor}(A)(A') \\
\{e^{\text{pre}} \mid (e_1^{\text{fld}}, \dots, e_n^{\text{fld}}) \mid (e_1^{\text{sup}}, \dots, e_k^{\text{sup}}) \mid e^{\text{post}}\} : A \\
\hline
\boxed{\Delta \mid fd, \dots \mid fd, \dots \vdash x^C.em : A} \\
\Delta \mid x_{\text{this}} \mid fd', \dots \mid fd_1, \dots, fd_n \vdash A > \Gamma < A' \quad \Delta \mid \Gamma, x_{\text{this}} :^R C' \vdash e^{\text{pre}} : \tau^{\text{pre}} \dashv \Gamma_0^{\text{fld}} \\
\forall i. fd_i = g_i f_i : \tau_i^{\text{fld}} \quad \forall i. \Delta \mid \Gamma_{i-1}^{\text{fld}} \vdash e_i^{\text{fld}} <: \tau_i^{\text{fld}} \dashv \Gamma_i^{\text{fld}} \quad \Delta \mid \Gamma_n^{\text{fld}} [x_{\text{this}} :^R C] \vdash e^{\text{post}} : \tau^{\text{post}} \dashv \Gamma' \\
\hline
\Delta \mid fd_1, \dots, fd_n \mid fd', \dots \vdash x_{\text{this}}^{C <: C'}.\text{evolver}(A)(A') \{e^{\text{pre}} \mid (e_1^{\text{fld}}, \dots, e_n^{\text{fld}}) \mid e^{\text{post}}\} : A \\
\hline
\boxed{\Delta \vdash x^C.d} \quad \Delta \vdash \tau \quad f \mathbf{any} \quad \Delta \mid x \mid gf : !C \mid gf : !C \vdash A > \Gamma < A' \quad \Delta \mid \Gamma, x :^R C \vdash e <: \tau \dashv \Gamma' \\
\hline
\Delta \vdash x^C.g m(A) : \tau(A') \mapsto e
\end{array}$$

Fig. 11. Member Validation

fine because an evolvable field can only ever be read outside of an evolver, and as such, it is only retrieved in its non-evolvable version, therefore guaranteeing the soundness of any access to the field. The overriding field definition then ends up in the overall resulting list of field definitions.

The second field-related relation simply checks that all field types are valid, that all field names are distinct, and that all the variable names of declared **evo** fields are disjoint as well.

For constructors, we are given a list of new fields that the class introduces, which need to be initialized by the constructor (a constructor cannot exist where fields are overridden, so this second kind of field definitions can be ignored here). A constructor further needs to call a superclass constructor, and so an argument list of the correct length needs to exist in the superclass' list of constructors. The relation  $\Delta \mid x, \dots \mid fd, \dots \mid fd, \dots \vdash A > \Gamma < A$  builds an initial local frame environment  $\Gamma$  for type-checking the code of the constructor. Recall that every callable has two argument lists  $A$ , one for the actual arguments of the callable (here  $A$ ), and one for the (uninitialized) local variables (here  $A'$ ). The relation checks that their types are well-formed, and no name appears twice or in the list  $x, \dots$ , which here serves to reserve the name of the **this**-variable for later. The lists of field definitions will come in when we discuss evolvers; they are empty here.

Given this initial environment, the rest of the rule checks that successively evaluating the parts of the constructor type-check under consecutively chained frame types (which keep track of the initialization status of variables). The pre- and post-expressions just need to type-check under any type; their result is ignored, but they may affect state. Field initialization expressions and

$$\begin{array}{c}
\boxed{g \sim g \quad \Delta \vdash s <: s \quad \vdash s \mathbf{dsj} s \dots} \\
\frac{\Delta \vdash \tau <: \tau' \quad \forall i. \tau'_i <: \tau_i \quad g \sim g'}{\dots \sim \dots \quad \mathbf{evo}(x) \sim \mathbf{evo}(x')} \quad \Delta \vdash g m(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau <: g' m(x'_1 : \tau'_1, \dots, x'_n : \tau'_n) : \tau' \\
\frac{\Delta \vdash s \mathbf{dsj} \emptyset \quad m \neq m' \vee n \neq k \quad g m(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau \mathbf{dsj} s'_1, \dots}{\Delta \vdash s \mathbf{dsj} \emptyset \quad \vdash g m(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau \mathbf{dsj} g' m'(x'_1 : \tau'_1, \dots, x'_k : \tau'_k) : \tau', s'_1, \dots} \\
\boxed{s, \dots < s, \dots > s, \dots \quad \Delta \mid s, \dots \mid s, \dots \vdash s, \dots} \\
\frac{\overline{\emptyset < \emptyset > \emptyset} \quad s = \cdot m(A) : \tau \quad s^l, \dots < s_1, \dots > s^r, \dots \quad s = \mathbf{evo}(x) m(A) : \tau \quad s^l, \dots < s_1, \dots > s^r, \dots}{\overline{\emptyset < \emptyset > \emptyset} \quad s, s^l, \dots < s, s_1, \dots > s^r, \dots \quad s^l, \dots < s, s_1, \dots > s, s^r, \dots} \\
\frac{\Delta \mid s, \dots \mid s_1^l, \dots, s_1^r \dots \vdash s''_1, \dots \quad \Delta \vdash s''_1 <: s'}{\Delta \mid \emptyset \mid \emptyset \vdash \emptyset \quad \Delta \mid s, \dots \mid s_1^l, \dots, s', s_1^r, \dots \vdash s''_1, \dots} \\
\frac{\Delta \mid s_1, \dots \mid \emptyset \vdash s_1, \dots \quad \vdash s \mathbf{dsj} s_1, \dots \quad \Delta \mid s_1, \dots \mid s_1^l, \dots, s_1^r, \dots \vdash s''_1, \dots \quad \vdash s' \mathbf{dsj} s''_1, \dots}{\Delta \mid s, s_1, \dots \mid \emptyset \vdash s, s_1, \dots \quad \Delta \mid s, \dots \mid s_1^l, \dots, s', s_1^r, \dots \vdash s', s''_1, \dots} \\
\boxed{\Delta \mid x, \dots \mid fd, \dots \mid fd, \dots \vdash A > \Gamma < A} \\
\frac{\Delta \mid x, \dots \mid \emptyset \mid fd, \dots \vdash \emptyset > \emptyset < \emptyset \quad \Delta \mid x', x, \dots \mid \emptyset \mid fd, \dots \vdash A > \Gamma < A'}{\Delta \mid x, \dots \mid \emptyset \mid fd, \dots \vdash A, x' : \tau > \Gamma, x' :^{\text{RW}} \tau < A'} \\
\frac{\forall g', \tau'. g' f : \tau' \notin fd', \dots \quad \Delta \mid x, \dots \mid fd, \dots \mid fd', \dots \vdash A > \Gamma < A' \quad x' \notin x, \dots \quad \Gamma = \Gamma', x' :^{\text{W}} \tau \quad \Delta \vdash \tau}{\Delta \mid x, \dots \mid g f : \tau, fd, \dots \mid fd', \dots \vdash A > \Gamma < A' \quad \Delta \mid x', x, \dots \mid \emptyset \mid fd, \dots \vdash A > \Gamma' < A'} \\
\frac{x' \notin x, \dots \quad g f : \tau' \in fd', \dots \quad \Delta \mid x', x, \dots \mid fd, \dots \vdash A > \Gamma < A'}{\Delta \mid x, \dots \mid \mathbf{evo}(x') f : \tau, fd, \dots \mid fd', \dots \vdash A > \Gamma, x' :^{\text{RW}} \tau < A'}
\end{array}$$

Fig. 12. Helper Relations for Checking Type and Member Definitions

super-constructor arguments need to match their expected types. Lastly, once the super-constructor has been called, the object is fully initialized, and so the post-expression has access to the previously unavailable this-pointer. This relates to the soundness issues discussed in Section 4.2. By requiring that every constructor fully initializes the fields of a class before calling the super-constructor, an object is always fully initialized after the super-constructor call, and calling overridden methods that rely on fields in subclasses will work just fine, as those fields have already been initialized.

Evolvers are similar to constructors, with three modifications: first, there is no super-constructor call. Second, the this-pointer is available from the start, but changes its type when we get to the post-expression. Third, the starting frame type  $\Gamma$  may contain additional variables based on the field declarations in the supertype and the current class. For every field of the superclass that is overridden (and therefore is declared as  $\mathbf{evo}(x)$ ), there is a local variable  $x$  of the full type (including evolvability) of the field. This is the only time that evolvable references can be retrieved from an evolvable field, based on the reasoning that it is going to be overwritten with a new reference by the evolution constructor (in many cases, by evolving the object now accessible through an evolvable reference).

$$\boxed{\Delta \mid \Gamma \vdash e <: \tau \vdash \Gamma} \quad \frac{\Delta \mid \Gamma \vdash e : \tau' \vdash \Gamma' \quad \Delta \vdash \tau' <: \tau}{\Delta \mid \Gamma \vdash e <: \tau \vdash \Gamma'}$$

$$\boxed{\Delta \mid \Gamma \vdash e : \tau \vdash \Gamma}$$

$$\frac{e \in \{\text{true}, \text{false}\}}{\Delta \mid \Gamma \vdash e : \mathbb{B} \vdash \Gamma} \quad \frac{x :^r \tau \in \Gamma}{\Delta \mid \Gamma \vdash x : [\tau] \vdash \Gamma} \quad \frac{x :^{\text{RW}} !M \in \Gamma}{\Delta \mid \Gamma \vdash !x : !M \vdash \Gamma[x :^W !M]} \quad \frac{\Delta \mid \Gamma \vdash e : \tau \vdash \Gamma' \quad \Delta \vdash [\tau].f : g_f \tau_f}{\Delta \mid \Gamma \vdash e.f : [\tau_f] \vdash \Gamma'}$$

$$\frac{\forall i. \Delta \mid \Gamma_{i-1} \vdash e_i <: \tau_i \vdash \Gamma_i \quad \Delta \vdash C(\tau_1, \dots, \tau_n) \quad \Gamma_n = \Gamma}{\Delta \mid \Gamma_0 \vdash \text{new } C(e_1, \dots) : !C \vdash \Gamma} \quad \frac{\Delta \mid \Gamma \vdash e_1 : \tau_1 \vdash \Gamma' \quad \Delta \mid \Gamma' \vdash e_2 : \tau_2 \vdash \Gamma''}{\Delta \mid \Gamma \vdash e_1 == e_2 : \mathbb{B} \vdash \Gamma''} \quad \frac{\Delta \mid \Gamma \vdash e <: \tau \vdash \Gamma' \quad x :^w \tau \in \Gamma \quad \Gamma'' = \Gamma'[x :^{\text{RW}} \tau]}{\Delta \mid \Gamma \vdash x = e : [\tau] \vdash \Gamma''}$$

$$\frac{\Delta \mid \Gamma \vdash e : \tau \vdash \Gamma' \quad \Delta \vdash [\tau].f : \cdot \tau_f}{\Delta \mid \Gamma' \vdash e.f <: \tau_f \vdash \Gamma''} \quad \frac{\Delta \mid \Gamma \vdash e : \tau_e \vdash \Gamma_0 \quad \forall i. \Delta \mid \Gamma_{i-1} \vdash e_i <: \tau_i \vdash \Gamma_i \quad \Delta \vdash \tau_e.m(\tau_1, \dots, \tau_n) : \tau \quad \Gamma_n = \Gamma'}{\Delta \mid \Gamma \vdash e.m(e_1, \dots, e_n) : \tau \vdash \Gamma'} \quad \frac{\Delta \mid \Gamma \vdash e_1 : \tau_1 \vdash \Gamma' \quad \Delta \mid \Gamma' \vdash e_2 : \tau_2 \vdash \Gamma''}{\Delta \mid \Gamma \vdash e_1; e_2 : \tau_2 \vdash \Gamma''}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathbb{B} \vdash \Gamma_1 \quad \Delta \mid \Gamma_1 \vdash e_2 : \tau_2 \vdash \Gamma_2 \quad \Delta \mid \Gamma_1 \vdash e_3 : \tau_3 \vdash \Gamma_3}{\Delta \mid \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_2 \sqcup_{\Delta} \tau_3 \vdash \Gamma_2 \sqcup_{\Delta} \Gamma_3} \quad \frac{\Delta \mid \Gamma \vdash e : !C' \vdash \Gamma_0 \quad \forall i. \Delta \mid \Gamma_{i-1} \vdash e_i <: \tau_i \vdash \Gamma_i \quad \Delta \vdash C' \Downarrow C(\tau_1, \dots, \tau_n) \quad \Gamma_n = \Gamma'}{\Delta \mid \Gamma \vdash e \Downarrow C(e_1, \dots, e_n) : !C \vdash \Gamma'}$$

Fig. 13. The typing rules for source-level MAY

Finally, method definitions are validated by checking that their return type is valid and their expression checks against this type under the initial frame type  $\Gamma$ .  $\Gamma$ 's initialization coopts the overwritten field variable scheme for evolvers with some arbitrary field name  $f$  to potentially provide an evolvable version of the `this`-pointer for `evo` methods, which are specified as `evo(x)`, in which case there will be a local variable  $x$  of type  $C!$ . Nothing happens in this regard if the evolvability parameter  $g$  of the method definition is  $\cdot$ .

**5.4.3 Helper relations.** Figure 12 shows the definitions of the key helper relations used and explained above. Standard relations like type validation  $\Delta \vdash \tau$ , and signature validation  $\Delta \vdash s$ , and overall definitions validation  $\vdash \Delta$  are omitted for space, and completely standard.

## 5.5 Expression Typing

Figure 13 shows the basic expression typing rule for MAY on the source-level. A special subsumption rule fulfills a role analogous to checking rules in bi-directional type systems, while the regular typing rules would be synthesis rules. Both kinds of rules have the same components: a program definition environment  $\Delta$ , an input frame typing  $\Gamma$  (left), an expression  $e$ , a type  $\tau$ , and an output frame typing  $\Gamma$  (right).

The program source code (e.g. method bodies) type-checks under simple frame typings that express current local variables (including e.g. local variables and method arguments).

Boolean values have type  $\mathbb{B}$ . Simple variable accesses require that a variable is readable and return a non-evolvable value (cf. Example (b) in Figure 8). Evolvable variable access require that a variable is both readable and writable, and of an evolvable type. The evolvable access de-initializes the variable (it becomes write-only), as the evolvable reference has been retrieved from it (cf. Example (g) in Figure 8). Field accesses function like simple variable accesses, except that fields always need

to be initialized. The relevant rule, like other member-access rules, makes use of a straightforward helper relation that uses the class/interface validation rule to retrieve a list of members of the relevant type.

Constructor calls check their arguments against the expected types, and return an evolvable reference to an instance of the constructed class. Equality expressions can compare arbitrary values and return a boolean. Variable assignment expressions require that the assigned value has the right type and that the variable is writable. They modify the variable to be read-write (which may be a noop) and return a non-evolvable version of the value that has been assigned.

Field assignment expressions require the receiver to have a non-evolvable field of the relevant name, and the assigned value to have the correct type. They return the previous value of the field, which may be an evolvable reference (cf. the various swapping examples in Figure 8). Method calls require a well-typed receiver that has an appropriate method. The method lookup relation checks whether the receiver type is evolvable, and if so, also allows `evo` methods. In any case, the arguments need to have the right types, and the overall expression's type is the return type of the method. Sequencing expressions return the value of the last expression, and otherwise just chain the frame types.

If-then-else expressions have a boolean condition and two expressions that are checked under the same frame type. They may return values of different types and different frame types, which need to be joined. A simple valid join relation on types only joins according to the class hierarchy (with `Object` a supertype of any interface), and the join of two frame types requires them to have the same domain, element-wise joining types of variables using the earlier relation, and accessibility towards the more restrictive one (i.e.  $W \sqcup RW = W$ ). Finally, evolution expressions require the receiver to be of an evolvable type of a direct superclass of the class to which the receiver should be evolved. As usual, the arguments need to type-check under the appropriate types, and the evolver returns an evolvable reference for an instance of the target class.

## 6 Dynamic Semantics

We define the reduction and (full) typing rules of `MAY` on an extended grammar whose additional components are shown in Figure 14 (the full typing rules can be found in the technical report [Mirrlees-Black et al. 2026a]). Expressions are extended with location references  $\ell$  and  $!\ell$ , which track whether a particular reference is an evolvable reference, and the call boundary expressions of method calls `mb(...)`, constructors `cb(...)`, and evolvers `eb(...)` that introduce call stack frames. Being a stateful language, the dynamic semantics does not rely on substitution, but explicitly tracks stack frames. The idea here is that the call boundaries both track the progress within a constructor or evolver, and in general allow us to know when to pop a stack frame off the stack, once the evaluation of a method/constructor/evolver is finished.

For ease of tracking the relevant data, the constructor/evolver boundaries carry around the name of the local `this` variable  $x$ , the location  $\ell$  of the constructed/evolved object, and the target class  $C$  in the superscript. Most constructor boundaries additionally carry the current class  $C$  as the last argument, which may be different from the target class in nested super-constructor calls. The different forms reflect the different stages of executing constructors/evolvers, see below.

There are two kinds of evaluation contexts, one for regular expressions and one for call boundaries. Both contexts are “flat” in the sense that they do not build recursive structures (they only do so in mutual recursion with expressions/contexts inserted into them). The intuition is still the same as with recursive evaluation contexts, but the flat structure supports our proof of preservation.

The remainder of Figure 14 describes stack frames  $G$ , stacks  $S$ , and heaps  $H$ , and otherwise bookkeeping structures for type checking. Stack frames are maps from variable names to variable states, which may either be a value or nothing, indicating that the variable is uninitialized; this

Location $\ell$			
Expression	$e ::= \dots \mid \ell \mid !\ell \mid \text{mb}(e) \mid \text{cb}^C(e \mid (e\dots) \mid (e\dots) \mid e, x, \ell, C)$ $\mid \text{cb}^C(e \mid e, x, C) \mid \text{cb}^C(e, \ell) \mid \text{eb}^C(e \mid (e\dots) \mid e, x, \ell) \mid \text{eb}^C(e, \ell)$		
Evaluation Ctx	$E ::= \bullet.f \mid \text{new } C(v\dots, \bullet, e\dots) \mid \bullet == e \mid v == \bullet \mid x = \bullet \mid \bullet.f = e$ $\mid v.f = \bullet \mid \bullet.m(e\dots) \mid v.m(v\dots, \bullet, e\dots) \mid \bullet; e$ $\mid \text{if } \bullet \text{ then } e \text{ else } e \mid \bullet \Downarrow C(e\dots) \mid v \Downarrow C(v\dots, \bullet, e\dots)$		
Call Boundary	$B ::= \text{mb}(\bullet) \mid \text{cb}^C(\bullet \mid (e\dots) \mid (e\dots) \mid e, x, \ell, C)$ $\mid \text{cb}^C(v \mid (v\dots, \bullet, e\dots) \mid (e\dots) \mid e, x, \ell, C) \mid \text{cb}^C(\bullet \mid e, x, C)$ $\mid \text{cb}^C(v \mid (v\dots) \mid (v\dots, \bullet, e\dots) \mid e, x, \ell, C) \mid \text{cb}^C(\bullet, \ell)$ $\mid \text{eb}^C(\bullet \mid (e\dots) \mid e, x, \ell) \mid \text{eb}^C(v \mid (v\dots, \bullet, e\dots) \mid e, x, \ell) \mid \text{eb}^C(\bullet, \ell)$		
Value	$v ::= \text{true} \mid \text{false} \mid \ell \mid !\ell$	Affine Type	$\mu ::= C \mid !C$
Frame	$G ::= \cdot \mid G, x \mapsto v \mid G, x$	Construction State	$\gamma ::= C \mid \top$
Stack	$S ::= \cdot \mid S; G$	Stack Typing	$\Sigma ::= \cdot \mid \Sigma; \Gamma$
Heap	$H ::= \cdot \mid H, \ell \mapsto C(f = v, \dots)$	Heap Typing	$X ::= \cdot \mid X, \ell \mapsto_C^Y \mu$

Fig. 14. Additional grammar constructs for run-time constructs and typing.

corresponds to the availabilities in frame typings discussed in Section 5. Stacks are literally stacks of stack frames. Heaps are maps from locations to objects, which have a concrete class  $C$ , and an (ordered, by declaration and inheritance) map from field names to values. Values are one of the two boolean values, regular locations  $\ell$ , and evolvable locations  $!\ell$ .

The other structures concern the typings for stacks  $\Sigma$ , and heaps  $X$ . Stack typings  $\Sigma$  are literal stacks of frame typings  $\Gamma$ , mirroring the structure of their corresponding values. Heap typings  $X$  are maps from locations to triples of a construction state  $\gamma$ , an evolution target  $C$ , and an affine type  $\mu$  (i.e., class types that are either evolvable or not). The construction state tracks what parts of an object have been initialized, while the evolution target indicates whether the object is currently in the process of evolving. We will explain this in more detail below.

## 6.1 Reduction Rules

Figure 15 shows most of core MAY's reduction rules. To save space, we have omitted rules for comparison expressions, if-then-else expressions, and sequencing, which are standard. The rules step between program configurations consisting of a heap  $H$ , a stack  $S$ , and a program expression  $e$ , under a constant set of program definitions  $\Delta$ .

*Evaluation Contexts.* We start with the rules for evaluation contexts on the first line. Recall that our evaluation contexts are flat, so we instead build trees from these rules, just as in a system without evaluation contexts. Importantly, each rule assumes that it is operating on the stack frame at the bottom of the given stack. Since the second rule deals with call boundaries, those rules cut off the frame on the bottom of the stack for the recursive premise, as we move “up” in the call stack.

Conversely, we see that all other rules assume that they are operating at or near the top of the call stack, and so always describe their input and output stacks in terms of the top frames. This makes sense because changes should always happen on the top of the stack.

*Local Variables and Fields.* The next five rules deal with local variables and fields, both reading and assigning. Their semantics mirror the considerations of the respective typing rules discussed in Section 5, avoiding the duplication of evolvable references. For example, the reduction rule for evolvable variables resets the variable in the stack frame to be uninitialized, and field assignment returns the old value of a field, implementing a swap operation.

$$\boxed{\Delta \vdash H \mid S \mid e \rightarrow e \mid S \mid H}$$

$$\frac{\Delta \vdash H \mid S \mid e \rightarrow e' \mid S' \mid H'}{\Delta \vdash H \mid S \mid E[e] \rightarrow E[e'] \mid S' \mid H'} \quad \frac{\Delta \vdash H \mid S \mid e \rightarrow e' \mid S' \mid H'}{\Delta \vdash H \mid S; G \mid B[e] \rightarrow B[e'] \mid S'; G \mid H'}$$

$$\frac{x \mapsto v \in G}{\Delta \vdash H \mid G \mid x \rightarrow [v] \mid G \mid H} \quad \frac{x \mapsto !\ell \in G}{\Delta \vdash H \mid G \mid !x \rightarrow !\ell \mid G[x] \mid H}$$

$$\frac{[v] \mapsto C(\dots, f = v_f, \dots) \in H}{\Delta \vdash H \mid G \mid v.f \rightarrow [v_f] \mid G \mid H} \quad \frac{}{\Delta \vdash H \mid G \mid x = v \rightarrow [v] \mid G[x \mapsto v] \mid H}$$

$$\frac{[v] \mapsto C(\dots, f = v_f, \dots) \in H \quad H' = H[[v] \mapsto C(\dots, f = v', \dots)]}{\Delta \vdash H \mid G \mid v.f = v' \rightarrow v_f \mid G \mid H'}$$

$$\frac{\Delta \mid H \vdash v.m(v_1, \dots) \rightsquigarrow e \mid G'}{\Delta \vdash H \mid G \mid v.m(v_1, \dots) \rightarrow \text{mb}(e) \mid G'; G \mid H} \quad \frac{}{\Delta \vdash H \mid G; G' \mid \text{mb}(v) \rightarrow v \mid G' \mid H}$$

$$\frac{\Delta \vdash C(v_1, \dots)_x \rightsquigarrow \{e^{\text{pre}} \mid (e_1^{\text{fld}} \dots) \mid (e_1^{\text{sup}} \dots) \mid e^{\text{pst}}\} \mid G' \quad H' = H, \ell \mapsto C() \quad \mathbf{fresh} \ell}{\Delta \vdash H \mid G \mid \text{new } C(v_1 \dots) \rightarrow \text{cb}^C(e^{\text{pre}} \mid (e_1^{\text{fld}} \dots) \mid (e_1^{\text{sup}} \dots) \mid e^{\text{pst}}, x, \ell, C) \mid G'; G \mid H'}$$

$$\frac{C'(g_1 f_1^{C'} : \tau_1, \dots, g_n f_n^{C'} : \tau_n;) \text{ extends } C'' \in \Delta \quad \Delta \vdash C''(v_1^{\text{sup}}, \dots)_{x_{C''}} \rightsquigarrow \{e^{\text{pre}} \mid (e_1^{\text{fld}} \dots) \mid (e_1^{\text{sup}} \dots) \mid e_{C''}^{\text{pst}}\} \mid G'' \quad \ell \mapsto C(f_1 = v_1, \dots) \in H \quad H' = H[\ell \mapsto C(f_1 = v_1 \dots, f_1^{C'} = v_1^{\text{fld}}, \dots, f_n^{C'} = v_n^{\text{fld}})]}{\Delta \vdash H \mid G; G' \mid \text{cb}^C(v^{\text{pre}} \mid (v_1^{\text{fld}}, \dots, v_n^{\text{fld}}) \mid (v_1^{\text{sup}} \dots) \mid e_{C'}^{\text{pst}}, x_{C'}, \ell, C') \rightarrow \text{cb}^C(\text{cb}^C(e^{\text{pre}} \mid (e_1^{\text{fld}}, \dots) \mid (e_1^{\text{sup}} \dots) \mid e_{C''}^{\text{pst}}, x_{C''}, \ell, C'') \mid e_{C'}^{\text{pst}}, x_{C'}, C') \mid G'; G; G' \mid H'}$$

$$\frac{C'(g_1 f_1^{C'} : \tau_1, \dots, g_n f_n^{C'} : \tau_n;) \text{ extends Object} \in \Delta \quad \ell \mapsto C(f_1 = v_1, \dots) \in H \quad H' = H[\ell \mapsto C(f_1 = v_1 \dots, f_1^{C'} = v_1^{\text{fld}}, \dots, f_n^{C'} = v_n^{\text{fld}})]}{\Delta \vdash H \mid G; G' \mid \text{cb}^C(v^{\text{pre}} \mid (v_1^{\text{fld}}, \dots, v_n^{\text{fld}}) \mid () \mid e^{\text{pst}}, x, \ell, C') \rightarrow \text{cb}^C(e^{\text{pst}}, \ell) \mid G, x \mapsto \ell; G' \mid H'}$$

$$\frac{}{\Delta \vdash H \mid G; G' \mid \text{cb}^C(!\ell \mid e, x, C') \rightarrow \text{cb}^C(e, \ell) \mid G, x \mapsto \ell; G' \mid H}$$

$$\frac{\Delta \mid H \vdash \ell \Downarrow C(v_1, \dots)_x \rightsquigarrow \{e^{\text{pre}} \mid (e_1^{\text{fld}} \dots) \mid e^{\text{pst}}\} \mid G' \mid H'}{\Delta \vdash H \mid G \mid !\ell \Downarrow C(v_1, \dots) \rightarrow \text{eb}^C(e^{\text{pre}} \mid (e_1^{\text{fld}} \dots) \mid e^{\text{pst}}, x, \ell) \mid G'; G \mid H'}$$

$$\frac{\text{class } C(g_1 f_1^C : \tau_1, \dots, g_k f_k^C : \tau_k; g_{k+1} f_{k+1}^C : \tau_{k+1}, \dots, g_n f_n^C : \tau_n) \in \Delta \quad \ell \mapsto C'(f_1 = v_1, \dots) \in H \quad H' = H[\ell \mapsto C(f_1 = v_1, \dots, [f_{k+1}^C = v_{k+1}^{\text{fld}}, \dots, f_n^C = v_n^{\text{fld}}], f_1^C = v_1, \dots, f_k^C = v_k)]}{\Delta \vdash H \mid G; G' \mid \text{eb}^C(v^{\text{pre}} \mid (v_1^{\text{fld}}, \dots, v_n^{\text{fld}}) \mid e^{\text{pst}}, x, \ell) \rightarrow \text{eb}^C(e^{\text{pst}}, \ell) \mid G[x :^R C]; G' \mid H'}$$

$$\frac{}{\Delta \vdash H \mid G; G' \mid \text{cb}^C(v, \ell) \rightarrow !\ell \mid G' \mid H} \quad \frac{}{\Delta \vdash H \mid G; G' \mid \text{eb}^C(v, \ell) \rightarrow !\ell \mid G' \mid H}$$

Fig. 15. The dynamic semantics for core MAY, except equality comparisons, if-then-else, and sequencing.

*Method Calls.* The next two rules concern method calls. All rules that initiate a call of some kind use an auxiliary predicate to look up the relevant code and initialize the new stack frame, mirroring the behavior in the corresponding typing rules. The definitions of these auxiliary predicates are in the technical report [Mirrlees-Black et al. 2026a]. We push the new frame on top of the stack and add a method call boundary to mark the corresponding call stack boundary. The second rule returns from a method call when its code is reduced to a value. This pops off the top-most stack frame and leaves the returned value.

The remaining rules all have to do with constructors and evolvers, which are similar.

*Constructors.* The first constructor rule uses an auxiliary predicate to look up the relevant constructor that should be called and initialize its stack frame, just as for methods. It uses this to initialize the call boundary, whose parts are the parts of the constructor plus, for the simplicity of both the typing and reduction rules, the name of the `this` pointer  $x$ , the location of the new object  $\ell$ , the ultimate class of the new object  $C$  (in the superscript), and potentially also the class that the current constructor belongs to,  $C'$ .

The constructor's call boundary goes through several phases. In the first phase, the preamble expression  $e^{\text{pre}}$ , the field values  $e_i^{\text{fld}}$ , and the super-constructor arguments  $e_i^{\text{sup}}$ , are evaluated in order. Once this is done, we assign the field values to the relevant fields, and then call the super-constructor (except if the superclass is `Object`, in which case we directly skip to the third phase after assigning fields), spawning a nested constructor call boundary. In the second phase, we simply wait for the super-constructor to complete its evaluation. In the third phase, the object is fully initialized from a soundness point of view, and the `this`-pointer is available. The constructor completes after having evaluated any remaining code in  $e^{\text{pst}}$ , returning an evolvable reference to the newly created object.

*Evolvers.* Evolvers behave in a similar way as constructors, but with differences that mirror the differences discussed in Section 5. Key here is the field assignment step, which uses the separate field definitions to add new fields and update overwritten fields in the object.

## 7 Soundness

We have proved the soundness of MAY using a standard progress-and-preservation approach. The proof relies on an extended typing relation  $\Delta \mid X \mid \Sigma \vdash e : \tau \dashv \Sigma$  that includes a heap typing  $X$  and generalizes frame types  $\Gamma$  to stack types  $\Sigma$ . As is common for substructural type systems, we reason about splits of heap types  $X$ , such that the objects on the heap, the stack, and each sub-expression of the current program are type-checked under their own views of the heap. Importantly, the heap type tracks which of those parts may have an evolvable reference to each object on the heap, and the splits guarantee that at most one such reference exists for each object. For non-evolvable references, the heap splits may overlap.

The full definitions and proofs are in the technical report [Mirrlees-Black et al. 2026a], but we state abstracted versions (abstracting away the validation rules for the different parts of the heap, etc.) of our theorems here:

**THEOREM 7.1 (PROGRESS (ABSTRACTED)).** *For all  $\Delta, X, \Sigma, \Sigma_o, e, \tau, H$ , and  $S$ , if:*

- $\vdash \Delta$  (well-formed definitions), and
- $\Delta \mid X \mid \Sigma \vdash e : \tau \dashv \Sigma_o$  (well-typedness), and
- $\Delta \vdash X \mid \Sigma \mid \Sigma_o \mid H \mid S$  (well-formed heap/stack)

*then either  $e = v$  for some  $v$ , or there exists some  $e', H'$ , and  $S'$ , such that  $\Delta \vdash H \mid S \mid e \rightarrow e' \mid S' \mid H'$ .*

See proof in the technical report [Mirrlees-Black et al. 2026a].

**THEOREM 7.2 (PRESERVATION (ABSTRACTED)).** *For all  $\Delta, X, \Sigma, \Gamma_o, e, \tau, H, S, e', H',$  and  $S',$  if:*

- $\vdash \Delta$  (well-formed definitions), and
- $\Delta \mid X \mid \Sigma \vdash e : \tau \dashv \Gamma_o$  (assumed well-typedness), and
- $\Delta \vdash H \mid S \mid e \rightarrow e' \mid S' \mid H'$  (assumed reduction), and
- $\Delta \vdash X \mid \Sigma \mid \Gamma_o \mid H \mid S$  (assumed well-formed heap/stack)

then there exists some  $\tau', X', \Sigma',$  and  $\Gamma'_o,$  such that:

- $\Delta \mid X' \mid \Sigma' \vdash e' : \tau' \dashv \Gamma'_o$  (result well-typedness), and
- $\Delta \vdash \tau' <: \tau$  (result subsumption), and
- $\Delta \vdash X' \mid \Sigma' \mid \Gamma'_o \mid H' \mid S' \mid X \mid \Sigma \mid \Gamma_o \mid H \mid S$  (well-related results,  $\Rightarrow$  well-formed heap/stack)

See proof in the technical report [[Mirrlees-Black et al. 2026a](#)].

## 7.1 Type Safety

Program validation is defined as follows:

$$\frac{\vdash \Delta \quad \Delta \mid \cdot \vdash e : \tau \dashv \cdot}{\vdash \langle \Delta \mid e \rangle : \tau}$$

**COROLLARY 7.3 (TYPE SAFETY).** *Any MAY program  $\mathcal{P}$  satisfying  $\vdash \mathcal{P} : \tau$  will either run forever or evaluate to a value whose type is a subtype of  $\tau$ .*

## 8 Implementation

We have implemented MAY as a compiler to native code using the QBE backend [[Carbonneaux 2024](#)], which includes an efficient implementation for object evolution. The difficulty naturally comes not with changing the class or methods on evolution, since this may be implemented by changing the class pointer, but with extending the number of fields available on an object. [Cohen and Gil \[2009\]](#) give three approaches for implementing object evolution: requiring every reference to be a handle so objects can be moved arbitrarily and transparently, using proxy objects by updating an object's header to forward to the new object when evolved, as well as the rather dramatic strategy of forcing a GC run to force an object to move and resize when it evolves.

We consider the first two of these approaches to have too high an overhead for programs that do not make substantial use of object evolution, due to the increased cost of double-pointer dereference and the read barriers required to check if an object is a proxy on every access. On the other hand, forcing a GC run presents too high an overhead to object evolution, and would require carefully designing the runtime to account for potentially regular evolution operations.

Instead, our implementation approach relies on whole-program compilation to determine the maximum possible size an object may evolve to, and pre-allocate the necessary space. For objects that cannot evolve, we pay no additional cost; they are allocated and treated as normal. That is, a program without evolvers behaves just like in a regular object-oriented language, for example Java, and does not pre-allocate additional space. The pre-allocation pays off when most objects fully evolve, since in this case, we are equivalent in runtime cost to just using fields that start as uninitialized and become initialized on evolution. For objects that do not fully evolve on average, or at least evolve to a subclass that is not the largest possible subclass, we have to pay for the additional memory usage and fragmentation that results from the unused memory. This is undesirable, but presents the best tradeoff of the approaches we have evaluated.

The applications we are envisioning evolve most objects towards their final state—this is particularly true when object evolution is used for initialization or processes like type checkers. In such a scenario, pre-allocation with evolution in fact reduces memory overhead compared to a scheme where one has to create extended copies of the relevant data structures, as in the end, the memory

consumption is the same, but in the meantime, one would have additional old objects requiring additional memory. In addition, this scheme can be made to work with dynamic code loading to avoid the reliance on whole-program compilation. This requires a moving garbage collector that can be triggered to resize objects on demand. As code loading is a rare and expensive activity, this would cause relatively little overhead, while preserving the soundness of the overall scheme.

Alternative methods we have considered are all similar to the approaches suggested by [Cohen and Gil 2009], but optimize for reducing memory usage in some way. However, each must either make the tradeoff of additional memory accesses or read barriers.

## 9 Case Study: A Type Checker for IMP

The IMP language is a simple imperative language used for educational purposes to teach programming language theory [Winskel 1993]. We have implemented a type checker in MAY to demonstrate that object evolution can be used for monotonic in-place updates to tree data structures, such as syntax trees [Mirrlees-Black et al. 2026b]. Additionally, this example illustrates the practical applications of attaching evolution permissions to interfaces.

The structure of the checker strongly follows the pattern laid out in Figure 4. We use the visitor pattern for both the untyped and typed syntax tree, and include a `getType()` function in the interface for typed nodes. We diverge from the standard pattern in two places. Firstly, when checking assignment to and dereference of locations, we resolve locations from their name in the source to an index in an array of valid store locations. Secondly, when checking binary operators, we resolve the generic untyped `Op` node into either `TypedAdd` or `TypedGt` depending on whether the operator is addition or greater-than comparison. This is a somewhat synthetic example of the realistic need to specialize as part of evolution. In a more advanced compiler, we could imagine specializing from an untyped identifier to one of “local variable”, “global variable” or “type name”.

In total, the typed and untyped language definitions, pretty printers, and type checking required about 500 lines of code. Of course, lines of code are not a perfect measure of the usability of a language feature; indeed, using a terser language, or even an approach compromising safety to include a nullable type field in the syntax tree, would reduce the line count. However, we do at least demonstrate that evolution is effective for one of the tasks that motivated the feature, and reduces complexity over alternative safe approaches that rely on copying data between stages.

This case study does also highlight a handful of current pain points in MAY’s design, but we believe them to be remediable. The language does not yet support exceptions or alternative error handling mechanisms, so failure of type checking results in aborting the process. The semantics of exceptions within evolvers introduces new design challenges, and this case study in particular motivates a desire to access evolvable fields outside of an evolver. Furthermore, accessing the elements of arrays containing evolvable objects requires mutability and repeated swap operations. Extending evolution permissions to arrays and adopting semantics for destructing them ought to resolve this issue.

## 10 Future work

*Memoization and Self-Ownership.* One common pattern in software is objects that only initialize their data on the first access, but then memoize the result and return this result for the rest of their lifetime. But also included in the rest of their lifetime is a check for whether they have already computed their result.

In cases where it is known who the first accessor will be, MAY can already model this by making sure that said first accessor has an evolvable reference to the object to let it finish off its initialization explicitly. But it would clearly be interesting to have a more transparent way of achieving the effect of objects transforming *themselves* on first access.

In principle, this is a feasible addition to MAY. Certain classes would be marked as self-owning, and their constructors would only return non-evolvable references. Certain methods could then be marked with an attribute like `evo`, giving access to an evolvable `this` pointer. The key difficulty here is that this method has to be callable through a non-evolvable pointer, which means that lots of places—parallel threads or just different methods on the same stack—could do this at the same time. Therefore, just like a thread-safe version of the regular version of memoization, the method implementation would likely have to use some form of lock, which then re-does the dynamic method lookup in case the object has evolved while the current method was blocked waiting for the lock (this is similar to the scheme proposed by Damiani et al. [2004] for a multithreaded variant of Fickle, though monotonicity means that we do not have to worry about non-existent fields or methods). The good news is that just like existing versions of this pattern, the lock can be avoided once the object has evolved into its memoized form.

*Gradual Typing.* Although MAY is not gradual, idioms from dynamically typed languages were a motivating rationale for its design. The mismatch between static type systems and such dynamic idioms could be mitigated to some extent by implementing MAY-like evolution in a gradual setting.

The design of the overall system would lend itself to relatively cheap run-time casts. Essentially, each object would keep a binary reference count for whether there currently is a typed evolvable reference to it. If not, a reference to the object can safely be cast to an evolvable one, allowing evolution actions without keeping track of evolvable types everywhere.

*Generics.* We have not explored the relationship between evolvable types and generic types. We do not believe that there is any fundamental incompatibility, as evolvable references must be to the exact run-time type, and so in a system with generics, would always be to a fully resolved type. However, there are many possible interactions, each of which would need to be addressed (evolvable reference to a type parameter, evolvable fields of type parameters, etc.).

*Object Initialization.* Existing work on object initialization [Liu et al. 2020; Servetto et al. 2013; Summers and Müller 2011] already allows null-safe mutually recursive object initialization, though they require a relatively local *commitment point* at which a mutually recursive structure is fully initialized. MAY, by contrast, allows the initialization process to be drawn out through several evolutionary steps, while suffering from the drawback that not all parts of a mutually recursive structure would be able to see the most precise types of the other parts, as their evolution needs to happen according to some order. It would be interesting to adapt the idea of commitment points to both constructors and evolvers in MAY, and we see no reason why this would not be possible.

## 11 Related Work

Typestate [Strom and Yemini 1986] has its origins in detecting uninitialized variable access, but has since been generalized as a technique for proving properties about stateful protocols in object-oriented languages [DeLine and Fähndrich 2004b]. Early systems like Fugue [DeLine and Fähndrich 2004a] use annotations inside another OOP language to relate typestate and run-time state. Avoiding the aliasing restrictions this imposed, [Fähndrich and Leino 2003] introduces *heap monotonic typestate* to enforce heap invariants that may only become stronger, not weaker, clearly possessing the same monotonic spirit as object evolution. The Plural system [Bierhoff and Aldrich 2008] was the first to incorporate *permissions* for typestate programs [Bierhoff and Aldrich 2007] to allow checking in the presence of aliasing, achieved through fractional permissions [Boyland 2003].

Dynamic object reclassification has been a persistent feature of dynamically typed OO languages. Smalltalk [Goldberg and Robson 1983] includes a `becomes:` message that exchanges the class identity of two objects. SELF's [Ungar and Smith 1987] prototype system relies on the ability to

reclassify objects at run-time by adding new functionality to a prototype that can be cloned to create new objects, or adding new fields (including methods) to objects themselves. These features have persisted into modern languages like JavaScript and Python.

Fickle<sub>II</sub> [Drossopoulou et al. 2002] incorporates object reclassification into a statically typed language by restricting it to state classes. Their restriction prevents fields from having state class types and does not support object evolution, but on the other hand, guarantees safety without any need for permissions. More recently, work in this line has explored synchronizing the state of objects with an outside knowledge base at specific synchronization points [Sieve et al. 2025], with similar reasoning and restrictions to Fickle<sub>II</sub>.

Our permission-free system for object evolution is clearly similar to the inheritance approach of [Cohen and Gil 2009]. As discussed, this system treats failure of evolution as a runtime error. We expect our inheritance-based evolution to extend clearly to mixins and shakeins if desired. Furthermore, we contribute a formalization of object evolution.

Typestate-oriented programming (TSOP) [Aldrich et al. 2009], first implemented in Plaid, structures programs explicitly around typestate, in a way that clearly resembles Fickle<sub>II</sub>'s object reclassification, but permits enforcing state changes within the type system, along with aliasing permissions. Borrowed permissions can be used to temporarily share these permissions [Naden et al. 2012]. Borrowing has no analogue for us, the only operation evolvability provides is evolution, which would consume the permission in an irrecoverable way.

Featherweight Typestate (FT) is a formalisation of the principles of TSOP [Garcia et al. 2014, 2010]. Its expressive permission system pairs an object's current type with a state guarantee, allowing typestate changes in the presence of aliasing, given sufficient permissions that the guarantee remains. Despite certain aspects of our permission system embedding into the complex FT system, FT has no analogy for evolvable fields or evolvers, requiring a swap to be performed to extract the permission from a field, instead of within an evolver.

## 12 Conclusion

In this paper, we present our approach to type-safe monotonic object evolution in the MAY language. We have presented its formal semantics, proved type-safety, described certain implementation details, and illustrated that it is effective for use in general-purpose programs, as well as its ability to represent program patterns that are otherwise cumbersome, impossible, or unsafe. More importantly, we hope to have at least garnered some new interest in the relatively under-explored idea of object evolution.

### Data Availability Statement

The implementation of MAY and the case study [Mirrlees-Black et al. 2026b] will be submitted for artifact evaluation. The proofs and omitted definitions are available in the technical report [Mirrlees-Black et al. 2026a].

### Acknowledgments

We thank Sophia Drossopoulou, Stephen Kell, and the anonymous OOPSLA reviewers for their valuable feedback and suggestions on earlier drafts of this paper. Jeremiah Ikosin, Junhao Liu, Alex Potanin, and Ross Tate provided additional valuable input and discussion on the preliminary findings of this work.

This research was supported partially by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DE250100667). The views expressed herein are those of the authors and are not necessarily those of the Australian Government or Australian Research Council.

## References

- Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA 2009* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 1015–1022. doi:10.1145/1639950.1640073
- Nada Amin and Ross Tate. 2016. Java and scala’s type systems are unsound: the existential crisis of null pointers. In *OOPSLA 2016* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 838–848. doi:10.1145/2983990.2984004
- Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *OOPSLA 2007* (Montreal, Quebec, Canada). Association for Computing Machinery, New York, NY, USA, 301–320. doi:10.1145/1297027.1297050
- Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: checking protocol compliance under aliasing. In *ICSE 2008* (Leipzig, Germany). Association for Computing Machinery, New York, NY, USA, 971–972. doi:10.1145/1370175.1370213
- John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS 2003* (San Diego, California, USA), Vol. 2694. Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72. doi:10.1007/3-540-44898-5\_4
- Quentin Carbonneau. 2024. QBE 1.2. <https://c9x.me/compile/>
- Tal Cohen and Joseph (Yossi) Gil. 2009. Three approaches to object evolution. In *PPPJ 2009* (Calgary, Alberta, Canada). Association for Computing Machinery, New York, NY, USA, 57–66. doi:10.1145/1596655.1596665
- Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2004. Re-classification and multi-threading: FickleMT. In *SAC 2004* (Nicosia, Cyprus). Association for Computing Machinery, New York, NY, USA, 1297–1304. doi:10.1145/967900.968163
- Robert DeLine and Manuel Fähndrich. 2004a. *The Fugue Protocol Checker: Is Your Software Baroque?* Technical Report MSR-TR-2004-07. Microsoft Research. <https://www.microsoft.com/en-us/research/publication/the-fugue-protocol-checker-is-your-software-baroque/>
- Robert DeLine and Manuel Fähndrich. 2004b. Typestates for Objects. In *ECOOP 2004* (Oslo, Norway), Vol. 3086. Springer Berlin Heidelberg, Berlin, Heidelberg, 465–490. doi:10.1007/978-3-540-24851-4\_21
- Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2002. More dynamic object reclassification: Ficklen. *ACM Trans. Program. Lang. Syst.* 24, 2 (March 2002), 153–191. doi:10.1145/514952.514955
- Manuel Fähndrich and Rustan Leino. 2003. Heap Monotonic Typestate. In *IWACO 2003*. Utrecht University, Utrecht, Netherlands, 15 pages. <https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/>
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA.
- Ronald Garcia, Eric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. doi:10.1145/2629609
- Ronald Garcia, Roger Wolff, Eric Tanter, and Jonathan Aldrich. 2010. *Featherweight typestate*. Technical Report. Technical Report CMUISR-10-115, Carnegie Mellon University.
- Jean-Yves Girard. 1987. Linear logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. doi:10.1016/0304-3975(87)90045-4
- Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, USA.
- JetBrains. 2011. The Kotlin Programming Language. <https://kotlinlang.org/>
- Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. doi:10.1145/197320.197383
- Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 175 (Nov. 2020), 28 pages. doi:10.1145/3428243
- Alexandra Mirrlees-Black, Haoyu Wu, Gregor Richards, and Fabian Muehlboeck. 2026a. *Type-Safe Monotonic Object-Evolution*. Technical Report. Australian National University. doi:10.25911/SAAJ-JE93
- Alexandra Mirrlees-Black, Haoyu Wu, Gregor Richards, and Fabian Muehlboeck. 2026b. Type-Safe Monotonic Object-Evolution (Artifact). doi:10.5281/zenodo.18810359
- Fabian Muehlboeck and Ross Tate. 2021. Transitioning from structural to nominal code with efficient gradual typing. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 127 (Oct. 2021), 29 pages. doi:10.1145/3485504
- Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *POPL 2012* (Philadelphia, Pennsylvania, USA). Association for Computing Machinery, New York, NY, USA, 557–570. doi:10.1145/2103656.2103722
- Manuel Serrano. 1999. Wide Classes. In *ECOOP 1999* (Lisbon, Portugal). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–415. doi:10.1007/3-540-48743-3\_18
- Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix: Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP 2013* (Montpellier, France). Springer Berlin Heidelberg, Berlin, Heidelberg, 205–229. doi:10.1007/978-3-642-39038-8\_9
- Riccardo Sieve, Eduard Kamburjan, Ferruccio Damiani, and Einar Broch Johnsen. 2025. Declarative Dynamic Object Reclassification. In *ECOOP 2025* (Bergen, Norway), Vol. 333. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl,

- Germany, 29:1–29:31. doi:10.4230/LIPICS.ECOOP.2025.29
- Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. doi:10.1109/TSE.1986.6312929
- Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *OOPSLA 2011* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 1013–1032. doi:10.1145/2048066.2048142
- The FFmpeg Project. 2000. FFmpeg – ffmpeg.org. <https://ffmpeg.org/>. [Accessed 10-10-2025].
- Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *SNAPL 2017* (Asilomar, California, USA), Vol. 71. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:17. doi:10.4230/LIPICS.SNAPL.2017.17
- Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL 2011* (Austin, Texas, USA). Association for Computing Machinery, New York, NY, USA, 447–458. doi:10.1145/1926385.1926436
- David M. Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *OOPSLA 1987* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 227–242. doi:10.1145/38765.38828
- Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction*. MIT Press, Cambridge, Massachusetts.